



Final Report
CM3203 – One Semester Individual Project (40 credits)

Title:
**Machine Learning Model to Identify Attacks on IoT
Devices**

Author: Shaikha Alshehhi

Supervised by: Amir Javed

Moderated by: Hiroyuki Kido

School of Computer Science and Informatics

Cardiff University

**A Final Year Project Submitted for the Degree of
BSc Computer Science with Security and Forensics**

May 2022

Abstract

The Internet of Things is a relatively new trend in the Internet domain. It is defined as a network of physically and digitally connected devices and sensors that generate and exchange massive amounts of data without the need for human intervention. Since human operators are no longer required, the IoT (Internet of Things) can process more data faster and more efficiently than ever before. The IoT paradigm has emerged due to the advancements in hardware, software, communication, and embedded computing technologies, as well as their dropping costs and improving performance. These devices must, however, be network-connected which makes them vulnerable to outside attacks. In recent years, there has been a lot of focus on the security of these IoT devices. Another significant recent trend is the massive amount of data generated by these devices every day, which has revived interest in technologies such as machine learning and artificial intelligence. This study investigates the applicability of machine learning techniques in enhancing the security of IoT devices, precisely the potential of integrating various machine learning models to construct a malware detection system for IoT devices.

Acknowledgements

I would like to take this opportunity to express my appreciation to the people who have supported me throughout this project. Firstly, I would like to express my special gratitude and thanks to my supervisor, Dr Amir Javed, for his continuous support and supervision while working on this project. His guidance and knowledge kept me in the right direction and made this an enriching and exciting project and enabled me to complete this research project to the best of my abilities.

I would also like to thank my parents, Ali Alshehhi and Ibtisam Alshehhi for their endless help and support, and for believing in my abilities to let me pursue my dream.

I will always be thankful for my little sister and brothers, Maryem, Abdullah and Mohammed, whose love and support have always kept me going.

Special thank you to my beloved sister, Fatma Alshehhi, and my best friend, Zainab Almusawi, for their constant support, and for all the precious memories we shared throughout this unforgettable journey.

Finally, I would like to thank everyone who has supported me and contributed to my happiness during this journey.

TABLE OF CONTENTS

ABSTRACT.....	II
ACKNOWLEDGEMENTS	III
LIST OF FIGURES	VI
LIST OF TABLES	X
CHAPTER 1 INTRODUCTION.....	1
1.1 Preface	1
1.2 Project Aim and Scope	2
1.3 Intended Audience.....	3
1.4 Report Structure	3
CHAPTER 2 BACKGROUND.....	5
2.1 Introduction to Machine learning	5
2.2 Related Work.....	7
2.3 IoT-23 Dataset	10
2.4 Machine Learning Approach to Malware Detection	13
2.5 Learning Evaluation	19
2.6 Python Libraries and Frameworks	24
CHAPTER 3 APPROACH	27
3.1 Description of the System	27
3.2 System Plan	28
3.3 Data Flow of the system.....	29
3.4 Development Methodology	33
CHAPTER 4 IMPLEMENTATION.....	36

4.1	Project structure.....	36
4.2	Data preparation and pre-processing	36
4.3	Autoencoder implementation	41
4.4	Train-Test Split.....	41
4.5	Clustering with K-means.....	45
4.6	Random Forest classifier	49
4.7	Dashboard implementation.....	51
4.8	Summary of the implementation process.	57
CHAPTER 5 RESULTS AND EVALUATION		59
5.1	Exploratory Data Exploration Results.....	59
5.2	Anomaly detection with Autoencoder.....	62
5.3	Malware Clustering with K-means.....	73
5.4	Malware Classification with Random Forest	78
5.5	Summary of the results.....	84
5.6	Limitations.....	85
CHAPTER 6 FUTURE WORK		86
6.1	Anomaly detection with an Autoencoder	86
6.2	Malware Clustering and classification	86
6.3	Dashboard.....	87
CHAPTER 7 CONCLUSION.....		88
CHAPTER 8 REFLECTION ON LEARNING		89
CHAPTER 9 REFERENCES		91

LIST OF FIGURES

Figure 2.1. The Architecture of an Autoencoder	13
Figure 2.2. Formula to Calculate Mean Squared Error	14
Figure 2.3. Formula to calculate Within Cluster Sum of Squares (WCSS)	16
Figure 2.4. Illustration of an Elbow point	17
Figure 2.5. The Equation used to calculate the Euclidean Distance.....	17
Figure 2.6. Voting Mechanism used by Random Forest.....	19
Figure 2.7. Confusion Matrix for binary classifier	20
Figure 2.8. AUC ROC Curve	22
Figure 3.1. Simple Data Flow Diagram.....	28
Figure 3.2. Detailed Data Flow Diagram	29
Figure 3.3. Agile Development Methodology	33
Figure 4.1. Description of the data	38
Figure 4.2. Data encoding	39
Figure 4.3. Label distribution	39
Figure 4.4. Dummy encoding.....	40
Figure 4.5. Dropping columns.....	40
Figure 4.6. Converting continues variables to categories.....	41
Figure 4.7. train-test split	42
Figure 4.8. Model Architecture	42
Figure 4.9. Model training.....	43
Figure 4.10. Reconstruction error	44
Figure 4.11. Autoencoder evaluation	45

Figure 4.12. Malware Distribution.....	46
Figure 4.13. Features and labels initialisation	47
Figure 4.14. plot of Elbow method	47
Figure 4.15. Define and train K-means	48
Figure 4.16. Plot clusters.....	48
Figure 4.17. K-means evaluation.....	49
Figure 4.18. train-test split for random forest.....	50
Figure 4.19. random forest model	50
Figure 4.20. confusion matrix.....	51
Figure 4.21 metrics computation.....	51
Figure 4.22 requirements.....	52
Figure 4.23 Procfile.....	52
Figure 4.24 server variable	53
Figure 4.25 app layout.....	53
Figure 4.26 Autoencoder - dashboard.....	54
Figure 4.27 Autoencoder training - dashboard.....	55
Figure 4.28 plot display mode	56
Figure 4.29 read csv files.....	56
Figure 4.30 plot cluster graph.....	57
Figure 4.31 plot classification results	57
Figure 5.1 Traffic distribution	60
Figure 5.2 malware distribution	60
Figure 5.3 histograms for all the numerical columns for benign traffic	61

Figure 5.4 histograms for all the numerical columns for malicious traffic.....	61
Figure 5.5 Model Loss	62
Figure 5.6 Model Accuracy	63
Figure 5.7 Precision over different threshold values	65
Figure 5.8 Recall over different threshold values	66
Figure 5.9 confusion matrix for a threshold of 0.003	67
Figure 5.10 Accuracy for different threshold values	68
Figure 5.11 Recall for different threshold values	69
Figure 5.12 Precision for different threshold values.....	69
Figure 5.13 F1-Score for different threshold values	70
Figure 5.14 AUC ROC for different threshold values	70
Figure 5.15 confusion matrix for a threshold of 0.000724.....	71
Figure 5.16 Autoencoder tab	72
Figure 5.17 Autoencoder training simulation	73
Figure 5.18 Elbow method.....	74
Figure 5.19 Clustering result.....	74
Figure 5.20 Malware distribution for k=2	76
Figure 5.21 Malware distribution for k=3	76
Figure 5.22 Malware distribution for k=5	77
Figure 5.23 Precious of random forest.....	80
Figure 5.24 Confusion matrix for random forest (left-train/right-test).....	81
Figure 5.25 Recall of random forest.....	82
Figure 5.26 F1-Score of random forest	82

Figure 5.27 classification and clustering tab	83
---	----

LIST OF TABLES

Table 2.1. Zeek conn.log features	11
Table 2.2. Detailed labels of the malicious traffic	12
Table 5.1. Reconstruction error for malicious samples – test set.....	64
Table 5.2. Reconstruction error for benign samples – test set	64
Table 5.3. Reconstruction error for benign samples – train set	64
Table 5.4. Autoencoder confusion matrix	65
Table 5.5. performance metrics for a threshold of 0.003	67
Table 5.6. performance metrics for a threshold of 0.000724	71
Table 5.7 summary of performance for different number of clusters	78
Table 5.8 summary of classifier performance.....	83

CHAPTER 1

Introduction

1.1 Preface

The real and digital worlds have rapidly converged in the previous two decades and the Internet was critical in bridging the two worlds. As embedded computers become smaller, more powerful, and more economical, they are being deployed in a wide range of devices and systems, making them Internet accessible and reachable from any device, at any time, from anywhere. These technical advancements have cleared the path for the Internet of Things (IoT) to arise. IoT devices have been deployed in a variety of fields such as health care, smart homes, manufacturing, and transportation [3]. The technology allows for a high level of automation and information sharing, which improves usability and functionality [5]. As the demand for and use of IoT is quickly growing, experts predict that there will be more than 22 billion linked IoT devices by 2025 [43]. The massive scale of IoT networks introduces new issues such as massive amounts of data, storage, security and privacy. IoT devices are becoming increasingly prevalent, extending the Cyber world into the physical world, resulting in new and more complicated security risks and concerns [48].

As the number of internet-connected devices expands and the general lack of security protection for such devices, IoT devices become increasingly vulnerable to malware attacks, giving attackers more opportunities to collect data or remotely exploit them. According to Kaspersky, there were 1.51 billion breaches of Internet of Things (IoT) devices from January to June 2021, up from 639 million in 2020 [34]. The fact that IoT employs technologies such as Software-Defined Networking (SDN), Cloud Computing (CC), and fog computing broadens the

threat picture for attackers [29]. As a result, in recent years, researchers have begun to investigate more complex security solutions to address this issue. Currently, IoT and Machine Learning-based approaches are applied in every aspect of human life [26]. Previous research has demonstrated that utilising machine learning in static and dynamic malware analysis methodologies could improve malware detection [14]. The rise of smart attackers, fuelled by the spread of machine learning tools and the rise of big data, has sparked interest in investigating the potential of machine learning technology to improve IoT security. Furthermore, since IoT devices generate a massive volume of data, typical data collection, storage, and processing solutions may not be effective [29]. Consequently, new methods must be utilized to maximise the value of IoT-generated data. In this perspective, machine learning is recognised as one of the most relevant computing paradigms for providing embedded intelligence in IoT devices. With the improvement and use of machine learning algorithms in various fields, including computer science, they have become increasingly popular in identifying malicious network traffic with better degrees of Accuracy and flexibility in a variety of situations and environments [26],[53]. As a result, due to their ability to keep up with malware growth, the development and deployment of machine learning algorithms for malware detection would give a high level of security value.

The report covers the process of building three Machine Learning algorithms that could be utilised in a malware detection system for IoT devices and testing them using the IoT-23 dataset [47], which is a novel dataset that contains both malicious and benign network captures from a variety of IoT devices.

1.2 Project Aim and Scope

The main aims of this project are to establish knowledge in machine learning methodologies and malware detection novelty techniques that will be used to complete this project,

followed by detecting ongoing cyberattacks on IoT devices and further categories them based on the type of attack from a given data set by utilising supervised and unsupervised machine learning algorithms and anomaly detection techniques. Another goal is to implement a simple dashboard that simulates the malware detection system. Finally, presenting the findings by offering a discussion and evaluation of the methods utilised.

The scope of this project focuses on investigating whether a combination of machine learning techniques can be utilised to build a malware detection system that does not only detect ongoing cyberattacks on IoT devices but further classify them based on the type of attack. The algorithms and techniques that are implemented and evaluated in this project are: Neural Networks to detect malicious traffic and K-means and Random Forest to group and classify malicious traffic based on malware type.

1.3 Intended Audience

This project is intended and can be beneficial for individuals conducting or planning to be involved in research in the field of malware detection and security applications involving IoT devices. It also aims to provide cybersecurity specialists and researchers with a good starting point in terms of employing novelty techniques and machine learning methodologies in the case of malware detection systems.

1.4 Report Structure

The report is structured as the following:

Chapter 1 introduces the project, its aims and scope and finally the intended audience.

Chapter 2 illustrates the Background research conducted on Machine Learning and approaches to Malware Detection, Related work and existing solutions, the Learning Evaluation techniques

applied and the used performance metrics and finally, the tools, libraries and frameworks used in this project.

Chapter 3 provides a description of the problem and demonstrates the approach to solving it.

Chapter 4 provides a discussion on the implementation of the project down to the coding stage. It includes the structure of the project, the data preparation process, the model's implementation and evaluation and finally the implementation of the dashboard.

Chapter 5 illustrates the findings and provides a discussion on the results obtained from the implemented models along with the limitations of the approach taken and the project

Chapter 7 discusses the potential future work that could be followed to improve the proposed solution.

Chapter 8 provides a conclusion with a summary of the results and the main findings.

Chapter 9 illustrates the reflection on learning and outcomes gained from working on this project.

CHAPTER 2

Background

This chapter will discuss the concepts that must be understood prior to the implementation stage, such as machine learning algorithms and evaluation methodologies.

2.1 Introduction to Machine learning

Machine learning is considered one of the most rapidly expanding computer science fields. Over the last couple of decades, it has become a common technique in every endeavour that demands information extraction from massive data sets, particularly, when human expertise is not available or cannot be utilised [29]. It is a subfield of artificial intelligence that uses algorithms to synthesise the underlying relationships between data and information and extract significant patterns and behaviours within arrays of data. It is used to solve problems in big data analytics, behavioural pattern detection, and information evolution [4],[42],[50]. Machine learning is described as the “field of study that gives computers the ability to learn without being explicitly programmed” by Arthur Samuel (1959) [4].

Machine learning algorithms use mathematical and statistical techniques to create behaviour models from large data sets where these data sets can be numbers, words, or images. These models can be later used to make predictions based on new input data [29]. Machine learning algorithms are utilised in a wide range of applications in today's technology, and they have an impact on our daily lives. Search engines, anti-spam software, fraud detection, facial detection and voice command recognition are all examples of machine learning-based technologies [50]. Furthermore, it is employed in real smart systems; for example, Google utilises machine learning

to analyse risks to Android endpoints and applications. Similarly, Amazon has developed Macie, a tool that employs machine learning to organise and classify data in its cloud storage service [29]. It is believed that the development of a slew of user-centric innovations will be aided by machine learning [4].

2.1.1 Supervised and Unsupervised Learning

Machine learning has branched into multiple subfields that deal with various sorts of learning tasks. Machine learning algorithms can be categorised into two types: Supervised or Unsupervised based on the underlying mappings between input data and predicted output presented during the learning phase of machine learning [50].

Supervised learning is a process that understands the underlying relationship between observed data (input data) and a target variable (label) in order to predict new unlabelled cases [7]. This learning method starts with training the computer using labelled data to generalise the underlying relationship between feature vectors (input) and supervisory signals (label) [4]. A trained model based on a supervised learning algorithm can anticipate hidden phenomena buried in unfamiliar or unknown data instances. Supervised learning algorithms can be further categorised into Classification or Regression algorithms [21]. In this project, a Random Forest classifier will be utilised.

Meanwhile, unsupervised learning can be defined as the training of a machine using data that has not been labelled [7]. Algorithms in this instance must learn the underlying relationships or features from the given data and then group the samples that have similar traits or characteristics without any prior training [7]. Unsupervised learning can be classified into two categories of algorithms: Clustering and Association [21]. The K-means clustering algorithm will be used in this project.

2.1.2 Deep Learning and Artificial Neural Networks

Deep Learning is a subclass of machine learning that uses a huge dataset to train Deep Artificial Neural Networks (ANNs) [46]. It eliminates some of the data pre-processing that machine learning generally entails. Deep Learning algorithms can ingest and interpret unstructured data such as text and photos, as well as automate feature extraction, which reduces the need for human specialists [30].

Artificial Neural Networks (ANNs) are machine learning algorithms with a human brain-inspired structure. The human brain is made up of billions of neurons that communicate with one another through electrical and chemical signals, allowing people to see, feel, and make decisions [35]. ANNs function by mathematically mimicking the human brain through a combination of data inputs, weights, and bias, and linking many artificial neurons in a multi-layered fashion [30]. These components work together to effectively recognise, classify, and characterise data objects. Although a single-layer neural network could produce approximate predictions, adding hidden layers could assist in refining and optimising the model for Accuracy [30]. The Artificial Neural Network model that will be utilised in this project is an Autoencoder.

2.2 Related Work

2.2.1 Machine learning in the security of IoT

The potentiality of machine learning has recently generated attention in a range of application domains. One area where machine learning technologies have seen an increase in interest is cybersecurity. [60]. For a variety of reasons, machine learning is a promising tool for IoT security. One reason is the massive volume of data generated by IoT networks, which machine learning algorithms require in order to provide intelligence to systems [29]. Another factor is that

IoT devices are vulnerable. The number of susceptible entry points for attackers grows in direct proportion to the number of devices on a network, and as the number of IoT devices grows in general, so does the number of attacks targeting IoT devices [5]. Furthermore, by utilising data retrieved from IoT devices, machine learning approaches will be able to allow IoT systems to make informed and intelligent decisions. As previously stated, machine learning is mostly utilised for security, privacy, attack detection, and malware analysis [29]. Experiments with machine learning-based detection models suggest that they have the potential to protect and secure IoT devices [5]. The upcoming section will further discuss some of the existing solutions and research conducted in the field of malware detection and classification.

2.2.2 Existing solutions

Data mining and machine learning approaches, such as Support Vector Machine, Random Forest, Naïve Bayes and Clustering, have been focused on malware detection in the last two decades. However, the rising diversification of malware and its subtypes has necessitated the development of novel methods for greater efficacy. In 2020, Kim et al [37] built a framework for developing an IoT botnet detection model. Their framework consisted of a botnet dataset, botnet training model, and botnet detection model. They utilised various machine and Deep Learning models, including Naïve Bayes, logistic regression, decision tree, random forest, CNN, and LSTM, in building their model. The botnet detection model included both a binary and a multiclass classification model that can classify 10 types of malicious traffic and benign traffic. Their study concludes that Random Forest and Decision Tree are the best performing machine learning models while Logistic regression scored the lowest metrics. They found that, for Logistic regression, benign traffic yields low precision, Recall, and F1-score indicating that errors occur frequently in benign classification. Whereas, among the various Deep Learning models, CNN showed the best

performance. According to Kim et al, building IoT botnet detection models based on a decision tree, random forest, and CNN would be a good strategy to improve botnet detection performance for a variety of IoT devices. Moreover, Stoian also implemented various machine learning algorithms such as Random Forest, Naïve Bayes and an Artificial Neural Network to classify attacks on IoT devices. The research concludes that the best results were achieved by the Random Forest algorithm, with an Accuracy of 99.5% [53]. In a study similar to Stoian's, Hasan et al found that the Random Forest algorithm performed best, with an Accuracy of 99.4%, followed by an Artificial Neural Network with the same percentage but lower scores on other measures [26]. Moreover, Hegde et al [27] utilised a multi-class decision forest and a multi-class Artificial Neural Network to conduct Big Data analysis on Mirai, Torii, Gagfyt, Kenjiro, Okiru, and several other trojans. They tested the models on a small dataset and gradually larger subsets of a larger dataset, measuring Accuracy, probability of detection, and probability of false alarm. Their research results indicate that the performance of a classifier improves as the size of the dataset, the amount of malicious traffic, and the diversity of malicious activity increases [27].

