

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technology

ANUPAM RAKSHIT - 194963IVCM

PRAGMATIC COMPARISON OF MACHINE LEARNING MODELS TO DETECT THE TYPE OF ATTACKS IN AN IOT NETWORK TRAFFIC

Master's Thesis

Supervisor:

Pelle Jakovits

Phd

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

ANUPAM RAKSHIT - 194963IVCM

**MASINÕPPEMUDELITE PRAGMAATILINE
VÕRDLUS RÜNNAKUTE TÜÜBI
TUVASTAMISEKS ASJADE INTERNETI
VÕRGULIIKLUSES**

Magistritöö

Juhendaja:

Pelle Jakovits

Phd

Tallinn 2022

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Anupam Rakshit

13.12.2021

Abstract

Rapid advancement of technology made the focus shift towards how fast and reliable any information can be shared over the internet. The use of Internet of Things devices has raised exponentially, with the aim of making life better and simpler. The usage of such devices also introduces many complications and security risks, if used neglecting cyber-hygiene. Through this paper, it is intended to analyse how several attacks in a network of IoT devices operating together, can be detected, and predicted for real-time intrusion detection. As data generated from any organisation of these devices can be huge in terms of size, efficient techniques of handling large datasets by considering some big data tools, leveraging the power of parallel computing are also covered. An extensive comparison of the techniques is performed to find out the optimal settings and the best performing model is found out. An application written in python that can be easily configured for several use-cases and used in the large-scale environment in a practical scenario, also its wide possibilities are discussed.

List of abbreviations and terms

RFID	Radio Frequency Identification
WSN	Wireless Sensor Networks
IoT	Internet Of Things
AI	Artificial Intelligence
SDN	Software Defined Network
NFV	Network Function Virtualization
ML	Machine Learning
IDS	Intrusion Detection System
NGFW	Next Generation Firewall
DDoS	Distributed Denial of Service
MITM	Man in the Middle
DL	Deep Learning
NIDS	Network Intrusion Detection System
MUD	Manufacturer Usage Description
TCP	Transmission control Protocol
UDP	User Datagram Protocol
ICMP	Internet Control Message Protocol
ARP	Address Resolution Protocol
EKS	Elastic Kubernetes Service

Table of Contents

AUTHOR'S DECLARATION OF ORIGINALITY	3
ABSTRACT	4
LIST OF ABBREVIATIONS AND TERMS	5
1 INTRODUCTION	8
1.1 MOTIVATION.....	9
1.2 PROBLEM STATEMENT.....	10
1.3 GOAL.....	11
1.4 SCOPE.....	12
2 BACKGROUND	13
2.1 IOT.....	13
2.2 MACHINE LEARNING (ML) AND ARTIFICIAL INTELLIGENCE (AI).....	15
2.2.1 <i>Types of ML</i>	17
2.2.2 <i>Algorithms</i>	18
2.2.3 <i>Evaluation Metrics</i>	26
2.2.4 <i>Efficiency metrics</i>	27
2.3 BIG DATA PROCESSING WITH APACHE SPARK.....	30
2.4 RELATED WORK.....	31
3 METHODOLOGY	34
3.1 APPLICATION ARCHITECTURE.....	35
3.1.1 <i>Running the application</i>	35
3.1.2 <i>Architecture</i>	36
3.2 CHOICE OF DATASET.....	39
3.3 DATASET ANALYSIS.....	42
3.4 TRAINING AND CREATING MODELS.....	43
3.5 TUNING WITH HYPER PARAMS.....	44
3.6 EVALUATION OF MODELS.....	45
3.7 APPLICATION OUTPUT.....	46
4 CATEGORIZING ATTACKS	47
4.1 ENVIRONMENTAL SETUP.....	47
4.2 DATA PREPARATION STEPS.....	50
4.2.1 <i>Data Pre-processing</i>	50
4.2.2 <i>Data Analysis</i>	55
4.2.3 <i>Dimensional Reduction</i>	61
4.3 BENCHMARKING WITH BINARY CLASSIFIERS.....	62
4.4 CATEGORIZING WITH MULTICLASS CLASSIFIERS.....	65
4.5 TUNING THE MODELS.....	69
5 EVALUATION TECHNIQUES	72
5.1 MECHANISM.....	73
5.2 METRICS.....	74
5.2.1 <i>Performance Metrics</i>	74
5.2.2 <i>Efficiency Metrics</i>	77
6 RESULTS AND COMPARISON	78
6.1 COMPARING WITH THE BENCHMARKS.....	79
6.1.1 <i>Outcome of Hyperparameter Tuning</i>	80
6.1.2 <i>Outcome of Feature Reductions</i>	83
6.1.3 <i>Outcome of Parallel Computations</i>	86

6.1.4 Overall Results.....	90
7 APPLICATION AND ITS POSSIBILITIES.....	92
7.1 REAL LIFE USE CASE.....	92
7.2 WAY FORWARD.....	94
8 CONCLUSION.....	95
REFERENCES.....	96
APPENDIX 1.....	99
APPENDIX 2.....	100
APPENDIX 3.....	100
APPENDIX 4.....	100
APPENDIX 5.....	102

1 Introduction

IoT devices are very widely accepted in the world to improve productivity and enhance the quality of life of the people, along with that, it also induces a huge risk of potential threats from adversaries and other cybercriminals. A few years ago, Hewlett Packard revealed in an article that almost 70% of the IoT devices in use contains serious security vulnerabilities [1]. There has been much research and surveys which addresses mostly the security and privacy in IoT architecture. Understanding the IoT space requires a classification of the whole system architecture where multiple devices are connected in a shared network and processing terabytes of data. The multiple layers of the IoT architecture are explained well in [2] and their respective security challenges. As studied in [3] the most prominent security concerns are identified in any of the four layers of the IoT architecture.

Considering the explosive increase of IoT data over the past few years, an efficient classification technique won't be sufficient soon. Igor and his fellow mates in their paper [4] proposed an efficient approach to detect network anomalies using machine learning and big data processing comparing the computation time and accuracy of the detection system. This suggests the significant move towards parallel computing with the capacity to handle a large amount of data and fast computation are all a major concern in terms of anomaly detection in any critical infrastructure.

Several types of research have been carried out regarding building efficient ML models for purposes like building more efficient IDS, NGFW, used in SDN, detect specific types of attacks like DDoS and other Volumetric attacks, MITM attacks etc. However, the idea of building models to detect anomalies or intrusion is not new. D.E. Denning in his paper An Intrusion-Detection Model [5] talks about a general-purpose model based on a hypothesis that could detect any intrusion by analysing the audit logs of the devices. Machine learning is the most sought technique used in many applications like fraud detection, malware detection, bot-net detection etc. because it can simulate the behaviour of any object in restricted environments. Fatima and others in their paper [6] performed a systematic review of the various security requirements, attack vectors and existing

security solutions in IoT networks and proposes Machine Learning and Deep learning approaches in dealing with different security problems in IoT networks.

1.1 Motivation

At present every human being is somehow connected to an IoT device, directly or indirectly some of his/her information is gathered and transmitted through the network. As a result of which a lot of information is available that needs to be guaranteed that the information is reached to the intended person and processed securely. The increase in demand for these devices explains the number of vendors currently present in the market. Most of their priority is a shorter Time-To-Market rather than security or data privacy which is mostly neglected or considered as an after-thought [7]. These devices are often very cheap and readily used without any vendor verification or background knowledge, which increases the risk of leaking private data over the internet. Those interested in this data can cause serious harm to an organization depending on the motive of the attack.

According to a report published by Kaspersky [8], cyberattacks in IoT devices escalated in the month of January to June. Some 1.51 billion devices were breached mostly using telnet remote access protocol which is a drastic increase from 639 million in 2020, as reposted by Kaspersky. Another article [9] published by Cyber Magazine claims an average of 5,200 IoT devices suffer cyber-attacks every month, as IoT devices collect and process many essential data every day, these attract most cybercriminals.

These findings confirm that the Covid-19 pandemic has only aggravated the vulnerabilities in IoT devices by prolonging the usage of these devices in minimal network settings, with a lack of security protocols. New technologies like artificial intelligence, deep learning models to predict and detect anomalies in the IoT network have presented many opportunities but also complicated the cyberspace and data security landscape. With the growth in the adaption of IoT devices, the demand for IoT security has also increased and there is a potential growth in AI-based systems using ML models like support vector machines, linear regression, decision trees, neural networks etc. to identify threats and potential attacks. Several signature-based detection mechanisms are also available for detecting malicious files and activities of systems, the earliest Intrusion Detection Systems (IDS) rely heavily on signature definitions. But with the advancement of technologies, malwares became more advanced using new techniques like

polymorphism, to change the pattern each time the object would spread from system to system. These change in the behavior of the malwares or any other attack vectors makes it difficult for signature-based systems to identify and flag potential threats. Hence, this study shifts towards behavior-based detection techniques like applying ML models in sophisticated IDS, to be able to detect as well as predict future anomalies in a system [10].

Most researches covering anomaly detection techniques using IoT network data uses sophisticated ML models to analyze and detect malicious data. However, there is a lack of study, as discussed in the related work section, regarding extensive comparison of the efficiencies along with performances of the models and their usage in analyzing and detecting the IoT network together with the efficiency of the system.

1.2 Problem Statement

Relevant work in this domain discussed in the above section highlights the extensive amount of research in IoT networks. Most relevant studies talk about coming up with an efficient model to detect a malicious device or network traffic. To mitigate any security incident, knowing the type of attack threat beforehand is critical. The more familiar the attack is, the quicker it can be resolved. In most scenarios, the analysis and identification of the threat consume a larger portion of the total time required to resolve the incident.

For this reason, it is important to be aware of what methods are efficient and highly performative and can be used reliably. ML-based applications are extensively used nowadays to detect anomalies but there are quite a several models that can correctly detect threats. Hence, it is required to have a comparative analysis of the performances of the models with a large dataset replicating the most practical dataset possible. This paper aims to solve the problem, it focuses on both binary and multiclass type of ML algorithms to compare the performances of them and provide a comprehensive analysis of the same.

Many IoT devices generate a huge amount of data at a regular interval of time. Not only detecting anomalies correctly from the dataset is important but to be able to carry out the whole process in a measurable amount of time is another high priority requirement. Generally, the dataset considered in most of the studies contains a limited number of attacks and there are inadequate studies as outlined in the literature review

section considering the distributed environment, parallel computations in terms of anomaly detection in IoT networks.

1.3 Goal

This study aims to provide a practical approach for a comprehensive analysis of the ML models in handling large volume of data, their performance comparison and behaviour in different environmental setups. The major goals of the study can be summarized as follows:

1. Using supervised ML techniques, perform a thorough comparative analysis of multi-class models against their respective trained binary models in terms of performance metrics (accuracy, test error, true positive, true negative, false positive, false negative, F-score, recall)
2. Identify the best model along tuned with an optimum configuration for the proposed system and validate it against an untrained dataset.
3. Analyse the impact of parallel computation while achieving the above-mentioned goals in terms of efficiency metrics such as Speedup, Efficiency, Scalability.

In order to reach the above major goals, the below mentioned secondary goals can be seen as important milestones:

- a. Design an application with pipeline architecture that is modular, highly configurable and performs the comparisons in a scalable manner.
- b. Leverage the advantages of cloud-environment to deploy the system that can handle a large set of data (up to 65 gigabytes).
- c. Demonstrate the reusability of the proposed application to read from a raw network data capture as the input dataset and perform identification of the attack types and deliver the best model.

1.4 Scope

This study attempts to take a practical and comprehensive approach to achieve the above-mentioned milestones. Taking into account the limited power and resources, the IoT devices are made of, the computation is carried on a cloud environment where the input data is emitted from the IoT devices. An efficient architecture is proposed which is capable of reading network traffic from these devices and uses ML algorithms to process the data and identify different types of attacks, in a parallel fashion.

The main source of data, this study relies on, contains 4 different categories of attacks and this paper focuses on 3 of those identifying 1 from each category of attacks. It depends on the existing models supported by Apache Spark and is limited to core Machine Learning techniques. An attempt is made to analyse the effects of neural network for categorization of the attacks but is limited due to resources and time consumption.

The efficiency of the parallel computation calculated is based to a standalone cluster configurations rather than simulating a typical large scale multi-cluster production environment that requires huge memory and computational power. The main aim of the paper considering the computational environment is to analyse the possibilities to incorporate state-of-art Big Data tools and be able to scale efficiently to handle larger amount of data efficiently when need arises. It also ensures the application built is ready to be used in a real big data cluster without significant modifications.

The following are the sections organised to achieve small milestones which cumulatively results in the major goals of this study. Section 2 provides some background information of the relevant terms used extensively in this study. Such fields may include IoT, Machine Learning and Deep learning, its various types, the algorithms ML uses to train the data and information on the evaluation metrics to understand how those are useful in terms of evaluating a model. It contains some insights about computing using Apache Spark and its significance in handling big data and parallel computing. And some related work around these topics to counter various problem scenarios with IoT network security. Section 3 provides us some insights about the application being developed for this study, help us understand about the choice of the dataset, the several steps required for Machine Learning process used in this study and its outcome. Section 4 details about the

identification and categorization of the attacks, discusses the approaches taken to meet the final goal, covers details of the implementation of the ML steps and the configurations used to run the application. Section 5 covers mostly validating the outcome of Section 4, evaluating the results and the evaluation techniques used. It also calculates the evaluation metrics for each ML models in consideration. It also covers the efficiency metrics of the system with the help of various data logged during the identification process. Section 6 contains the results of all the approaches taken in Section 4. It compares each metrics to understand the performance of the ML models in different settings. Section 7 covers the real-life use cases of the application and highlights various possibilities it can be useful to. Finally Section 8 concludes the study by analysing the results of the ultimate outcome of the study and justifies the goal of the thesis.

2 Background

Some of the phrases extensively used in this thesis are machine learning (ML), IoT, artificial Intelligence, Apache Spark etc. This study expects some basic knowledge in these fields to grasp the importance of the goals it intends to achieve. This section provides some background information of these relevant fields and their significance in this thesis. It also covers few of the techniques heavily used in the field of machine learning and provides some insights about metrics used to evaluate the outcome of the milestones.

2.1 IoT

The phrase ‘Internet of Things’ started as early as in 1999, when Kevin Ashton presented a paper on a new device linking RFID to the supply chain of Proctor & Gamble (P&G) [11]. It was discovered that within the last decade, the number of IoT devices registered was as large as 15 billions, which means an average of 2 devices per person [12]. Thanks to the cheap integrated chips and the fast connectivity through internet, literally any device from a hairpin to large aeroplane, can be termed as an IoT device or a system with numerous such devices.

As the usage of these devices increase, more information is shared through the network with an aim to make life easier. At present every human being is somehow connected to an IoT device, directly or indirectly some of his information is gathered and transmitted through the network.

Industrial Internet of Things (IIoT) is referred to the fourth revolution of the industries and are more focused on the usage of IoT device in a business setting. The concept is quite similar to that of the consumer usage of the IoT devices but with huge number of sensors, wireless networks, cloud technology, big data and analytics with a common aim to improve the business process.

Any IoT architecture can typically be described as an architecture with 3-4 tier commonly used. Below figure shows 4-layer IoT architecture overview. It depicts the security challenges faced in a IoT device mapping them with four layers of the architecture involved:

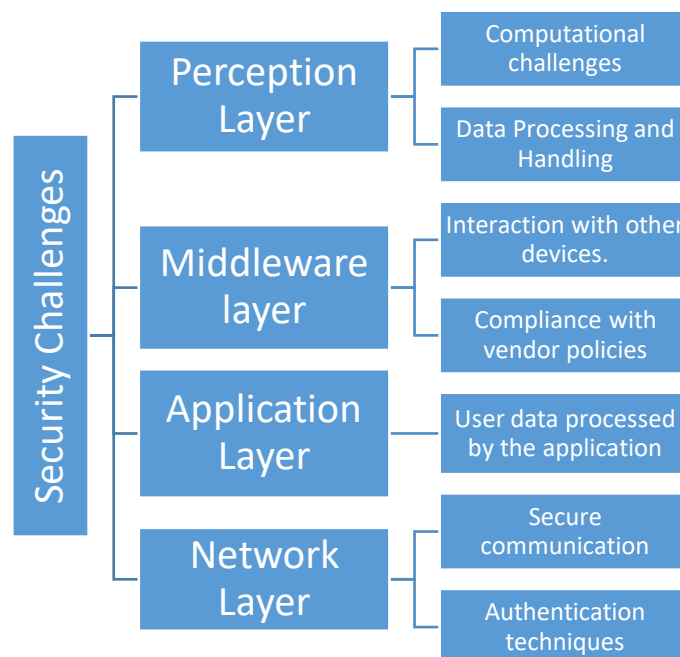


Figure 1. Four-layer IoT architecture [3]

There can be different types of threats based on the vulnerabilities present in the system. Based on the above layers, the possible threats on any IoT architecture in various sectors can be summarized as follows:

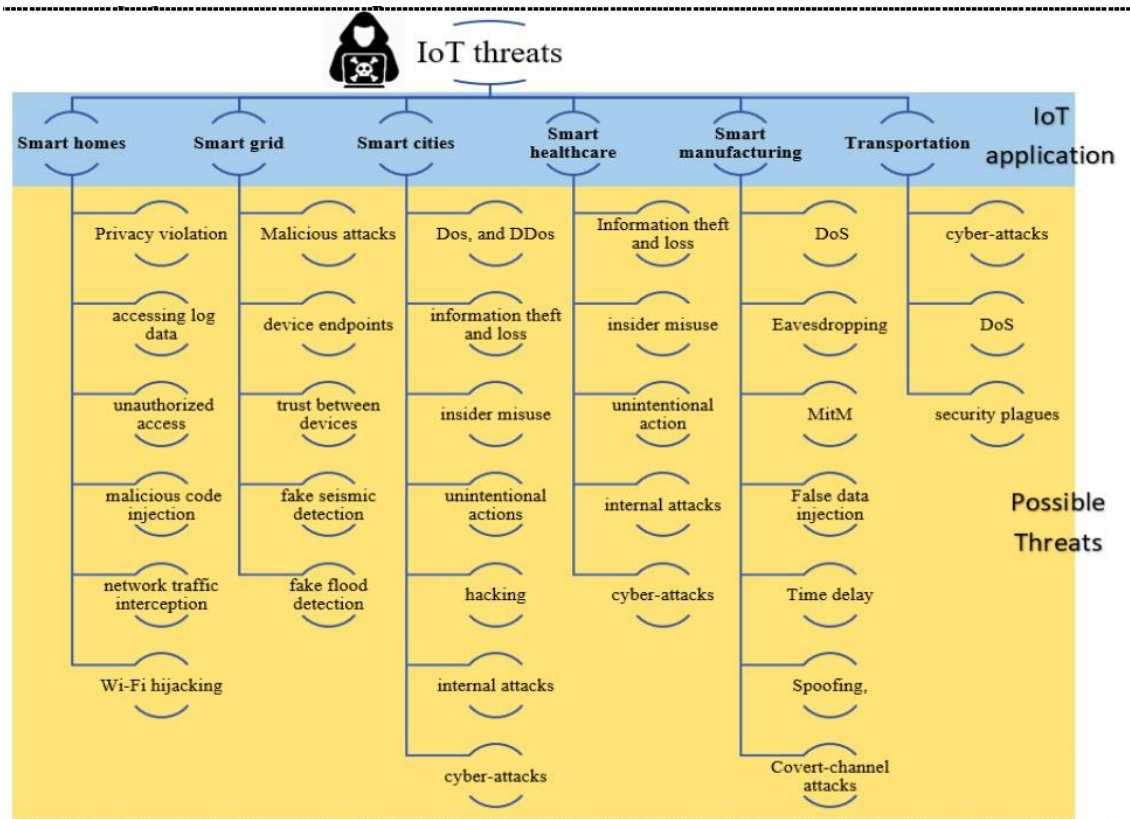


Figure 2. Possible threat in IoT architecture [13]

The above figure denotes the wide area of implications of the IoT devices and corresponding attacks in that domain. It shows how each of the field dealing with separate type of functions are prone to specific types of attacks. Most vulnerable of them would be the smart city appliances which are generally available for public usage. It also provides us an idea of the potential research areas in such fields and provides some clue for more possibilities in future research work related to security and privacy for related devices in these fields.

Attacks like network traffic injection, DoS, DDoS, eavesdropping attacks, MitM attacks are some of most common attacks in today's IoT world. These attack types are some of the major focus areas of this study.

2.2 Machine Learning (ML) and Artificial Intelligence (AI)

Machine learning as the name says is a method for building applications which can learn from its own data during the process to gain better accuracy overtime without any human interference [14]. It is a part of Artificial Intelligence (AI), where models are created with definite algorithms to train the machines with sample "training" data. These models once

trained are then used for predictions or analysis of the result over a larger and more practical dataset without any re-programming.

Few of the machine learning subsets includes computational statistics, focusing mostly on making predictions using computers. The study of mathematical optimization deals with improvement of the algorithms, mathematical theories, and its application domains to the field of machine learning. Data mining, yet another field uses machine learning for exploratory analysis of data and predictive analysis includes its application in business problems.

Following figure illustrates the field of machine learning in terms of Artificial Intelligence and relative fields like Deep Learning.

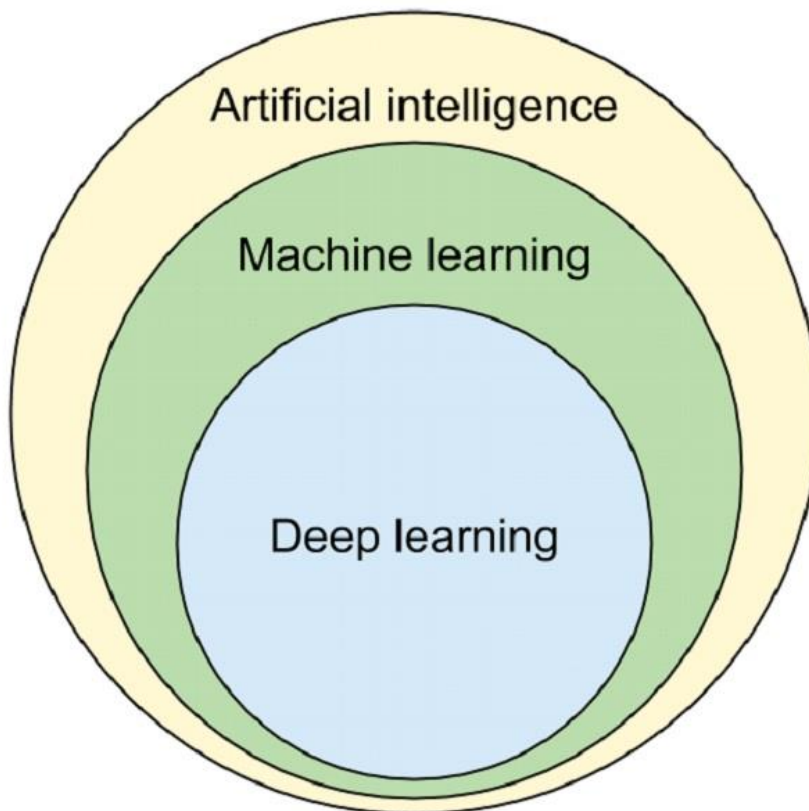


Figure 3. Machine learning as a sub-field of Artificial Intelligence [15]

2.2.1 Types of ML

Machine learning can generally be of the following types:

Supervised Learning uses labelled examples from the past data to predict unknown events. The known datasets are analysed with an algorithm that infers function to make prediction about the output values. The learning algorithm also validates itself by comparing the future outputs with the intended ones and calculate metrics that significantly help us understand any accuracy of the model [16].

Un-supervised Learning is quite the opposite of supervised learning method i.e., there are not enough evidence from the past data for the machines to learn. Essentially when the training dataset does not include instructions or appropriate labels of the type of records it contains, unsupervised learning techniques are considered. Unsupervised ML techniques specializes on algorithms that can learn about possible future events from an unlabelled data.

Semi-supervised Learning typically falls somewhere in between the above two categories where the training dataset may or may not contain indications of the type of data it contains. Semi-Supervised Learning (SSL) was introduced to observe the behaviour of using labelled and un-labelled data as input dataset together in machine learning.

Reinforcement Machine Learning includes a special type of algorithms where it learns from the environment by producing actions and discovers errors or rewards. This type of learning mechanism helps to automatically determine the ideal behaviour of a system within a specific context to maximize its performance.

Generally, in machine learning two mostly used modes of learning techniques are – supervised and unsupervised learning. Binary Classifiers can be used as the supervised domain of learning modes, as the training dataset is labelled which means manual identification of the data is necessary if those are normal/malicious before the model is trained [17].

Binary Classifiers can detect only two states/class of the data, for example, spam or not spam, malicious or normal data i.e., this type of classifiers can only be used on a single attack type of data, in our application. These models can predict the Bernoulli distribution

of the data. Bernoulli distribution is the discrete probability that predicts a case where the state of a particular data can be either 0 or 1 [18].

2.2.2 Algorithms

Popular algorithms that can be used in binary classification are:

- a. Logistic Regression
- b. Decision Trees
- c. Random Forest
- d. Gradient-Boosted Trees
- e. Naïve Bayes

In this paper most of the above algorithm fit for the classification problem are taken for evaluation and comparison – Decision Trees, Random Forest, Gradient Boosted Trees, and Naïve Bayes are described in detail in the following sections. A brief background of the Multi-layer Perceptron is also described in the below section which is used for multi-class classification with an attempt to explore the basic configuration of a neural network.

2.2.2.1 Logistic Regression

Logistic Regression is a popular Generalized linear model to predict categorical response [19]. It can be used to predict a binary outcome by using a binomial logistic regression as well as predict a multiclass outcome by using a multinomial logistic regression by using the family parameter that is used to switch between the two variants. Multinomial logistic regression can also be used for binary classification by setting the family to “multinomial”, as it produces two set of coefficients and two intercepts.

Multinomial Logistic Regression produces K set of coefficients, or a matrix of $K \times J$ dimensions where K is the number of outcomes and J is the number of features. The conditional probabilities of the outcome classes $k \in 1, 2, \dots, K$ are modelled using the softmax function:

$$P(Y = k|X, \beta_k, \beta_{0,k}) = \frac{e^{\beta_k \cdot X + \beta_{0,k}}}{\sum_{k'}^{K-1} e^{\beta_{k'} \cdot X + \beta_{0,k'}}$$

- X is the vector of the explanatory variables describing the observation I,
- where β_k is the vector of weights corresponding to the outcome k,
- $\beta_{m,k}$ is the regression coefficient associated with the m^{th} explanatory variable and k^{th} outcome.

2.2.2.2 Decision Trees

A simple predictive analysis approach used in machine learning to achieve conclusions about the item's target value. Tree models where the target takes a discrete set of variables called classification trees. In this tree structure, the leaves represent the class labels, and the branches represents the conjunctions of the features that led to that decision.

Decision trees could of two types:

Classification tree analysis is when the outcome is predicted a discrete value.

Regression trees where the outcome is continuous and is considered a real number.

Classifying with Decision Tree:

As the name suggests, Decision Trees classifies data items by taking decisions in every step of the process. It poses a series of questions on the features associated with the data. The whole process is distributed into multiple steps in the form of a hierarchy where each question is considered as a node and a decision is made when each node is executed. Based on this decision another node is executed which can be referred as the child node. This complete process resembles a tree structure where each node results in a 'yes' or 'no' result [20].

Decision Trees are well suitable to handle a mix of real-valued, categorical values as well as some missing features. Decision Trees normally support classification problems but can also be modified to handle regression problems. For these, two most common measures are entropy and Gini index.

If the training set, E is used to classify m classes, and $p_i (i = 1, 2, 3 \dots m)$ be the part of the items that belongs to class i . We measure of the set E with the entropy of the probability distribution $(p_i)_{i=1}^m$.

The entropy is measured as:

$$\sum_{i=1}^m p_i \log p_i = \begin{cases} \min, & \text{when a single } p_i = 1 \\ \max, & \text{when } p_i = p_{i+1} \end{cases},$$

The Gini index measure by:

$$\left(1 - \sum_i^m p_i^2\right) = 0, \text{ when } E \text{ has a single class items}$$

In general, Entropy quantifies the measure of information in a random variable, or more specifically its probability distribution. From the above equation, we can infer that if the data is skewed, it contains less information hence low entropy and more equally distributed data, the entropy is high.

Information gain is typically the “surprise” of the event. Information is expected to be high in dataset with low probability and low with high probability distribution. Hence, dataset with equal probability distribution have more surprises and high entropy. Information gain is measured with Kullback-Leiblar divergence always has non-negative values [21].

Although, single decision trees often provide good results, these can also be applied as a collection of multiple trees. This process is called ensembles of decision trees which are often excellent classifiers. *Random Forests* and *Boosted Trees* are two most efficient ensemble strategies of decision trees considered in this paper are discussed thoroughly in the next sections.

2.2.2.3 Random Forest

Random Forest, an ensemble of decision tree, builds a forest with the combination of learning models from numerous decision trees to generate accurate and stable results [22]. According to Wikipedia, a random forest can be defined as a decision tree with each internal node representing a test which decides the outcome as “yes” or “no” on an

attribute. This iteration is carried over until a class label is found. Each branch determines the outcome of the test and the final node of the branch with the result is the leaf node.

Feature importance is a vital quality of random forest and is measured by the relative importance of each feature on the prediction. Too much increase in the number of features would also increase the risk the model of overfitting and similarly, too less of the features may result in underfitting. Hence, it is necessary to find out what features are of utmost importance in determining the predictions for each tree.

In decision trees, the features and labels from the training data is used to generate some set of rules to make the predictions, on the other hand random forest would randomly select the features and labels to formulate the rules and then averages the results after several decision trees.

Random Sampling is the process where the estimator learns from random data points while it is being trained. The overall idea is to gain low variance among the features of the dataset although some samples might be used multiple during random selection in a single tree which is known as *bootstrapping*. At the next step when the trained model is tested in the test data, the predictions are made by averaging the predictions of each decision tree. This process of separate bootstrapped subsets of data being trained in an individual tree and finally, averaging out the prediction results is known as *bootstrap aggregating* [23].

2.2.2.4 Gradient Boosted Trees

Unlike the Random Forests, which calculates the average at the end of the process, Gradient Boosted Decision trees (GBT) rather starts combining the decision trees at the start of the process. GBTs is also an ensemble of Decision tree, it also combines the results of various decision trees underneath to generate the overall result.

A series of sequential steps designed to take decisions in each step to answer a question and provide predictions in terms of probabilities, costs, and other consequences of making the decision. GBTs are easy to understand but comes with some serious flaws as explained below:

Overfitting: When the process is tuned with one large tree with maximum depth, fair chance of overfitting arises. This may also happen due to the presence of noise.

Error due to Bias: Limiting the target functions with too many restricting functions like linear equation often results in bias in the result.

Error due to variance: This occurs as decision trees are highly dependent on the training dataset. If the dataset is changed slightly, high variance could occur which would result in variance error hence, large change in the result.

Gradient Boosts can achieve better results than random forests but needs to be tuned very carefully. If the data has larger noise, GBTs are not a good fit, as it results in overfitting. GBTs perform well when the dataset is unbalanced, for instance, performing real time risk assessment. However, GBT are not suitable for multiclass analysis of the dataset.

2.2.2.5 Naïve bayes

Bayes theorem is used in many inferential statistics and many advanced machine learning models. Bayesian reasoning is the probability of updating the hypothesis considering new evidence where frequentist statistical approaches were not developed [24].

If A and B are two events in a sample space Ω and P be the probability distribution of those events in Ω such that $0 < P(A) < 1$ and $0 < P(B) < 1$, and $P(\Omega) = 1$, and the occurrence of each events is not dependent on that of the other, i.e., they are mutually exclusive, then the simplest form of Bayes theorem can be denoted as:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

- where $P(A/B)$ is the conditional probability of A given that B has already occurred,
- $P(B/A)$ is the conditional probability of the occurrence of B given that A has already occurred.

Naïve Bayes classifiers are based on applying Bayes theorem with strong (naive) independence assumptions among its features [25]. It is a family of simple probabilistic multiclass classifiers that can be trained very efficiently. Despite its unrealistic assumption, its often very effective and successful in certain cases in medical diagnosis,

system performance management etc. and often competes with more sophisticated techniques [26].

The different types of bayes classifier Apache Spark supports are – Multinomial naïve Bayes, Complement naïve Bayes, Bernoulli naïve Bayes, Gaussian naïve Bayes. These multinomial, complement, Bernoulli models are typically used in document classification, where each feature is a term, and each observation is a document. In this study the gaussian naïve bayes classifier is considered as from the feature investigation done for *col_03* in the below section, it can be observed that the graph somewhat resembles a gaussian distribution.

2.2.2.6 Multi-layer Perceptron

MLP (Multi-Layer Perceptron) is a typical neural network which consists of three layers – input layer, hidden layer, output layer [27]. This is a very basic neural network and due to its simplicity can be often termed as “vanilla” neural network. This algorithm is considered for categorizing the attacks in the input dataset which paves the way for deep learning.

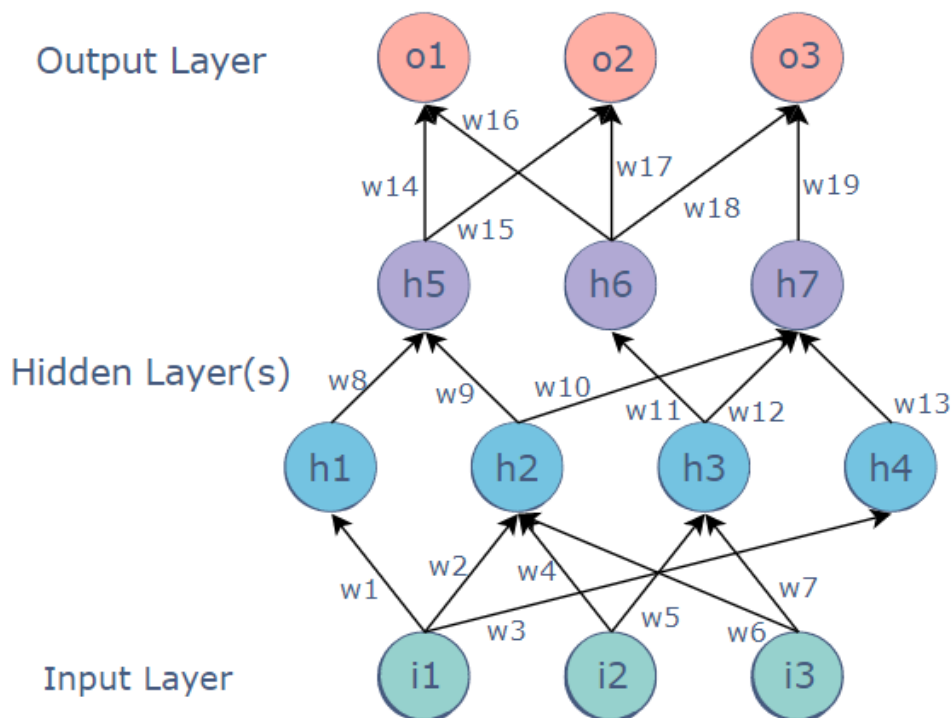


Figure 4. MLP architecture [28]

The figure above shows the three-basic layer of MLP, where $w_1, w_2 \dots$ are the weights of the connections. The input layer receives the input data, and the output layer provides the prediction and probabilistic measure of the model, and the hidden layer does all the computation analysis of the model. There can be arbitrary number of the hidden layers present in between the input and the output layer depending on the resources and computational power of the system. Hence its name as the multi-layer perceptron as it extends the simplest and the oldest Perceptron network which was initially used for binary classification.

MLP can be used as a bi-directional neural network i.e., forward propagation and backward propagation.

Feed-forward Network: The layers in the MLP can be designed to work in such a manner that the input layer accepts the input and propagates the result of each neuron in a forward direction to the output layer. Inputs are multiplied with their calculated weights

and fed to the activation function whose result is an input to another neuron. From the figure above, we can calculate the input of node h7 as:

$$\underline{h7 = h2 * w10 + h3 * w12 + h4 * w13}$$

Feed-forward Networks are typically less complex, fast, and easy to design. However, it cannot be efficiently used in deep learning due to absence of dense layers and back propagation [29].

BackPropagation: MLP can also be tuned to use back-propagation technique. It is quite the opposite of feed-forward propagation which means the output of a neuron is used as the input of another neuron to improve the error. In conventional MLP, supervised machine learning methods are used to calculate the loss in each node. For each node, the wrights are randomly assigned to calculate the value and then compared with the trained data and then re-calculated and re-adjusted to minimise the error. The variance of the calculated result and the provided result in the input data is known as the error. They self-adjust depending on the calculated outputs and the training input.

This is an important advantage of the MLP to use back-propagation technique which makes it useful in deep learning networks due to the presence of dense layer and its ability to self-correct its outcome for every neuron.

Apache Spark's MLP Classifier is based on feed forward neural network where all nodes map the inputs to output by linear combination of the weights (w) and bias (b) and applies an activation function. Hence, if MLPC have K + 1 layers, the above function can be represented as:

$$y(x) = f_k(\dots + f_2(w_2^T f_1(w_1^T x + b_1) + b_2)\dots + b_k)$$

The intermediate layers (hidden layer) use the sigmoid function:

$$f(z_i) = \frac{1}{1 + e^{-z_i}}$$

And the nodes in the output layers use the softmax function defined as:

$$f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^N e^{z_k}}$$

, where N = number of nodes in the output layer corresponds to the number of classes.

2.2.3 Evaluation Metrics

Evaluation is the vital step which let us know how well the model performed. To test various models, numerous evaluation metrics are considered and compared with each other. There are many evaluation metrics each with its own benefits and drawbacks.

Spark provides evaluators like *BinaryClassificationEvaluator* and *MulticlassClassificationEvaluator* are used to evaluate the models and calculate certain metrics and are described in detail in the below sections. Python sklearn library is also used to calculate metrics for comparison.

2.2.3.1 Confusion Matrix

Confusion Matrix is calculated in almost every machine learning process. It calculates the true positive(tp), true negative(tn), false positive(fp), false negative(fn) predictions. It is extremely useful for measuring precision, recall, accuracy, and area under ROC. It is typically denoted as a matrix: [30]

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Figure 5. Confusion Matrix

- where:
 - TP -> True Positive i.e., the predicted value is positive and is same as that of the actual value.
 - TN -> True Negative i.e., the predicted value is negative and its true.

- FN -> False Negative i.e., the predicted value is negative and its incorrect. (Type Error 1)
- FP -> False Positive i.e., the predicted value is positive and its incorrect. (Type Error 2).

2.2.3.2 Recall

$$Recall = \frac{TP}{TP + FN}$$

Recall signifies, out of all positive classes how many of them are predicted correctly by the model. The higher the value the better it is.

2.2.3.3 Precision

$$Precision = \frac{TP}{FP + TP}$$

Precision signifies, out of all predicted positive class how many are actually positive.

2.2.3.4 F-Score

$$F - measure = \frac{2 * Recall * Precision}{Recall + Precision}$$

F-score makes two models comparable, especially when the Recall and Precision of them differ. It uses Harmonic Mean instead of Arithmetic Mean by punishing the extreme values more.

2.2.4 Efficiency metrics

In essence of parallel computing, several CPUs execute one or multiple tasks simultaneously at a given span of time. In software programming, two common patterns of parallel computing are *pipeline* and *divide-and-conquer* [31]. In a serial programming the performance is often measured in terms of time and memory requirements of the program. On the other hand, in parallel computing, we have a myriad of measures i.e., speedup, scaled speed up, efficiency, iso-efficiency, serial factor etc.

A brief description of the metrics considered for this study are as follows:

2.2.4.1 Speedup

Time and memory are both dominant metrics in parallel computing as in serial computing. When there is available memory to run two computations, we would prefer the algorithm that completes the tasks faster. There is not much of a significance to have a lesser memory to run an application, if it takes much longer time to complete. Hence time complexity is a crucial metric in both the computation techniques to measure the performance of an algorithm. But in parallel computing, if we want to analyse the time complexity of the application, we will have to calculate the time complexity of all the processors participating in computations. Additional architectural details of interconnected topology and memory access properties are also required to be known. The architecture of the system and the algorithm together defines the parallel system; hence we consider the time complexity of the whole system.

In practice, to calculate the speedup(S) of the parallel systems with that of the serial system, the following ration is defined [32]:

$$S(p) = \frac{\text{execution time in serial computation, } T(1)}{\text{execution time in parallel computation, } T(p)}$$

There are some diversities in terms of defining the Speedup of an application which results in at least 5 different types of definitions of speedup [32]:

Relative Speedup, Maximum Relative Speedup, Real Speedup, Absolute Speedup, Asymptotic Real Speedup.

There are certain limitations as well to increase the number of processors to increase speedup, details of which are beyond the scope of this study.

2.2.4.2 Efficiency

It is defined as the ratio of the speedup and the number of processors (P). Efficiency is closely related to *Speedup* and depending on the variety of the speedup, efficiency may also change. Efficiency is the measure of the usage of the computational capacity and can be denoted as:

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \times T(p)}$$

- where $S(p)$ is the speed up of the application for p processing units.

Efficiency essentially measures the fraction of time for which the parallel executions have actually happened. The CPUs cannot devote 100% of their time to compute the parallel algorithm, rather some amount of resource is used to communicate among the machines, administration work required for the parallel execution etc. An ideal parallel computation would have $E(p)$ as 1, but in practice, parallel systems have efficiency ranging from 0 to 1.

2.2.4.3 Scalability

It measures how scalable the parallel computation is i.e., when the number of processors is kept on increasing keeping the algorithm constant, the efficiency will reach a certain saturation point, after which the efficiency won't increase with the number of processors. This is when the processors are not utilized efficiently and there are no further computations available for the processors. A parallel program is scalable when the efficiency increases with the algorithm size keeping the number or parallel executions constant i.e., its performance continues to improve with the increase in size (both in terms of complexity of the algorithm and the system resources) [32].

Asymptotic scalability is one of the measures of scalability and as defined by Nussbaum and Agarwal [33] of a machine for a given algorithm is the ratio of the asymptotic speed up of the machine to that of an ideal Extensive Reads Extensive Writes Parallel RAM (EREW PRAM). It is mathematically denoted as:

$$S_I = \frac{T_{par_I}(S)}{T_{seq}(S)},$$

$$\text{and } S_R = \frac{T_{par_R}(S)}{T_{seq}(S)}$$

- where S_I is the asymptotic speed up of an ideal machine and S_R is the asymptotic speed up of a regular machine, then

$$\psi(S) = \frac{S_I(S)}{S_R(S)} = \frac{T_{par_I}(S)}{T_{par_R}(S)}$$

- here $\psi(S)$ is the asymptotic scalability of the parallel system.

2.3 Big Data Processing with Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing [34]. The two most important quality of Spark used in big-data and machine learning are:

- It can be used in its own environment in the standalone mode or in a distributed fashion collaborating with multiple machines working in tandem with other big-data processing tools, and
- It is very fast and performative in terms of processing large-scale data. Spark natively supports several languages viz. Java, Scala, Python, R and can be deployed in a variety of ways.

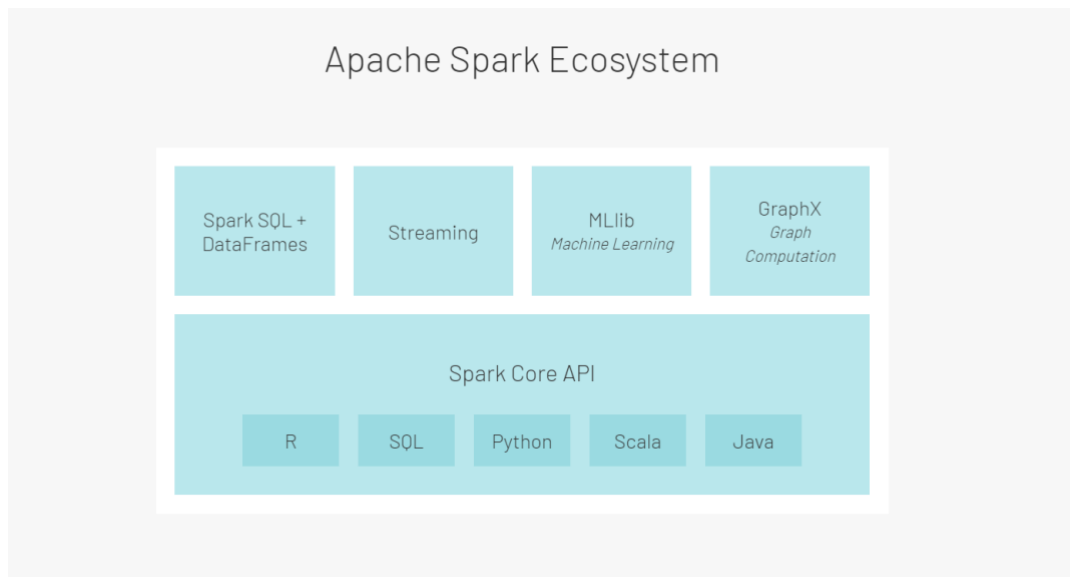


Figure 6. Spark Ecosystem [35]

The high-level architecture Spark uses is it consists of two main components: the driver and workers.

The driver accepts the user application/code and converts into multiple tasks and assigned to each of the workers/executors in a distributed mode. This split of the tasks in between the two requires some resource managers to monitor the jobs and assignments of each worker.

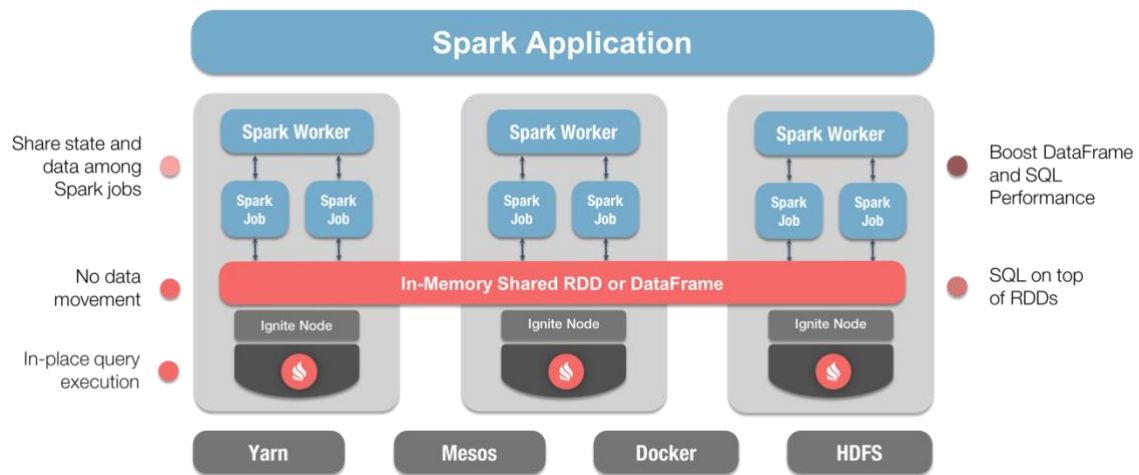


Figure 7. Apache Spark distributed Processing [36]

Spark provides its own Cluster manager out-of-the-box which can be used in the standalone mode. It also has the feasibility to collaborate with other resource managers, for instance, Yarn, Apache Mesos, Kubernetes etc [37].

Built on top of Apache Spark, MLlib is a scalable ML library that offers high quality algorithms which can be tuned with multiple parameters, as necessary. Moreover, it does all the computing in a blazing speed up to 100 times faster than MapReduce [38].

2.4 Related Work

Numerous researches have been done in building efficient ML models for purposes like building more efficient IDS, NGFW, used in SDN, detect specific types of attacks like DDoS and other Volumetric attacks, MITM attacks etc. D.E. Denning in his paper An Intrusion-Detection Model [5] talks about a general-purpose model which could detect any intrusion analysing the audit logs of the devices. Machine learning is the most sought technique used in many applications like fraud detection, malware detection, bot-net detection etc. because it can simulate the behaviour of any object in restricted environments. Fatima and others in their paper [6] proposed a comprehensive survey of different ML and DL techniques and their limitations in IoT networks by analysing the security requirements in IoT architectures and current security threats.