However, this project takes a different method of solving the problem. It combines two machine learning algorithms with a Deep Learning model. **Random Forest**, a supervised algorithm, and **K-means**, an unsupervised algorithm, as the Machine Learning models. In addition to an **Autoencoder** as the Deep Learning model. Since most of the research in this field has focused on a specific type of machine learning algorithms that can be employed and comparing them, this study will focus on investigating whether these three models can be integrated to build a single malware detection system for IoT devices.

2.3 IoT-23 Dataset

The dataset used in this project is the IoT-23 dataset. The dataset was published in 2020 by Parmisano, Garcia, and Erquiaga with the purpose of providing a large dataset of real and labelled IoT malware infections and IoT benign traffic for researchers to develop machine learning algorithms [47]. The IoT network traffic was captured in the Stratosphere Laboratory, AIC group, FEL, CTU University, Czech Republic and it consists of 23 network captures [47]. These captures are divided into 20 network captures from infected IoT devices (malicious) and 3 captures from normal IoT devices (benign). In each malicious scenario, a specific malware sample was executed in a Raspberry Pi. The network traffic captured for the benign scenarios was obtained by capturing the network traffic of three not infected real IoT devices: a Philips HUE smart LED lamp, an Amazon Echo home intelligent personal assistant, and a Somfy smart door lock. According to the authors “both malicious and benign scenarios run in a controlled network environment with unrestrained internet connection like any other real IoT device” [47].

The dataset includes three files: **PCAPs**, which are the actual network capture files, **README** files that contain information about each of the captures and **conn.log.labeled** files. The conn.log.labeled is a Zeek conn.log file that was obtained by running the Zeek network analyser using the original PCAP file [47]. The conn.log.labeled file has the flows of the capture network connection as a normal Zeek conn.log file with two new columns for the labels. For the purpose of this project, PCAP files were discarded and only the conn.log.labeled files were utilised. The PCAP files are created by the network capture tool Wireshark and can only be viewed with it; hence, utilising them would only add unnecessary complexity to this project. A description of each column in the conn.log.labeled file is summarised in Table 2.1 [13]. Furthermore, the dataset also includes labels to define the relationship between flows connected to the malicious

activity which provides more specific information to network malware researchers and analysts.

Table 2.2 indicates the definitions of detailed labels the authors provided [47].

Table 2.1. Zeek conn.log features

Feature	Type	Description
ts	Time	Timestamp.
uid	String	Unique ID of Connection.
id.orig_h	String	The IP address where the attack happened, either IPv4 or IPv6.
id.orig_p	Integer	The port used by the responder.
id.resp_h	String	The IP address of the device on which the capture happened.
id.resp_p	Integer	The port used for the response from the device where the capture happened.
proto	String	The network protocol used for the data package.
service	string	The application protocol.
duration	Float	Time of last packet seen – time of first packet seen.
orig_bytes	Integer	The amount of data sent to the device.
resp_bytes	Integer	The amount of data sent by the device.
conn_state	String	Connection state.
local_orig	bool	whether the connection originated locally.
local_resp	bool	whether the response originated locally.
missed_bytes	Integer	Number of missing bytes in content gaps.
history	String	Connection state history.
orig_pkts	Integer	Number of packets being sent to the device.
orig_ip_bytes	Integer	Number of bytes being sent to the device.

resp_pkts	Integer	number of packets being sent from the device.
resp_ip_bytes	Integer	Number of bytes being sent from the device.
tunnel_parents	String	The id of the connection.
label	String	The type of capture, benign or malicious.
detailed-label	String	If the capture is malicious, the type of capture.

Table 2.2. Detailed labels of the malicious traffic

Label	Description
Attack	This label indicates that an attack of some sort was launched from the infected device to another host. Attacks are defined as any flow that attempts to exploit a susceptible service by analysing its payload and behaviour such as a telnet login brute force.
Benign	This label indicates that neither suspicious nor malicious activities were found in the connections.
C&C	This label indicates that the infected device was connected to a CC server. This traffic flow is characterized by periodic connections to a malicious domain and/or download of suspicious binaries.
DDoS	This label indicates that the infected device is executing a Distributed Denial of Service attack. This attack is detected by the number of flows directed to the same IP address.
FileDownload	This label indicates that a file is being downloaded to the infected device.
HeartBeat	This label indicates that packets sent on this connection are used to keep a track of the infected device by the C&C server.
Mirai	Connections have characteristics of a Mirai botnet.
Okiru	Connections have characteristics of a Okiru botnet.
PartOfAHorizontalPort Scan	Connections are used to do a horizontal port scan to gather information to perform further attacks. These attacks are Characterized by the pattern in which the connections shared the same port, a similar number of transferred bytes, and numerous different destination IP addresses.
Torri	Connections have characteristics of a Torri botnet

2.4 Machine Learning Approach to Malware Detection

As previously mentioned, this paper will discuss how a combination of a Neural Network, an unsupervised model and a supervised model can be utilised to build a Malware Detection system. The algorithms and approaches utilised in this project will be further discussed in the upcoming sections.

2.4.1 Autoencoder

An Autoencoder is a type of neural network that is trained to attempt to copy its input to its output in an unsupervised manner [23]. It compresses the data into a lower-dimensional code before reconstructing the output from this representation [12]. It has a hidden layer h that describes the code on the inside. The code, also known as the latent-space representation, is a compact compression of the input [12]. An Autoencoder consists of 2 components: encoder and decoder. The encoder is a fully connected feedforward neural network that compresses the data into a latent-space representation and can be represented by the encoding function $h=f(x)$ [23],[12]. Meanwhile, the decoder's purpose is to reconstruct the input from the latent-space representation. The decoder is also considered a neural network, and it can be represented by the decoding function $r=g(h)$ [23]. The Autoencoder can thus be expressed as a whole by the function $g(f(x)) = r$, where r should be as close to the original input x as possible. [28]. Figure 2.1 illustrates the architecture of an Autoencoder [28].

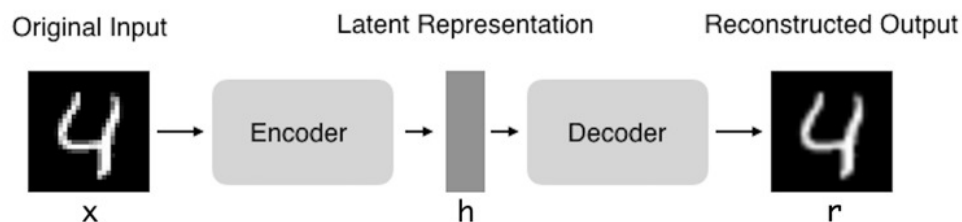


Figure 2.1. The Architecture of an Autoencoder

There are 4 hyperparameters that should be set before training an Autoencoder. The code size, or the number of nodes in the middle layer, is the first hyperparameter. It determines the amount of compression achieved; hence, when the code size is smaller, more compression is obtained. The second hyperparameter is the number of layers that determines the deepness of the model. While a larger depth adds to the model's complexity and yields better compression, a smaller depth is faster to process. Another parameter that should be set before training the model is the number of nodes per layer. As the input to each of these layers becomes smaller across the layers, the number of nodes in the Autoencoder typically decreases with each consecutive layer. The reconstruction loss, also known as the loss function, is the final parameter to specify before beginning the training process. The loss function is heavily dependent on the desired type of input and output. While Mean Squared Error (MSE) and L1 are the most frequently used loss functions for reconstruction, binary cross-entropy can be utilised in cases where the inputs and outputs are within the range of 0 to 1 [12]. For the purpose of this project, Mean Squared Error (MSE) loss function will be used. The mean squared error (MSE) measures the average of the squares of the difference between the true and the predicted values [8]. Figure 2.2 shows the equation used to define the mean squared error (MSE) [8].

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Figure 2.2. Formula to Calculate Mean Squared Error

Although the purpose of an Autoencoder is to attempt to copy the input to the output, for the purpose of this project, it is not the goal. Instead, the aim is to find a structure in the data by learning a compressed version of it and extracting important features. However, an Autoencoder

can theoretically memorise the input data if the hidden layers are greater than or equal to the input layer, or if the hidden units are given adequate capacity. Therefore, this issue can be avoided by utilising an undercomplete Autoencoder where the number of hidden units in the model is limited [23]. In undercomplete Autoencoders, h (hidden layers) is constrained to have a smaller dimension than x (the input) [23]. Consequently, the Autoencoder is forced to capture the most important features of the training data. The learning process can be described as minimizing a loss function $L(x, g(f(x)))$, where L is a loss function that penalises $g(f(x))$ for being dissimilar from x , such as the mean squared error (MSE) [23].

It is worth mentioning that Autoencoders are data-specific, which means they learn features particular to the given training data and can only achieve decent compression results on data that are similar to what they were trained on [12],[28]. They differ from normal data compression algorithms for the same reason, and so cannot be utilised as general-purpose compressors [23].

2.4.2 K-means clustering

Clustering is a valuable tool in data science. It is an unsupervised learning approach that determines cluster structure in a data set based on the highest similarity within a cluster and the highest dissimilarity between clusters [51]. Clustering methods are classified statistically as probability model-based approaches and non-parametric approaches. Clustering methods for non-parametric approaches are typically based on an objective function of similarity or dissimilarity measures [58]. Non-parametric approaches can be divided into hierarchical and partitional methods, with partitional methods being the most popular [58]. The K-means algorithm is the oldest and most widely used partitioning method and has been extensively explored and utilised in several substantive domains [31]. It is the most basic unsupervised learning algorithm for clustering problems. The goal of K-means is to find underlying patterns by grouping similar data

points together. K-means searches a dataset for a fixed number of clusters (k) to achieve this goal. Where a cluster is a collection of data points that have been grouped together due to similarities [22].

The algorithm assigns data points to one of the k clusters iteratively depending on their proximity to the cluster centroid [41]. K-means requires the number of clusters to be specified in advance and since the number of clusters is generally unknown, valid indices can be utilised to find the optimal number of clusters [51]. The Elbow Method using WCSS (Within Cluster Sum of Squares) is considered one of the most popular approaches to finding the optimal number of clusters for the K-means algorithm. Within Cluster Sum of Squares (WCSS) can be defined as the square of the average distance of all the points within a cluster to the centroid of that cluster [15]. WCSS can be computed using the equation in Figure 2.3 where Y_i is centroid for observation X_i [15].

$$WCSS = \sum_{i \in n} (X_i - Y_i)^2$$

Figure 2.3. Formula to calculate Within Cluster Sum of Squares (WCSS)

The Elbow Method clusters the dataset using K-means clustering for a range of k values, then for each value of k, it plots the WCSS values [20]. When plotting WCSS against k, an ‘elbow’ can be seen, indicating a considerable decrease in the rate of increase [15]. Reasonable performance can be achieved when the number of clusters corresponds to the elbow point; yet, this is still judgmental because what is considered an elbow is assessed visually. Figure 2.4 shows a possible illustration of an ‘elbow’ [16].

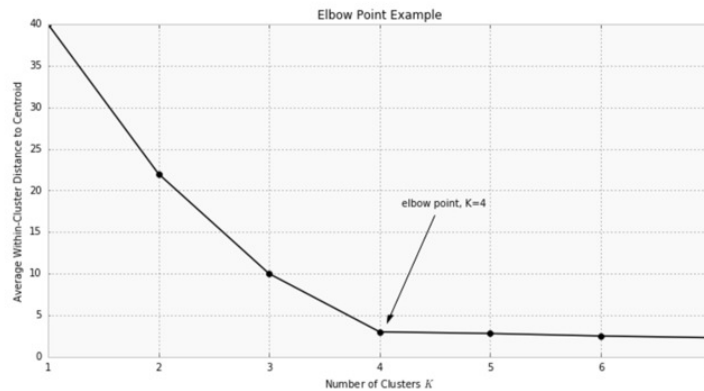


Figure 2.4. Illustration of an Elbow point

After finding the optimal number of clusters (k) and randomly initialising the centroid for each cluster from the data, each data point (x) must be assigned to a cluster. The assignment is determined by the distance between the data point (x) and the cluster centroid. Hence, the Euclidean distance of each point in the dataset with the identified centroids must be calculated. A data point is assigned to a cluster based on the shortest distance between it and the cluster's centroid (smallest Euclidean distance). Euclidean distance can be calculated using the equation in Figure 2.5 [41].

$$d(x, y) = \sqrt{\sum_{i=1}^n (y_i - x_i)^2}$$

Figure 2.5. The Equation used to calculate the Euclidean Distance

The fourth step is to find the new centroid by averaging the points in each cluster group. Finally, repeat the previous three until the centroids of the newly formed clusters no longer change or the data points remain in the same cluster, or the maximum number of iterations is reached. However, K-means can produce arbitrarily terrible clusters in some instances if cluster initialization is done randomly [1]. This is when the K-means++ algorithm will be used. It presents a strategy for initialising the cluster centres before utilising the standard K-means clustering

algorithm, ensuring a smarter initialisation of the centroids, and enhancing the clustering quality [1]. The first step in K-means++ initialisation is to choose the first cluster at random from the data points. This is similar to K-means, only instead of picking all the centroids at random, only one centroid is chosen. The next step is to calculate the Euclidean distance between each data point (x) and the chosen centroid. The third step is to select a new centroid from the data points, with the chance of selecting a point as a centroid being proportional to its distance from the previous centroid. Then steps 2 and 3 are repeated until the chosen number of clusters k is reached [1].

2.4.3 Random Forest Classifier

Random forest is a flexible, user-friendly machine learning method that, in most circumstances, produces great results even without hyper-parameter tuning. Due to its simplicity and versatility, it became one of the most widely used algorithms. The algorithm has the advantage of being able to solve classification and regression issues, which make up the majority of existing machine learning systems. Moreover, it is frequently used for IoT malicious traffic classification [3], [17], [49]. The algorithm is based on ensemble learning which means that it integrates several classifiers to solve a complex problem and increase the model's performance [50].

Random Forest is a supervised learning algorithm made up of a collection of decision trees, collectively referred to as "Forest" [50]. It uses feature randomness and bagging to ensure that, for each tree, the features used in the starting iteration are randomly selected and the dataset is randomly sampled with replacement during training [3]. Feature randomness and bagging add more diversity to the dataset and ensure that decision trees have a low correlation [9]. The determination of the algorithm's prediction varies depending on the type of problem. For this project, the problem is considered a classification task; hence, it takes each tree's prediction and selects the final prediction using a voting mechanism [14]. Figure 2.6 illustrates the mechanism of

voting used by Random Forest [57]. Moreover, one of the drawbacks to Random Forest is the risk of overfitting. However, the greater the number of trees in the forest, the higher the Accuracy and the less chance of overfitting because the overall variance and prediction error are reduced by averaging uncorrelated trees [14].

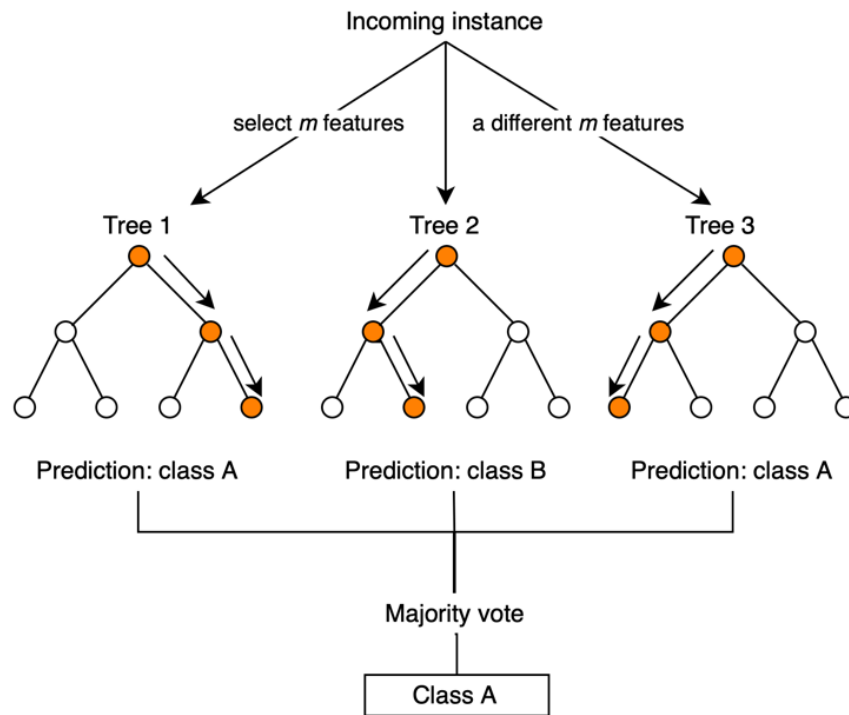


Figure 2.6. Voting Mechanism used by Random Forest

2.5 Learning Evaluation

A model's strength must be tested by putting it through a performance evaluation. Model performance can be evaluated in a variety of ways, but distinct problems necessitate unique choices when selecting performance evaluation metrics. This section will discuss the evaluation methods and performance metrics used to evaluate the models implemented in this project.

2.5.1 Confusion Matrix

A confusion matrix in the context of machine learning is an $N \times N$ table that is used to evaluate the performance of a classification model, where N indicates the number of classes [40]. The matrix compares the true classes to the machine learning model's predictions. This provides a comprehensive picture of the classification model's performance and the types of errors it makes [40]. Figure 2.7 illustrates a confusion matrix for a binary classifier where each cell in the matrix represents an evaluation factor [40].

		True Class	
		Positive	Negative
Predicted Class	Positive	TP	FP
	Negative	FN	TN

Figure 2.7. Confusion Matrix for binary classifier

These evaluation factors can be described as follow:

True Positive (TP): The number of correctly predicted samples in the positive class.

False Positive (FP): The number of incorrectly predicted samples in the positive class.

True Negative (TN): The number of correctly predicted samples in the negative class.

False Negative (FN): The number of incorrectly predicted samples in the negative class.

Since performance indicators are derived from the confusion matrix, excellent indicators will rely on small numbers in the off-diagonal element, ideally zero-diagonal, and large numbers in the diagonal elements [40]. This means a high rate of True Positive and True Negative with a low rate of False Positive and False Negative. Confusion matrices are used to calculate performance metrics such as Accuracy, Precision, Recall, F1 Score and AUC ROC. These performance metrics are further described in the list below:

Accuracy: The number of all correct predictions divided by the total number of predictions [6]. In terms of the confusion matrix, it can be defined as the sum of TP and TN divided by the sum of TP, TN, FP and FN. Accuracy is calculated using the formula:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision: The ratio of True Positives to the total number of predicted positive samples [6]. A Precision score of 1 means the model classified all the positive samples correctly. A low Precision score, on the other hand, implies a high number of False Positives, which might be caused by an imbalanced class or untuned model hyperparameters [6]. Precision is calculated using the formula:

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

Recall: The ratio of True Positives to the total number of positive samples [6]. High Recall indicates high True Positive and low False Negative rates. Recall is calculated using the formula:

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

F1-score: The weighted average of Precision and Recall. It takes into consideration both False Positives and False Negatives. Although it is not as intuitive as Accuracy, F1 is frequently more

useful than Accuracy, especially if the class distribution is unequal [6]. F1-score is calculated using the formula:

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

AUC-ROC curve: The ROC (Receiver Operating Characteristics) curve is a performance measurement for the classification problems across all possible classifications. It is used to measure how well the model can distinguish between classes. The ROC curve plots two parameters, True Positive Rates (TPR) and False Positive Rates (FPR) where TPR correspond to the Recall score of the model and FPR is the ratio of False Positives to the sum of False Positives and True Negatives [6]. Whereas AUC measures the area under the ROC curve. As can be seen in Figure 2.8, the larger the area under the curve (AUC) the higher the rate of True Positive; thus, a better model.

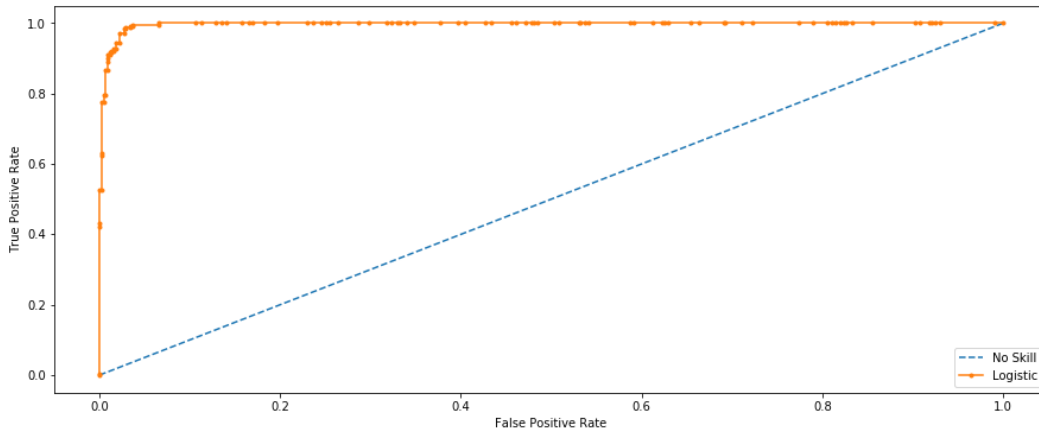


Figure 2.8. AUC ROC Curve

2.5.2 Multiclass Classification

Binary and multiclass are the two types of classification algorithms. Multiclass classification is described as classifying data samples into three or more unique classes, as opposed to binary classification, which is defined as classifying data samples into one of two classes [52]. One of the aims of this project is to classify malicious network traffic into more than two types of

malware; hence, this task is considered a multiclass classification. Accordingly, the evaluation metrics, excluding Accuracy and AUC ROC, must be calculated for each class separately and then calculate the average to measure the overall performance (macro-averaging) [52].

2.5.3 Clustering

In unsupervised learning, data is not labelled which makes the evaluation process complex. Hence, classification performance metrics cannot be employed. Any evaluation metric should not consider the absolute values of the cluster labels, but rather whether this clustering defines data separations that are similar to some ground truth set of classes or satisfy some assumption that members of the same class are more similar than members of different classes according to some similarity metric [59]. Clustering evaluation metrics are categorised into two types: metrics that require the labels of the data and metrics that do not. A combination of both is utilised in this project. All clustering evaluation metrics in this project are outlined below:

Purity: It quantifies the extent to which clusters contain a single class. It is calculated by first allocating each cluster to the most common class in the cluster, and then measuring the Accuracy of this assignment by counting the number of correctly assigned samples and dividing it by the total number of samples [59].

Rand Index and Adjusted Rand Index: The measure of similarity between two clusters by looking at all pairs of samples and calculating how many of them are assigned to the same or different clusters in the predicted and true clusters. Rand Index can range from 0 to 1, where one is the highest achievable score [61]. Whereas Adjusted Rand Index is an adjusted version of the Rand Index meaning that any random clustering is given a score of 0 [61].

Mutual Information: Given the true label assignment and the clustering algorithm assignment, Mutual Information measures the similarity between these two assignments while ignoring permutations [61].

Calinski-Harabaz Index: It is the ratio of the sum of between-clusters dispersion and of within-cluster dispersion, where dispersion is defined as the sum of distances squared [56].

Davies-Bouldin Index: It signifies the average similarity between clusters, where the similarity is a is the ratio of within-cluster distances to between-cluster distances. Unlike other performance metrics, the smaller the Davies-Bouldin Index the better clustering performance which indicates that the clusters are farther apart and less dispersed [56].

2.6 Python Libraries and Frameworks

2.6.1 Python Libraries

This project will be implemented using the Python programming language for a variety of reasons. Python is a high-level programming language that is one of the most often used by data scientists. Its significance in scientific and research domains originates from its simple syntax and ease of use. Importantly, it has a large collection of libraries that deals with data science application and supports the implementation of machine learning projects. These libraries will be utilised in essential areas of the project including the data analysis process, models' construction and evaluation. There are several libraries that are utilised in this project such as Pandas, NumPy, Scikit-learn, Keras, TensorFlow, Matplotlib and Seaborn.

NumPy is a Python open-source package that allows for quick mathematical computations on arrays and matrices [11]. Due to its straightforward syntax and high-performance matrix calculation capabilities, NumPy has become an important library for any scientific computation in Python, including machine learning. Similar to NumPy, Pandas is a data manipulation library for numerical data and time series. It defines three-dimensional and two-dimensional data using data frames and series, respectively. It also has options for indexing massive datasets and searching them quickly. Furthermore, data reshaping, handling missing data and connecting datasets are some of the important features of this library. The most crucial aspect of data analysis and machine learning is data cleaning, processing, and analysis, all of which Pandas supports [11]. Scikit-learn is a Python library that provides a uniform interface for supervised and unsupervised learning algorithms such as K-means and Random Forest that will be implemented in this project [10]. Hence, by offering an abstract level object that can be utilised to apply the algorithms, the Scikit-learn package can help avoid writing so many lines of code and implementing the models from scratch. Keras is another library that will be utilised in the implantation of this project. It is a free open-source Python API for developing Neural Networks [36]. It is built on a simple structure that allows data scientists to design and define Deep Learning models in a clean and efficient manner. Keras is built on top of TensorFlow, a mathematical framework that can also be used to build Neural Networks and Deep Learning models [36]. Finally, both Matplotlib and Seaborn are data visualisation libraries that will be used to visualise data in order to better comprehend it before processing and training it [18].

2.6.2 Dash Framework

Dash is another tool that will be used in this project, precisely for the implementation of the dashboard. It is a python framework, that is created by Plotly, for building interactive web

applications. Dash is built using Flask, Plotly.js, and React.js and it only requires a basic understanding of HTML and CSS [55]. Dash is ideal for delivering data insights and implementing interactive charting with Python, making it ideal for this project.

2.6.3 Jupyter Notebook

Jupyter Notebook is a free and open-source web tool that allows the user to create and share documents that include live code, equations, visualisations, and text. Its user-friendly interface enables users to build and organise workflows in data science, scientific computing and machine learning [33]. Jupyter Notebook is appropriate for use in this project because it allows for all documentation, code execution, output observation, and result visualisation to be done in a single file. Furthermore, because each section is self-contained, a certain block may be tested without needing to execute the code from the start [33].

CHAPTER 3

Approach

The project aims to investigate the ability of Machine Learning to detect network traffic on IoT devices and classify them as malicious or benign and further classify malicious traffic based on the different types of malware. This solution was developed for the purposes of Scientific Research to examine if the proposed strategy is appropriate for addressing the project's problem. Since it is not intended for profit or commercial usage, a user-friendly visual user interface (GUI) was not a priority. Due to the limited time frame, the major goal of this project was to conduct an investigation and provide a good starting point rather than a fully functional and successful software. However, a simple dashboard was developed to simulate what a potential fully functioning system would look like.

3.1 Description of the System

This solution is a Machine Learning approach to identify attacks on IoT devices at a high level. The system starts by taking network traffic data from the bro log file and classifying them as malicious or benign using a Neural Network (Autoencoder). The system then clusters the identified malicious traffic based on the different malware groups using K-means clustering. However, since it is an unsupervised machine learning model, K-means will only group the samples into clusters rather than detecting the types of malware; hence, including a supervised learning algorithm is necessary to identify malware classes. A Random Forest classifier is utilised to achieve this requirement. Figure 3.1 illustrates the data flow within the system in a simplified matter.

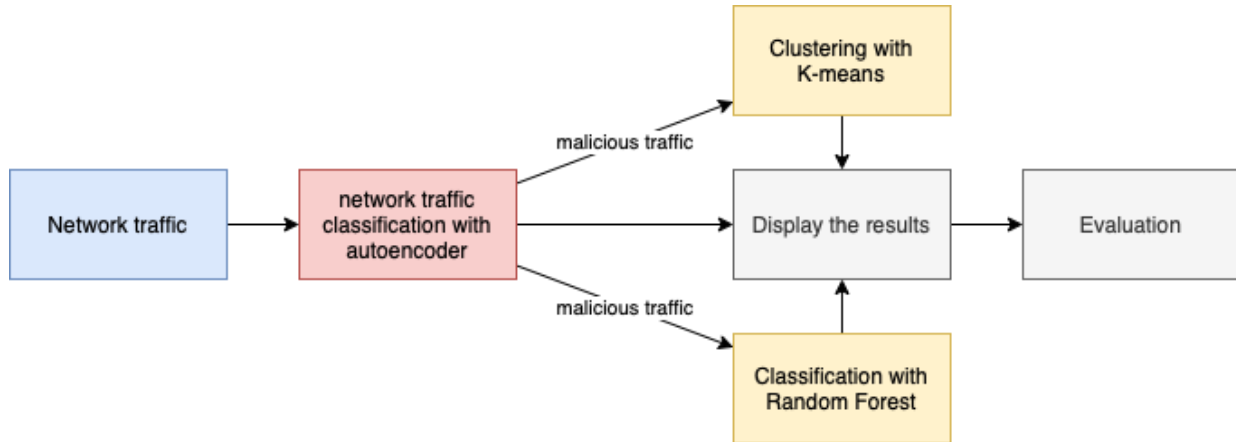


Figure 3.1. Simple Data Flow Diagram

3.2 System Plan

The produced system was implemented based on the IoT-23 data set. The data set contains 540,715 benign and 762,536 malicious network traffic from IoT devices. For the machine learning models, the implementation was conducted using Python Programming Language and the script was written in three Jupyter notebook files:

- EDA.ipynb includes the Exploratory Data Analysis (EDA) of the dataset.
- Autoencoder.ipynb includes the Autoencoder implementation and its evaluation.
- MLmodels.ipynb includes the pre-processing of the dataset, the implementation of the K-means and the Random Forest algorithms, and the evaluation of said models.

The code was split into three sections in order to make smaller and more focused files. It is simpler to navigate these smaller files, as well as to comprehend their contents. Whereas the source code for the Dashboard is stored in separate file called dashboard.

The source code for the entire project can be found in the code file while the data frames are in the dataset folder (df.csv is the pre-processed dataset while OriginalDataFrame.csv is the original concatenated datasets).

3.3 Data Flow of the system

This section will further explain each component of the solution in a detailed matter based on the Data Flow diagram illustrated in Figure 3.2.

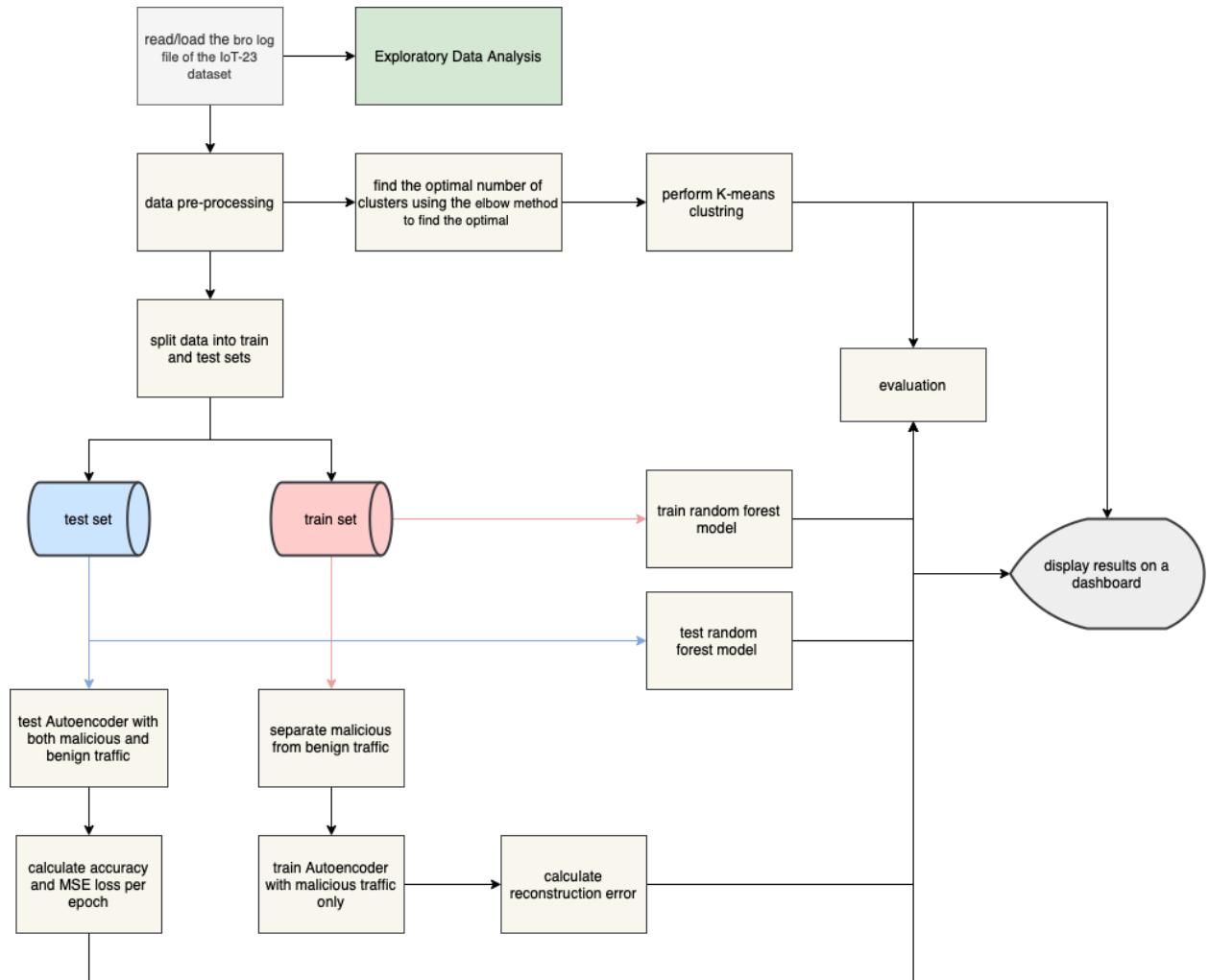


Figure 3.2. Detailed Data Flow Diagram

3.3.1 Exploratory Data Analysis

Conducting Exploratory Data Analysis (EDA) is the initial step before any data processing. This phase is significantly important when it comes to examining and investigating the data collection as well as summarising its primary features. It makes it easier to detect patterns, spot anomalies, and test assumptions, which will aid in determining how to best manage the data set.

Furthermore, it aids in the comprehension of dataset variables and their relationships. Data visualisation tools such as graphs and tables will be used in the EDA.

3.3.2 Data Pre-Processing

Data pre-processing is a crucial stage in Machine Learning because the quality of data and the relevant information that can be extracted from it has a direct impact on the models' capacity to learn; consequently, pre-processing the data before feeding it into the models is critical. The pre-processing phase includes:

1. Handling any null values by dropping rows or columns that contain null values,
2. Normalization of the data by converting continuous variables into categories
3. Handling Categorical Variables by converting them to numerical values using Label Encoder and One-Hot Encoding,
4. And the final step of the data pre-processing phase is to drop any categorical feature that is used for identification purposes.

3.3.3 Anomaly detection using an Autoencoder

The first step in identifying malware attacks on IoT devices is to classify network traffic as malicious or benign. As mentioned above, this will be achieved using a Neural Network, an Autoencoder to be precise. The model will be implemented with 4 fully connected layers with 12, 6, 6 and 20 neurons respectively. The model will be trained, using only benign network traffic from the pre-processed dataset, with a batch size of 32 for 100 epochs.

Since Autoencoders can only reconstruct samples similar to what they were trained on, it is logical to expect that malicious traffic will yield higher reconstruction errors than benign traffic. Hence, a threshold, based on the reconstruction error for benign traffic, must be set in a way that

it can distinguish between benign and malicious traffic. The rationale for this is that any sample that is given into the model and results in a reconstruction error higher than the predefined threshold can be identified as malicious traffic. The model will be tested over different threshold values using both malicious and benign traffic. For every threshold value, Accuracy, precision, Recall, F1 Score and ROC AUC will be calculated in order to determine the optimal threshold value.

It is important to note that, for the purpose of this project and the limited time frame, no hyperparameter tuning will be conducted; instead, the focus will be to find a threshold value that limits the number of False Positives and captures as many malicious traffic as possible while also yielding the best performance metrics.

3.3.4 Malware Clustering

The next step is to group the network traffic identified by the Autoencoder as malicious into clusters based on malware types using K-means. In this step, the same dataset will be used but any benign traffic will be excluded. Before clustering, the Elbow method will be used to find the optimal number of clusters. The number and the different types of malware are known since the dataset is labelled; yet, it is important to know how the model learns and processes the data over a different number of clusters rather than forcing it to split the data into a specific number of groups right away. The model will be tested with a different number of clusters (k) each time and for every k, performance metrics will be calculated. Those metrics are Purity, Rand Index, Adjusted Rand Index, Mutual Information, Calinski-Harabasz Index and Davies-Bouldin Index. Further evaluation will include manually observing the data in each cluster to potentially understand how the model processes the data.