In his paper M. Hasan et al. discussed various machine learning approaches to detect anomalies and attacks in IoT infrastructure [39]. ML models used in [39] includes Support Vector Machines (SVM), Logistic Regression (LR), Decision Tree (DT), Random Forest (RF) along with Artificial Neural Network (ANN). An extensive comparison of the accuracy of the models is done with an experimental approach. The author has also compared the Classification types proposed in relevant papers [40] [41] and provided clear explanation of the various steps involved in the process of Machine Learning like Feature Selection, Data pre-processing etc.

With unsupervised pre-training and compression capabilities, the applications of DL are useful in resource constraint networks [42]. Bahsi et al. in his paper [43] showed that it is possible to get high accurate results from an unsupervised trained model with lesser feature set. The author attempted to compare the performance results of a generic model for multiple IoT devices with that of separate ML models and concluded that considering individual models for each IoT device can provide in better results rather than having a single model for all the devices.

Most datasets generated few years back, lacks modern low foot-print attack environment. Countering these challenges Moustafa et al. presented a comprehensive comparison of the existing dataset and its demerits, created an UNSW-NB15 dataset to evaluate NIDS [44]. Yet another dataset from the Machine Learning Repository provides an extensive attack vector with a motive to violate Confidentiality, Integrity and Availability in mind. For each attack, a set of features in csv file, labels (benign, malicious), and the original network capture is provided [45].

Apart for achieving the accurate models, processing a large dataset is also critical to be able to achieve in a considerable amount of time. Despite the popular research topics around machine learning and parallel data processing, there are relatively lesser studies focusing on parallel data processing and Big Data technologies in the context of IoT security. Hadoop is a big data processing framework which works on the concept of Map-Reduce algorithm. Here, the first stage is mapping the data and distribution into computing nodes and the second stage is reducing the input data which essentially is a parallel processing and aggregating the results computed in these nodes. Branitskiy et al. in his paper [46] discussed about the importance of the time taken for computation and

parallelly processing the input dataset using Apache Spark. Efficiency indicator ACC of a model is measured by,

$ACC = \frac{TP+TN}{P+N}$, where P and N are the amount of positive and negative attack detections and TP and TN are amount of true positive and true negative attack detections respectively. They emphasize on the fact that is it not enough to measure the classification problem in terms of parallel processing and hence proposed an integral indicator ACC/t , where t is the training and testing time of the classifier. However, distributed processing of Apache Spark is leveraged only for binary classification of the data. Amanulla et al. in their paper [47] highlighted 5 different use cases for IoT security where Big Data technologies and Deep Learning could be a potential solution. Based on their survey about the state-of-the-art researches focused on big-data technologies and deep learning, they have developed a thematic taxonomy about the potential challenges and proposed guidelines for future researchers to encourage successful application in this domain.

Now that Apache Spark is quite the buzz around parallel processing and computing large amount of data, numerous researches can be found around the various configurations, modes and settings to be used to gain an optimum cluster resource. In this paper [48], the author provides a comprehensive analysis of the outcome of the various configurations of Spark in a distributed setting. The author used Python APIs to detect multiple anomalies in a network traffic viz. Dos, DDoS, PortScan etc, with about 80 features in the input dataset. To understand the effect performance of the components of Spark, various settings were observed like the effect of memory size on accuracy of the models, the effect of memory on the algorithm's execution time, varying the number of executors the algorithm's execution time is measured etc. However, it lacks the study of dimensional reduction and its impact on accuracy in training the models. It also missed the description of the environmental setup of overall application. An older version of Apache Spark is used, 2.4.0, which cannot be configured to scale its executors dynamically (on-demand) and lacks the support of cloud computations.

3 Methodology

This section illustrates the pragmatic approach taken to achieve the overall outcome of the study. It will walk through the various steps taken and discusses the reason why certain steps were chosen.

The project created categorizing the types of attack uses machine learning approach, which is the most sought method that is used in most classification techniques. Essentially, two types of classifiers are taken into consideration in this study viz:

Binary Classifiers - to predict the anomalous data points. Measuring the accuracy, this sets a benchmark for the further advance classification methods by detecting the malicious data from a particular type of attack. As binary classifiers are capable of detecting as either positive or negative results, it will help predict accurate results when the attack type of the dataset is already known.

Multiclass Classifiers - to identify the actual type of the attack in the dataset containing a mix of multiple attack types. When the dataset contains several types of attacks, binary classifiers are not suitable due to their nature of detecting either the data is malicious or not. As the major focus is to categorize and identify these attacks in the dataset and not just detect if malicious or not, multiclass classification techniques are useful in this scenario.

Once the models are prepared, then these are compared thoroughly using various metrics calculated on the predicted data and finally the models are tuned with hyper parameters for possible scopes of improvement. All these models must go through strict evaluation techniques for proper assessment and getting better optimized results.

The whole process requires a large amount of CPU resources for training the model, particularly when the dataset is huge. In this case, an average size of a dataset of a particular attack type (for instance, *SSL_renegotiation_attacks*) is 6.36GB, and a total of 59.77 GB for all attack types, which means a lot of computation power is required to get accurate results. In order to process such a large number of input data efficiently, in a considerable amount of time, the latest technologies of Cloud Computing are used. One of the examples, would be Apache Spark, the latest version of which can be deployed in Kubernetes, where multiple executors can be spawned up on demand and can go to sleep

when not necessary. Apart from that Spark itself can be set up on distributed mode in a standalone cluster, this is efficient in terms of a single machine with enough computation power. In this mode, Spark can be setup to use multiple executors and compute the tasks in a parallel fashion to increase efficiency and time required to train and analyse the models is also significantly reduced.

The following sections covers the steps required in achieving the milestones and the steps necessary to create a machine learning application that can be run in a distributed environment.

3.1 Application Architecture

A Python based application is developed for the comparison of various configurations and settings to measure the categorization of the attack types. This application is robust and highly configurable and scalable to handle large amount of data. It is de-coupled from the Spark architecture and can be easily deployable to cloud based architecture, for instance EKS cluster.

Few dependencies/libraries used in the application directly and also for investigative purposes are:

Python 3.9 – primary language of the application

Pandas – data analysis and manipulation library,

Numpy – library to perform numerous mathematical operations on arrays of data,
matplotlib – visualization library for plotting various graphs and charts,

Sklearn – machine learning library by python containing multiple algorithms and evaluation functions.

Spark-MLlib (Apache Spark 3.1.1) libraries – Spark’s out-of-the-box library to use machine learning algorithms in a large scale and fast manner.

3.1.1 Running the application

The same application can be run on 2 different modes – *binary*, *multiclass*. It can also be deployed as 2 separate applications behaving differently based on the modes desired. It

accepts this mode-identifier as a runtime argument along with the algorithm which is supposed to train the models and predict the outcome of the test data.

The application can be run with a command as simple as:

1. *python main.py -mbinary -erandom_forest* – given the python interpreter is already configured in the machine and path to the application is already set in the environment variable.
2. Another command: *python main.py --help* to get an overview of the instructions about running the application and various supported arguments.
3. A third optional parameter can be used in the application when the input dataset is the raw network capture. It supports both “.pcap” and “.pcapng” file formats and requires *tshark* to be installed in the machine. The following command can be used to run the application with raw network capture as the input

python main.py -mbinary -erandom_forest -pcap true

3.1.2 Architecture

The high-level flow diagram of the working of the application can be represented as below:

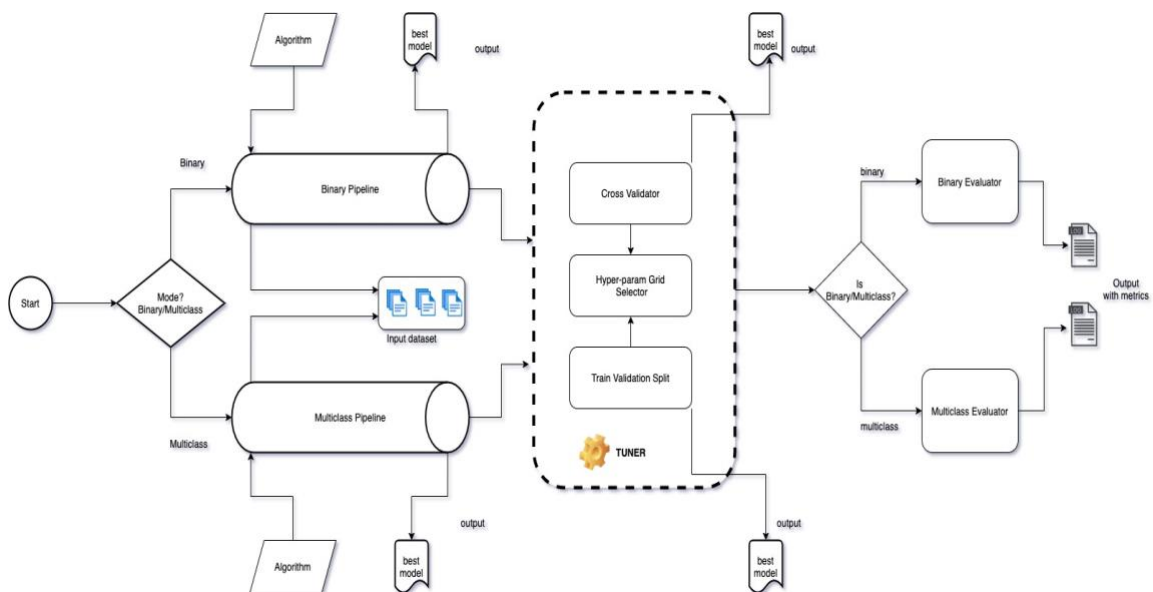


Figure 8. Flow Diagram of the Application - Appendix 6

The application can be started by running the command mentioned in the above section. It decides the mode based on the argument passed in the command and the selected estimator is highlighted. As soon as the application is run, the measurement process starts, it logs the start time and end time when it ends to measure the exact time taken for the whole process to finish.

The input data is mounted in a separated volume, given the size of the dataset, it is decoupled from the application which make it highly flexible to switch to any dataset with minimum code changes.

From the architecture design above, we can see the separate pipelines created depending on the mode of the application. However, the outcome of each pipeline is fed to a common *Tuner*. This is designed in such a fashion to provide high configurability and flexibility to select the components of the application for various comparison purposes. In other words, the Tuner module can be easily removed from the whole system when desired. This significantly helps in selecting the suitable modules and parameters when the outcomes are measured.

Now, the pipeline itself contains various steps and modes which are again highly configurable. The below diagram illustrates the high-level data flow for an individual pipeline which is similar in both the modes.

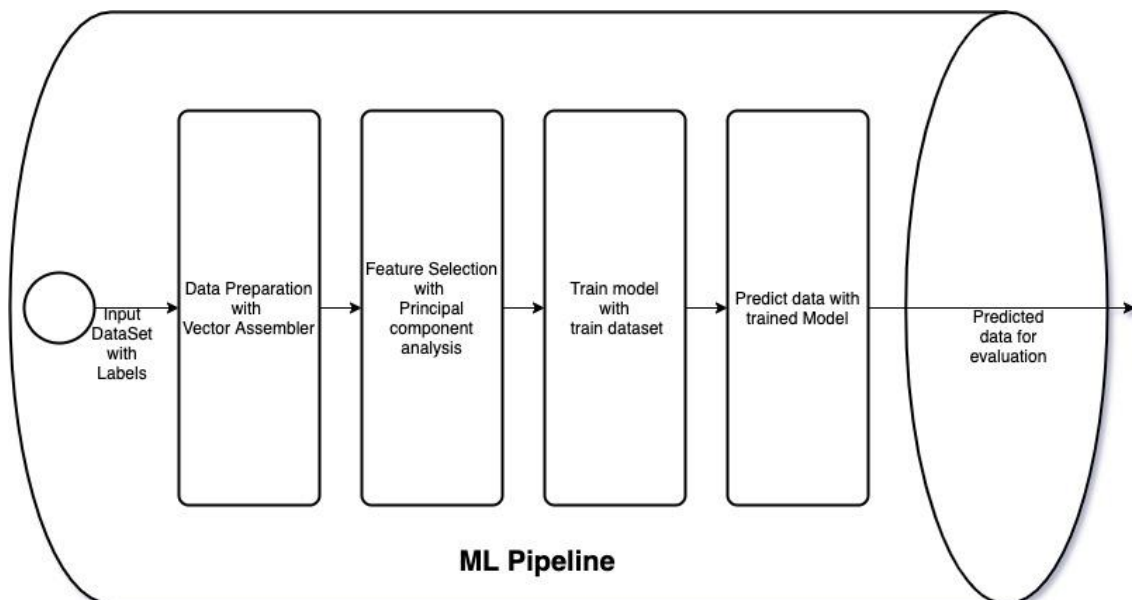


Figure 9. Flow Diagram of a Pipeline

From the figure above we can see the different stages in a pipeline considered in this study. Once the dataset is ready, a spark compatible dataframe is created and fed to the Vector Assembler which assembles all the features into a single column for further analysis. The next stage is to reduce the dimension of the dataframe and select only those features that are less correlated with each other. In this stage, the principal components are calculated from the features that will be used to train the models. The components are configurable i.e., metrics are calculated using different values of the component and compared to analyse the effect on the result.

The model once trained is generated as the output of the pipeline for re-usability and reduce the compute time for further steps. For training purposes, the dataset is split into train and test dataset, and a major part of it is used to train the models to ensure all possible cases are covered.

This model is then used to predict the remaining dataset and evaluated to create benchmarks in binary mode and categorize different attacks in multiclass mode. These models are then fed to the tuner module for achieving the best possible parameters and the best models are saved as an output for further re-use.

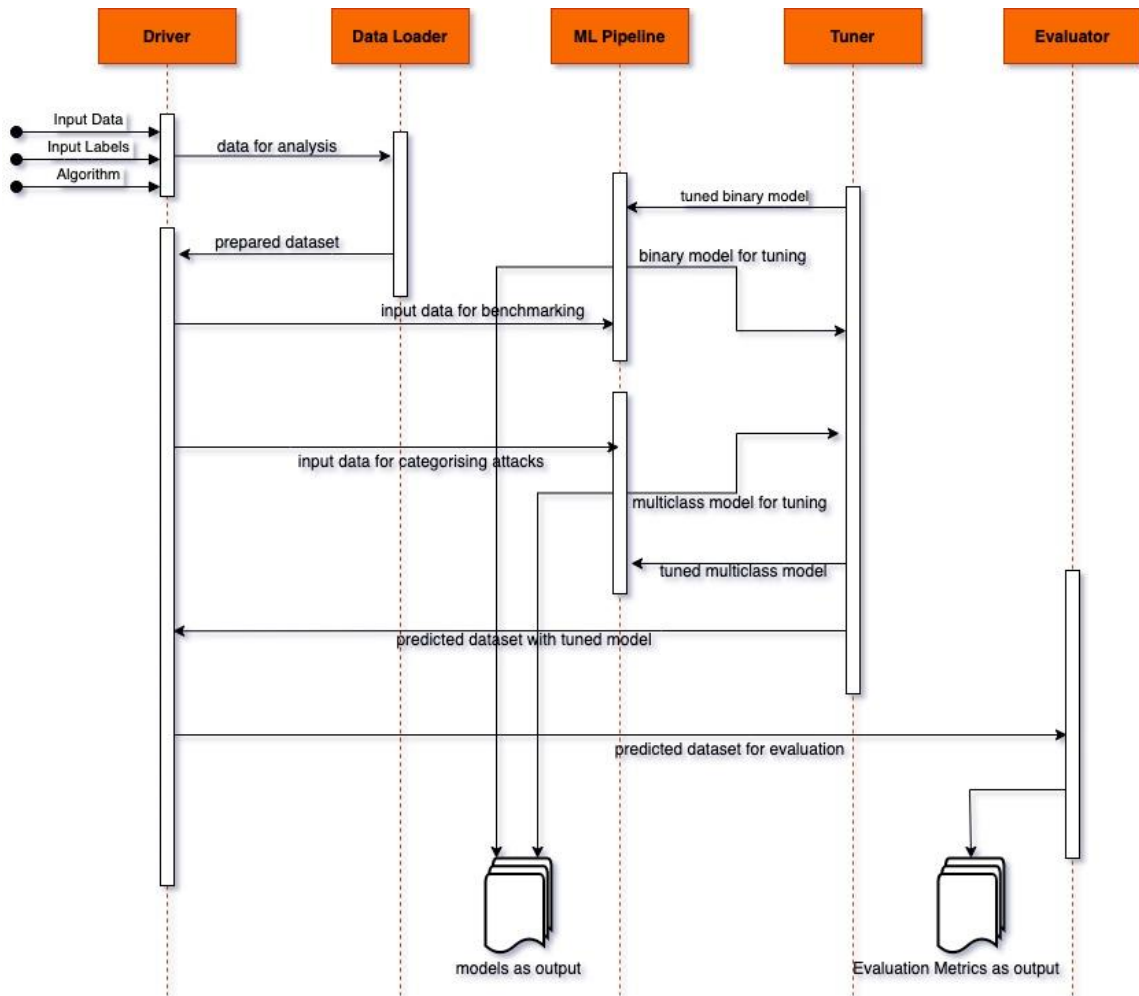


Figure 10. Sequence diagram of the data flow.

The above diagram depicts the process sequence of the whole application that helps us understand the working of the different processes in each module. It illustrates the type of interaction between the modules and in what order the execution is taking place.

3.2 Choice of Dataset

Multiple datasets were considered before initiating the process. Several facts for each dataset in terms of the achievement of the goal of this project, were assessed as described below:

Medbiot [49] dataset was as a viable candidate initially as it contains appropriate labels for normal and malign data. It could be best fitted to the binary classification type of machine learning models, and it contains 3 different types of anomalies, namely bashlite,

mirai, torii malwares which disrupts the IoT networks causing misbehaviour in their functionalities.

TON_IoT [50] dataset or Telemetry Operating systems' data and Network traffic dataset contains a huge variety of IoT devices exposed to a variety of attacks and are classified with proper labels. It also covers Linux and Windows operating systems data when exposed to several attacks in their testbed.

However, these datasets mentioned above fulfils the goals of this project up to some extent but leaves potential gaps in certain factors. *Medbiot* focused the most on malwares attacking multiple IoT devices and its behaviours. *TON_IoT* has a huge variety of dataset and attack types, but the size of each attack types is quite low. The big data processing and tools will not showcase its capabilities in terms of parallel computing and handling large dataset. One of the datasets best suited for this purpose is the Kitsune Network Attack dataset [45]. Kitsune, in Japanese folkore, is a fox-like character with multiple tails, which can shift into many forms, and it grows stronger with experience. Resembling this, Kitsune, is a novel ANN based online, unsupervised, and efficient NIDS.

Kitsune Network Attack dataset is a cybersecurity dataset on a commercial IP-based surveillance system and an IoT network. The dataset contains 4 different types of network attacks which very well, with some modifications, serves our purpose.

Below are the characteristics of the dataset in brief:

Name	Value
Dataset Characteristics	Multivariate, Sequential, Time-Series
Associated Task	Classification, Clustering
Total no. of instances	27170754
Missing values	N/A
No. of attributes	115

Table 1. Dataset description.

The attack types covered in this dataset are:

a. Reconnaissance

OS Scan – Tools like Nmap can scan through the network to find vulnerable hosts.

Fuzzing – SFuzz can scan various camera webservers by sending random commands and look for vulnerabilities.

b. Man In the Middle

Video Injection – Video Jack injects a malicious recorded video into live video streams which are popular in free streaming websites.

ARP Mitm – Ettercap can perform an ARP poisoning attack by intercepting all LAN traffic.

Active Wiretap – Wiretapping is performed in exposed cables.

c. Denial of Service

SSDP flood – Cameras are used as zombie servers overloading the DVR servers by spamming with UPnP advertisements.

SYN DOS – A camera's video stream is disabled by overloading its server.

SSL Reneg – A camera's video stream is disabled by overloading with multiple SSL renegotiation packets.

d. BotNet Malware

Mirai – Telnet is used to infect IoT devices with Mirai malwares by exploiting through various vulnerabilities and scans through the network for more vulnerable victims.

A total of 9 attack dataset are present in the combination of above types. Each of dataset is organized in the following manner:

Each attack type consists of 2 separate files:

<prefix>_dataset.pcap – contains the original raw network capture of N packets truncated to 200 bytes for privacy reasons.

<prefix>_dataset.csv – contains a N * N matrix of feature vectors for a total of 115 features.

<prefix>_labels.csv – contains a N * 1 matrix of labels identifying the traffic whether benign or malign.

- where prefix is the type of attacks mentioned above.

3.3 Dataset Analysis

Whenever a packet arrives, the behavioural snapshots of the hosts are captured chronologically. The captured traffic typically ...

- originates from the packet source's MAC and IP address
- originates from the source's IP.
- traffic through the denoted channel between a packet's source and destination IPs
- traffic through the channel between the source and destination's TCP/UDP socket.

The following network packets are captured to extract the set of features:

"frame.time_epoch", "frame.len", "eth.src", "eth.dst", "ip.src", "ip.dst", "tcp.srcport", "tcp.dstport", "udp.srcport", "udp.dstport", "icmp.type", "icmp.code", "arp.opcode", "arp.src.hw_mac", "arp.src.proto_ipv4", "arp.dst.hw_mac", "arp.dst.proto_ipv4", "ipv6.src", "ipv6.dst"

From the above captures, feature vectors, which essentially are the recent temporal statistics describing the context of the packet's channel and the communication addresses, are extracted from the traffic in a single window. The feature extractor extracts the same set of features from a total of 5 times damped windows of approximately 100ms, 500ms,

1.5s, 10s, 1min thus totalling 115 features. In case there are no features for a particular traffic, the feature vectors are zeroed hence, the features are always 115.

The path of the dataset is provided as the input to any of the modes of the application and the dataset is loaded at runtime and fed to the DataLoader module for further analysis. The feature distribution is analysed, and investigation is performed for possible skewed data which would otherwise produce biased results.

Once the results of the distribution analysis are satisfactory, the relation among the features are calculated. A single column is taken as a reference and its relationship with all other columns is measured with an identifier called correlation coefficient. This step is necessary in identifying the principal components of the dataset which helps in further dimensional reduction of the large dataset keeping accuracy unaffected.

3.4 Training and Creating models

The next stage in the pipeline process after dataset preparation is the training the models. Training models, in a machine learning process, essentially means obtaining good values out of the weights in the dataset and the bias from labelled data. In a supervised mode, the training is performed by analysing many records of data and its categories identified by the labels and to come up with an effective model to minimise loss.

In this study, for the binary mode of the application, which is used to create the benchmarks, the input dataset is randomly split in to 6:4 for training and testing purpose i.e., 60% of the input data is randomly selected for training the models and the remaining 40% of it is used to predict the attack types.

The algorithms taken into consideration for training the models are mostly decision trees and some ensemble form of those trees. The algorithms selected for benchmarking the performance metrics in binary classification mode are *decision trees*, *random forest*, *gradient boosted trees*, and *naïve bayes*.

The Multiclass mode of the application also randomly splits the input data into 70:30 for training and testing the models respectively. Since the input data for multiclass mode contains a collection of multiple attacks, the training data selected ensures all possible categories of the dataset are included for learning.

The algorithms selected for multiclass mode are decision trees, random forest and multi-layer perceptron (MLP) for analysis and comparison against those benchmarks. However, for thorough comparison random forest and decision trees are only selected as MLP embarks the initial stage of Neural Networks with Deep learning which demands high computing resources and significant amount of time and is beyond the scope of this study.

3.5 Tuning with Hyper params

In any machine learning process, selection of hyper parameters is a very important step in selecting the best model. Parameters which define the model architecture are known as the hyperparameters and this process of tuning the models with multiple parameters is called hyperparameter tuning.

While selecting the model, we cannot rely on the default parameters to have the best results rather a range of parameters are explored and evaluated to find out the best possible model. Hyper parameters are known to control the learning process of the model and typically the machine itself performs this evaluation with the parameters as the input.

Hyperparameters tuning help us understand few aspects of model architecture, for example, the degree of polynomial features used for binary classification, maximum depth to be used in a decision tree, number of trees in a random forest, number of networks required in a neural network etc.

The application developed in this study uses a tuner module for hyperparameters tuning. This module is configurable and can be used in both the binary and multiclass mode. A list of param grids is defined that are suitable for all the algorithms supported by the application.