3.3.5 Malware Classification

For malware classification, the Random Forest algorithm will be utilised. The purpose of including a supervised model is to identify the different classes of malware rather than only grouping the data into clusters. The model will be trained and tested using the same data set used for the K-means model which will serve as the controlled variables in the experiment/hypothesis testing. The first step will be to split the data into train and test sets. The model will be first trained using the train set, and then using the trained model, the model will predict the labels of the test set. Finally, the classifier will be evaluated using a variety of performance metrics and confusion matrices. It is important to note that classifying malicious traffic is a multiclassification task; thus, binary performance measurements cannot be simply utilised. Consequently, performance metrics will be calculated for each class, and the overall classification's quality will be determined by the average of the same metrics calculated for each class (macro-averaging).

3.3.6 Dashboard

This step includes displaying the result of all three models on a simple dashboard. For the Autoencoder section on the dashboard, the system will simulate and animate the training process of the Autoencoder by displaying the Accuracy and the Loss values for each epoch. Additionally, using a graph, the system will display the data that are being fed to the system as points where malicious and benign traffic will be coloured red and green respectively.

For the clustering and classification section, the system will display the data as clusters that resulted from the unsupervised model (K-means clustering) where each cluster will be coloured differently. Additionally, it will show the identified classes that resulted from the supervised model (Random Forest classification) by listing all the malware classes identified and their distribution within the classes using a pie chart.

The dashboard will be implemented using the Dash framework. It is a python framework created by Plotly for creating interactive web applications [55]. Since the dashboard will only be a simulation and not a live version of any of the models, all data that will be used is going to be retrieved from the models and stored in CSV files beforehand.

3.4 Development Methodology

In order to achieve the aim of this project, a development methodology had to be followed. The development methodology ensures that the development and testing of the system's components are managed, and at the end of the project a clear set of findings and results are delivered and presented. As a result, in this project, the Agile software development methodology was adopted. Since it focuses on incremental delivery, continuous planning, and continuous learning, the agile development strategy is a suitable strategy. It breaks the project into small incremental builds and iterations [24]. As can be seen in Figure 3.3, the client will be able to see and review the working result immediately after each iteration, which is an advantage of the methodology [24].



Figure 3.3. Agile Development Methodology

This methodology was chosen over Waterfall since Waterfall requires the planning stage to be completed before the implementation stage can begin eliminating the flexibility element

required in this project. However, with Agile, any modification to the requirements at any step of the process, even late in the development phase, can be integrated without fear of losing the entire project or starting over [24]. Consequently, this makes it suitable for this project where the timeframe is limited and there is a high chance of requirements alteration. Finally, the supervisor will submit feedback after each iteration, which will provide opportunities for any adjustments or improvements to the system in subsequent iterations. Despite the fact that Agile is a software development strategy for teams, planning is simpler with a single developer [24]. The work was broken down into smaller tasks that needed to be completed on a weekly basis and goals to be achieved on monthly basis making any changes or modifications to the project, as it progressed, flexible. The tasks were prioritized and modified based on the supervisor's feedback in the weekly meetings. The project consisted of 6 iterations. The list below states each iteration's objectives and deliverables.

1st iteration: Loading the data set and conducting an exploratory data analysis

Deliverables: Data is ready to be pre-processed

2nd iteration: pre-processing of the data needed for each model

Deliverables: Data is pre-processed and ready to be used for the training of the models.

3rd iteration: Implementing the Autoencoder

Deliverables: Autoencoder is fully implemented and can predict new data.

5th iteration: Evaluating the Autoencoder

Deliverables: model is evaluated over different threshold values and performance metrics are presented clearly.

4th iteration: Implementing the K-means and Random Forest algorithms.

Deliverables: The data is clustered and classified and models are ready to be tested.

5th iteration: Evaluating K-means and Random Forest.

Deliverables: K-means is evaluated over a different number of clusters while performance metrics for Random Forest are calculated for each class. The evaluation results are displayed clearly.

6th iteration: Dashboard implementation.

Deliverables: The web app is successfully running. It simulates the training process of the Autoencoder and displays the classification results of the Autoencoder and random forest and the clustering results of K-means.

CHAPTER 4

Implementation

This section will discuss the implementation of the models utilized in this project and the Dashboard.

4.1 Project structure

Instead of using a single modular structure with numerous sections, the code was split into sections, each in its own file, in order to make the implementation process simple and the code easy to debug and read. Python is the programming language utilised throughout this project, as previously stated. Python's popularity in the scientific and research fields stems from its ease of use and straightforward syntax, which makes it simple to learn even for persons without an engineering background. Because of its capacity to quickly generate complex tasks, it allows developers to focus on the machine learning element rather than scripting the task [60]. Furthermore, the machine learning models' script is written in Jupyter notebook since it is a convenient and flexible tool that allows all documentation, code execution, output observation, and visualisation of the result to be done in one file. Also, because each section is self-contained, a certain block can be tested without having to execute the code from the beginning, making it easier to access specific functions. Whereas the script of the Dashboard is written in normal Python files.

4.2 Data preparation and pre-processing

In Machine Learning, data preparation and pre-processing are important steps that help improve the quality of data and promote the extraction of relevant insights from the data. Data pre-processing refers to the technique of preparing raw data for the training of Machine Learning

models. It is a data mining approach that converts raw data into an accessible and readable format. This section illustrates how the dataset was pre-processed and prepared for the machine learning models.

4.2.1 Data loading

Importing the dataset from the sources into the analysis tool is the initial step toward data analysis and machine learning models' implementation. The dataset to be analysed is in a conn.log file format. Accordingly, Pandas' Python library was used to parse the log files since it is a widely used tool for data analysis, exploration, and modification. Pandas' `read_table()` method takes the path of a file as an argument and converts it to a data frame. For the purpose of this project, multiple datasets were imported and concatenated into one data frame using the `concat()` function.

4.2.2 Data description

Using Pandas' `info()` method, information about the data frame including the columns, data types, non-null values and memory usage was generated. According to the result generated by the `info()` method, the dataset contains 21 columns and 1,303,279 rows with each column containing valuable information about the captured network traffic. A description of each column can be found in the background section. Moreover, not all 21 columns are used in the study since some of them are unrelated and others have no entries. According to Anthi's study on Intrusion Detection systems, features that reflect identifiable attributes, such as IP addresses and timestamps, should be removed [2]. This will ensure that the model is not dependent on specific network configurations and that the features of network behaviour are captured rather than network actors and devices. As a result, before beginning the implementation process in this project, some columns were eliminated. Further explanation can be found in the upcoming sections. As can be

seen in Figure 4.1, the dataset does not include any null values; thus, the pre-processing step can be carried out without the need to handle null values.

```
... <class 'pandas.core.frame.DataFrame'>
Int64Index: 1303279 entries, 0 to 399998
Data columns (total 21 columns):
#   Column              Non-Null Count  Dtype
---  -
0   ts                  1303279 non-null  object
1   uid                 1303279 non-null  object
2   id.orig_h           1303279 non-null  object
3   id.orig_p           1303279 non-null  float64
4   id.resp_h           1303279 non-null  object
5   id.resp_p           1303279 non-null  float64
6   proto               1303279 non-null  object
7   service             1303279 non-null  object
8   duration            1303279 non-null  object
9   orig_bytes          1303279 non-null  object
10  resp_bytes          1303279 non-null  object
11  conn_state          1303279 non-null  object
12  local_orig          1303279 non-null  object
13  local_resp          1303279 non-null  object
14  missed_bytes        1303279 non-null  float64
15  history             1303279 non-null  object
16  orig_pkts           1303279 non-null  float64
17  orig_ip_bytes       1303279 non-null  float64
18  resp_pkts           1303279 non-null  float64
19  resp_ip_bytes       1303279 non-null  float64
20  label               1303279 non-null  object
dtypes: float64(7), object(14)
memory usage: 218.8+ MB
```

Figure 4.1. Description of the data

4.2.3 Encoding categorical data

This step includes converting any categorical data to numerical ones. Categorical data is information that is divided into distinct categories within a dataset. Mathematical equations serve as the foundation for Machine Learning models; thus, preserving category data in the equation can cause problems because the equations only require numbers.

One column that was encoded is the label column which specifies whether network traffic is malicious or benign. The strings "Benign" and "Malicious" were converted to 0 and 1, respectively, and stored in a new column called `mal_flag` (where 1 means that the traffic is malicious and 0 means that it is benign). As shown in Figure 4.2, the function `labelEncoder()` was used to accomplish this translation. It is a function that comes with the SciKit-learn library, and it translates labels into numeric values.

```
# Create column traffic_type to show benign and malicious
df['label'] = df['label'].astype(str)
df['name_location'] = df['label'].str.find('Benign')
df['name_location'] = df['name_location'].apply(lambda x: 'Malicious' if x == -1 else 'Benign')
df.rename(columns = {"name_location": "traffic_type"}, inplace=True)
```

Python

```
# label code- Using label encoding to code the Malicious as 1 and Benign as 0. New flag variable is named mal_flag
from sklearn.preprocessing import LabelEncoder
df['mal_flag'] = LabelEncoder().fit_transform(df['traffic_type'])
```

Python

Figure 4.2. Data encoding

The label column does not only specify whether a sample is malicious or benign but also includes the type of malware within malicious traffic. Hence, the label column was manipulated in a way that, for malicious traffic, it specifies the type of malware only and for the benign traffic uses the label “Benign”. Moreover, the function `value_counts()` was applied to the label column to check the frequency of each label as can be seen in Figure 4.3. Malicious traffic labelled as Okiru-Attack, C&C-FileDownload and C&C-Torii were removed due to their low frequency. Although C&C and C&C-Heartbeat are also considerably low in frequency, yet; they were not removed from the data frame not only to investigate how the model would process these types of attacks but also to see whether their low occurrence would impact the training of the models.

```
# Labels count based on the traffic type
df['label'].value_counts()
```

Python

Benign	548715
DDoS	399240
Okiru	199727
PartOfAHorizontalPortScan	163167
C&C-HeartBeat	322
C&C	80
C&C-Torii	14
C&C-FileDownload	11
Okiru-Attack	3

Name: label, dtype: int64

Figure 4.3. Label distribution

The columns, `proto` and `conn_state`, were also encoded using Dummy encoding as shown in Figure 4.4. Dummy variables take the values 0 or 1 to represent the absence or presence of a specific category influence that can change the outcome. In this scenario, the value 1 denotes the presence of that variable in a certain column, whereas the other variables are assigned the value 0.

The number of columns is the same as the number of categories in dummy encoding. Using Pandas' `get_dummies()` method, dummy variables were created for the `proto` and `conn_state` columns.

```
# create dummy variables
dataFrame = pd.get_dummies(dataFrame, columns = ['proto'])
dataFrame = pd.get_dummies(dataFrame, columns = ['conn_state'])
```

Python

Figure 4.4. Dummy encoding

4.2.4 Dropping columns

As previously mentioned, features that reflect identifiable attributes should be dropped. Using the `drop()` method some features were removed. The list below includes the removed feature and states the justification for dropping said feature. Figure 4.5 shows the code for the operation.

- `ts`, `uid`, `id.orig_h`, `id.orig_p`, `id.resp_h`, `id.resp_p`: categorical values used for identification.
- `service`, `local_orig`, `local_resp`, `missed_bytes`: missing so much data.
- `history`: contains categorical values that are not related to the problem.
- `traffic_type`: contains categorical values that are not needed anymore since they were encoded in a previous step.

```
dataFrame.drop(['ts', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h',
               'id.resp_p', 'service', 'local_orig', 'local_resp', 'missed_bytes',
               'history', 'traffic_type'], axis=1, inplace=True)
```

Python

Figure 4.5. Dropping columns

4.2.5 Handling continuous variables

Working with raw, continuous numeric characteristics has the disadvantage of skewed value distribution. This means that some values will appear frequently, while others will appear infrequently. Aside from that, there is the issue of varying ranges of values in any of these attributes. Using these features directly can result in a slew of problems and have a negative impact on the model. As a result, a technique, such as binning, exists to deal with this. Binning refers

to dividing a list of continuous variables into groups. Using Pandas `qcut()` function, the continuous numeric features: `duration`, `orig_bytes`, `resp_bytes`, `orig_pkts`, `orig_ip_bytes`, `resp_pkts`, and `resp_ip_bytes` were converted into discrete ones as illustrated in Figure 4.6.

```
# Convert variables - Converting continuous variables to bin numerical data into groups (Normalizing)
dataFrame['duration'] = pd.qcut(dataFrame['duration'].rank(method='first'), q=20, labels=False )
dataFrame['orig_bytes'] = pd.qcut(dataFrame['orig_bytes'].rank(method='first'), q=10, labels=False )
dataFrame['resp_bytes'] = pd.qcut(dataFrame['resp_bytes'].rank(method='first'), q=10, labels=False )
dataFrame['orig_pkts'] = pd.qcut(dataFrame['orig_pkts'].rank(method='first'), q=10, labels=False )
dataFrame['orig_ip_bytes'] = pd.qcut(dataFrame['orig_ip_bytes'].rank(method='first'), q=10, labels=False )
dataFrame['resp_pkts'] = pd.qcut(dataFrame['resp_pkts'].rank(method='first'), q=10, labels=False )
dataFrame['resp_ip_bytes'] = pd.qcut(dataFrame['resp_ip_bytes'].rank(method='first'), q=10, labels=False )
```

Python

Figure 4.6. Converting continues variables to categories

4.3 Autoencoder implementation

This section will discuss the implementation process of the Autoencoder.

4.4 Train-Test Split

Machine learning algorithms and Neural Networks are based on the idea that it fits a model on a train set before making predictions on another dataset called the test set. In order to avoid memorization and overfitting and prevent sampling fluctuations, the train set is frequently set to be larger than the test set. Moreover, both the train and test set should have the same statistical properties. Using the train-test split approach, the model can be evaluated on unseen dataset providing an unbiased final model performance metric.

Accordingly, the first step of the Autoencoder implementation is to split the data. The data was split into train and test sets using scikit-learn's `train_test_split()` function. As mentioned in the Approach section, the Autoencoder must be trained with benign traffic only but tested with both malicious and benign network traffic. Therefore, all malicious samples were removed from the train set. Figure 4.7 shows the code used for this step.

```

X_train, X_test = train_test_split(df, test_size=0.3, random_state=RANDOM_SEED)
X_train = X_train[X_train.mal_flag == 0]
X_train = X_train.drop(['mal_flag', 'label'], axis=1)

y_test = X_test['mal_flag']
X_test = X_test.drop(['mal_flag', 'label'], axis=1)

X_train = X_train.values
X_test = X_test.values

X_train.shape

```

Python

Figure 4.7. train-test split

4.4.1 The architecture of the model

The Autoencoder employs an input layer with four completely connected layers, each containing 12, 6, 6, and 20 neurons respectively. The encoder takes up the first two layers, while the decoder takes up the last two. Additionally, during training, L1 regularisation was employed. Whereas for the activation functions, the rectified linear unit and Hyperbolic tangent were used.

```

input_dim = X_train.shape[1]
encoding_dim = 12

input_layer = Input(shape=(input_dim, ))

encoder = Dense(encoding_dim, activation="tanh", activity_regularizer=regularizers.l1(10e-5))(input_layer)
encoder = Dense(int(encoding_dim / 2), activation="relu")(encoder)

decoder = Dense(int(encoding_dim / 2), activation="tanh")(encoder)
decoder = Dense(input_dim, activation="relu")(decoder)

autoencoder = Model(inputs=input_layer, outputs=decoder)

autoencoder.compile(optimizer='adam', loss='mean_squared_error', metrics=['accuracy'])

# print an overview of the model
autoencoder.summary()

```

Python

Model: "model_1"

Layer (type)	Output Shape	Param #
=====		
input_2 (InputLayer)	[(None, 20)]	0
dense_4 (Dense)	(None, 12)	252
dense_5 (Dense)	(None, 6)	78
dense_6 (Dense)	(None, 6)	42
dense_7 (Dense)	(None, 20)	140
=====		
Total params:	512	
Trainable params:	512	
Non-trainable params:	0	

Figure 4.8. Model Architecture

Additionally, the behaviour of the model was customised with two callbacks:

1. ModelCheckpoint which saves the best performing model, in terms of Accuracy, to an H5 file format during training, and
2. TensorBoard which exports the training progress in a TensorBoard-compatible format for visualisation [54].

As can be seen in Figure 4.9, training metrics for each epoch were recorded using the History object and stored in the history variable. Finally, with a batch size of 32 samples, the model was trained for 100 epochs using the train set and validated using the test.

```

nb_epoch = 100
batch_size = 32

checkpointer = ModelCheckpoint(filepath="model12.h5", verbose=0, save_best_only=True)
tensorboard = TensorBoard(log_dir='./logs', histogram_freq=0, write_graph=True, write_images=True)

history = autoencoder.fit(X_train, X_train, epochs=nb_epoch, batch_size=batch_size,
                        shuffle=True, validation_data=(X_test, X_test), verbose=1,
                        callbacks=[checkpointer, tensorboard]).history

```

Output exceeds the [size limit](#). Open the full output data [in a text editor](#)

```

Epoch 1/100
9063/9063 [=====] - 12s 1ms/step - loss: 2.5432 - accuracy: 0.9003 - val_loss: 2.5635 - val_accuracy: 0.9362
Epoch 2/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.2729 - accuracy: 0.9184 - val_loss: 0.2437 - val_accuracy: 0.7880
Epoch 3/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.0139 - accuracy: 0.8882 - val_loss: 0.2410 - val_accuracy: 0.7469
Epoch 4/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.0114 - accuracy: 0.8825 - val_loss: 0.2308 - val_accuracy: 0.7598
Epoch 5/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.0099 - accuracy: 0.9083 - val_loss: 0.1665 - val_accuracy: 0.7564
Epoch 6/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.0088 - accuracy: 0.9128 - val_loss: 0.1444 - val_accuracy: 0.7628
Epoch 7/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.0083 - accuracy: 0.9122 - val_loss: 0.1341 - val_accuracy: 0.7396
Epoch 8/100
9063/9063 [=====] - 12s 1ms/step - loss: 0.0079 - accuracy: 0.9124 - val_loss: 0.1135 - val_accuracy: 0.7620
Epoch 9/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.0067 - accuracy: 0.9104 - val_loss: 0.1497 - val_accuracy: 0.7620
Epoch 10/100
9063/9063 [=====] - 12s 1ms/step - loss: 0.0058 - accuracy: 0.9112 - val_loss: 0.1204 - val_accuracy: 0.7559
Epoch 11/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.0055 - accuracy: 0.9117 - val_loss: 0.1010 - val_accuracy: 0.7629
Epoch 12/100
9063/9063 [=====] - 11s 1ms/step - loss: 0.0054 - accuracy: 0.9126 - val_loss: 0.1035 - val accuracy: 0.7832

```

Figure 4.9. Model training

4.4.2 Reconstruction error

Following the training of the model, the reconstruction error must be calculated. Although the model was successfully trained, it can only predict benign traffic as it was only trained with this type of traffic. Accordingly, it can reconstruct benign traffic samples with a small

reconstruction error, whereas malicious traffic would yield a higher reconstruction error. Thus, the reconstruction error would be used as a threshold value to distinguish between malicious and benign traffic.