Apache Spark MLlib supports tools like *Cross Validator* and *Train Validation Split* for model selection and accepts the following required parameters:

Estimators: an algorithm or the whole Pipeline of the ML process.

Evaluators: an instance of the Evaluator for the above estimator to train, validate and evaluate the models.

On a very high level, for each of the modes, the tuning works as follows:

- Once the dataset preparation is finished, the application creates the Binary Pipeline with the stages including the vector assembler, principal component analyser, and the corresponding algorithm for the model.
- This pipeline along with the algorithm, the input dataset and the param grid are then provided as input of the Tuner module for tuning with both Cross Validator and Train Validation split tools.
- Before feeding to the tuner module, the input dataset is again randomly split as 9:1 to test and validation dataset. 90% of dataset is then tuned with both Cross Validator and Train Validation Split to determine the best model. The under-lying working strategies of these tools are discussed in the below sections. Essentially, these tools iterate through the param map and further splits the train dataset into a third category called validation dataset based on some parameters. For each of the combinations of the params, several models are created and evaluated with the validation dataset to output the best model. This best model is then selected and saved that can be re-used for identifying similar attacks.
- This best model is used to transform the remaining 10% of the dataset split from the original input data to predict the attacks and calculate the performance metrics for further comparative analysis.

The tuning process could consume a lot of time based on the hyper-params provided as the input which can be improved by performing the whole process in parallel. The parallelism can be controlled by *parallelism* parameter (1 means sequential and more than 1 is the level of parallelism required). However, it should be carefully used not to exceed the cluster resources.

3.6 Evaluation of Models

The next step of the machine learning process is to make sure that the models being created can be used to predict future dataset and should be able to categorize attacks in a new dataset without much computing time. In both the modes, the base model without the

tuner is evaluated with metrics and then the tuned model is also evaluated with the same metrics for extensive comparison.

There are two specific evaluators for binary and multiclass mode used in this study: *BinaryClassificationEvaluator* and *MulticlassClassificationEvaluator*. These are supported by Spark and takes the dataset and the label column name as input and outputs the metric calculated like accuracy, precision, areaUnderROC etc. Generally, in a classifier evaluation, considering only pure accuracy which measures if a record is malicious or not, is not a reliable metric because there might a case where 95% of the data are normal. If the model predicts all records as good data in which the accuracy would be 95% regardless of the input. For this reason, additional metrics like precision, F-score, recall etc are calculated to have an un-biased comparison.

For binary mode, additional python based sklearn library is used for measuring metrics like confusion matrix, true negative, true positives etc. along with those mentioned above to ensure a wide perspective of evaluation criteria are selected for the benchmarking.

3.7 Application Output

As important are the steps discussed above for any machine learning application, starting from the data preparation step to creating an efficient model to evaluating those models, the outcome of the whole process is also a key step towards the process. To be able to re-use the trained models in the future with minimum changes and resources is considered as one of the goals of this study.

To be able to achieve that, application developed is kept as modular as possible keeping in mind comparison of the results are done in multiple settings by tweaking the configurations with several parameters requiring minimal changes.

The evaluated models are saved as the output of the application for reusability. Each of the modes saves the best model for each of the combinations, for instance,

- In both the modes, when decision tree algorithm is used, the model generated without the tuner module as the base model.

- When the tuner is used, the best model tuned with the CrossValidator, and
- The best model tuned with TrainValidationSplit tool.

Similar is the case when other algorithms are used, for instance random forest and MLP in case of multiclass mode.

Apart from this, the time taken for the application to complete the whole learning process for each configuration is calculated and logged in output files. For each setting of the application separate run commands are used with several combinations of arguments, separate files are generated, the evaluation metrics, sample predicted dataset etc. are logged as an output of the independent steps of the application.

4 Categorizing Attacks

This section aims to achieve few of the major goals this thesis is focused on, for example, milestones like dataset preparation and analysis, investigating the performance of the models while having reduced set of dimensions, setting benchmarks with binary classifier, categorizing and identifying the types of the attacks in a dataset containing data with multiple attack types are achieved in a pragmatic manner. Overview of the cluster setup, tuned to perform the computations in a parallel fashion are also covered in this section.

4.1 Environmental Setup

A virtual machine has been used for the purpose of installing Apache Spark and exploring its various configurations to run the application in a distributed mode. The most recent version of the tools is considered while developing this application making it capable of handling large amount of data. Few details of the VM used and the resources allocated:

Name	Library	Version
Programming language	Python	3.6
Runtime environment	Pyspark	3.0.2
IDE	PyCharm	2021.1.3(Professional Edition)
RAM	62GB	
DISK Volume	25 + 100GB (additional)	
No. Of Cores	16	
Computation Framework	Apache Spark	3.1.1
No. Of worker	16	
Cores assigned per worker	1	
Memory per executor	3	

Table 2. Environmental Details

Apache Spark is downloaded from the official website: `spark-3.1.1-bin-hadoop2.7.tgz` is the package used for installation. It is extracted and installed in the additional volume attached to the main instance and mounted in a file path `ws/`. It is then configured for 5 dedicated executors in the `spark-defaults.conf` file located in `$SPARK_HOME/conf` directory. Spark env variables are set in the `~/.bashrc` file to use the appropriate python libraries for the executors and in the `$SPARK_HOME/conf/spark-env.sh`.

Python is installed by default when the instance is created in Openstack. Using cloud environment comes with flexibility in terms of setting up security groups and access policies. The instance has its dedicated security group, and an additional volume is attached to it for more disk usage.

A glimpse of how the application is ran in a virtual machine with multiple executors can be referred from the figure below:

URL: spark://rakshit-thesisv2.cloud.ut.ee:7077

Alive Workers: 16

Cores in use: 16 Total, 0 Used

Memory in use: 48.0 GiB Total, 0.0 B Used

Resources in use:

Applications: 0 Running, 63 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers (16)

Worker Id	Address	State	Cores	Memory	Resources
worker-20210722174910-172.17.66.135-37955	172.17.66.135:37955	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084304-172.17.66.135-42317	172.17.66.135:42317	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084305-172.17.66.135-46745	172.17.66.135:46745	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084308-172.17.66.135-45399	172.17.66.135:45399	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084311-172.17.66.135-34293	172.17.66.135:34293	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084313-172.17.66.135-42303	172.17.66.135:42303	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084316-172.17.66.135-34195	172.17.66.135:34195	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084318-172.17.66.135-46513	172.17.66.135:46513	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084321-172.17.66.135-40851	172.17.66.135:40851	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084324-172.17.66.135-43643	172.17.66.135:43643	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084326-172.17.66.135-33211	172.17.66.135:33211	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084329-172.17.66.135-44677	172.17.66.135:44677	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084331-172.17.66.135-38679	172.17.66.135:38679	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084334-172.17.66.135-32883	172.17.66.135:32883	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084336-172.17.66.135-37083	172.17.66.135:37083	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	
worker-20210724084339-172.17.66.135-37989	172.17.66.135:37989	ALIVE	1 (0 Used)	3.0 GiB (0.0 B Used)	

Figure 11. Apache Cluster Overview

From the figure above, we can observe the no. of executors being used to train the models and predict the classification.

One of the major advantages of using Apache Spark is that it is built for parallel processing and is installed in a distributed environment. This allows it to handle a large dataset in a parallel fashion which makes the computational process much faster given appropriate resources are used. In this experiment, a 16 core CPU is used with 64 GB of RAM which is distributed among 16 executors running parallel each with 3GB of memory with dedicated 1 core each to ensure smoother computation and low physical disk usage.

A *spark-submit.sh* script is present in the root directory which is used to submit this application to the spark cluster. The *main.py* python file contains the main entry point of the application.

The following are the various scenarios executed in spark to try out various options before coming out to the optimal configuration:

```

spark-submit.sh -- Edited
### WITH PCA
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator gbt --mode binary > logs/pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator decision_tree --mode binary > logs/pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator decision_tree --mode multiclass > logs/pca/multiclass/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator random_forest --mode binary > logs/pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator random_forest --mode multiclass > logs/pca/multiclass/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 --executor-memory 4G outlier-detection/main.py --estimator perceptron --mode multiclass > logs/pca/multiclass/output-$(date +%b-%d-%Y-%H:%M:%S').txt
## WITH PCA, K=18

```

Figure 12. Sample run script configurations

In the figure above, the file contains the several modes with different combinations of the runtime arguments to analyse the generated model in each step. The spark executors and memory allocated per executors are also tweaked to investigate the effect of the environment in the outcome models.

4.2 Data Preparation Steps

The raw dataset gathered from the UCI repository is in the csv and pcap format. These files could not be directly fed into the machine learning process. The application picks up the relevant file with a specific attack type and creates a dataframe with appropriate id and labels and does some analysis before feeding to the machine learning pipeline. The following series of sections describes what steps are performed with the data in both binary and multiclass mode.

4.2.1 Data Pre-processing

The dataset was prepared by parsing the network traffic and then converting the comma separated data into feature vectors, we analyse each type of attack dataset in the following manner.

In binary mode, a specific attack type of data, at a particular time is considered to create a base model. For example, for analysing *SSL_Renegotiation* attack there is a dataset file containing about records and a labels file with same no. of records. For the labels file to be kept separate, was intentional, to make unsupervised learning hassle free. But for supervised learning, the labels are required to be attached with the original data in a separate column which is the reason the dataset are joined together to create a single input dataset. This join process in a distributed environment is handled with intense care to

ensure that proper labels are attached with the correct rows of the original input data, after joining. This process is further described in the below section.

4.2.1.1 Creating Dataframe and generating unique Ids

The first step is to create a Spark compatible dataframe from the csv files. In binary mode, two different dataframes are created – one with the main data and a labels dataframe. Since the binary mode is run only with a single type of attack dataset, the data loading part is less complicated and costly than in multiclass mode.

If we consider the SSL_Renegotiation attack type, the following code snippet shows how to create a the dataframe in spark:

```
# Load Data
data = "../input/network-attack-dataset-kitsune/SSL Renegotiation/SSL_Renegotiation_dataset.csv"
df_data = spark.read.load(
    data,
    format="csv", sep=";",
    inferSchema="true",
    header="false")

# Load Labels
labels = "../input/network-attack-dataset-kitsune/SSL Renegotiation/SSL_Renegotiation_labels.csv"
labels_schema = StructType([StructField("id", LongType(), False),
                             StructField("label", StringType(), False)])
df_labels = spark.read.load(
    labels,
    format="csv",
    sep=";",
    inferSchema="false",
    schema=labels_schema,
    header="true")
```

Once the dataframes are created, the next step is to generate unique ids which will be used for joining. For this purpose, *pyspark sql.function's monotonous_increasing_id* library is used with a little modification. As spark does all the computations in a distributed fashion, when this library is used, it creates internal partitions to store the metadata information of the input dataset. The internal partitions with the partition id and their respective sizes can be viewed as follows:

partition_id	partition_size
26	46584
29	46593
19	46576
0	46604
22	46589
7	46574
34	46577
32	46589
43	46623
31	46575
39	46600
25	46574
6	46575
9	46571
27	46557
17	46544
41	46603
28	46575
33	46565
5	46572

only showing top 20 rows

Figure 13. Spark's internal partitions

From the above output, we can see the sizes vary in each partition and it is not guaranteed that the data is ordered throughout the partitions. Hence a join operation among the dataset and the labels at this stage will not provide accurate results.

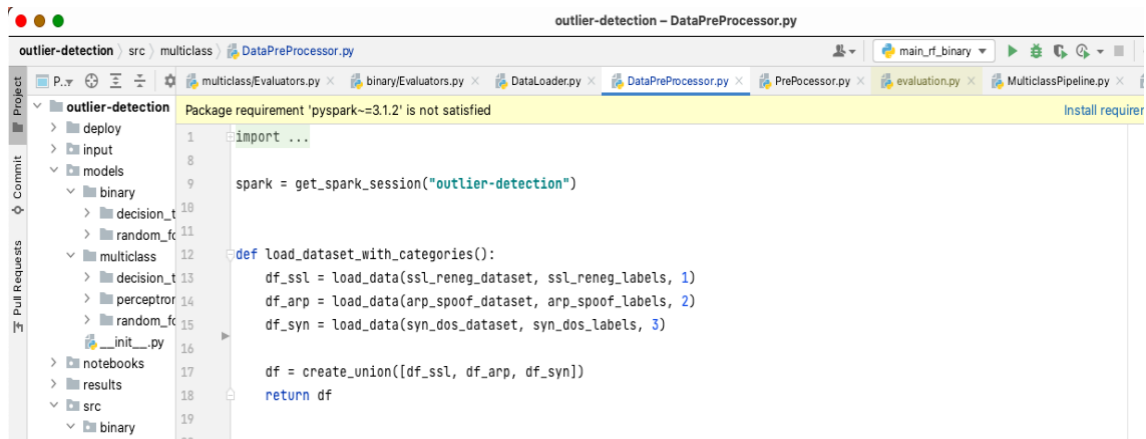
For this reason, an extra modification is required while generating these ids. In the next step, the offsets are calculated for each partition and summed up and finally joined with the original dataset as shown below:

```
# Final step joining with the original dataset
df_data = df_partition
    .join(broadcast(partitions_offset), "partition_id")
    .withColumn("id", partitions_offset.partition_offset+df_partition.row_offset+1)
    .drop("partition_id", "row_id", "row_offset", "partition_size", "partition_offset")

# Verify the id creation and redundant rows cleanup with schema
df_data.printSchema()
```

The outcome of the above code snippet is the input dataframe, for the binary mode, supported by Spark created from the csv files. This dataframe is now ready to be joined with the labels dataframe.

In multiclass mode, the above steps are repeated for each type of attack which are provided as an input for categorization and can be achieved in python as shown below:



```

outlier-detection - DataPreProcessor.py
outlier-detection | src | multiclass | DataPreProcessor.py
Package requirement 'pyspark==3.1.2' is not satisfied
1 import ...
8
9 spark = get_spark_session("outlier-detection")
10
11
12 def load_dataset_with_categories():
13     df_ssl = load_data(ssl_reneg_dataset, ssl_reneg_labels, 1)
14     df_arp = load_data(arp_spoof_dataset, arp_spoof_labels, 2)
15     df_syn = load_data(syn_dos_dataset, syn_dos_labels, 3)
16
17     df = create_union([df_ssl, df_arp, df_syn])
18     return df
19
20

```

Figure 14. Load multiple datasets for categorization in Pyspark



```

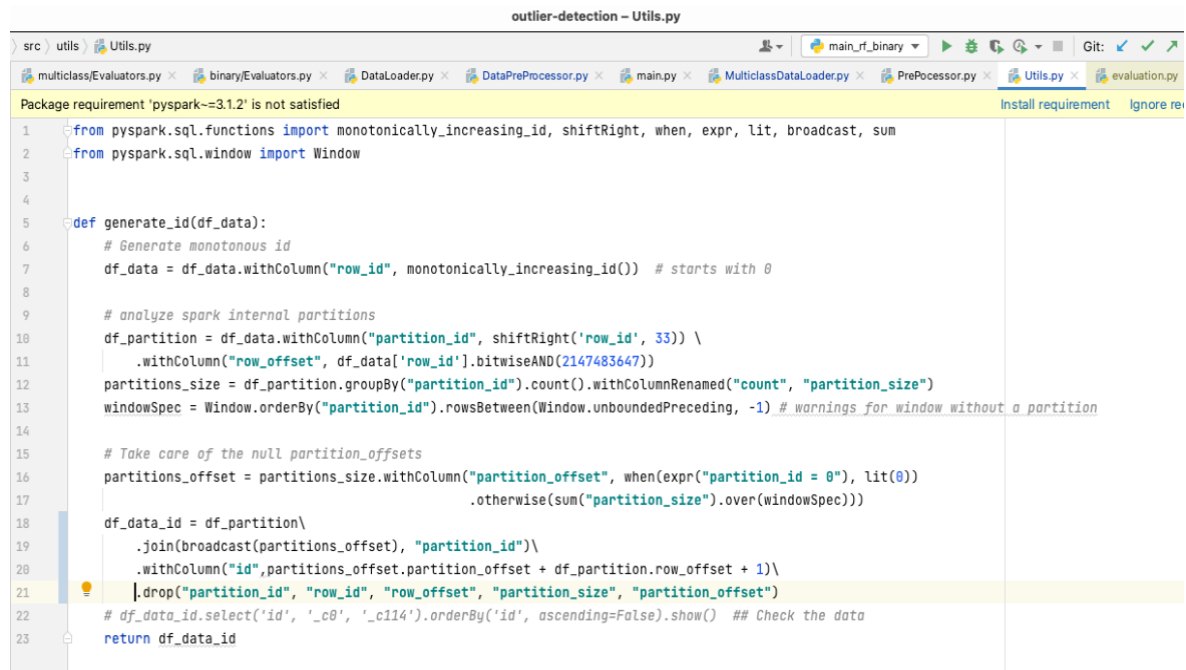
10 def load_data(data_path, labels_path, multiclass_param):
11     df_data = spark.read.load(
12         data_path,
13         format="csv", sep=",", inferSchema="true", header="false")
14     print("Loaded dataset. ")
15
16     labels_schema = StructType([StructField("id", LongType(), False),
17                                 StructField("label", IntegerType(), False)])
18     df_labels = spark.read.load(
19         labels_path,
20         format="csv",
21         sep=",",
22         inferSchema="false",
23         schema=labels_schema,
24         mode="DROPMALFORMED",
25         header="true")
26     df_labels = df_labels.withColumn('label', when(df_labels.label == 1, lit(multiclass_param)).otherwise(lit(0)))
27
28     print("Loaded labels. ")
29     df_labels.select("id", "label").orderBy('id', ascending=False).show()
30
31     # Pass through data pre-processor
32     df = pre_process_data(df_data, df_labels)
33     return df
34
35

```

Figure 15. Function load_data creating the dataframes

The function *load_data* takes in the input data path as string, the labels data path and the multi-class labels as an argument and returns a dataset joined with proper labels attached to the original rows.

For the scope of this study, three types, namely, `ssl_renegotiation`, `arp_spoofing`, and `syn_dos` type of attacks are taken into consideration in the multiclass mode, as shown in Figure 13. Once the dataframes are create for each of those attack types, unique ids for each of the dataframes are created as follows:



```

outlier-detection - Utils.py
src | utils | Utils.py
Package requirement 'pyspark==3.1.2' is not satisfied
1 from pyspark.sql.functions import monotonically_increasing_id, shiftRight, when, expr, lit, broadcast, sum
2 from pyspark.sql.window import Window
3
4
5 def generate_id(df_data):
6     # Generate monotonous id
7     df_data = df_data.withColumn("row_id", monotonically_increasing_id()) # starts with 0
8
9     # analyze spark internal partitions
10    df_partition = df_data.withColumn("partition_id", shiftRight('row_id', 33)) \
11        .withColumn("row_offset", df_data['row_id'].bitwiseAND(2147483647))
12    partitions_size = df_partition.groupBy("partition_id").count().withColumnRenamed("count", "partition_size")
13    windowSpec = Window.orderBy("partition_id").rowsBetween(Window.unboundedPreceding, -1) # warnings for window without a partition
14
15    # Take care of the null partition_offsets
16    partitions_offset = partitions_size.withColumn("partition_offset", when(expr("partition_id = 0"), lit(0))
17        .otherwise(sum("partition_size").over(windowSpec)))
18
19    df_data_id = df_partition\
20        .join(broadcast(partitions_offset), "partition_id")\
21        .withColumn("id", partitions_offset.partition_offset + df_partition.row_offset + 1)\
22        | drop("partition_id", "row_id", "row_offset", "partition_size", "partition_offset")
23    # df_data_id.select('id', '_c0', '_c114').orderBy('id', ascending=False).show() ## Check the data
24    return df_data_id

```

Figure 16. Function to generate id

This process is carried for all the attack types and finally, each of the attack types are identified with labels and are ready to be joined with their respective data.

4.2.1.2 Join with Labels

Finally, each row in the dataframes created for both binary and multiclass mode, are uniquely identified with the generated id field and is ready to join with its corresponding label's dataset in the following manner:

```

#JOIN
df = df_data.join(df_labels, on=["id"], how="inner")
df.count()

```

The join strategy considered is *inner join*, on the id column, which means, it will consider on the common rows from both the data and the label's dataset and ignore the unmatched ids.

Similar steps are performed for all the attack types in the multiclass mode. The following table shows an overview of the label's distribution of the dataset with 3 types of attacks labelled respectively:

```

+-----+-----+
|label|  count|
+-----+-----+
|   0|6238152| |
|   2|1145272|
|   1| 92652||
|   3| 7038|
+-----+-----+

```

Figure 17. Labels distribution for the input data.

- where 0 is normal traffic,
- 1 is *SSL_renegotiation* attacks,
- 2 is *arp_spoof*,
- 3 is *syn_dos* types of attacks.

The above-described process can be done for all other attack types which are not considered in the study and in such case, the table would contain 10 classes. Now, these datasets are ready for further analysis.

4.2.2 Data Analysis

This section focuses on the pre-training steps performed to get a better understanding of the data being used for training the models. Now that the dataset is loaded the spark, it can automatically infer the schema of the dataset. The dataset with the feature vectors are loaded as Spark dataframe and labels are automatically inferred by Spark as: *_c0*, *_c1*, *_c2*, *_c3* ... *_c113*, *_c114*.

It may be quite tedious to analyse all 115 columns individually, hence let us consider a single column for the analysis and try to understand the behaviour of it. For instance, the column *_c3* of the *SSL_reneg* dataset is taken for analysis and if we describe the dataframe in spark, the output can be tabularized as follows:

```
df_ssl.select("_c3").summary().show()
```

summary	_c3
count	2207571
mean	181.08887765214948
stddev	87.20518378016997
min	1.0
25%	132.0193251824308
50%	168.5805563917662
75%	210.9960072134343
max	476.62876150643086

+ Code + Markdown

Figure 18. Description of _c03 col.

From the above figure we get a summary of the specific statistics of the column “_c03” by using the spark’s *summary()* function to get an overview of the exploratory data analysis. To have a more concrete view of the distribution of the data for that column, the following graph can be generated.

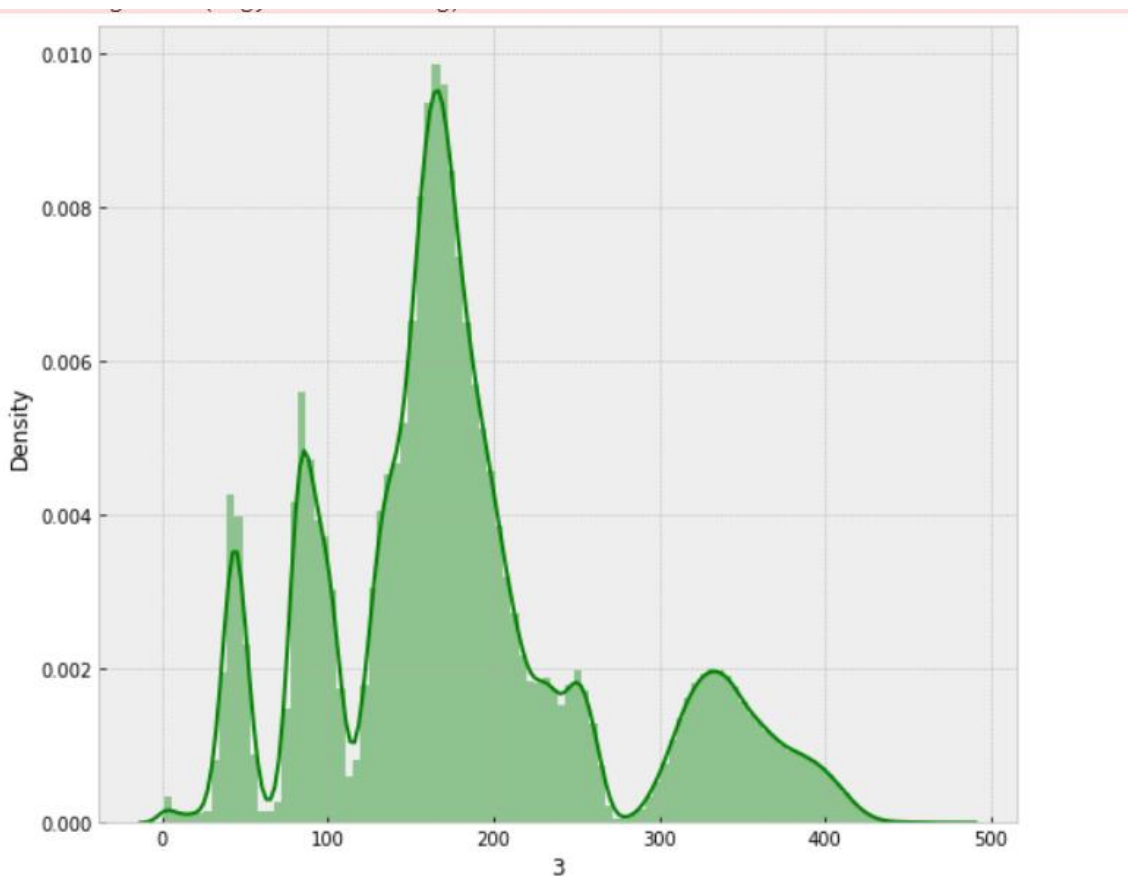


Figure 19. Feature distribution of col03

In a similar fashion, the feature distribution analysis can be performed on all the features in the each of the dataset. For reference, the feature distribution of all SSL_Reneg dataset is attached in the appendix.