Firstly, given the trained model, Scikit-learn's `predict()` function predicted the labels of the test set. The method accepts one argument, the new data, and returns the learned labels for each object in the array. As shown in Figure 4.10, the reconstruction error was then calculated using the formula discussed in the Background section and stored in a data frame. Using the `describe()` method, descriptive statistics of the reconstruction error were generated including the mean, minimum, maximum and different percentile values of the reconstruction error.



Figure 4.10. Reconstruction error

4.4.3 Model evaluation over different threshold values

Since the model is trained using benign traffic only, it cannot predict the labels of non-benign traffic. Hence, the model is expected to return a high reconstruction error when tested with

non-benign traffic. That is, a threshold must be defined such that if a network traffic reconstruction error is greater than the pre-defined threshold, it is classified as malicious.

Initially, the threshold was set as the mean of reconstruction error for benign traffic, but it yielded a high False Negative rate (misclassified malicious traffic). When tested with a smaller threshold value, the rate of False Negative decreased. Hence, it was important to test the model and calculate its performance metrics over different threshold values in order to define the optimal value that would maximise the number of correctly predicted malicious traffic. The influence of the threshold value on the proposed model is further discussed in the Results and Evaluation section. The model's Accuracy, Precision, Recall, F1-Score and AUC ROC were calculated over 30 different threshold values that range between 0 to 0.003 and stored in a data frame. These metrics are provided by the sklearn.metrics module as functions which increase the efficiency of the implementation. Figure 4.11 shows the implementation of this step.

```
performance = pd.DataFrame(columns = ["threshold", "Accuracy", "Precision", "Recall", "F1-Score", "ROC AUC"], index=range(30))

val = np.linspace(0, 0.003, num=30)

# test with different thresholds
for x, th in enumerate(val):
    performance.iloc[x,0] = th
    # determine label
    y_pred = [1 if e > th else 0 for e in error_df.reconstruction_error.values]
    # calculate metrics
    A = metrics.accuracy_score(y_test, y_pred)
    P = metrics.precision_score(y_test, y_pred, average='binary', pos_label=1)
    R = metrics.recall_score(y_test, y_pred, average='binary', pos_label=1)
    F1 = metrics.f1_score(y_test, y_pred, average='binary', pos_label=1)
    roc_auc = metrics.roc_auc_score(y_test, y_pred)

    # store metrics in a data frame
    performance.iloc[x,1] = A
    performance.iloc[x,2] = P
    performance.iloc[x,3] = R
    performance.iloc[x,4] = F1
    performance.iloc[x,5] = roc_auc
```

Figure 4.11. Autoencoder evaluation

4.5 Clustering with K-means

As mentioned in the approach section, the purpose of employing K-means and Random Forest algorithms in the system is to classify data samples that are identified as malicious, by the Autoencoder, based on the different types of malware. However, it is not the aim of this project to

build the system but to investigate the feasibility of integrating an Autoencoder, K-means and random forest models to build an actual malware detection system for IoT devices. Hence, each model was built and tested separately to test the hypotheses. This section will illustrate the implementation of the K-means model.

4.5.1 Data initialization

In order to assess the feasibility of incorporating K-means into the system, the model must be tested only using malicious traffic. As a result, only malicious traffic from the IoT-23 dataset was used in the process. The dataset consists of 5 types of attacks as can be seen in Figure 4.12.

DDoS	399240
Okiru	199727
PartOfAHorizontalPortScan	163167
C&C-HeartBeat	322
C&C	80
Name: label, dtype: int64	

Figure 4.12. Malware Distribution

4.5.2 Model implementation

K-means clustering is an unsupervised machine learning approach that detects unique categories of groups in unlabelled datasets without the need for training, allowing data to be clustered into a number of clusters (k). The algorithm requires the specification of the number of clusters that it will generate. Therefore, finding the optimal number of clusters (k) is an important step toward achieving a high performing model.

Scikit-learn library provides an implementation of the algorithm meaning that there is no need to build the model from scratch. The `kmeans()` method accepts various parameters but the ones that were manually specified for this model are: **n_clusters**, **init** and **random_state** [44]. The **init** represents the initialisation method while the **random_state** determines random number

generation for centroid initialization. **n_clusters**, on the other hand, determines the number of clusters to form and the number of centroids to generate [44].

The first step toward building the model is to find the ideal number of clusters (k). This was achieved using the elbow. As mentioned in the background section, it chooses the optimum value of k based on the distance between the data points and their associated clusters using the sum of squared distance (WCSS). As can be seen in Figure 4.13, the algorithm iterates the value of k from 1 to 10 and initialises a `kmeans()` function with the k as the **n_clusters**. It calculates the value of WCSS for each k using the `inertia_` attribute of `kmeans()` and plots the result.

```
# Using the elbow method to find the optimal number of clusters
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

Python

Figure 4.13. Features and labels initialisation

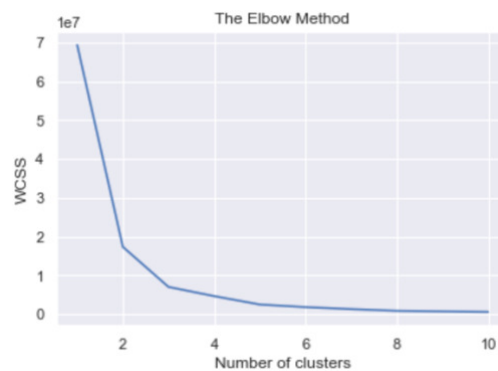


Figure 4.14. plot of Elbow method

The final K-means model was initialised with a random seed of 42 and `kmeans++` as the method for initialisation. Whereas the number of clusters is the elbow point observed from the chart generated in the previous step.

```
# define the model
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)

# train the model
y_kmeans = kmeans.fit_predict(X)
```

Python

Figure 4.15. Define and train K-means

Visualising the clusters generated by a clustering algorithm is an essential step toward evaluating its performance. Using the matplotlib library, the clusters and their centroids were displayed using a scatter plot with each cluster being coloured differently. The scatter plot illustrates the relationship between the number of bytes being sent to the device (`orig_ip_bytes`) and the number of bytes being sent from the device (`resp_ip_bytes`) for each malicious network traffic. The data were represented as points in an x, y coordinate system, with each point representing one observation along two axes of variation.

```
plt.figure(figsize=(7,7))

plt.scatter(X[y_kmeans == 0, 4], X[y_kmeans == 0, 6], s = 50, c = 'crimson', label = 'Cluster 1')
plt.scatter(X[y_kmeans == 1, 4], X[y_kmeans == 1, 6], s = 50, c = 'dodgerblue', label = 'Cluster 2')
plt.scatter(X[y_kmeans == 2, 4], X[y_kmeans == 2, 6], s = 50, c = 'green', label = 'Cluster 3')

plt.scatter(kmeans.cluster_centers_[0, 4], kmeans.cluster_centers_[0, 6], marker="X", s = 175, c = 'gold', label = 'Centroids')

plt.title('Clusters of Malicious traffic', fontsize=16)
plt.xlabel('orig_ip_bytes')
plt.ylabel('resp_ip_bytes')
plt.legend()
plt.show()
```

Python

Figure 4.16. Plot clusters

4.5.3 Model Evaluation

As previously stated, the dataset consists of 5 types of attacks which logically means that there should be 5 clusters. However, according to the result of the Elbow method, only 3 clusters can be identified. Accordingly, further investigation was necessary to understand the behaviour of the model. Similar to the structure of the Elbow method algorithm, the model was tested with a different value of k, and, for each value, clustering performance metrics were calculated.

The model was evaluated using 6 clustering evaluation methods: Purity, Rand Index, Adjusted Rand Index, Mutual Information, Calinski-Harabasz Index, and Davies-Bouldin Index.

Further details on these metrics can be found in the background section. It is important to note that Purity, Rand Index, Adjusted Rand Index and Mutual Information require the true labels of the samples which are not usually available in real-life. However, labels are available for this dataset which means that they can be utilised to test the hypothesis of this project. These methods are provided by the sklearn.metrics module as functions making the evaluation process more efficient.

```
# compute contingency matrix (also called confusion matrix)
contingency_matrix = metrics.cluster.contingency_matrix(y, y_kmeans)
# Purity
p = np.sum(np.amax(contingency_matrix, axis=0)) / np.sum(contingency_matrix)
# Rand Index
ri = np.round((metrics.rand_score(y, y_kmeans)),4)
# Adjusted Rand Index
amis = np.round((metrics.adjusted_mutual_info_score(y, y_kmeans, average_method='arithmetic')),4)
# Mutual Information
mis = np.round((metrics.mutual_info_score(y, y_kmeans, contingency=None)),4)
# Calinski-Harabasz Index
chs = np.round(calinski_harabasz_score(X, y_kmeans),4)
# Davies-Bouldin Index
dbs = np.round(davies_bouldin_score(X, y_kmeans),4)
```

Figure 4.17. K-means evaluation

4.6 Random Forest classifier

Random Forest algorithm is the supervised learning classifier, for this project it is used to classify malicious network traffic based on the different types of malware. It is a meta estimator that fits several decision tree classifiers on various sub-samples of the dataset and, by using averaging, it improves the predictive Accuracy and control over-fitting [45]. Although Random Forest ensembles can be built from scratch, the scikit-learn Python machine learning library provides an implementation of the algorithm. This section will discuss the implementation of the classifier.

4.6.1 Train-Test Split

As mentioned previously, splitting the data into train and test sets is an essential step in building a machine learning model as it is used to evaluate the performance of a machine learning model. As shown in Figure 4.18, the data is split into train and test sets with a ratio of 70:30 using

scikit-learn's `train_test_split()` function. The dataset only includes malicious traffic with the same features and labels used for the implementation of the K-means model.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 1)
```

Python

Figure 4.18. train-test split for random forest

4.6.2 Model implementation

The model is defined using the method `RandomForestClassifier()` with `n_estimators` as a parameter. `n_estimators` is an integer that is used to specify the number of trees in the forest [45]. The default value of `n_estimators` is 100 and it is the same value used for this model. The model is fitted on the train set and the predictions are made on both the train and test sets. The complete implementation of the model is shown in Figure 4.19.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn.metrics import confusion_matrix

classifier = RandomForestClassifier(n_estimators=100)

#Train the model
classifier.fit(X_train,y_train)

# prediction on the training set
y_pred_train = classifier.predict(X_train)

# prediction on test set
y_pred = classifier.predict(X_test)
```

Python

Figure 4.19. random forest model

4.6.3 Model evaluation

As mentioned in the background section, the task in this project is considered a multiclass classification since it aims to classify malicious network traffic into different types of attacks. Hence, the performance metrics were first calculated for each class individually and then the mean of the results is calculated to find the overall metrics. Before calculating the performance metrics, a confusion matrix was used to calculate the number of True Positives, True Negatives, False Positives and False Negatives of each class for both the train and test sets. Figure 4.20 shows the implementation of this step.

```
# calculate TP, FP, FN, and TN
TP_train = np.diag(conf_matrix_train)
FP_train = conf_matrix_train.sum(axis=1) - np.diag(conf_matrix_train)
FN_train = conf_matrix_train.sum(axis=0) - np.diag(conf_matrix_train)
TN_train = conf_matrix_train.sum() - (FP_train + FN_train + TP_train)
```

Python

Figure 4.20. confusion matrix

The overall Accuracy and AUC ROC, and each class's precision, Recall and F1 scores, for both the train and test sets, are calculated using the formulas described in the background section and stored in a data frame with the mean of each metric. Figure 4.21 shows the computation of the performance metrics.

```
# accuracy
A_train = (np.sum(TP_train)+np.sum(TN_train))/ (np.sum(TP_train)+np.sum(TN_train)+np.sum(FP_train)+np.sum(FN_train))
A_test = (np.sum(TP_test)+np.sum(TN_test))/ (np.sum(TP_test)+np.sum(TN_test)+np.sum(FP_test)+np.sum(FN_test))
```

```
# roc_auc
train_prob = classifier.predict_proba(X_train)
tests_prob = classifier.predict_proba(X_test)

roc_auc_ovo_train = metrics.roc_auc_score(y_train, train_prob, multi_class="ovo", average="macro")
roc_auc_ovo_test = metrics.roc_auc_score(y_test, tests_prob, multi_class="ovo", average="macro")
```

Python

```
# precision
P_train = TP_train/(TP_train+FP_train)
P_train_m = np.mean(P_train)

P_test = TP_test/(TP_test+FP_test)
P_test_m = np.mean(P_test)

# recall
R_train = TP_train/(TP_train+FN_train)
R_train_m = np.mean(R_train)

R_test = TP_test/(TP_test+FN_test)
R_test_m = np.mean(R_test)

# F1 scores
F1_train = 2*((P_train+R_train)/(P_train+R_train))
F1_train_m = np.mean(F1_train)

F1_test = 2*((P_test+R_test)/(P_test+R_test))
F1_test_m = np.mean(F1_test)
```

Python

Figure 4.21 metrics computation

4.7 Dashboard implementation

The purpose of implementing the dashboard is to provide a simulation of the actual malware detection system. The dashboard does not deploy the implemented models in the backend but uses the results generated from them to simulate the system. The dashboard is implemented using the Dash framework which is a python framework created by Plotly for creating interactive web applications. This section will discuss the implementation of the Dashboard.

4.7.1 Web-app structure

The Dash project folder is structured as follows:

```
|-- app.py
|-- app_utils.py
|-- Procfile
|-- requirements.txt
|-- assets
|-- data
```

The requirements.txt file describes all the Dash app's Python dependencies that would be installed during the deployment process. Whereas the Procfile is a text file that declares which commands would be run by Dash Enterprise on the start-up.

```
dash==1.0.0
gunicorn==19.9.0
scipy==1.2.1
numpy==1.16.3
pandas==0.24.2
```

Figure 4.22 requirements

```
web: gunicorn app:server --workers 4
```

Figure 4.23 Procfile

The assets folder contains all the CSS files that were used to style the web page while the data folder consists of all the CSV files used to build the simulation. The content of each file is described as follows:

- history.csv: The training history of the Autoencoder including the Accuracy and loss per epoch
- error.csv: Reconstruction error yielded from testing the Autoencoder with both malicious and benign traffic
- df_cluster.csv: The data frame used for clustering with an additional column that specifies the cluster predicted by K-means
- df_class.csv: The data frame used for classification with an additional column that specifies the class predicted by random forest

The app.py file contains the Python code and is called by the command in the Procfile. It also contains a line that defines the server variable so that it can be exposed for the Procfile as shown in Figure 4.24. The last file within the Dash project folder is the app_utils.py. It includes supportive functions that are called within the app.py file.

```
--
19 app = dash.Dash(
20     __name__, meta_tags=[{"name": "viewport", "content": "width=device-width"}]
21 )
22 app.title = "Malware Detection System"
23 server = app.server
```

Figure 4.24 server variable

The app.py is composed of two parts. The first part is the "layout" of the app, and it describes what the application looks like. The second part describes the interactivity of the application. The layout is composed of a tree of components such as dcc.Tab, html.Div and dcc.Graph. Most of the components are pure HTML. However, the Dash Core Components module (dcc) contains higher-level components that are interactive and are generated with JavaScript, HTML, and CSS through the React.js library.

The app consists of two dcc.Tab which initialises two tabs for the web app. One tab displays all the components related to the Autoencoder and the second tab illustrates the results of the K-means and Random Forest algorithms.

```
179 app.layout = html.Div([
180     dcc.Tabs([
181         dcc.Tab(label='Autoencoder', children=[
182             html.Div(
183                 style={"height": "100%"},
301         dcc.Tab(label='Malicious traffic classification', children=[
302             html.Div(
303                 style={"height": "100%"}.
```

Figure 4.25 app layout

4.7.2 Anomaly detection with an Autoencoder:

As previously mentioned, the Autoencoder's purpose is to classify network traffic based on reconstruction error. In an actual system, it is expected to take network traffic as input and classifies them based on reconstruction error. The classification process is visualized as a chart where the x axis represents the order in which the data series was fed into the system and the y axis represents the reconstruction error. Each network traffic that is fed into the system is placed on the chart as a point where red indicates malicious traffic and green indicates benign traffic.

Using the threshold value that yielded the best performance metrics (refer to the results and evaluation for further explanation about the threshold value), the `setColour()` function assigns a colour for each point based on its reconstruction error. Following the colour assignment, the `error_graph()` displays all the points on the graph. Figure 4.26 includes the code used for the implementation of this step.

```

531 def SetColor(x):
532     if(x < 0.000724):
533         return "green"
534     else:
535         return "red"
536
537 def error_graph(
538     graph_id,
539     history,
540     yaxis_title,
541     value
542 ):
543     if history:
544         layout = go.Layout(
545             margin=go.layout.Margin(l=50, r=50, b=50, t=50),
546             yaxis={"title": yaxis_title},
547             xaxis=dict(range=[-0.01, 1], autorange=False, zeroline=False),
548             )
549
550     re_error = error(value)
551
552     anomaly_detect = go.Scatter(
553         x=np.linspace(0,1,200),
554         y=re_error,
555         mode="lines+markers",
556         line=dict(color="#797676"),
557         marker = dict(color=list(map(SetColor, re_error)))
558     )
559
560     figure = go.Figure(data=[anomaly_detect], layout=layout)
561     return dcc.Graph(figure=figure, id=graph_id)
562
563     return dcc.Graph(id=graph_id)

```

Figure 4.26 Autoencoder - dashboard

4.7.3 Autoencoder training:

In addition to the visualization of the Autoencoder classification result, a simulation of the training process of the model is included in the system. The app allows visualization of the metrics as they are updated inside the model. The app uses the data stored during the actual training of the Autoencoder, which is stored in the History.csv file as previously mentioned, to simulate the process as an animated graph.

The `update_graph()` function initializes graphs for both the training and validation of the model based on the data stored in the history.csv. It updates the Accuracy and loss graphs as the number of epochs increases in order to simulate the model training process. As long as the app is running, the number of epochs is increasing, and the graphs are updating.