The input dataset contains 115 features which was generated from 5 window frame. In a single window almost 19 attributes are captured from the network traffic to generate the feature vectors containing the context of the packet's channel and its relevant communicating parties, which might infer the presence of some degree of correlation among the vectors.

In the next step, we investigate the correlation among the different variables in the features dataset. In a similar method as above, we consider the col03 as the reference col and calculate all the columns which are highly correlated with it as depicted in the following figure:

```
There is 21 strongly correlated values with col 3:
3      1.000000
18     1.000000
37     0.999270
68     0.999270
0      0.977193
15     0.977193
30     0.976664
65     0.976664
6      0.967565
21     0.967565
71     0.966821
44     0.966821
9      0.909070
24     0.909070
74     0.908317
51     0.908317
87     0.730647
12     0.707987
27     0.707987
77     0.707355
58     0.707355
Name: 3, dtype: float64
```

Figure 20. Feature correlation of co03.

In the above correlation matrix, we observe 21 strongly correlated columns with col03, this indicates that there are features which doesn't provide any additional significance to the machine learning model. Hence, we try to find the range of features that are less

relevant to each other and provides some unique value the ML model to train by measuring the correlation coefficient of the vectors.

We use Spark's *corr()* function to compute the correlation matrix of a vector column. This function accepts the dataframe and column name as arguments and the method it uses is *Pearson Correlation Coefficient*, by default.

Pearson Correlation Coefficient is the measure of linear correlation between two variables for measuring the relationship, or association among them, developed by Karl Pearson [51].

For the above scenario, it is measured as the ratio of the variance of `_col03` and any other col and the product of their standard deviation which is essentially the normalised measurement of the covariance of the other columns with respect to `_col03`.

Correlation coefficient can have three types of values as: [52]

- Positive Correlation – In this case, both the variables behave similarly i.e., if one of variable increases, the other will increase too.
- Neutral Correlation – This type indicates no relation among the variables at all i.e., the behaviour of the other variable is not dependent on the first one.
- Negative Correlation – Negative value of the correlation coefficient means both the variables behave in a opposite manner but still related to each other.

In this data analysis step, we are generally interested in the neutral correlated variables only as the tightly coupled variables can negatively impact the performance of the algorithm.

To understand the relation of all the columns in the input dataset and get an overview of all the related features to that of the `col03`, in our case, we generate a heatmap as shown below:

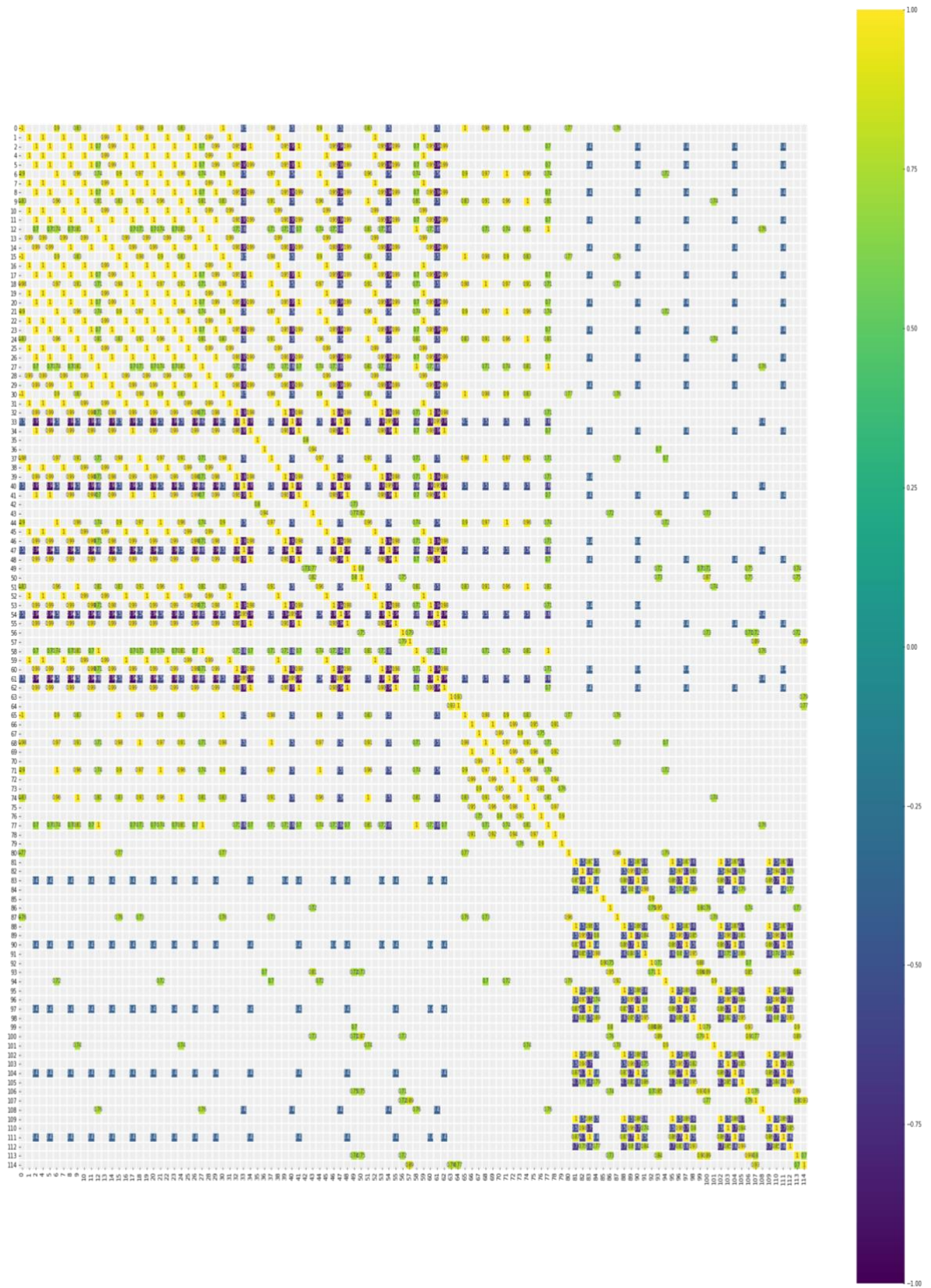


Figure 21. Heat map of the correlated features to col03

The figure above is generated with a range of correlation coefficient as $-1 < 0 < +1$. A value towards $+1$ suggests the variables are positively correlated to each other and similarly, a value towards -1 implies the variables are negatively correlated. The higher the absolute value, the stronger is the correlation among the vectors. Hence, we attempt to find a range of values that are of less relevant to each other which is why threshold values of $+0.7$ and -0.4 is considered as a reasonable assumption for calculating the less correlated features. Now we have the heat map, colour coded with highly coupled features, but we know there are outliers in the dataset which impacts the correlations, but it's still doesn't provide enough information about the actual correlation of the features whether they are linearly varied or exponential, hence, this is incomplete. This correlation map provides a more high-level understanding and may not be accurate in interpreting the relationships among the features which leads us to the next step of plotting dedicated graphs considering two individual columns at a time.

A sample of the pair-plot of the features are plotted as shown below:

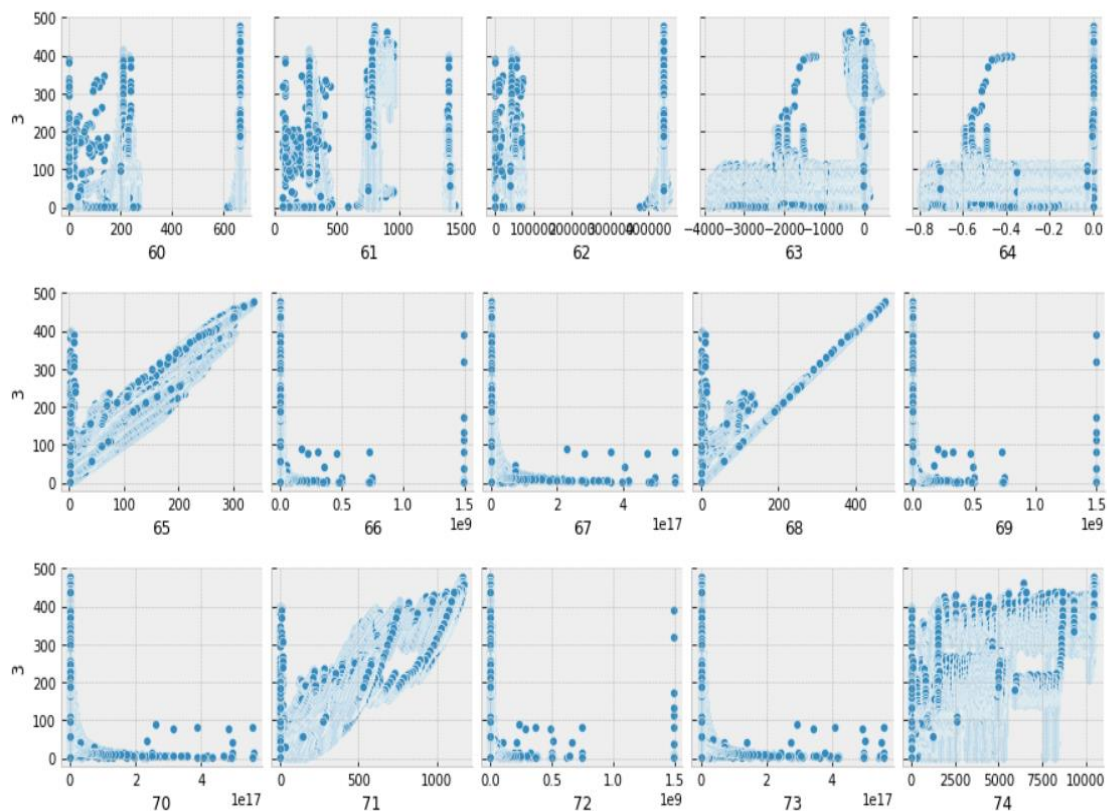


Figure 22. Sample pair plot of columns.

In the above figure a sample pairplot of `_col160` to `_col174` are provided as an example to understand the relationship of these columns with that of `_col103`. We can see that few of

the feature vectors are linearly related and identify few outliers in the relationship plot. More details of other columns can be found in the appendix.

This data analysis helped us understand the how efficient the input data is and the need to further process the data by removing the strongly related features without affecting the performance of the algorithm. This leads us to the next step of dimension reduction which is covered thoroughly in the next section.

4.2.3 Dimensional Reduction

Until this stage, we understand how too many input variables may have high correlation among each other and is not of much learning for the machine in the process. Having too many input variables often make predictive modelling task challenging leading to a popular term called “*curse of dimensionality*”. [53]

In the field of machine learning, curse of dimensionality refers to the problems that involves the learning a “state-of-nature” from a finite data sample in a high dimensional feature space. In this study, we analyse how the performance of the algorithm is impacted having large number of dimension and attempt to achieve the higher or at least the same results in a reduced dimension.

The application can be configured to run on both the settings i.e., to train models with original set of features and similar activity by reducing the features set. We use spark’s inbuilt library called Principal Component Analysis (PCA) to control the number of relevant features. PCA is a statistical procedure that converts a set of observations possibly correlated variables into a set of values of linearly uncorrelated variables by using orthogonal transformations, called principal components, denoted by K .

This step is one of stages in the ML Pipeline which is carried out after assembling the feature vectors as described below:

```

11 def process_multiclass_pipeline(df: dataframe.DataFrame, estimator):
12     assembler = VectorAssembler(
13         inputCols=[x for x in df.columns if x != "label"],
14         outputCol="features_v",
15         handleInvalid="skip"
16     )
17
18     # PCA
19     pca = PCA(k=get_k(), inputCol="features_v", outputCol="features")
20
21     algo = ()
22     if estimator.lower() == "perceptron":
23         # features_col = len(assembler.getInputCols())
24         features_col = pca.getK()
25         print("Number of features column: ", str(features_col))
26         algo = get_perceptron_estimator(features_col)
27     else:
28         algo = get_estimator_for_multiclass(estimator)
29
30     # Split the data into training and test sets (30% held out for testing)
31     (trainingData, testData) = df.randomSplit([0.7, 0.3])
32
33     pipeline = Pipeline().setStages([assembler, pca, algo])

```

Figure 23. PCA instantiation in ML Pipeline.

In the above figure, we select the value of K as the number of principal components and configure the mode application. The results are obtained with multiple values of K and evaluating the models which would help us understand the improvements, in having the dimensions reduced. The evaluation and comparison of the results in different configurations are covered in the Results and Comparison section.

4.3 Benchmarking with Binary Classifiers

From the above step, we have the prepared and well analysed dataframe that is ready to be fed into the ML Pipeline. In this section, we move a step ahead towards one of our major goals to develop an efficient model that can identify the attacks by setting some benchmarks with binary classification. The final model is then compared with these results and other evaluation metrics to understand the efficiency of the model.

Few examples of how the application is configured to run in binary mode with commands are as below:


```

## BINARY MODE
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator gbt --mode binary > logs/without_pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator decision_tree --mode binary > logs/without_pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator random_forest --mode binary > logs/without_pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt

```

Figure 24. Run commands for binary mode.

The common classifiers like Random Forest, Decision Trees, GBT, Naïve Bayes and Logistic Regression are used to train the data and result in the best possible model. The evaluation metrics for each of the models are compare amongst each other and tuned to the best performing criteria. Below are some code snippets of the usage of the algorithms for binary classification:

```

_estimators = dict()

def estimators_for_classifiers():
    global _estimators
    # Add Param Grid for new algorithms

    _estimators = {
        'random_forest': RandomForestClassifier(LabelCol="Label", featuresCol="features", numTrees=10),
        'decision_tree': DecisionTreeClassifier(LabelCol="Label", featuresCol="features"),
        'gbt': GBTClassifier(LabelCol="Label", featuresCol="features"), # only supports binary classification
        'lr' : LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.0, family="multinomial"),
        'nb' : NaiveBayes(smoothing=1.0, modelType="gaussian"),
        'fm' : FNCClassifier(LabelCol="Label", featuresCol="features", stepSize=0.001)
    }
    return _estimators

def get_estimator_keys():
    return estimators_for_classifiers().keys()
estimators_for_classifiers() : 'nb'

```

Figure 25. Algorithms supported for Binary classification.

The figure above shows the supported algorithms by the application in binary mode. The keys for their corresponding instances are expected as the runtime parameters, so that the application can be run parallelly to train the input dataset with different models simultaneously.

```

29
30 # Split the data into training and test sets (40% held out for testing)
31 (trainingData, testData) = df.randomSplit([0.6, 0.4])
32
33 pipeline = Pipeline().setStages([assembler, pca, algo])
34
35 # Tune multiclass
36 tdf_cross, tdf_train = tune_multiclass(df, estimator)
37
38 print("Training model ...")
39 pipeline_model = pipeline.fit(trainingData)
40 pipeline_model.write().overwrite().save("models/multiclass/"+estimator+"/base_model")
41 print("Training complete. Base model saved in 'models/multiclass/*'")
42
43 transformed_df = pipeline_model.transform(testData)
44
process_multiclass_pipeline()

```

Figure 26. ML Pipeline instance creation.

The above screenshot shows the creation of the pipeline in binary mode. The input data is split to training and test dataset as 6:4 i.e., 60% of the input data is randomly selected to train the algorithms and the rest 40% to test the resultant model and predict the outcome. The pipeline instantiation is decoupled from the selected algorithm hence no matter what algorithm is selected, the application behaves in a similar fashion.

Algorithms selected for this thesis are mostly decision trees or some ensemble of decision trees for instance, *random forest*. During the training process, random forest creates multiple trees for each decision made for each input. The following diagram provides us an overview of a single tree with a configured depth and its parameters.

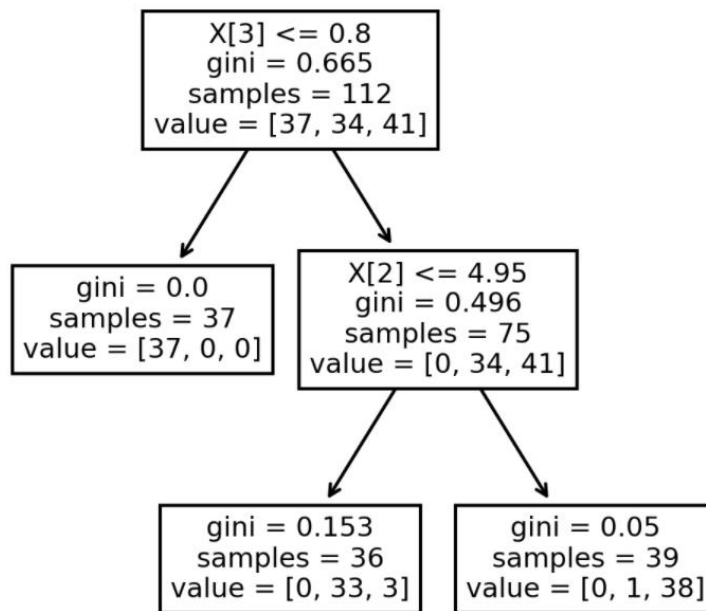


Figure 27. A sample tree from Random Forest training process.

From the above figure, we can see how random forest is making the decision from the samples with the conditions based on the measure the *gini_index*. The corresponding values are highlighted in the nodes of the diagram above.

Once the model is trained, it is used to predict the test data and the same model is saved as the outcome of this pipeline process. The following table shows a brief overview of the performances of the algorithms in binary mode using default values in Spark, and the metrics calculated using Spark’s default libraries:

Metrics	Decision Tree	Gradient Boosted Trees	Random Forest	Naïve Bayes
Accuracy (in %):	99.74	99.96	99.81	70.84
Test Error:	0.00258468	0.000315617	0.00181969	0.291578
True Negative:	812694	815199	814914	356481
False Positive:	2709	204	489	458922
False Negative:	1270	262	2090	13985
True Positive:	686302	687310	4206	685482

Table 3. ML Pipeline performance in binary mode.

The values above shows that although all the algorithms achieved very good accuracy in the sample dataset, out of the three GBT has the highest accuracy, by a very small margin. Random Forest outperforms decision trees when default set of parameters are used in both the cases. A more comprehensive and detailed analysis with numerous configurations of the application is covered in the Results and Comparison section.

4.4 Categorizing with Multiclass Classifiers

In the above binary method, we use dataset with limited attack type to analyse if the data is malicious or not. In real life, the scenario we need to detect not one but many types of attacks in a network traffic, so applying 10 different algorithms for 10 different types of

attacks would consume lot of compute power and increase the computation time drastically, which is not beneficial.

Hence, to simulate the real-life scenario up to some extent, all the attack types are coagulated into a single dataset and fed into the existing machine learning system and the models are enhanced and tuned to classify multiple classes, analyse and the same metrics are compared in detail in the next section.

To achieve our goal i.e., to detect the attack type of the input data, in this study, *MultiClass-Classification* techniques are being adapted. An ML Pipeline configured with such techniques can categorize the different attack types in the input dataset. More than one models are trained, compared to achieve the best possible metrics and tuned to get the best accuracy in the huge set of input records.

We extend the functionality of the same classifiers based on Decision trees as described in the previous section i.e., Random Forest, Decision Tree and considering Multiple Layer Perceptron based on neural network.

The application is run in the multiclass mode for categorizing attacks and supporting Multiclass Classification. The following are some run commands with the algorithms dynamically selected:

```
### WITHOUT PCA
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator gbt --mode binary > logs/without_pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator decision_tree --mode binary > logs/without_pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator decision_tree --mode multiclass > logs/without_pca/multiclass/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator random_forest --mode binary > logs/without_pca/binary/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator random_forest --mode multiclass > logs/without_pca/multiclass/output-$(date +%b-%d-%Y-%H:%M:%S').txt
# spark-submit --master spark://rakshit-thesisv2.cloud.ut.ee:7077 outlier-detection/main.py --estimator perceptron --mode multiclass > logs/without_pca/multiclass/output-$(date +%b-%d-%Y-%H:%M:%S').txt
```

Figure 28. Commands to run the application in multiclass mode.

When we consider the decision tree algorithm for the ML Pipeline to train the input dataset with, we instantiate the algorithm in the application as shown below:

```

@keyword_only
def __init__(self, featuresCol="features", labelCol="label", predictionCol="prediction",
             probabilityCol="probability", rawPredictionCol="rawPrediction",
             maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0,
             maxMemoryInMB=256, cacheNodeIds=False, checkpointInterval=10, impurity="gini",
             seed=None, weightCol=None, leafCol="", minWeightFractionPerNode=0.0):
    """
    """

```

Figure 29. Decision tree and parameters used for training.

Decision trees are robust to noisy data and can handle continuous values well but still those values must be fit into buckets. It can handle multiple labels and provide a probabilistic measure of the predicted value belonging to a specific class.

In the above chunk of code, we see the parameters with which the Decision Tree algorithm is instantiated in pyspark. We can also visualize a single tree to get a better observation how the decisions are made at each step. To visualize, the tree the main params in focus are:

criterion = “gini”. The function to measure the quality of a split. It takes the value “gini” for Gini impurity and “entropy” for the Information Gain.

random_state = 100. This controls the randomness of the estimator and is an integer value. Random state parameter controls the random number generator used and is only in effect when the randomization is considered while splitting or fitting.

max_depth = 3. An integer value depicting the maximum depth of a tree.

min_samples_leaf = 5. A minimum no. of samples required to be a leaf node. When there at least *min_samples_leaf* in each left and the right branches of a node, a split point is considered.

When the pipeline is run with the algorithm configured with the above parameters, it creates certain steps based on some decision which can be visualized as trees as bellows:

[35]:

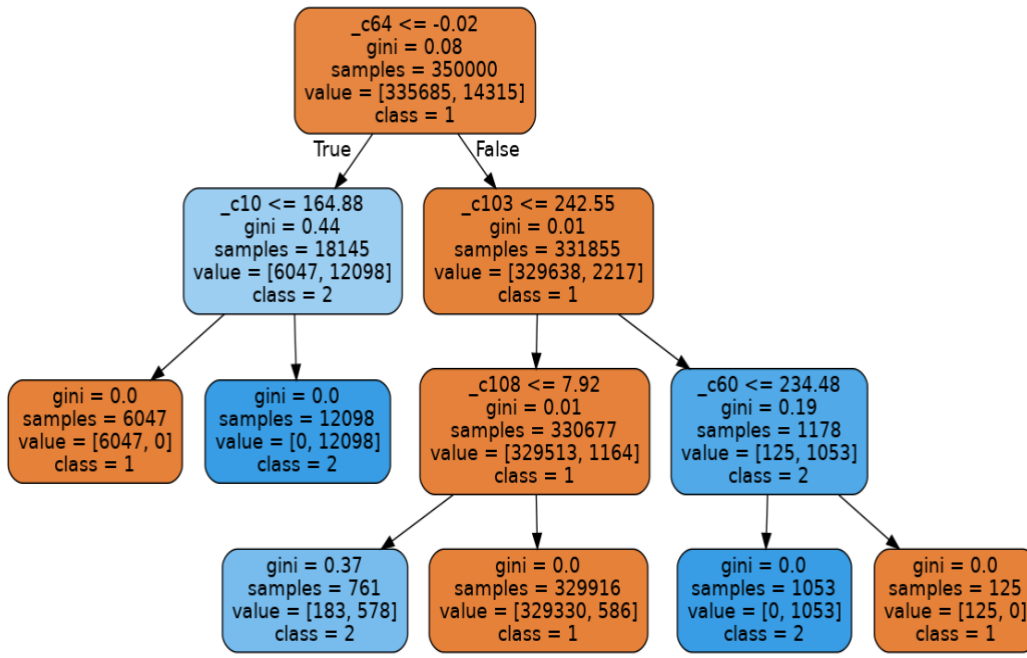


Figure 30. Sample tree from decision tree algorithm.

From the figure above, we understand how the decisions are made for each node considering the parameters listed above. Maximum depth of the leaf node considered in this setting is 3. We observe that just like the binary classification, the gini index is used to make the decision and apart from that the class of that node is also determined in multiclass classification. We also calculate the feature importance for each column in making the decision according to which the most important features can be listed in a descending order as follows:

[37]:

	FeatureImportance
_c64	0.576334
_c10	0.310004
_c103	0.071550
_c108	0.033520
_c60	0.008591

Figure 31. Feature Importance of the selected tree.

Random forest has a built-in function to calculate the feature importance of each tree which is the measure of the *gini* index in a classification problem. The above figure

shows how the mentioned columns are important to make the decision as the above sample tree.

In this thesis, the ML Pipeline uses random forest to train the input dataset with different modes of the application and is experimented with varying parameters in the spark environment. The model is the output of the pipeline and is used for further tuning purposes results are gathered in the form of metrics used to evaluate the models are discussed in the next section.

For Multi-Layer Perceptron, as the algorithm selected in the application, it uses backpropagation and uses loss function for optimization and L-BGFS as an optimization routine.

```
def get_perceptron_estimator(length_of_features):  
    layers = [length_of_features, 5, 4, 3]  
    return MultilayerPerceptronClassifier(labelCol='label', featuresCol='features', maxIter=100,  
                                         layers=layers, blockSize=128, seed=1234)
```

Figure 32. MLP instance parameters used in the thesis

The above figure shows how the instance of Multi-layer Perceptron algorithm used in this paper which is supported by spark.

Here,

layers determine the layers of the neural network which consists of

[total number of features, size of hidden-layer, size of hidden-layer, output-classes size],

blockSize determines the stack of the input data within internal partitions. This size is adjusted according to the size input data when it is bigger than the required.

4.5 Tuning the models

From the previous sections, we have the trained models for all set of features as well as with reduced features, as the output of the pipeline. At this stage the models are sent to the tuner module for hyper parameters tuning.

The tuner module is common for both binary and multiclass mode. To ensure unbiased tuning process, no special care is taken for either binary classification or in multiclass classification problems.

This module has 2 separate tuning tools supported by Spark *TrainValidationSplit* and *CrossValidator* as shown below:

```
def evaluate_with_train_validation_split(df: dataframe.DataFrame, estimator, pipeline, evaluator):
    print("Tuning with train validation split...")
    train, test = df.randomSplit([0.9, 0.1], seed=12345)

    paramGrid = get_param_grid(estimator)

    tvs = TrainValidationSplit(estimator=pipeline,
                              estimatorParamMaps=paramGrid,
                              evaluator=evaluator,
                              # 80% of the data will be used for training, 20% for validation.
                              trainRatio=0.8)

    model = tvs.fit(train)
    tf_df = model.transform(test)
    tf_df.select("features", "label", "prediction").show(5)

    return model.bestModel, tf_df
```

Figure 33. Train Validation Split Process for hyper-params tuning

Figure 33 refers to the usage of the two supported hyper-param tuning tools by Spark in this thesis. We note the parameters passed as an argument to the instances created for this purpose.

For *TrainValidationSplit* tool, it accepts the binary/multiclass pipeline prepared with user-defined algorithms in the previous steps. A set of *paramGrid* is essentially some arrays of values of the respective algorithms to get the best possible results. An instance of the evaluator is also expected by the tool to evaluate the models against. The final crucial parameter expected is the *trainRatio* which indicates the strategy for preparing the validation dataset.

So, the input dataframe is essentially split into 3 parts – the train dataset, validation dataset and the test dataset. The tuner tools internally split the train data, it is fitted with, with the ratio provided as the value. In this study, the tuner is configured to split the train dataset as 80% for training the models and 20% for validation from the 90% of the original input dataset fed to it.

It is then fitted to the train dataset and emits the model as the output which is then used to transform the test dataset. It creates the prediction column with the predicted values in the

transformed dataset. This dataset is then evaluated with the calculated metrics and compared with other models for performance.

```
def evaluate_with_cross_validation(df: dataframe.DataFrame, estimator, pipeline, evaluator):  
    print("Tuning with cross validation...")  
    train, test = df.randomSplit([0.9, 0.1], seed=12345)  
  
    paramGrid = get_param_grid(estimator)  
  
    crossval = CrossValidator(estimator=pipeline,  
                             estimatorParamMaps=paramGrid,  
                             evaluator=evaluator,  
                             numFolds=2)  
  
    model = crossval.fit(train)  
    tf_df = model.transform(test)  
    tf_df.select("features", "label", "prediction").show(5).show()  
  
    return model.bestModel, tf_df
```

Figure 34. Cross Validation Process for hyper-params tuning

Figure 34 refers the usage of the CrossValidator tool supported by Spark. Similar to the train validation split, it also accepts the pipeline, paramGrid, evaluator as the input parameters but the only difference is the strategy it creates the validation dataset.

It accepts *numFolds*, as the fourth parameter which denotes the number of folds the train dataset will be splitted to.

For instance, in this study *numFolds=2*, this means the train dataset will contain 2 sets of train and validation data. Each set will be again split to contain 2/3 of the data as the train dataset and 1/3 as the validation dataset.

After we create the instance of the CrossValidator tool, it is fitted to the dataset which was 90% of the original input data and returns the model. This model transforms the test dataset and returns the transformed dataset with the new *prediction* column added to it. This column contains the values of the predicted categories of attacks and are evaluated against the metrics mentioned earlier.

```

def get_param_grid(estimator):

    if estimator == "random_forest":
        algorithm = get_estimator(estimator)
        return ParamGridBuilder() \
            .addGrid(algorithm.numTrees, [5, 10, 20]) \
            .addGrid(algorithm.maxDepth, [5, 10]) \
            .addGrid(algorithm.cacheNodeIds, [True, False]) \
            .build()

    elif estimator == "decision_tree":
        algorithm = get_estimator(estimator)
        return ParamGridBuilder() \
            .addGrid(algorithm.maxDepth, [3, 6, 9, 12]) \
            .addGrid(algorithm.maxBins, [30, 50]) \
            .addGrid(algorithm.cacheNodeIds, [True, False]) \
            .build()

    elif estimator == "gbt":
        algorithm = get_estimator(estimator)
        return ParamGridBuilder() \
            .addGrid(algorithm.maxDepth, [3, 6, 9, 12]) \
            .addGrid(algorithm.maxBins, [30, 50]) \
            .addGrid(algorithm.cacheNodeIds, [True, False]) \
            .build()

    elif estimator == "perceptron":
        layers = [23, 10, 6, 4]
        algorithm = MultilayerPerceptronClassifier(labelCol='label', featuresCol='features', maxIter=100,
                                                    layers=layers, blockSize=128, seed=1234)

        return ParamGridBuilder() \
            .addGrid(algorithm.maxIter, [75]) \
            .addGrid(algorithm.blockSize, [150, 200]) \
            .addGrid(algorithm.layers, [[23, 15, 10, 4]]) \
            .build()

```

Figure 35. Hyper params grid for some of the supported algorithms.

Cross Validator can be expensive if we do not choose the *paramGrid* carefully. The above screenshot provides the list of values for all the parameters for various algorithms, the models are tuned with. Train Validation Split only evaluates each combination of parameters once but Cross Validator performs each combination multiplied to the value of *numFolds* times which is the reason Train Validation Split is less expensive than Cross validator but produces lesser accurate results when the training dataset is not large enough.

5 Evaluation Techniques

At this stage, we have the results of following scenarios:

- The base model as the output of the ML Pipeline, in binary mode, with all the features set.

- The base model as the output of the ML Pipeline, in multiclass mode, with reduced features set.
- The tuned models from all the algorithms in binary mode
- The tuned models from all the algorithms in multiclass mode.

In the next step, we evaluate the performance of the outcome of each of the scenarios in details.

5.1 Mechanism

All those models are used to transform and predict the test dataset in each scenario and the transformed dataset is returned. This resulting dataset is then passed through the evaluator module for measuring the evaluation metrics which will be used to compare and justify the results.

```
def evaluate_binary_classifier(tf_df):
    evaluator = BinaryClassificationEvaluator(
        labelCol="label", rawPredictionCol="prediction")

    # Accuracy
    accuracy = evaluator.evaluate(tf_df)
    areaUnderROC = evaluator.evaluate(tf_df, {evaluator.metricName: "areaUnderROC"})
    print("areaUnderROC: ", str(areaUnderROC))
    print("Accuracy: ", str(accuracy))
    print("Test Error = %g " % (1.0 - accuracy))
    # Confusion Matrix
    tf_df.crosstab("prediction", "label").show(30, False)
    metrics_sklearn(tf_df)
```

Figure 36. Evaluator in binary mode.

In binary mode, the evaluator uses *BinaryClassificationEvaluator* as shown above. It accepts the original *label* column and the *rawPredictionCol* as the arguments and calculates the metrics like, accuracy, testError, areaUnderROC, recall, precision, F-score etc. It mostly uses the Spark supported metrics but to have a wider perception, *sklearn* library is also used to calculate some of the metrics.

```

def evaluate_multiclass(df: dataframe.DataFrame):
    # Select (prediction, true label) and compute test error
    evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction")
    accuracy = evaluator.evaluate(df, {evaluator.metricName: "accuracy"})
    weightedPrecision = evaluator.evaluate(df, {evaluator.metricName: "weightedPrecision"})
    weightedFMeasure = evaluator.evaluate(df, {evaluator.metricName: "weightedFMeasure"})
    # fMeasureByLabel = evaluator.evaluate(df, {evaluator.metricName: "fMeasureByLabel"})
    weightedRecall = evaluator.evaluate(df, {evaluator.metricName: "weightedRecall"})
    weightedFalsePositiveRate = evaluator.evaluate(df, {evaluator.metricName: "weightedFalsePositiveRate"})
    weightedTruePositiveRate = evaluator.evaluate(df, {evaluator.metricName: "weightedTruePositiveRate"})
    print("Test Error = %g \nAccuracy: %s \nFPR: %s\nTPR: %s\nF-measure: %s\nPrecision: %s\nRecall: %s"
          % ((1.0 - accuracy), accuracy*100, weightedFalsePositiveRate, weightedTruePositiveRate, weightedFMeasure, weightedPrecision, weightedRecall))

```

Figure 37. Evaluators used in multiclass mode.

In the multiclass mode, the evaluator used is `MulticlassClassificationEvaluator` supported by Spark. As similar to the `BinaryClassificationEvaluator`, it also accepts the label and the prediction column to calculate the metrics. In this mode, the metrics are calculated with the spark's supported metrics only.

It is also noted that the same evaluators for both the modes are used for hyper-parameters tuning wherein the evaluation was done in the validation set in order to select the parameters. But in this case, the evaluation is performed on the actual test dataset to analyse the model performance.

5.2 Metrics

This section is divided into two parts; the first provides an overview of the results of the evaluator module and various metrics collected to give us an idea about the performance of the models in terms of their benchmarks, and the second section sheds some light about the performance of parallel computation of the application using the big tools such as Apache Spark.

5.2.1 Performance Metrics

For each of the scenario mentioned in the previous section all the metrics are collected and logged in separate file as an application output respective of the algorithms.

For instance, in binary mode, for benchmarking process, the following table depicts the evaluation metrics calculated for all features set and without hyper-parameters tuned model. The below table 4 captures the performance metrics calculated with the help of sklearn library of python:

Metrics	Decision Tree	Gradient Boosted Trees	Random Forest	Naïve Bayes
Accuracy (in %):	98.74	99.97	99.83	68.54
True Negative:	812694	815199	814914	356481
False Positive:	2709	204	489	458922
False Negative:	1270	262	2090	13985
True Positive:	686302	687310	685482	673587
Recall	0.9981	0.9996	0.9969	0.9796
Precision	0.9960	0.9997	0.9992	0.5947
F-Score	0.9971	0.9996	0.9981	0.7401

Table 4. Benchmarks of model performance in binary mode calculated with sklearn.

Similarly, multiclass mode, with similar conditions, the models generated for categorizing attacks having the following evaluation metrics calculated with Spark's default metrics library:

Metrics	Decision Tree	Random Forest	MLP
Accuracy (in %):	99.07	99.00	82.91
Test Error:	0.0092	0.0099	0.1709
Recall	0.9907	0.9900	0.8262
Precision	0.9897	0.9901	0.678
F-Score	0.9901	0.9879	0.745

Table 5. Model performance in multiclass mode trained with all features set.

It is observed from metrics in Table 5, that GBT which only supports binary classification, clearly outperforms the other two different types of models when used in a similar environmental condition in terms of accuracy. On the other hand, random forest provides a consistent result in both binary and multiclass classification problems having accuracy of about 99%. However, if we compare with the benchmarks, gradient boosted trees out-performs random forest, decision tree and naïve bayes algorithms in terms of accuracy metrics, but as this algorithm only works with binary mode of classification, it may not be considered as a suitable candidate for comparison.

However, for MLP the accuracy is quite low, one of the potential reasons could be that MLP is configured as a very basic neural network and no tuning, as mentioned earlier. The MLP algorithm supported by Spark uses standard feed forward network and initializes the weights with random values which is then back propagated to learn better but when the hidden layer is very limited, the learning efficiency could be very low. Hence to optimize this Filiberto, Yaima et.al. in their paper [54] showed that rather than

randomly initializing the weights, if they are set from the conditional features which maximizes the measure quality of similarity as proposed in Rough Set theory. The experimental study for problems of function and classification shows better results with the proposed method of initializing weights when the data is fed from the input layer to the hidden layer.

Another interesting approach was proposed by Krzysztof Halawa in his paper [55] that as the hidden layers of the MLP uses the sigmoid function as the activation function of the feed forward network, if the last hidden layer uses the mean of squared errors before passing on to the output layer, it can improve the learning process of the MLP models trained earlier and need not start from the beginning. This might be useful for an already existing model trained with legacy data and a pipeline consisting of very large hidden layers with initial weights being randomly selected.

This essentially means, there are numerous possibilities to improve the neural networks for better performance and would require some extra effort. From the comparison table, we already achieved the best results in terms of accuracy and other metrics, which puts additional efforts in exploring neural networks beyond the scope of this paper.

5.2.2 Efficiency Metrics

Although ML jobs can be iterative and time consuming, one of the major goal of this paper was also to focus on the performance, and efficiency of the proposed application. Apache Spark can be setup with multiple worker nodes and has the capability to distribute the task among its worker nodes and compute in a parallel fashion. This functionality of Spark was leveraged, and the pipeline was set with a fixed optimal configuration and ran in several underlying configurations of Apache Spark tweaking the worker instances, cpu and memory.

This setup simulates a serial processing environment with minimal configuration to run the application having 1 worker instance with 1 processing unit with 3 GB of memory and gradually increasing the worker instances to 15 each with 1 core of the processing unit keeping the memory constant in order to calculate the efficiency metrics like *Speedup, Efficiency, Scalability*.

For the below metrics, considering the serial computation setup of the environment as the ideal machine and the pipeline configuration as the best algorithm, the efficiency metrics are calculated as:

Env Config	Total Processing Time	Speedup	Efficiency	Scalability
1 worker with 1 CPU	2 hrs 49 mins 08secs	n/a	n/a	n/a
5 workers with 5 CPUs	37mins 18 secs	4.53	0.9	4.5
10 workers with 10 CPUs	21 mins 06 secs	8.01	0.8	8.0
15 workers with 15 CPUs	17 mins 18 secs	9.75	0.65	9.7

Table 6. Efficiency metrics of the pipeline run in different configurations

In the above table, a few of the result are provided for the configuration variations described above. The outcome of the metrics comparison and their change in behaviour with the system resources are covered in more details in the next section.

6 Results and Comparison

This section covers the results of each outcome from the previous steps both in terms of performances of the ML models and the application performance in terms of parallel computations. The performance metrics calculated from the multiclass classification are compared with the corresponding benchmarks and the results of the model being evaluated with two different Evaluators are discussed.

And finally, we compare with the outcomes of the relevant studies as mentioned in the Related Work section and justify the need of this study.

6.1 Comparing with the benchmarks

In this section we look at the outcome of the several steps performed until now. The very first outcome of the application is the models as the output of the ML Pipeline. For each configuration the application is run, the models are saved into the project directories as:



Figure 38. The output of the ML Pipeline

These models are trained and well-tuned with the parameters mentioned above and ready to be re-used for a fresh set of test dataset.

6.1.1 Outcome of Hyperparameter Tuning

The application was run with various conditions like, binary mode with large features set, multiclass mode with large features set, binary mode with reduced features set, multiclass mode with reduced set of features, the outcome of the binary classification base model. The two hyper param tuning tools uses different strategy to evaluate the models and select the best one, hence we look at the results of each and compare which one of them is the most efficient. The results of those scenarios are saved a log file available with the project source code in the github.

The results of some of the scenarios are plotted in a bar chart comparing the multiclass mode models with the benchmarks as shown below:



Figure 39. Comparison of binary and multiclass modes with all features.

The above figure compares the accuracy of the Decision Tree algorithm in multiclass mode with its benchmarks in the binary mode. It compares all the model outputs of the application i.e., the base model without the hyperparameter tuning, the best model tuned with Cross Validator and the best model tuned with Train Validation Split. The overall accuracy is quite promising for all the models in the multiclass mode. As we can see the

model tuned with Cross Validator has lower accuracy than the other two models which is unlikely the case in the multiclass mode.

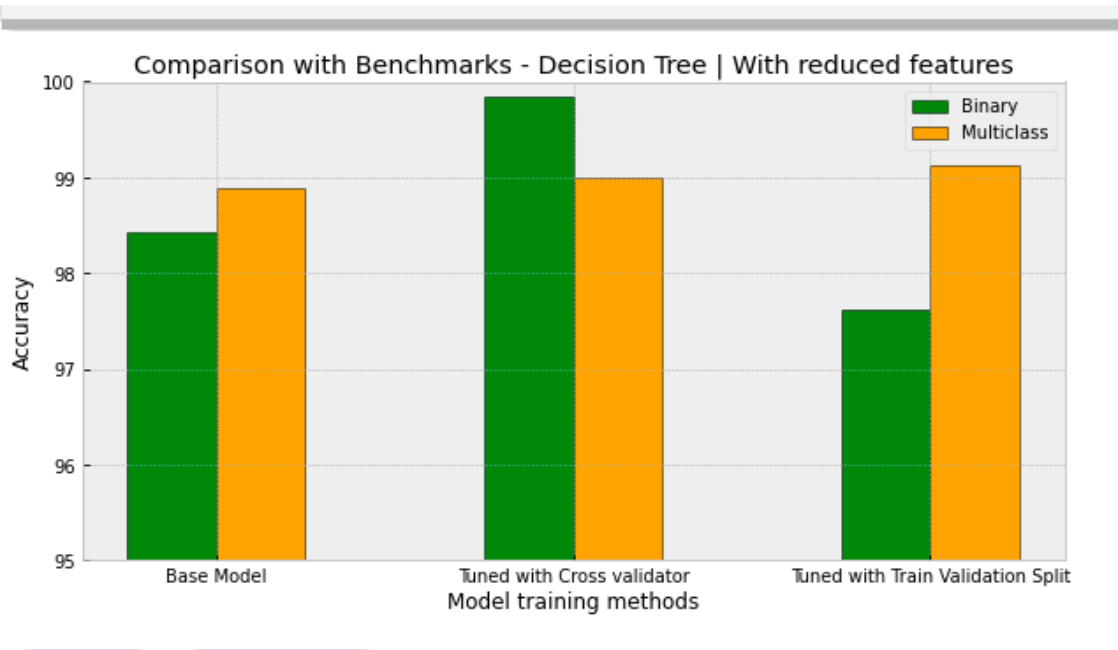


Figure 40. Comparison of binary and multiclass mode with reduced (k=25) features

The above figure depicts the comparison of the accuracy of decision tree algorithm trained in the multiclass mode with that of its benchmark but with feature reduction mechanism applied to the ML Pipelines. We observe that when feature reduction is used, the accuracy of binary classifier drops however, the accuracy of the multiclass classifier is consistent throughout the process. The base model itself achieves accuracy more than 98% in both the modes, but when tuned with hyperparameters, the accuracy of the model in binary mode drops when train validation split is used, whereas it increases significantly for Cross Validator. On the other hand, the accuracy of the model of the multiclass mode improves then the base model when tuned with hyperparameters.

The next two figures compare the performance of Random Forest algorithm with its benchmarks in terms of accuracy as the evaluation metric.

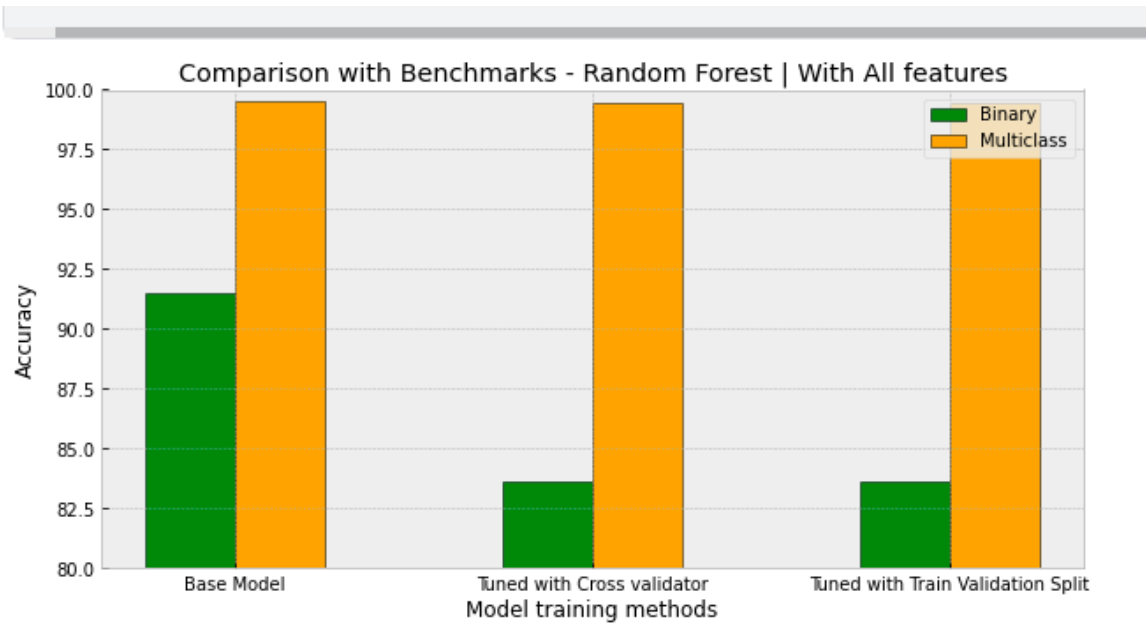


Figure 41. Comparison of Random Forest in both modes with all features.

When Random Forest is used to train the input data with all the features, the accuracy significantly improves in multiclass mode compared to its benchmarks. The performance is quite stable across the process and performs quite well even with default parameters.