```

325 def update_graph(
326     graph_id,
327     graph_title,
328     y_train_index,
329     y_val_index,
330     hisroty,
331     display_mode,
332     yaxis_title,
333 ):
334     if hisroty:
335         layout = go.Layout(
336             title=graph_title,
337             margin=go.layout.Margin(l=50, r=50, b=50, t=50),
338             yaxis=dict(title=yaxis_title),
339         )
340
341         history_df = pd.read_json(hisroty, orient="split")
342
343         epoch = history_df["step"]
344         y_train = history_df[y_train_index]
345         y_val = history_df[y_val_index]
346
347         trace_train = go.Scatter(
348             x=epoch,
349             y=y_train,
350             mode="lines",
351             name="Training",
352             line=dict(color="#D03535"),
353             showlegend=True,
354         )

```

Figure 4.27 Autoencoder training - dashboard

Moreover, the app also allows the user to change the display mode of the metrics graphs while the simulation is running. Using radio buttons, the user can specify the display mode.

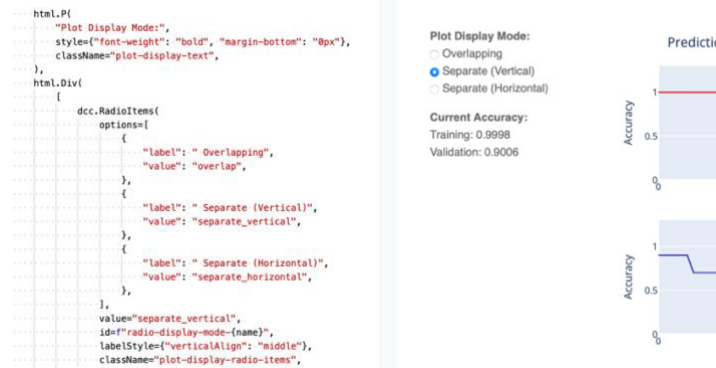


Figure 4.28 plot display mode

4.7.4 Clustering and Classification visualisation:

In order to visualize the clustering and classification results, the necessary CSV files were imported from the data folder and converted into data frames using the Pandas' `read_csv()` method.

```

47
48 df_cluster = pd.read_csv(DATA_PATH.joinpath("df_cluster.csv"))
49 df_class = pd.read_csv(DATA_PATH.joinpath("df_class.csv"))
50

```

Figure 4.29 read csv files

For the clustering results, the data is visualised as a scatter plot where each point is coloured based on the cluster it belongs to. The scatter plot illustrates the relationship between the number of bytes being sent to the device (`orig_ip_bytes`) and the number of bytes being sent from the device (`resp_ip_bytes`) for each malicious network traffic. The construction of the clustering graph is divided into two functions: `clusters_graph()` plots the actual graph and returns it as a figure while `update_clusters_graph()` executes the `clusters_graph()` function and places the returned figure in its assigned position in the layout of the app.

```

580 y_kmeans_test = df_cluster.iloc[:, 22].values
581 test = df_cluster.drop(columns=['mal_cat', 'cluster'])
582 test = test.values
583
584 def clusters_graph(graph_id, history):
585     if history:
586         layout = go.Layout(
587             margin=go.layout.Margin(l=50, r=50, b=50, t=50),
588             xaxis=dict(range=[-0.1, 1.1], autorange=False, zeroline=False))
589         C1 = go.Scatter(
590             x=test[:, 1],
591             y=test[:, 2],
592             mode='markers',
593             opacity=0.5,
594             marker=dict(color=y_kmeans_test*5, size=12,
595                         line=dict(width=1, color='grey')))
596         figure = go.Figure(data=[C1], layout=layout)
597         return dcc.Graph.figure=figure, id=graph_id
598     return dcc.Graph(id=graph_id)
599
600 @app.callback(
601     Output("div-malclass-graph", "children"),
602     [Input("run-log-storage", "data")])
603
604 def update_clusters_graph(history):
605     graph = clusters_graph("error-graph", history)
606     return [graph]
607
608

```

Figure 4.30 plot cluster graph

For the classification results, the data is visualised as a pie chart that displays the attacks identified and their distribution within the dataset. Similar to clustering, the construction of the classification visualization is divided into two functions: `class_graph()` plots the actual pie chart and returns it as a figure while `update_class_graph()` executes the `class_graph()` function and places the returned figure in its assigned position in the layout of the app.

```

609 def class_graph(history, graph_id):
610     fig = px.pie(df_class, values="predection", names="label")
611     return dcc.Graph.figure=fig, id=graph_id
612
613 @app.callback(
614     Output("div-class-graph", "children"),
615     [Input("run-log-storage", "data")])
616
617 def update_class_graph(history):
618     graph = class_graph(history, "class-graph")
619     return [graph]

```

Figure 4.31 plot classification results

4.8 Summary of the implementation process.

The implementation process was divided into 4 separate sections. The first part included pre-processing the dataset and conducting an Exploratory Data Analysis (EDA) followed by the complete implementation of the Autoencoder and its evaluation. The third part was to implement

K-means and random forest algorithms and evaluate them. Finally, using the results of the implemented model, a simple dashboard was implemented to provide a simulation of a possible malware detection system for IoT devices.

CHAPTER 5

Results and Evaluation

The aim of this project has been achieved. An Autoencoder, a K-means model and a Random Forest mode have been successfully developed. The Autoencoder has shown the ability to detect 90% of malicious traffic after being trained with benign samples only. Whereas the K-means and Random Forest models' performance varied based on the different categories of attacks within the dataset encouraging further analysis of the features and malware distribution within the dataset. Detailed results, discussion of the findings and the limitations are further discussed in the upcoming sections.

5.1 Exploratory Data Exploration Results

Exploratory Data Analysis (EDA) was the first step of the implementation of this project. This phase was significantly important in examining and investigating the dataset before pre-processing the dataset and building the models. Furthermore, it helped in terms of comprehending the dataset variables and their relationships.

Figure 5.1 illustrates the distribution of network traffic within the dataset. 58.5% of the samples are malicious traffic while 41.5% are benign samples which is a balanced distribution of data. Whereas the distribution of malware categories within the malicious samples is imbalanced. As shown in Figure 5.2, DDoS attacks, Okiru botnets and port scans dominate the malware categories while C&C and C&C-HeartBeat take up only 0.05% of the data.

Data composition by traffic type

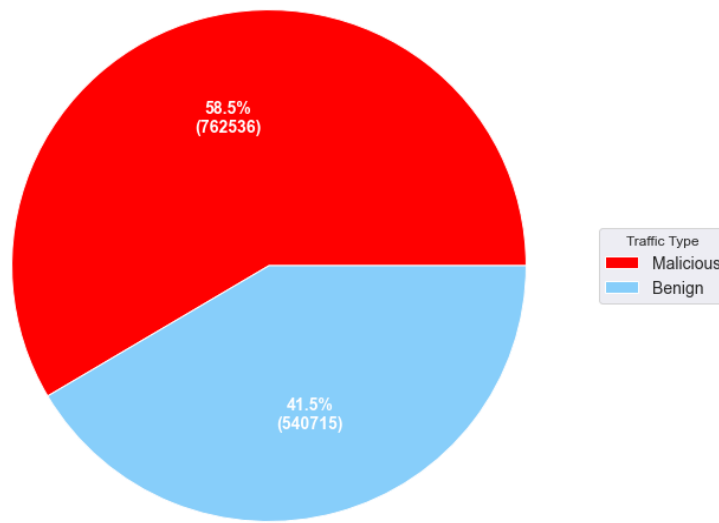


Figure 5.1 Traffic distribution

Data composition by malware type

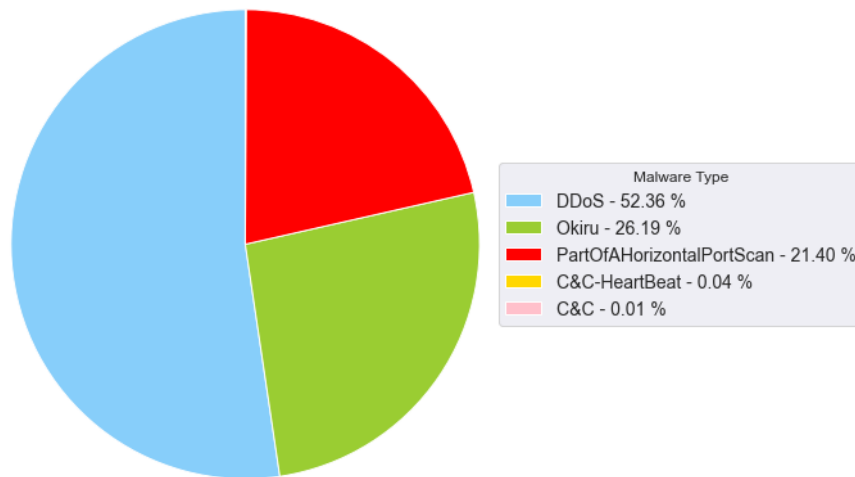


Figure 5.2 malware distribution

5.1.1 Summary of EDA results:

The distribution of the numeric variables shows that malicious traffic and benign traffic has clear distinction across all numeric features of the dataset; hence, it is expected that machine learning algorithms would be able to separate the two classes. However, the analysis indicates that

within malicious traffic, the distinction across the numeric features is less. Consequently, machine learning algorithms might not fully distinguish between the different categories of malware within the dataset. Further discussion on the impact of features on the proposed models can be found in the upcoming sections.

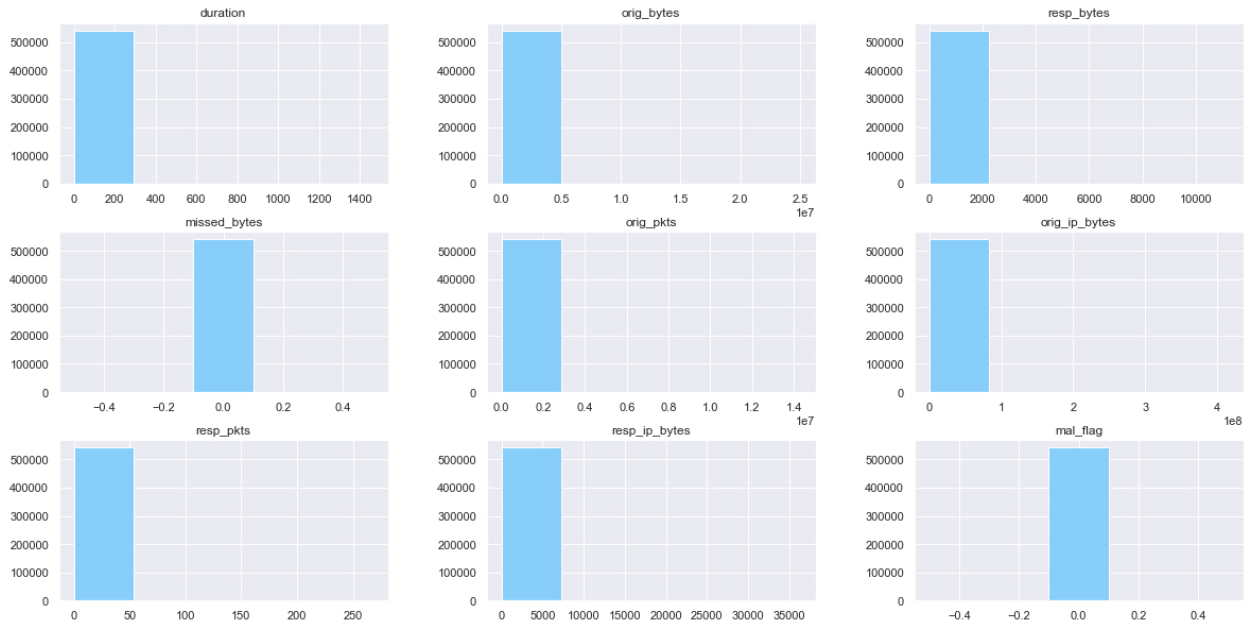


Figure 5.3 histograms for all the numerical columns for benign traffic

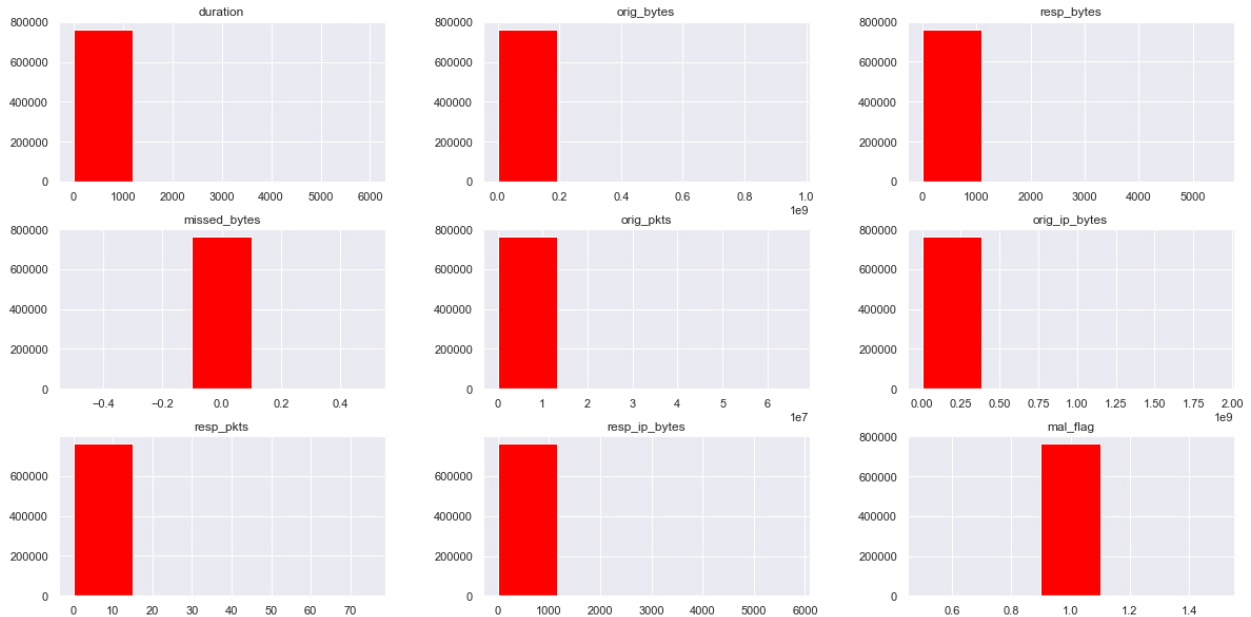


Figure 5.4 histograms for all the numerical columns for malicious traffic

5.2 Anomaly detection with Autoencoder

The first function of the malware detection system for IoT devices is to detect malicious traffic using an Autoencoder. The purpose of employing an Autoencoder is to test whether it can successfully predict malicious traffic if it was only trained on benign traffic. The hypothesis is that the malicious traffic will yield higher reconstruction error in comparison to the benign traffic. Hence, the network traffic will be classified as malicious if its reconstruction error surpasses a fixed threshold.

5.2.1 Loss and Accuracy

The Autoencoder was trained on 290,445 benign traffic and it was then used to reconstruct both benign and malicious network traffic. As previously stated, an Autoencoder aims to learn lower-dimensional representation for higher-dimensional data while minimising the reconstruction error. As can be seen in Figure 5.5, the model's reconstruction error for the train and validation sets is converging and decreasing sharply as the training proceeds. It is important to note that the validation set consists of both malicious and benign network traffic which explains why the reconstruction error for the validation set is higher than that for the training set.

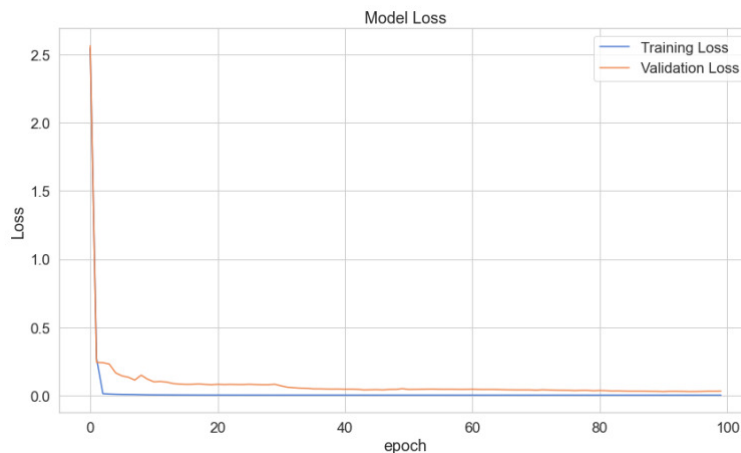


Figure 5.5 Model Loss

Figure 5.6 shows the accuracy of the model for the train and validation sets. The model's Accuracy on the train set is considerably stable as the training proceeds while for the validation set, the model's Accuracy is fluctuating but it is still lower than that for the train set at some points.

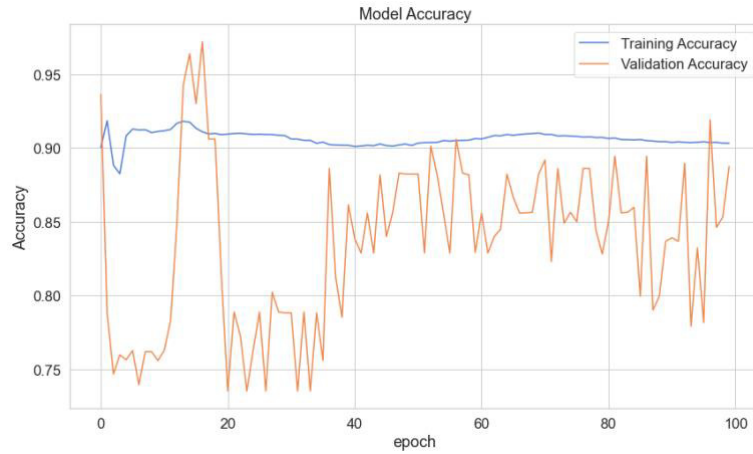


Figure 5.6 Model Accuracy

5.2.2 Reconstruction Error

Based on the model's loss and Accuracy alone it can be determined that the model performs poorly on malicious traffic. Nonetheless, it cannot be determined, just by observing the chart, whether the reconstruction error for the train set is low enough so that malicious traffic can be detected successfully when fed to the model. Therefore, it is important to investigate the reconstruction error distribution for both the train and test set in order to determine the difference between the reconstruction error yielded by benign traffic and malicious traffic. Tables 5.1 and 5.2 shows the reconstruction error distribution for the malicious and the benign traffic, from the test set, respectively. Meanwhile, Table 5.3 shows the reconstruction error distribution for the benign samples from the train set. As can be seen in Table 5.1 the mean of reconstruction error for malicious traffic is 0.0471 which equals approximately 14 times the mean of reconstruction error for benign traffic (0.0032). While 75% of the benign traffic is reconstructed with an error of 0.0004 or lower (75th percentile), the majority of the malicious samples resulted in a significantly higher

reconstruction error of 0.0011 or higher (25th percentile). Hence, the hypothesis that malicious traffic will have higher reconstruction error in comparison to benign traffic is proven correct.

Table 5.1. Reconstruction error for malicious samples – test set

reconstruction_error	
count	175527.000000
mean	0.047060
std	0.083844
min	0.000054
25%	0.001138
50%	0.032950
75%	0.045701
max	3.971564

Table 5.2. Reconstruction error for benign samples – test set

reconstruction_error	
count	124473.000000
mean	0.003199
std	0.024347
min	0.000049
25%	0.000117
50%	0.000162
75%	0.000449
max	1.450583

Table 5.3. Reconstruction error for benign samples – train set

reconstruction_error	
count	290007.000000
mean	0.003202
std	0.024369
min	0.000049
25%	0.000117
50%	0.000162
75%	0.000449
max	1.450583

5.2.3 Precision and Recall

As previously mentioned, Precision is used to measure the ratio of True Positives to the total number of samples classified as Positive. Whereas Recall measures the ratio between the

number of True Positives to the total number of actual Positive samples. Table 5.4 shows the confusion matrix for the Autoencoder where each cell in the matrix represents an evaluation factor. For this Autoencoder, Positive refer to malicious traffic while negative refers to benign traffic. As can be seen in Figure 5.7, the model's Precision increases as the reconstruction error increases. This implies that the False Positive rate decreases with the increase of the reconstruction error indicating that the model is classifying benign traffic correctly. On the other hand, the model's Recall decreases as the reconstruction error increases which indicates a high False Negative rate. Low Recall indicates that the model is misclassifying malicious traffic as benign. Figure 5.8 shows the model's Recall over different reconstruction error values.

Table 5.4. Autoencoder confusion matrix

		<i>Predicted class</i>	
<i>True Class</i>	Class	Benign	Malicious
	Benign	TN	FP
	Malicious	FN	TP

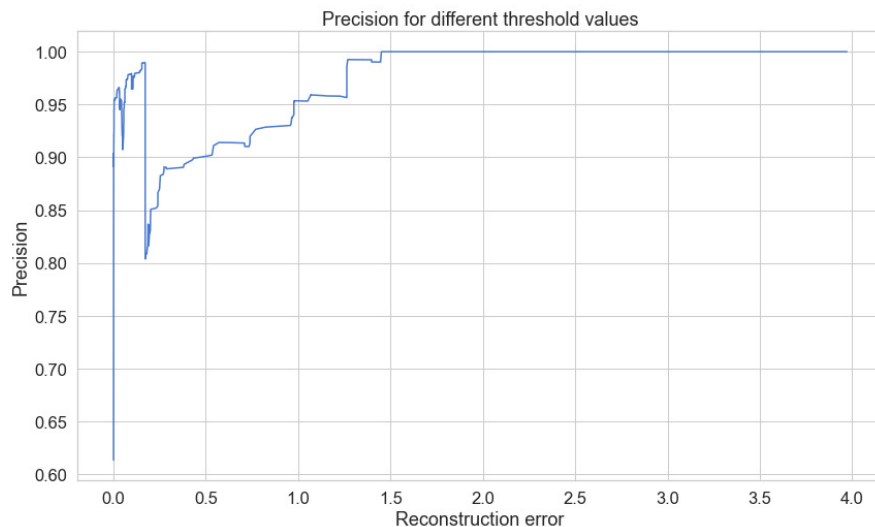


Figure 5.7 Precision over different threshold values

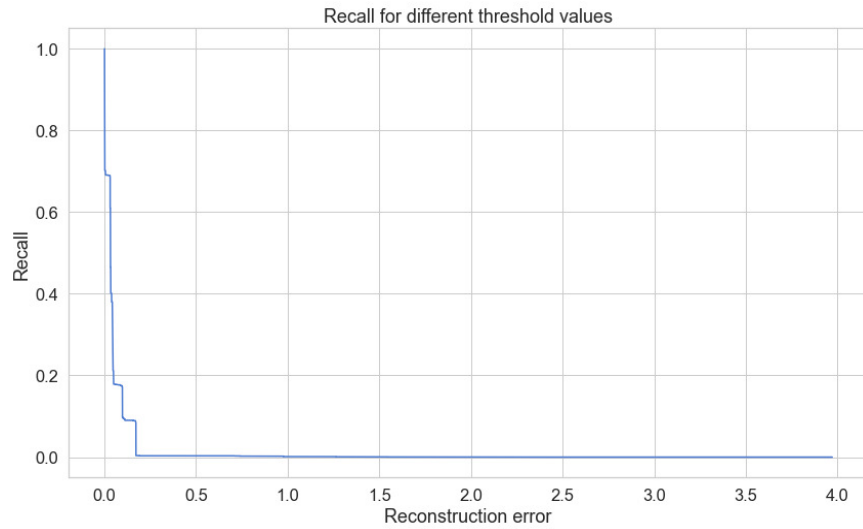


Figure 5.8 Recall over different threshold values

The Precision and Recall scores indicate that the model can reconstruct benign traffic successfully; hence, they yield low reconstruction error. Meanwhile, it cannot reconstruct malicious traffic successfully (high reconstruction error and False Negative rate) meaning that the model cannot predict new values. However, using the calculated reconstruction error from the network traffic data itself, unseen network traffic can be classified as benign or malicious. If the reconstruction error is larger than a predefined threshold, the network traffic can be classified as malicious.

Accordingly, a threshold must be defined so that it limits the number of False Negatives (misclassified malicious traffic) to a manageable degree and captures most of the malicious traffic. It is important to note that the consequences of misclassifying malicious traffic are considered worse than classifying benign traffic as malicious. Missing malicious traffic is far more expensive than incorrectly labelling benign traffic as such, hence, the focus was to reduce the number of misclassified malicious traffic. In order to determine the optimal threshold value, the model was tested over different reconstruction error values and for each value, Accuracy, Precision, Recall, F1-score and AUC ROC were calculated.

Initially, the model was tested with the mean of the reconstruction error for being traffic (0.003) as the threshold meaning that any sample that was reconstructed with an error greater than 0.003 is classified as malicious. Figure 5.9 shows a confusion matrix of the true classes and predicted classes for the test set using a threshold of 0.003. The model classified 95% of the benign traffic and only 57.5% of the malicious traffic correctly with an overall accuracy of 80%. The model scored Precision and Recall scores of 95% and 70% respectively indicating a high False Negative rate (42% of malicious traffic were misclassified). Table 5.5 summarises the performance metrics of the Autoencoder with a threshold of 0.003.

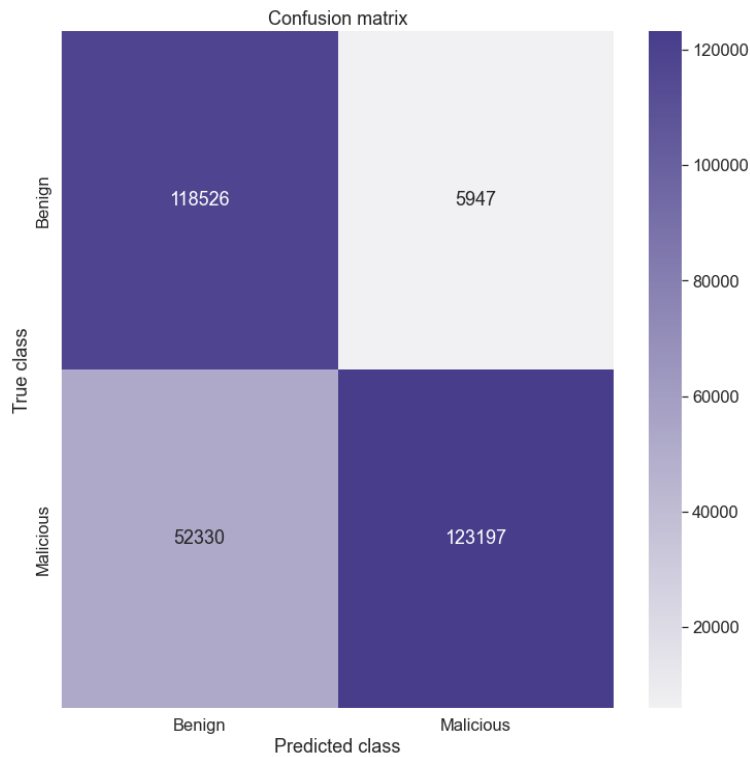


Figure 5.9 confusion matrix for a threshold of 0.003

Table 5.5. performance metrics for a threshold of 0.003

Accuracy	Precision	Recall	F1-score	AUC ROC
80%	95%	70%	80%	82%

Nonetheless, the model was tested with different threshold values in order to obtain better performance metrics. As mentioned previously, the aim is to limit the number of False Negatives which is obtained by scoring high Recall. According to Figure 5.8, the Recall increases as the reconstruction error decreases; hence, the model must be tested with values smaller than 0.003 to yield a higher Recall score. Accordingly, the model was tested with 30 threshold values ranging from 0 to 0.003.

5.2.4 Influence of threshold value on the proposed model

Accuracy: As previously mentioned, the aim is to capture most of the malicious traffic; hence, achieve a high accuracy score. Figure 5.10 shows the accuracy of the model over different threshold values. The highest Accuracy score was 89% and it was achieved by setting the threshold to 0.000724.

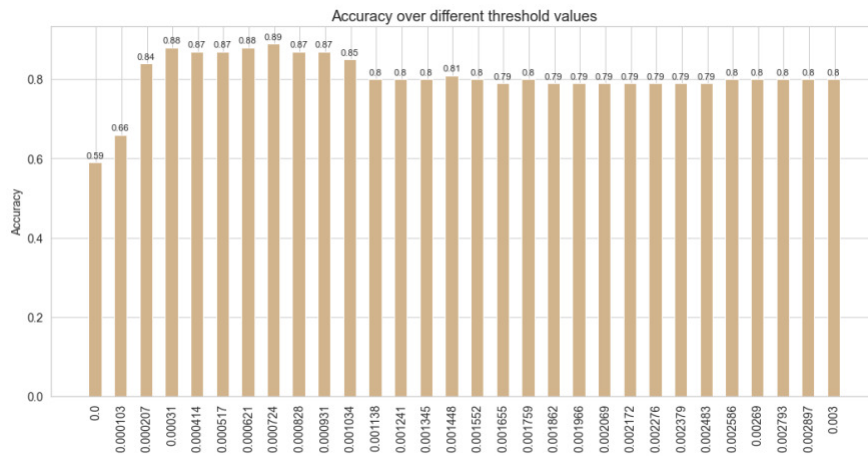


Figure 5.10 Accuracy for different threshold values

Recall: Figure 5.11 shows the model's Recall scores over different threshold values. The chart shows that the Recall decreases as the reconstruction error decreases. The highest achieved Recall is 1 with threshold values of 0 and 0.000103. Although it is important to have a high

Recall to minimise the rate of False Negatives, the optimal threshold cannot be determined by observing one performance metric.

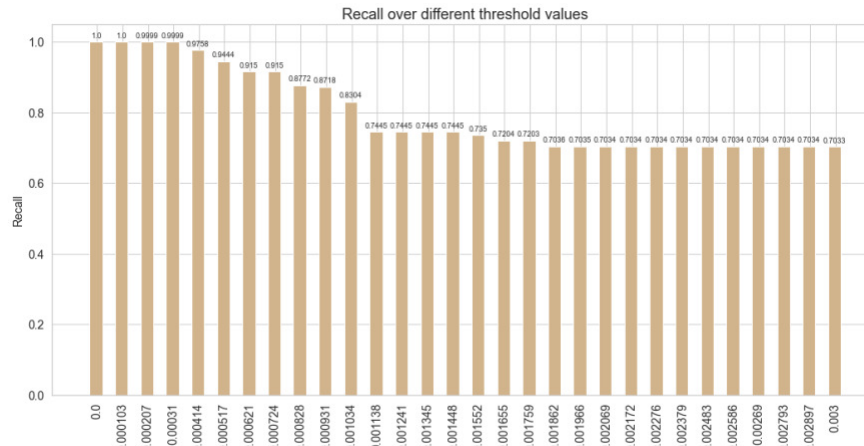


Figure 5.11 Recall for different threshold values

Precision: Figure 5.12 shows the model's Precision over different threshold values. The chart shows that the Precision increases as the reconstruction error increases. The model achieved its highest Precision of 95% with threshold values of 0.002586, 0.00269, 0.002793 and 0.002897. Unlike Accuracy and Recall, the Precision score improves with a higher threshold value. The model in this case is classifying benign traffic correctly and at the same time classifying a significant amount of malicious traffic as benign as well.

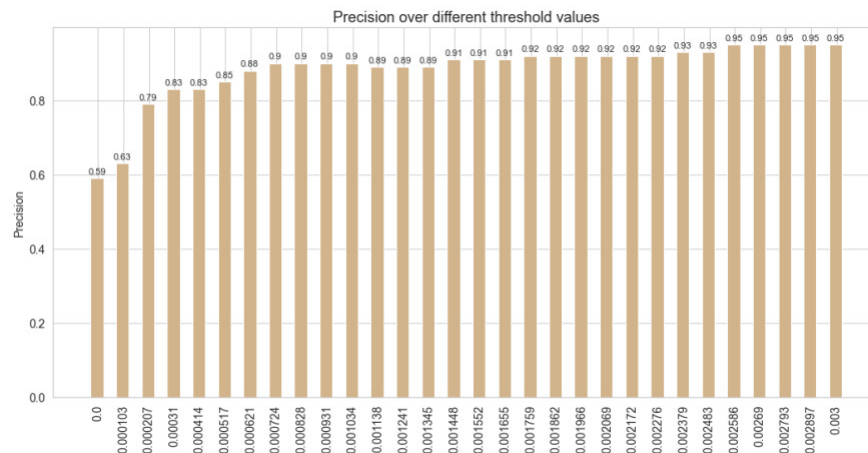


Figure 5.12 Precision for different threshold values