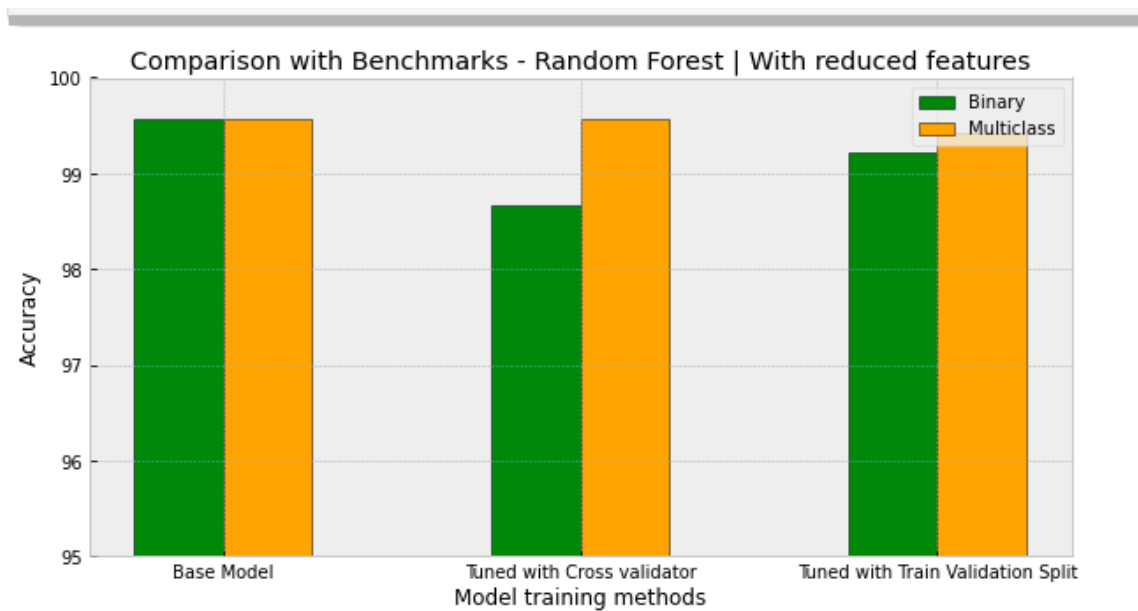


Figure 42. Random Forest comparison with both modes with reduced (k=25) features.

Figure 42 shows the performance of Random Forest with feature reduction techniques are stages are incorporated in the ML Pipeline. Multiple experiments are conducted with

variations of principle components i.e., varying the value of k in the PCA step of the pipeline, with different values such as 18, 20, 25 etc., close to the original set of parameters collected in the input dataset. So, with reduced set of features, the accuracy improves in the case of binary classification, but the base model remains consistent. In the multiclass mode, the model either outperforms than its corresponding benchmarks or is at par with its peer but shows inconsistent results while tuned with both the tuner modules.

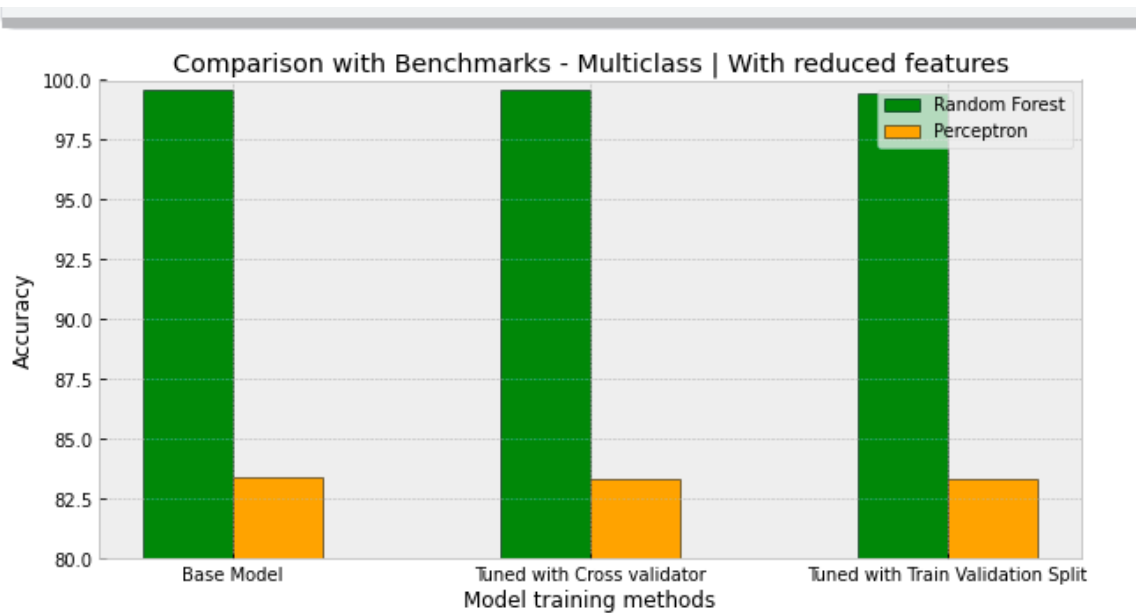


Figure 43. MLP vs Random Forest with reduced ($k=25$) features.

The above figure compares the accuracy of Multi-layer Perceptron with that of Random Forest and it clearly shows that the current setting is not quite suitable for even a native neural network to efficiently categorize the different attacks in the dataset.

6.1.2 Outcome of Feature Reductions

In this section, the comparison with the multiclass classification is performed with that of its benchmarks, focusing on the impact of feature reduction techniques. For the comparison to be comprehensive, all the evaluation metrics are compared for the supported models in both binary and multiclass modes.

Below are two tables containing the values for the metrics of the models that are hyperparameters tuned with the *Cross Validator* tool. The reason for choosing Cross Validator tuned model is that it uses extensive tuning process for deciding the best model

out of all the parameters. In most cases, the performance of the tool is quite consistent with the expense of computational resources.

In the tables below, one special case is considered i.e., when the application is using Multi-Layer Perceptron algorithm in the pipeline. As it works only for multiclass classification, it is compared with the metrics when the application is run with gradient boosted trees and naïve bayes as the algorithms as they support only binary classification.

Metrics	Decision Tree		Random Forest		GBT	MLP
	Binary	Multiclass	Binary	Multiclass	Binary	Multiclass
Accuracy (in %)	99.64	99.67	99.90	99.46	99.88	82.91
Test Error	0.0035	0.0032	0.00091	0.0053	0.001212 63	0.1709
Recall	0.9722	0.9967	0.9986	0.9946	0.99906	0.8262
Precision	0.9997	0.9966	0.9994	0.9937	0.9982	0.6779
F-Score	0.9858	0.9962	0.9990	0.9941	0.9986	0.7450

Table 7. Comparing model performances with its benchmarks trained with all features set.

The table above shows that decision tree performs equivalently in both the binary and the multiclass mode i.e., when categorizing attacks, if decision tree is considered, the performance of the model retains the same accuracy as in the binary classification. The test error is a bit greater in the binary than that of multiclass mode. As for the metrics recall, precision and f-score, the model behaves almost similar in both the cases although it slightly improves for metrics recall and f-score.

For the case of random forest, we see the accuracy drops slightly in multiclass mode and so is the case for all other metrics which means the algorithm is less correct in the multiclass mode than in binary mode when it trains with all the features of the model.

For the special case, it is clearly observed that MLP has a very low performance as compared to the binary classifier using gradient-boosted trees.

Metrics	Decision Tree		Random Forest		GBT	MLP
	Binary	Multiclass	Binary	Multiclass	Binary	Multiclass
Accuracy	99.85	99.00	98.66	99.57	99.75	83.34
Test Error	0.0014	0.0099	0.0133	0.0042	0.00248	0.1665
Recall	0.9971	0.9900	0.9733	0.9957	0.9983	0.8334
Precision	0.9971	0.9896	1.0	0.9957	0.9960	0.6945
F-Score	0.9971	0.9887	0.9865	0.9953	0.99720	0.7576

Table 8. Comparing model performances with its benchmarks trained with reduced ($k = 25$) set of features.

The above table has the metrics of the models tuned with the algorithms mentioned with feature reductions enabled in the ML Pipeline. And from the metrics above it can be inferred that the accuracy for Decision Tree slightly drops in multiclass mode than in the binary mode with reduced set of features even though it retains 99% accuracy mark. The similar is the case for the other metrics as well, although it performed well but did not outperform the benchmark value. In the case of Random Forest, it is the opposite i.e., in multiclass mode the accuracy and most of the metrics are better than its benchmarked value when trained with reduced set of features.

Comparing both the tables above, it can be observed that in binary mode, Decision Tree, algorithm improved accuracy and overall performance when it was used to trained with

reduced set of features, although GBT and Random Forest had a slight decline on the performance. In multiclass mode, Random Forest and MLP has improved performance when trained with reduced set of features as compared to Decision Tree whose accuracy decreased slightly with the no. of features.

6.1.3 Outcome of Parallel Computations

This section covers the behaviour of the application in terms of processing speed with varying configurations in the computational cluster environment. The overall cluster is initially restricted to only 1 Spark worker instance and 1 CPU with 3 GB of memory, which is just sufficient to run the application in its optimal configuration. In this restricted environment, the pipeline is run in multiclass mode selecting *random_forest* as the estimator. The steps performed in the pipeline are – input dataset and feature preparation, model training, and identifying the attack types using predictions as a typical multiclass classification problem.

With these steps in the pipeline, configured with the optimal settings, and no further changes are made in terms of complexity throughout the experimentation. The variations are performed only in the underlying environmental setups and the performance metrics are calculated as shown in the previous section. From table 6, we can see the application processing time decreases with the increase in the system resources and the Speedup of the application increases, but is true only up to certain number of additional resources as seen in the below figures:

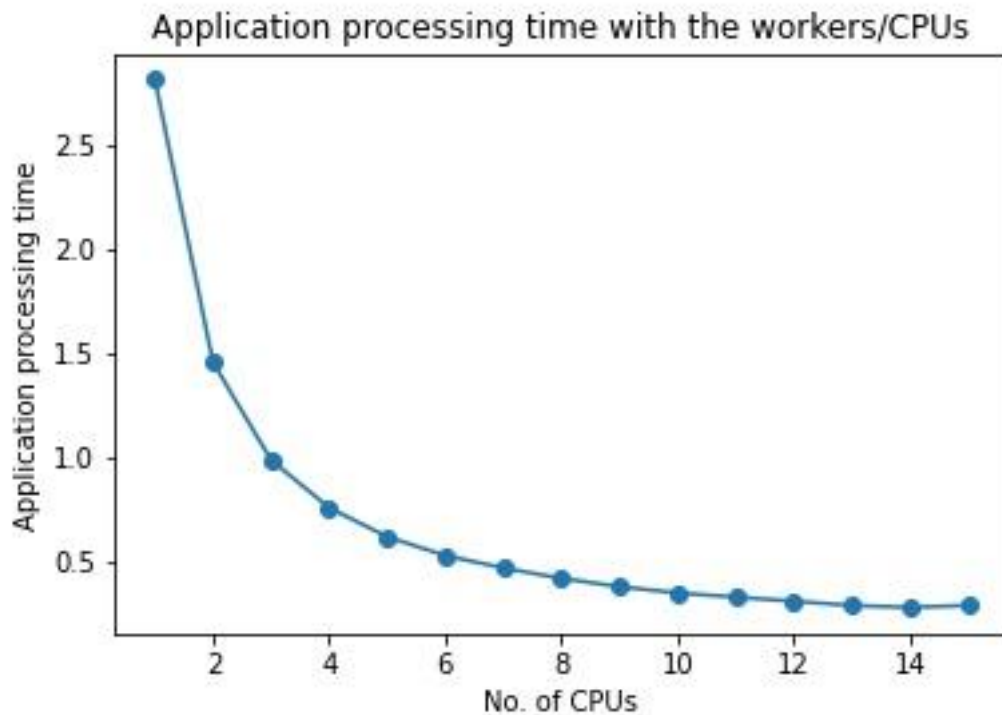


Figure 44. Processing Time with System Resources

The figure above shows how the application finishes faster keeping the application settings intact and increasing the system resources. We can also notice that the processing time reduces drastically up to a certain extent and then gradually saturates over addition of new resources. This justifies that if we keep the algorithm constant and keep on increasing the CPUs and other resources, it does not guarantee better performance. There is an optimal point where we receive the best performance in terms of the processing time. The impact of this behaviour further elicits the parallel computation's effectiveness in terms of Speedup, Efficiency and Scalability.

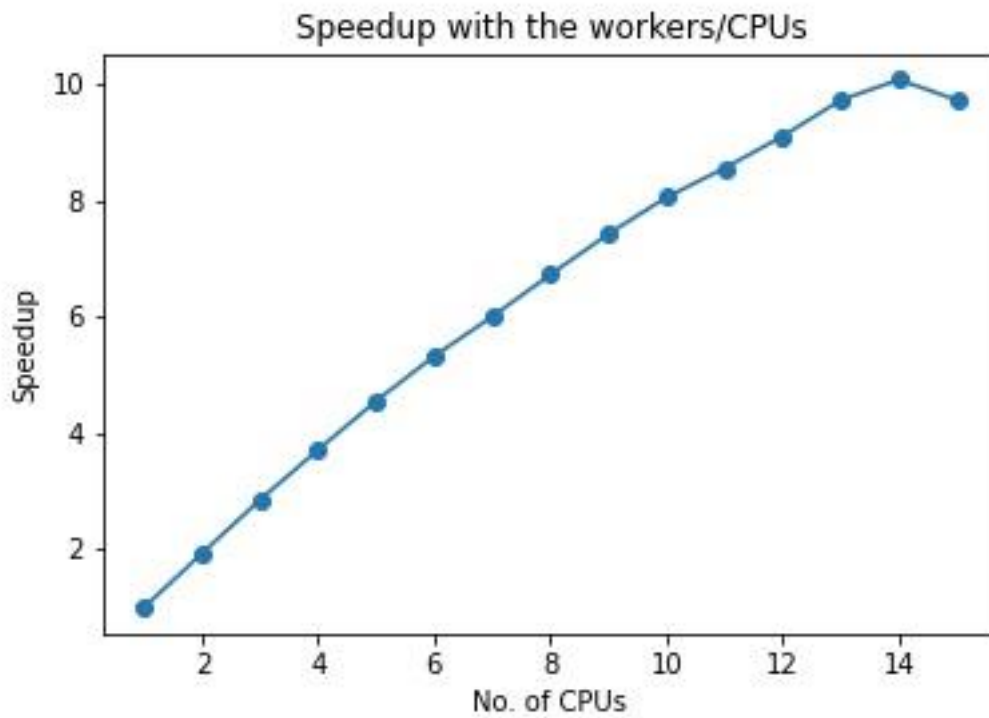


Figure 45. Speedup with system resources

Figure 45 denotes the Speedup of the parallel computation measured during the model training and attack detection process. It can be observed that the Speedup increases exponentially with the increase in the number of processors at first, but gradually when it reaches a certain saturation point, it stays stagnant for instance, from 13 worker instances to 15 we can see minimal or almost no increase in the value. Let us see some more analysis of efficiency and scalability metrics in the below sections.

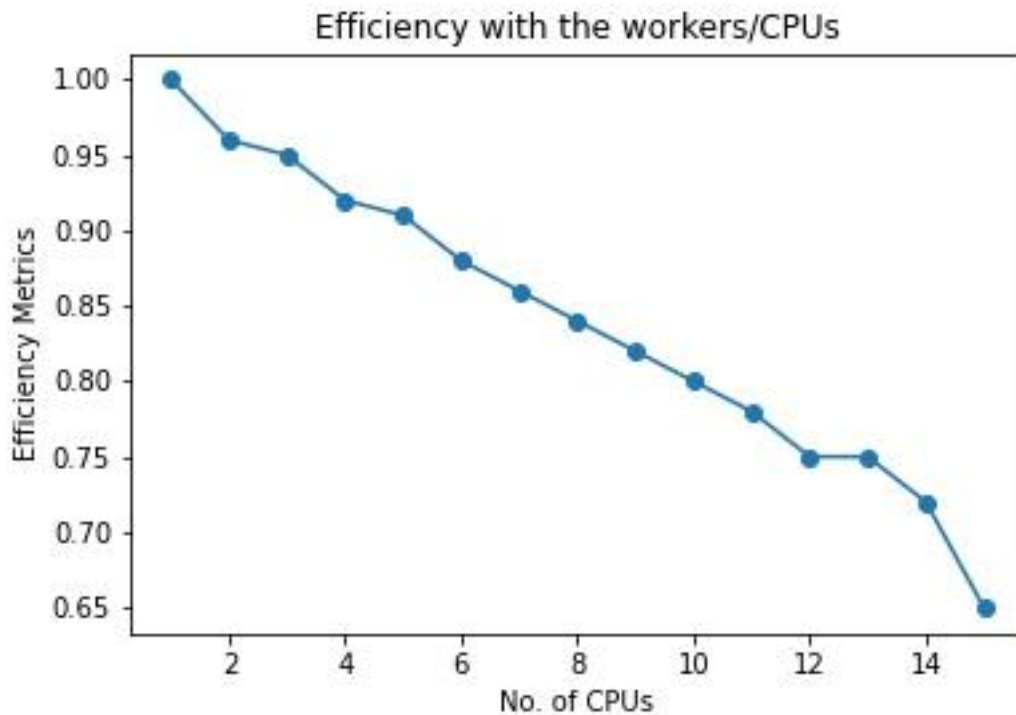


Figure 46. Efficiency with system resources.

The above figure shows the variations of efficiency of the parallel computation with the number of processors. Efficiency of a parallel computation is closely related to speedup and tells us how much work is truly done and how much is expended. From the graph, we can say that keeping the algorithm constant has decreased the efficiency of the parallel computations. In order to keep the efficiency intact, we need to add more complexities to the application, for instance, add more volume to the data or add more computation steps like more parameters in the paramGrid while the algorithms are tuned, as we keep on increasing the number of worker nodes and processors, and this suggests the CPUs ran out of tasks quickly as the number or workers were kept on increasing.

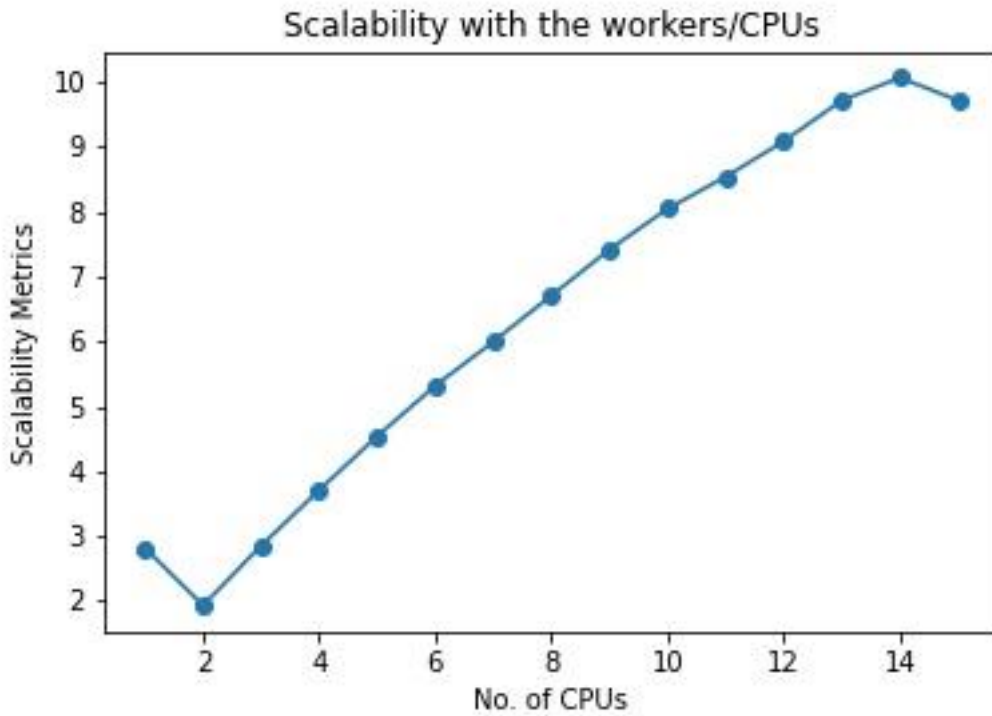


Figure 47. Scalability with system resources.

We observed how speedup tends to saturate on reaching a certain point and the efficiency decreases keeping the algorithm constant. This observation leads us to the next point i.e., how scalable the proposed application is considering the current settings. Figure 47 helps us understand the change in performance of the parallel system as the system resources increases. With a sharp increase in the performance as we scale i.e., increase the size of the system, similar to that of the speedup, it saturates after reaching a certain point i.e., from 13 to 15 worker nodes. From this observation we can infer that in order to keep increasing the scalability of our application we need to scale (increase) the problem size as well i.e., increase the complexity of the pipeline by adding more computation steps, along with the machine size.

6.1.4 Overall Results

With the comparison performed in the previous sections, it provides us an overview of the efficient ML model capable of detecting the attacks and categorizing into its respective types. It can be understood that models developed for multiclass mode of the application, can outperform its corresponding benchmarks in certain scenarios.

For example, with feature reduction process, the accuracy of decision tree varies significantly even though it is tuned with the similar tools. GBT only supports binary classification and that does not meet our requirements of being able to categorize the attacks, hence, it is only considered for benchmarking purposes. MLP performs poorly with the current settings and is not suitable for categorizing the attack types. Random Forest on the other hand clearly outperforms all other models in terms of consistency, ability to support both binary classification and multiclass classification problems. Also, it performs efficiently when tuned with the correct set of parameters with the CrossValidator tool.

Comparing the results with the relevant work, this study provides a comprehensive analysis of the steps and the measure of the metrics to compare various approaches in resulting an efficient model which was lacking in this paper [6]. This paper [6] provided a comparison of a wider perspective machine learning and its possible areas of interest. In paper [39] the author provided extensive comparison of the models and their overall performances; however, the evaluation metrics were lacking in the study to understand how exactly the models behaving during training the dataset. Although it was able to achieve good accuracy it considered relatively smaller dataset and focused only on getting accurate models. It did not cover the aspect of handling large datasets which is more practical as covered in this thesis. In their work [47] the author proposed a framework dealing with Deep Learning and Big Data Technologies for IoT security but lacked experimental and actual implementation of the framework, its working methods and outcome of the framework. It did not provide enough clarity on the evaluation process, tuning and feature analysis steps in the framework proposal and did not mention about the scalability of the framework so that it is easily adopted and deployed in cloud environments which was extensively covered in this thesis. This thesis gave an extensive working model of an application capable of categorizing different types of attacks in a large dataset. Also, it provides comprehensive analysis of the results of the most common algorithms used in machine learning when used in different settings of the application in the ML process.

7 Application and its Possibilities

The following sections describes how to use the application with real network capture from any IoT system. It also covers various possibilities and future aspects of the application in terms of analysing an exposed network with the best trained model without worrying about the size of the dataset.

7.1 Real Life Use case

In a practical scenario, a network of IoT devices generating data is captured in different formats, generally into *.pcap* files, which are then provided for analysis. The application created for this project, although uses a dataset which contains the feature vectors of the features of the IoT network traffic. It can be used directly with the raw *.pcap* files but with some additional steps as described below:

With the collaboration of the external libraries, Yisroel Mirsky et.al. used in their paper [45]. Tools like *Wireshark*, which could capture network traffic in pcap format, and a python library called *scapy* is used as an alternative tool to parse the raw file and convert them into the csv format are used here.

This file is then passed through the *FeatureExtractor* python class which can extract features from raw input data with the help of the parameters already described in the *Dataset Analysis* section. Tshark is required to parse the *.pcap* file and convert them into human readable *.tsv* format.



```
210
211
212
213 def pcap2tsv_with_tshark(self):
214     print('Parsing with tshark...')
215     fields = "-e frame.time_epoch -e frame.len -e eth.src -e eth.dst -e ip.src -e ip.dst -e tcp.srcport -e tcp.dstport" \
216             "-e udp.srcport -e udp.dstport -e icmp.type -e icmp.code -e arp.opcode -e arp.src.hw_mac -e arp.src.proto_ipv4" \
217             "-e arp.dst.hw_mac -e arp.dst.proto_ipv4 -e ipv6.src -e ipv6.dst"
218     cmd = ''' + self._tshark + ' -r ' + self.path + ' -T fields '+ fields + ' -E header=y -E occurrence=f > '+self.path+".tsv"
219     subprocess.call(cmd,shell=True)
220     print("tshark parsing complete. File saved as: "+self.path+".tsv")
221
222 def get_num_features(self):
223     return len(self.nstat.getNetStatHeaders())
```

Figure 48. Function that parses pcap to tsv

The above function uses tshark, a terminal version of Wireshark, if tshark is not installed, then it uses scapy library of python to parse the network capture and output the *.tsv* file with the labels/headers of the *.pcap* file. This file is then fed to the feature extractor respective to the estimator selected while running the application.

```

49
50
51 def get_FE_instance(path):
52     return FE(path)
53
54 ## binary_feature_extractor
55 def convert_write_to_csv(path, fe):
56     vecs = []
57     while True:
58         v = fe.get_next_vector()
59         if (len(v) == 0):
60             break
61         vecs.append(v)
62     with open(path, 'w', newline='') as file:
63         mywriter = csv.writer(file, delimiter=',')
64         mywriter.writerows(vecs)

```

Figure 49. Binary Classifier feature Extractor

The above function shows how the feature extractor class extracts feature vector for binary classification mode. The feature vectors for the multiclass classification mode are also done in a similar fashion for the considered input dataset. The output of these extractor can be then fed to the pipeline, no matter the size, respective to the mode of operation. It can be ran using the *spark-submit.sh* script provided in the project path, which also contains multiple commands to run, provided the necessary dependencies are already installed in the environment.

One of the benefits of using this application is the supported ML algorithms can be toggled by the user while running the application, without the need to modify the code. These algorithms are highly configurable and can be passed as an argument to the runtime script; multiple applications with different ML algorithms can also be run parallely in the same environment, if required, for obtaining quick results.

Upon running the application, several metrics are captured in the log to perform further detailed analysis like calculating the performance metrics in terms of parallel computation as described in the next section. A typical output of the application with all the details captured is provided in the Appendix 5.

7.2 Way forward

This study although compares the most common models in a big data framework taking the advantages of parallel computing to train the machine learning models in the most accurate way, however, there are multiple attack types which were beyond the scope of the paper.

For example, each of the 4 different categories of attacks have one or two attack datasets counting a total of 9 attacks in the original dataset which means the ML models would have to detect 9 types of attacks from the whole dataset. A minor change in code is required to fulfil this requirement i.e.,

- adding the path to the dataset and its corresponding labels to the Paths.py file
- define labels to the extra attacks that would be covered in the multiclass/DataLoader.py file.

These 2 changes are sufficient to analyse the whole large dataset in the same cluster. The existing cluster setup is well configured to handle this large dataset in a parallel manner.

Another possibility of improvement would be to use dynamic cluster resources which will be required on-demand. Currently, the cluster Apache Spark uses is standalone i.e., the executors it is configured with should be pre-configured and should be up and running all the time. Even if the execution has finished and the goal has been met, the cluster would still use the dedicated resources assigned to it. A potential solution to this is to use a Kubernetes cluster to manage Spark workers. Kubernetes (K8s) is an orchestration framework which uses container technology to use minimal required resources and is capable for on-demand availability. The current version of Spark used in this project supports this installation. The advantage of using K8s for managing Spark is it is possible to create many executors when the computation demands and once the job is finished, it can be shut down to save resources. Given the time and knowledge it requires to set-up a proper cluster with additional networking, maintenance etc. the scope diverges with the goal of this thesis.

Yet another area of improvement would be the removal of Kitsune from the process of using this application with real IoT network traffic. As of now, this paper relies on Kitsune to parse the pcap files and convert in to feature vectors whereas Spark has its

own feature extractors which may be directly used in this application to generate the feature vectors from the raw dataset. Apart from this python has inbuilt library to parse the pcap files into comma separated files which is pretty much easier to use.

8 Conclusion

As Internet of Things is widely involved with day-to-day activities of most of the people, there has been a huge threat to the data being used and shared across the network. Many people may be oblivious to the fact that the data they use in IoT devices may be already leaked when the devices are itself compromised and there is always a risk of losing sensitive data stored in these devices, if mishandled, for example, in case of spoofing attacks, the attacker may constantly eavesdrop into the network packets being shard and try to gain information of the user and the devices being used.

This study showed how we can detect multiple attacks in a IoT network traffic by developing an application capable of handling large amount of data. The application was designed to be scalable and modular and focused on the reusability of the application by generating tuned models as the output for later use.

The significance of multiclass classification in terms of categorizing the attack types and its advantages over binary classification problems are discussed in this thesis. This study helped us understand why it is necessary to consider parallel processing, computation time, modularity of the application while dealing with IoT data and its security analysis.

It focused on machine learning techniques to analyse, predict, and categorize the attack types as multiple labels in the input dataset. It provided an extensive comparison of the outcome of the hyperparameters tuning process and discuss the significance of such process in the process. It also provided some insight about the model performances by using efficient feature reduction techniques by reducing the features from 115 to 18, 23 and 25. It measured the performances of the model with multiple evaluation metrics and compared the same with other models run in different mode of the application.

Moreover, the efficiency of the parallel computation is also measured as the application trains, tests the input data and the models under consideration are evaluated, tuned with hyper parameters. It leverages the big data processing techniques, parallel computing, and resource efficiency by adapting cutting-edge tools and discusses about various methods of on-demand resources by adapting containerized technology to run this application in a more efficiently.

References

- [1] Hewlett Packard, "HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack," PALO ALTO, 2014.
- [2] H. HaddadPajouh, A. Dehghantanha, R. M. Parizi, M. Aledhari and H. Karimipour, "A survey on internet of things security: Requirements, challenges, and solutions," *Internet of Things*, p. 100129, 2019.
- [3] C. L. a. B. Palanisamy, "Privacy in Internet of Things: From Principles to Technologies," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 488-505, 2019.
- [4] I. Kotenko, I. Saenko, A. Kushnerevich and A. Branitskiy, "Attack detection in IoT critical infrastructures: a machine learning and big data processing approach," *IEEE*, no. 10.1109/EMPDP.2019.8671571, pp. 340-347, 2019.
- [5] D. E. Denning, "An Intrusion-Detection Model," *EEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 222-232, 1987.
- [6] F. Hussain, R. Hussain, S. A. Hassan and E. Hossain, "Machine Learning in IoT Security: Current Solutions and Future Challenges," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 1686-1721, 2020.
- [7] K. H. O. A. A. S. a. Y. J. J. Wurm, "Security analysis on consumer and industrial IoT devices," *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), Macau*, pp. 519-524, 2016.
- [8] C. Cyrus, "IoT Cyberattacks Escalate in 2021, According to Kaspersky," *IoT World Today*, 17 September 2021. [Online]. Available: <https://www.iotworldtoday.com/2021/09/17/iot-cyberattacks-escalate-in-2021-according-to-kaspersky/>. [Accessed 27 September 2021].
- [9] D. Meharchandani, "Cyber Threats Haunting IoT Devices in 2021," *Security Boulevard*, 21 September 2021. [Online]. Available: <https://securityboulevard.com/2021/09/cyber-threats-haunting-iot-devices-in-2021/>. [Accessed 27 September 2021].
- [10] K. N. Junejo and J. Goh, "Behaviour-Based Attack Detection and Classification in Cyber Physical Systems Using Machine Learning," in *ACM Symposium on Information, Computer and Communications Security*, 2016.
- [11] K. Ashton, "That internet of things' thing," vol. 22, no. 7, pp. 97-114, 2009.
- [12] D. Evans, "The internet of things – how the next evolution of the internet is chaging everything," *White Paper. Cisco Internet Business Solutions Group (IBSG)*, vol. 1, pp. 1-11,

2011.

- [13] A. Selamat and Z. Iqal, "Open Challenges in Internet of Things Security," *Journal of Physics: Conference Series*, vol. 1447, 2019.
- [14] Ibm Cloud Education, "Machine Learning," Ibm Cloud Education, 15 July 2020. [Online]. Available: "<https://www.ibm.com/in-en/cloud/learn/machine-learning>". [Accessed 13 May 2021].
- [15] S. V, N. S and P. M, "AN EMPIRICAL SCIENCE RESEARCH ON BIOINFORMATICS IN MACHINE LEARNING," *Journal of Mechanics of Continua and Mathematical Sciences*, no. Special Issue No. 7, 2020.
- [16] Expert.ai Team, "What is Machine Learning? A Definition.," Expert.ai, 06 May 2020. [Online]. Available: <https://www.expert.ai/blog/machine-learning-definition/>. [Accessed 13 May 2021].
- [17] Computer Science and Engineering, IIT Kanpur, India, "Binary Classification," [Online]. Available: https://www.cse.iitk.ac.in/users/se367/10/presentation_local/Binary%20Classification.html. [Accessed 13 May 2021].
- [18] J. Brownlee, "4 Types of Classification Tasks in Machine Learning," *Machine Learning Mastery*, 8 April 2020. [Online]. Available: <https://machinelearningmastery.com/types-of-classification-in-machine-learning/>. [Accessed 13 May 2013].
- [19] A. Spark, "Classification and Regression," Apache Spark, [Online]. Available: <https://spark.apache.org/docs/latest/ml-classification-regression.html#logistic-regression>. [Accessed 2021].
- [20] C. & S. S. L. Kingsford, "What are decision trees?. *Nature biotechnology*," vol. 26(9), pp. 1011-1013, 2008.
- [21] D. J. C. M. Kay, "Introduction to Information Theory," in *Information Theory, Inference and Learning Algorithms*, Cambridge University Press, 2004, p. 630.
- [22] N. Donges, "A COMPLETE GUIDE TO THE RANDOM FOREST ALGORITHM," *builtin*, 3 September 2020. [Online]. Available: <https://builtin.com/data-science/random-forest-algorithm>. [Accessed 13 May 2021].
- [23] W. Koehrsen, "An Implementation and Explanation of the Random Forest in Python," *towards data science*, 30 August 2018. [Online]. Available: <https://towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76>. [Accessed 13 May 2021].
- [24] D. Berrar, "Bayes' theorem and naive Bayes classifier," *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics; Elsevier Science Publisher: Amsterdam, The Netherlands*, pp. 403-412, 2018.
- [25] A. Spark, "Classification and Regression," Apache, [Online]. Available: <https://spark.apache.org/docs/latest/ml-classification-regression.html#naive-bayes>. [Accessed 2021].
- [26] I. a. o. Rish, "An empirical study of the naive Bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, 2001, pp. 41-46.
- [27] P. C. S. Abirami, *Advance in Computers*, Elsevier Ltd. All, 2020.
- [28] Educative, Inc, "What is a multi-layered perceptron?," Educative, Inc, [Online]. Available: <https://www.educative.io/edpresso/what-is-a-multi-layered-perceptron>. [Accessed 13 May 2021].
- [29] Great learning Team, "Types of Neural Networks and Definition of Neural Network," Great learning Team, 29 April 2020. [Online]. Available: <https://www.mygreatlearning.com/blog/types-of-neural-networks/>. [Accessed 13 May 2021].
- [30] J. Czakon, "24 Evaluation Metrics for Binary Classification (And When to Use Them)," 7 May 2021. [Online]. Available: <https://neptune.ai/blog/evaluation-metrics-binary>

- classification. [Accessed 13 May 2021].
- [31] D. Padua, in *Encyclopedia of parallel computing*, Springer Science & Business Media, 2011, p. 5.
- [32] S. Sahni and V. Thanvantri, "Parallel Computing: Performance Metrics and Models," *Research Gate*, 2002.
- [33] D. Nussbaum and A. Agarwal, "Scalability of parallel machines," *Communications of the ACM*, vol. 34, no. 3, pp. 57-61, 1991.
- [34] Apache, "Apache Spark," Apache Spark, 2009. [Online]. Available: <https://spark.apache.org/>. [Accessed 13 May 2021].
- [35] Databricks, "Apache Spark™," Databricks, 2009. [Online]. Available: <https://databricks.com/spark/about>. [Accessed 12 May 2021].
- [36] A. B. Chaudhri, "Apache Ignite vs Apache Spark: Integration using Ignite RDDs," GridGain, 7 August 2018. [Online]. Available: <https://www.gridgain.com/resources/blog/apacher-ignitetm-and-apacher-sparktm-integration-using-ignite-rdds>.
- [37] I. Pointer, "What is Apache Spark? The big data platform that crushed Hadoop," InfoWorld, 16 March 2020. [Online]. Available: <https://www.infoworld.com/article/3236869/what-is-apache-spark-the-big-data-platform-that-crushed-hadoop.html>. [Accessed 13 May 2021].
- [38] Databricks, "Machine Learning Library (MLlib) Guide," Databricks, [Online]. Available: <https://spark.apache.org/docs/latest/ml-clustering.html>.
- [39] M. Hasan, M. M. Islam, M. I. I. Zarif and M. Hashem, "Attack and anomaly detection in IoT sensors in IoT sites using machine learning approaches," *Internet of Things*, vol. 7, p. 100059, 2019.
- [40] M.-O. a. A. F.-X. Pahl, "All eyes on you: Distributed Multi-Dimensional IoT microservice anomaly detection," in *2018 14th International Conference on Network and Service Management (CNSM)*, IEEE, 2018, pp. 72-80.
- [41] A. A. Diro and N. Chilamkurti, "Distributed attack detection scheme using deep learning approach for Internet of Things," *Future Generation Computer Systems*, vol. 82, pp. 761-768, 2018.
- [42] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol and L. Bottou, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion.," *Journal of machine learning research*, vol. 11, no. 12, 2010.
- [43] S. a. B. H. N{\~o}mm, "Unsupervised anomaly based botnet detection in IoT networks," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2018, pp. 1048-1053.
- [44] Slay, N. Moustafa and J., "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," *2015 Military Communications and Information Systems Conference (MilCIS)*, no. 10.1109/MilCIS.2015.7348942., pp. 1-6, 2015.
- [45] Y. Mirsky, T. Doitshman, Y. Elovici and A. Shabtai, "Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection," *Network and Distributed System Security Symposium*, 2018.
- [46] A. Branitskiy, K. I. and S. I. B., "Applying Machine Learning and Parallel Data Processing for Attack Detection in IoT," *IEEE Transactions on Emerging Topics in Computing*, no. 10.1109/TETC.2020.3006351, pp. 1-1, 2020.
- [47] M. Amanullah, R. A. Habeeb, F. Nasaruddin, A. Gani, E. Ahmed, A. Nainar, N. Akim and M. IMRAN, "Deep learning and big data technologies for IoT security," *Science Direct*, pp. 495-517, 2020.
- [48] N. T. A. D. b. o. A. Spark, "Pwint P. H; Shwe, T," *IEEE*, no. 10.1109/AITC.2019.8920897, pp. 222-226, 2019.
- [49] A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi and S. Nomm, "MedBioT: Generation

- of an IoT Botnet Dataset in a Medium-sized IoT Network.,” *ICISSP*, pp. 207-218, 2020.
- [50] A. Alsaedi, N. Moustafa, Z. Tari, A. Mahmood and A. Anwar, “TON_IoT Telemetry Dataset: A New Generation Dataset of IoT and IIoT for Data-Driven Intrusion Detection Systems,” *IEEE Access*, 2020.
- [51] J. Deng, Y. Deng and K. H. Cheong, “Combining conflicting evidence based on Pearson correlation coefficient and weighted graph,” *International Journal of Intelligent Systems*, vol. 36, no. 12, pp. 7443-7460, 2021.
- [52] J. Brownlee, “How to Calculate Correlation Between Variables in Python,” *Machine Learning Mastery*, 27 April 2018. [Online]. Available: <https://machinelearningmastery.com/how-to-use-correlation-to-understand-the-relationship-between-variables/>.
- [53] J. Brownlee, “Introduction to Dimensionality Reduction for Machine Learning,” *Machine learning Mastery*, 06 May 2020. [Online]. Available: <https://machinelearningmastery.com/dimensionality-reduction-for-machine-learning/>.
- [54] Y. Filiberto, R. Bello, Y. Mota and G. Ramos, “Improving the MLP Learning by Using a Method to Calculate the Initial Weights of the Network Based on the Quality of Similarity Measure,” *Research Gate*, no. 10.1007/978-3-642-25330-0_31, pp. 351-362, 2011.
- [55] K. Halawa, “A method to improve the performance of multilayer perceptron by utilizing various activation functions in the last hidden layer and the least squares method,” *Springer*, vol. 34, no. DOI 10.1007/s11063-011-9199-4, p. 293–303, 2011.

Appendix 1

- The project and its findings are hosted in GitHub and can be found in the following link:

00anupam00/comparative-analysis: This repo contains the source for Thesis project. An anomaly detection framework aiming to detect outliers in IoT data streams/network capture. (github.com)

- The feature correlation graphs for all columns with respect to _col03 is present here:

comparative-analysis/results/feature_corr_col3 at main ·
00anupam00/comparative-analysis (github.com)

- The feature distributions of the columns can be found in this link:

comparative-analysis/results/feature_dist at main · 00anupam00/comparative-analysis (github.com)

- The online notebooks where various experiments were conducted in a smaller setup are hosted here:

comparative-analysis/notebooks at main · 00anupam00/comparative-analysis (github.com)

Appendix 2

All versions of the notebooks are hosted in Kaggle and can be found here:

<https://www.kaggle.com/anupamrakshit/outlier-detection?scriptVersionId=62277441>

Appendix 3

Spark environment variables set in the `~/.bashrc` script

```
export SPARK_HOME="/ws/spark/spark-3.1.1-bin-hadoop2.7"
export PATH="/ws/spark/spark-3.1.1-bin-hadoop2.7/bin:/ws/spark/spark-3.1.1-bin-hadoop2.7/sbin:$PATH"
#export PYSPARK_PYTHON="/home/ubuntu/ws/outlier-detection/py-venv/bin/python3"
export PYSPARK_PYTHON="/usr/bin/python3"
export PYTHONPATH="/usr/bin/python3"
#export PYTHONPATH="/home/ubuntu/ws/outlier-detection/py-venv/bin/python3"

#alias mkctl="microk8s kubectl"
alias run='sh ws/outlier-detection/spark-submit.sh > ws/logs/spark-submit_`date +%Y-%m-%d.%H:%M:%S`.txt'
ubuntu@rakshit-thesis:~$
```

Appendix 4

VM is created in OpenStack cloud environment with the following details:

Project / Compute / Instances / rakshit_thesisv2

rakshit_thesisv2

Overview Interfaces Log Console Action Log

Name	rakshit_thesisv2
ID	832d3ffd-ec97-4851-81fd-c418d0c05406
Description	-
Project ID	8fba445a3a6c-4495a5b0d8b22ab6e570
Status	Active
Locked	False
Availability Zone	nova
Created	19 Jun 2021, 5:29 a.m.
Age	6 months, 1 week

Specs

Flavour Name	m2.medium
Flavour ID	1be51d8f-ec8c-43b7-af7b-5c676e763cdd
RAM	64GB
VCPUs	16 VCPU
Disk	20GB

IP Addresses

provider_64_net	172.17.66.135
------------------------	---------------

Security Groups

Security Groups

default

- ALLOW IPv6 from default
- ALLOW IPv4 3306/tcp from 0.0.0.0/0
- ALLOW IPv4 80/tcp from 0.0.0.0/0
- ALLOW IPv4 icmp from 0.0.0.0/0
- ALLOW IPv4 1883/tcp from 0.0.0.0/0
- ALLOW IPv4 8080-8200/tcp from 0.0.0.0/0
- ALLOW IPv4 to 0.0.0.0/0
- ALLOW IPv4 8043/tcp from 0.0.0.0/0
- ALLOW IPv4 from default
- ALLOW IPv6 to :*/0
- ALLOW IPv4 4040/tcp from 0.0.0.0/0
- ALLOW IPv4 22/tcp from 0.0.0.0/0

Metadata

Key Name	rakshit_pc
Image Name	ubuntu18.04
Image ID	21cd7486-a2a6-458c-9b05-9207c0b63819
sw_runtime_python_ve...	3.7

Volumes Attached

Attached To	bb7c36a6-f9fa-4c79-af90-be44c25ab341 on /dev/sda
Attached To	rakshit_thesis_100 on /dev/sdb

Appendix 5

Sample output of the application that logs various metrics and parameters is shown below:

```

output-Dec-13-2021-12/00/16.txt — Edited
Importing Scapy Library
Application started at : 2021-12-13 12:00:38.842461
Running application with the following arguments:
Estimator: gbt
Mode: binary
Pcap File: False
Selected Estimator is: gbt
Binary Classification...
Loaded dataset: /home/ubuntu/comparative-analysis/input/arp/dataset.csv
Loaded labels.
Training model ...
Training complete. The base model is saved in 'models/binary/*'.
Total time required for training: 0:03:17.258441
Tuning binary pipeline...
Tuning with train validation split...
Tuning with cross validation...
The best models are saved in 'models/binary/*'.

Evaluating estimator: gbt
Evaluation results with default params:
areaUnderROC: 0.9984762371420678
Accuracy: 0.9984762371420678
Test Error = 0.00152376
+-----+
|prediction_label|0   |1   |
+-----+
|1.0             |1286|686561|
|0.0             |814117|1011 |
+-----+

Metrics from sklearn:
Accuracy: 99.85000000000001%
Confusion Matrix:
True Negative: 814117
False Positive: 1286
False Negative: 1011
True Positive: 686561
Recall : 0.9985296085355425
Precision: 0.99813039818448
F_score: 0.9983299634511374

Evaluation results for hyper params tuned with cross validation:
areaUnderROC: 0.9975123567102879
Accuracy: 0.9975123567102879
Test Error = 0.00248764
+-----+
|prediction_label|0   |1   |
+-----+
|1.0             |448 |114223|
|0.0             |135786|193 |
+-----+

Metrics from sklearn:
Accuracy: 99.74%
Confusion Matrix:
True Negative: 135786
False Positive: 448
False Negative: 193

```

```
output-Dec-13-2021-12/00/16.txt — Edited

Metrics from sklearn:
Accuracy: 99.85000000000001%
Confusion Matrix:
True Negative: 814117
False Positive: 1286
False Negative: 1011
True Positive: 686561
Recall : 0.9985296085355425
Precision: 0.99813039818448
F_score: 0.9983299634511374

Evaluation results for hyper params tuned with cross validation:
areaUnderROC: 0.9975123567102879
Accuracy: 0.9975123567102879
Test Error = 0.00248764

+-----+
|prediction_label|0      |1      |
+-----+
|1.0             |448    |114223|
|0.0             |135786 |193    |
+-----+

Metrics from sklearn:
Accuracy: 99.74%
Confusion Matrix:
True Negative: 135786
False Positive: 448
False Negative: 193
True Positive: 114223
Recall : 0.9983131729827996
Precision: 0.9960931708976114
F_score: 0.9972019363822479

Evaluation results for hyper params tuned with train validation split:
areaUnderROC: 0.9977677524746442
Accuracy: 0.9977677524746442
Test Error = 0.00223225

+-----+
|prediction_label|0      |1      |
+-----+
|1.0             |201    |114074|
|0.0             |136033 |342    |
+-----+

Metrics from sklearn:
Accuracy: 99.78%
Confusion Matrix:
True Negative: 136033
False Positive: 201
False Negative: 342
True Positive: 114074
Recall : 0.9970109075653755
Precision: 0.9982410851017283
F_score: 0.9976256170990551
Application finished at : 2021-12-13 13:18:21.991336
Total time taken to process: 1:17:43.148875
```


Appendix 6

High resolution images describing the architectures can be found here:

https://github.com/00anupam00/comparative-analysis/blob/b1f43fff75e879e46643c64e6371fa83bfbd0083/Application_High_level.jpg

https://github.com/00anupam00/comparative-analysis/blob/b1f43fff75e879e46643c64e6371fa83bfbd0083/Sequence_diagram.jpg

https://github.com/00anupam00/comparative-analysis/blob/b1f43fff75e879e46643c64e6371fa83bfbd0083/ML_Pipeline.jpg

Non-exclusive License for Publication and Reproduction of Graduation Thesis

I, Anupam Rakshit (author's name) (date of birth:
.....12.04.1991.....)

1. grant Tallinn University of Technology free license (non-exclusive license) for my thesis

PRAGMATIC COMPARISON OF MACHINE LEARNING MODELS TO DETECT THE TYPE OF ATTACKS IN AN IOT NETWORK TRAFFIC

(title of the graduation thesis)

supervised by Pelle Jakovits,

(supervisor's name)
to be

1.1. reproduced for the purposes of preservation and electronic publication, incl. to be entered in the digital collection of TalTech library until expiry of the term of copyright.

1.2. published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of TalTech library until expiry of the term of copyright.

2. I am aware that the author also retains the rights specified in clause 1.

3. I confirm that granting the non-exclusive license does not infringe third persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

Anupam Rakshit *(signature)*

13.12.2021 *(date)*