F1 score: F1 score is the harmonic mean of Precision and Recall. A high F1 score indicates low False Positives and False Negatives simultaneously. Figure 5.13 shows the model's F1 score over different threshold values. The model achieved its highest F1 score of 91% with the threshold ranging from 0.00031 to 0.000931, precisely with a threshold value of 0.000724.

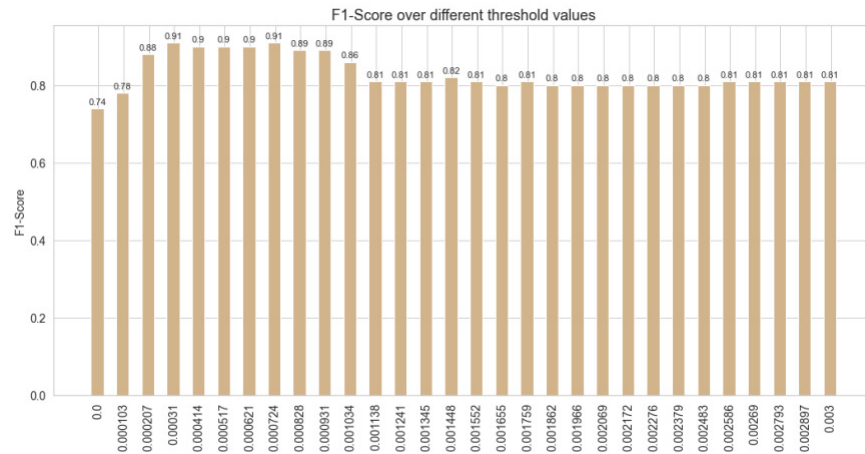


Figure 5.13 F1-Score for different threshold values

AUC ROC: Another performance metric that was used to evaluate the model is Area Under the ROC curve (AUC ROC). As can be seen in Figure 5.14, the highest achieved AUC is 89% with a threshold value of 0.000724.

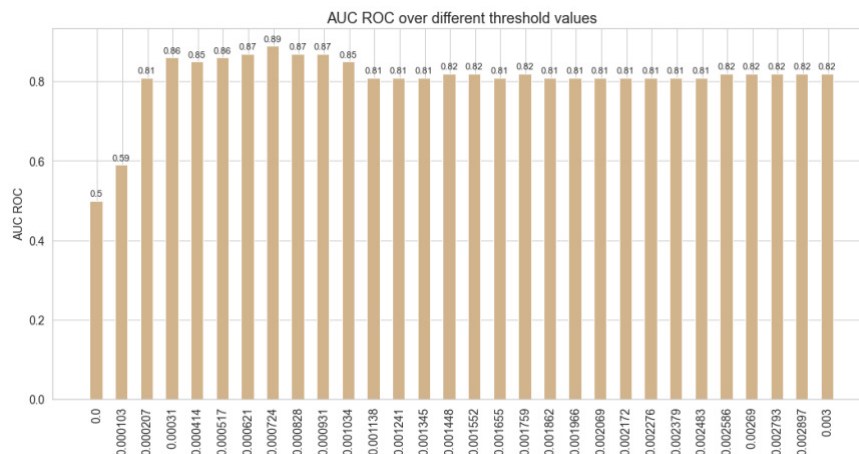


Figure 5.14 AUC ROC for different threshold values

With a threshold of 0.000724, the model achieved its highest Accuracy, F1 score and AUC ROC. Although it does not result in the highest Precision and Recall scores separately, it achieves the best Precision and Recall simultaneously with the F1 score. Figure 5.15 shows a confusion matrix of the true classes and predicted classes for the test set when the threshold is set to 0.000724. The model classifies 90% of the malicious traffic and 84% of benign traffic correctly. With a threshold value of 0.003, 42% of malicious traffic were misclassified. Whereas, with a threshold of 0.000724, only 10% were classified incorrectly indicating an improvement of the model. Table 5.6 summarises the performance metrics of the Autoencoder with a threshold of 0.000742.

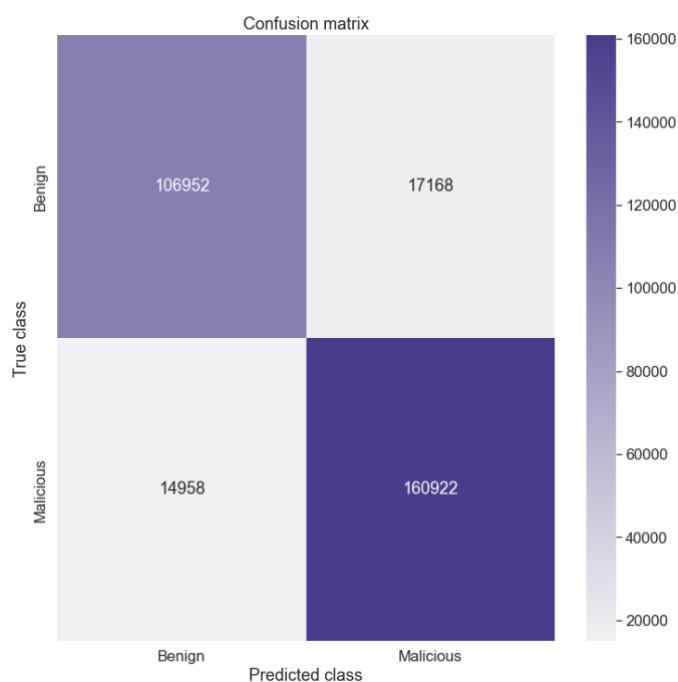


Figure 5.15 confusion matrix for a threshold of 0.000724

Table 5.6. performance metrics for a threshold of 0.000724

Accuracy	Precision	Recall	F1-score	AUC ROC
89%	90%	91%	91%	89%

In conclusion, according to the results and the performance metrics, an Autoencoder can be utilised in a malware detection system for IoT devices. Nonetheless, its performance can be improved with hyperparameter tuning and experimenting with the layers' structure. In this project, the focus was to experiment only with the threshold value and the model's performance showed noticeable improvement as the threshold value changed.

5.2.5 Dashboard - Autoencoder

As mentioned in the approach section, the dashboard only simulates the malware detection system and does not involve any machine learning training or testing in the backend. Yet, the data used to create the dashboard was retrieved as CSV files from the actual models that are implemented in this project. The first tab in the dashboard displays elements related to the Autoencoder. As can be seen in Figure 5.16, it displays all the traffic that are fed into the system as points where the green points mean benign traffic and red means malicious. It also stimulates the training process by displaying the model's Accuracy and Loss as the number of epochs increases.

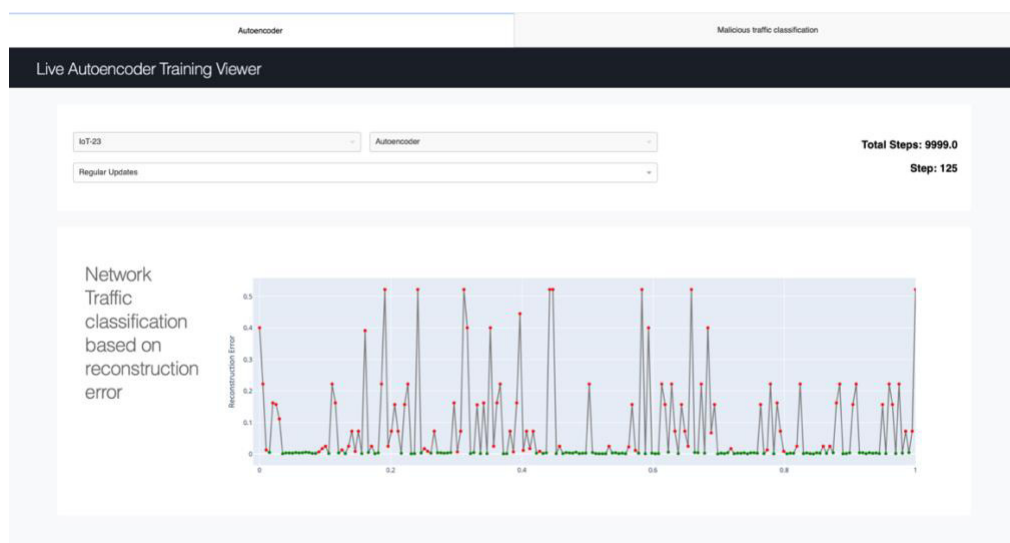


Figure 5.16 Autoencoder tab

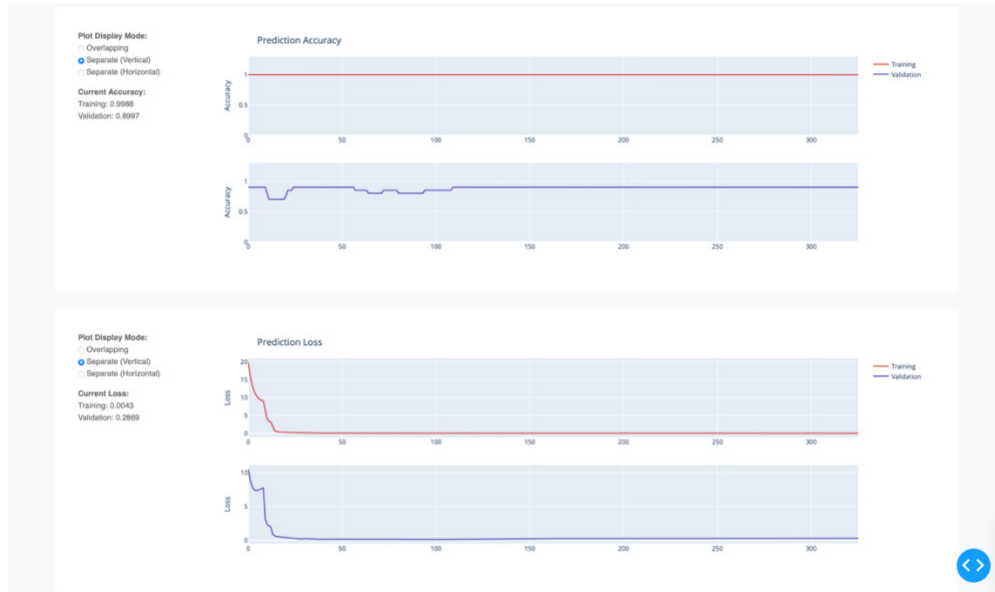


Figure 5.17 Autoencoder training simulation

5.3 Malware Clustering with K-means

As part of the malware detection system, the K-means algorithm was utilised. Its purpose is to group malicious traffic into different clusters based on the type of attack. For an actual system, it will cluster traffic classified as malicious by the Autoencoder. In order to simulate the process, K-means was tested and evaluated using the same dataset used for the Autoencoder to ensure that the hypothesis testing environment is controlled. The model was tested with 762,536 malicious traffic. A breakdown of the malicious traffic categories is illustrated in Figure 5.2. Three classes of malicious activity dominated the malicious traffic samples: DDoS attacks, Okiru botnets and port scans; with only a few samples composing each of the remaining two classes.

5.3.1 Elbow Method

As mentioned in the Background section, K-means clustering aims to group similar data samples in the form of clusters where the number of clusters must be predefined. Hence, the first step was to find the optimal number of clusters. As mentioned in the background section, the elbow

method is an algorithm that initialises a K-means model with a different number of clusters each time, and for each value, it calculates the WCSS (sum of squared distance). Using the Elbow Method, the chart in Figure 5.18 was generated. It shows the number of clusters against its corresponding WCSS. As can be seen in the chart, the elbow point is located at the point [3,1] meaning that the optimal number of clusters is 3.

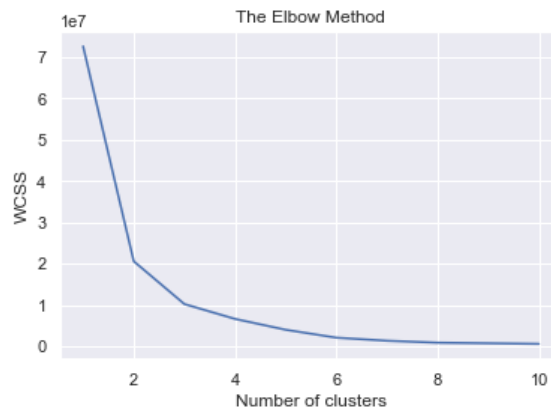


Figure 5.18 Elbow method

Figure 5.19 shows the clusters generated by the model. The samples are plotted based on the number of bytes being sent to the device (orig_ip_bytes) and the number of bytes being sent from the device (resp_ip_bytes) in each network flow.

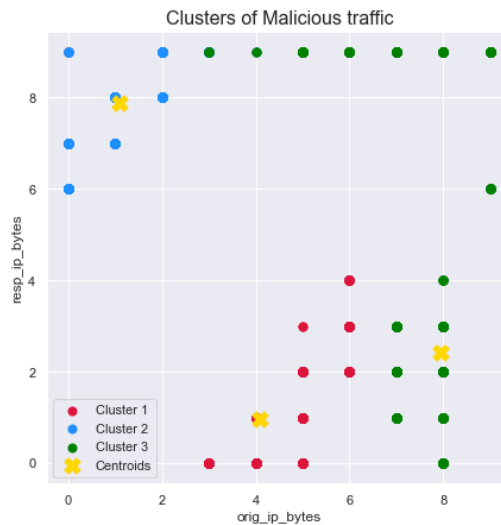


Figure 5.19 Clustering result

5.3.2 Influence of the number of clusters on the K-means model

K-means clustering is an unsupervised learning task meaning that it cannot be evaluated since using classification performance metrics. A dependent variable or label is required in any classification performance evaluation, whether it is a confusion matrix or log loss. Although labels are available for the IoT-23 dataset, K-means clusters the data without the labels meaning that the resulting clusters are not mapped to the actual labels. Hence, classification performance metrics still cannot be used, and clustering must be evaluated using different performance metrics.

The model's performance was evaluated using the following clustering performance metrics: Purity, Rand Index, Adjusted Rand Index, Mutual Information, Calinski-Harabaz Index and the Davies-Bouldin Index. Clustering is highly dependent on the predefined number of clusters; therefore, performance metrics were calculated for a total of 9 models where each model was initialised with a different number of clusters (k) ranging between 2 to 10 clusters.

Cluster analysis for $k=2$:

Initialising the model with $k = 2$ resulted in the highest Adjusted Rand Index with a value of 0.76. Figure 5.20 shows the distribution of malware within each cluster. DDoS attacks take up 98.6% of cluster 2 while port scans and Okiru botnets dominate cluster 1. The malware distribution within the clusters can be justified by the fact that Okiru botnets and port scans are considerably similar in terms of their features within the used dataset.

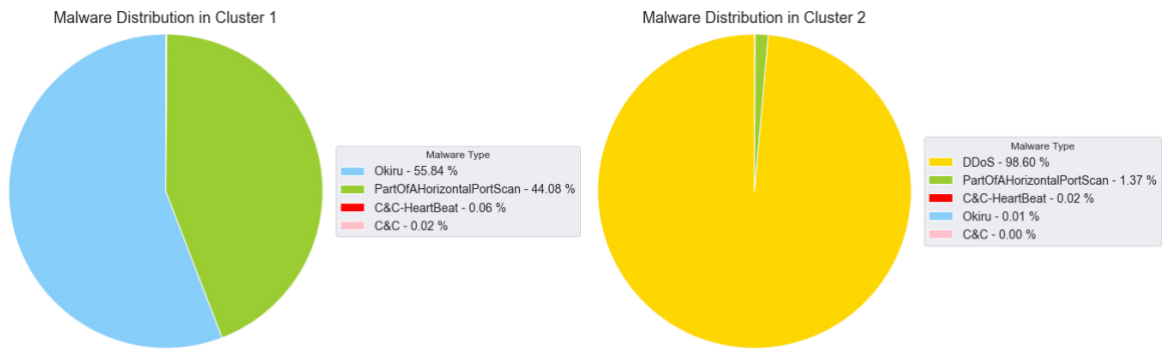


Figure 5.20 Malware distribution for k=2

Cluster analysis for k=3:

Initialising the model with $k = 3$ yielded the second highest Adjusted Rand Index and the best score of the Davies-Bouldin Index. Figure 5.21 shows the distribution of malware within each cluster. DDoS attacks and port scans dominate approximately 99% of clusters 2 and 3 respectively, while Okiru botnets take up 65.5% of cluster 1 followed by port scans (34.39%). As can be seen in Figure 5.21, C&C and C&C-HeartBeat take up a significantly small percentage of each cluster and are not clustered separately. The frequency of C&C and C&C-HeartBeat within the dataset is drastically low in comparison to the rest of the malicious traffic causing an imbalance in the dataset which affected the clustering result negatively.

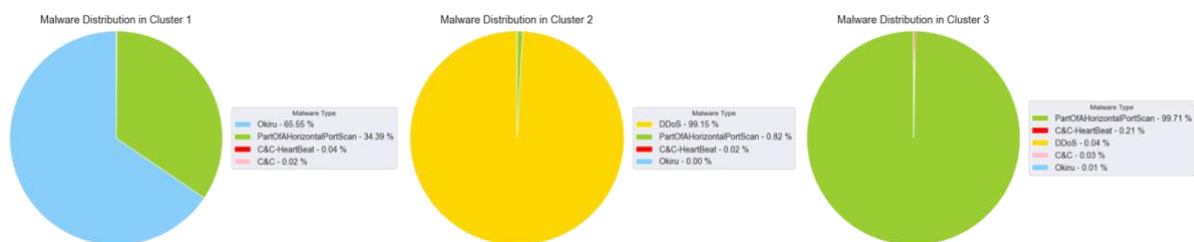


Figure 5.21 Malware distribution for k=3

Cluster analysis for $k=5$:

Since there are 5 types of attacks within the dataset, it is logical to expect that the optimal number of clusters should be 5 clusters. However, none of the performance metrics indicated that 5 is the ideal number of clusters. Hence, cluster analysis was necessary to check whether the model can cluster C&C and C&C-HeartBeat in separate clusters. Initialising the model with 5 clusters yielded the malware distribution illustrated in Figure 5.22. Similar to the previous case, DDoS attacks, port scans and Okiru botnets take up the majority of the clusters with DDoS attacks and port scans dominating two clusters each this time. As can be seen in Figure 5.22, the model failed to cluster C&C and C&C-HeartBeat correctly, rather, it separated DDoS attacks and port scans into two clusters each this time. Accordingly, it achieved a worse Rand Index, Adjusted Rand Index and Davies-Bouldin Index than K-means with $k=3$ since clusters are not significantly farther apart or less dispersed.



Figure 5.22 Malware distribution for $k=5$

Cluster analysis for $k=10$:

The highest achieved Purity, Mutual Information and Calinski Harabaz Index are 0.97, 0.95 and 10216306.97 respectively. They were achieved by setting k as 10. The Calinski-Harabasz index is often larger for convex clusters than for alternative cluster concepts, such as density-based clusters that can be acquired by the DBSCAN algorithm [59]. Additionally, it increases as the density and separation of clusters increase, hence, 10 yielded the highest score. Increasing the number of clusters also increases the Purity of the model [61]. Accordingly, it cannot be used as a trade-off between the number of clusters and the quality of clustering.

A summary of the model performance by the number of clusters is illustrated in Table 5.7

Table 5.7 summary of performance for different number of clusters

Number of clusters	Purity	Rand Index	Adjusted Rand Index	Mutual Information	Calinski-Harabasz Index	Davies-Bouldin Index
2	0.7855	0.8806	0.7684	0.6589	2288132.3991	0.4559
3	0.8577	0.9014	0.7682	0.7366	3393273.8243	0.2890
4	0.8576	0.7746	0.6517	0.7385	3542642.0773	0.4107
5	0.9646	0.8240	0.7169	0.8954	5136044.3660	0.4702
6	0.9656	0.7728	0.6614	0.9028	5689648.3370	0.4561
7	0.9638	0.7510	0.6343	0.9284	6656948.4408	0.4375
8	0.9684	0.7522	0.6431	0.9503	8685519.0776	0.5423
9	0.9684	0.7522	0.6422	0.9504	9154139.8397	0.4127
10	0.9701	0.7470	0.6280	0.9528	10186509.6364	0.4037

5.4 Malware Classification with Random Forest

The Random Forest classifier was used in this project because, when compared to other classification algorithms, it delivers the greatest prediction rate in network traffic datasets [7], [8], [10]. Additionally, it has an effective technique for estimating missing data, which results in a high level of predictive accuracy and maintains the correctness of the generalisation even when a large

portion of the data is missing. As previously mentioned, it can minimise overfitting and can handle datasets with many features.

As mentioned in the approach section, the purpose of implementing Random Forest is to classify traffic, identified by the Autoencoder as malicious, based on the different types of attacks. Same as for the K-means clustering, the model was trained and tested using the IoT-32 data set excluding all the benign traffic within the dataset. The model was trained with 70% of the data set and tested with the remaining 30%. The task is a multiclass classification since it requires classifying instances into one of 5 distinct classes; hence, performance metrics, excluding Accuracy and AUC ROC, were calculated for every class separately for the train and test sets. The mean of each metric was calculated to determine the model's overall performance (macro-averaging).

5.4.1 Performance metrics

Accuracy and AUC ROC: The model scored an overall Accuracy of 99.9% for both the train and test set indicating that the model classified 99.9% of the samples correctly. Furthermore, it archived an overall AUC ROC of 99.7% for both the train and test sets. The AUC ROC result implies that the model has a good measure of separability meaning that it can successfully distinguish between classes, consequently, classifying the samples correctly.

Precision: As can be seen in Figure 5.23, DDoS attacks, port scans and Okiru botnets scored a Precision of 100% for both train and test sets. The result indicates 0 False Positives for these

classes meaning that none of these malware classes was misclassified. However, C&C and C&C-HeartBeat attacks are predicted incorrectly by the model.

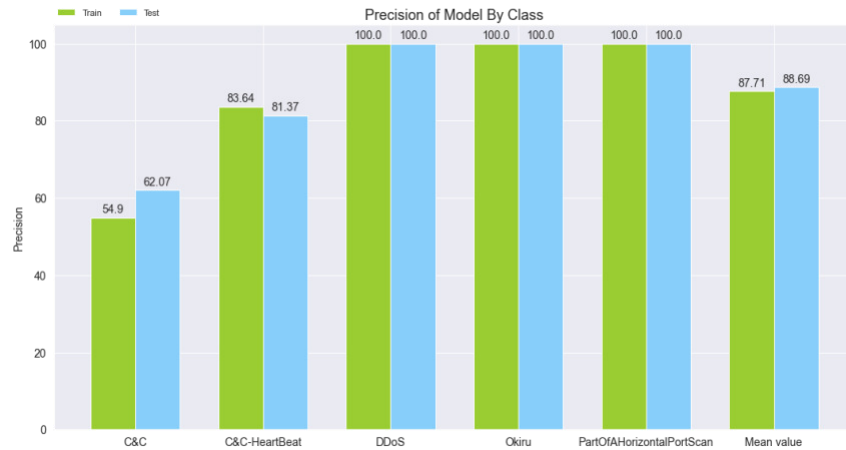


Figure 5.23 Precious of random forest

Figures 5.24 show confusion matrices of the model's prediction on the train and test sets respectively. According to the confusion matrices, C&C and C&C-HeartBeat are often misclassified as port scans. This can be justified by the fact that features of C&C and C&C-HeartBeat samples, such as orig_bytes, resp_bytes, duration and orig_ip_bytes, are similar in value to port scans. As mentioned previously, there is an imbalance in the data set since C&C and C&C-HeartBeat have very low proportions in comparison to the rest of the classes meaning that there is an inadequate amount of data for the model to learn from. Consequently, the model fails to classify C&C and C&C-HeartBeat correctly.

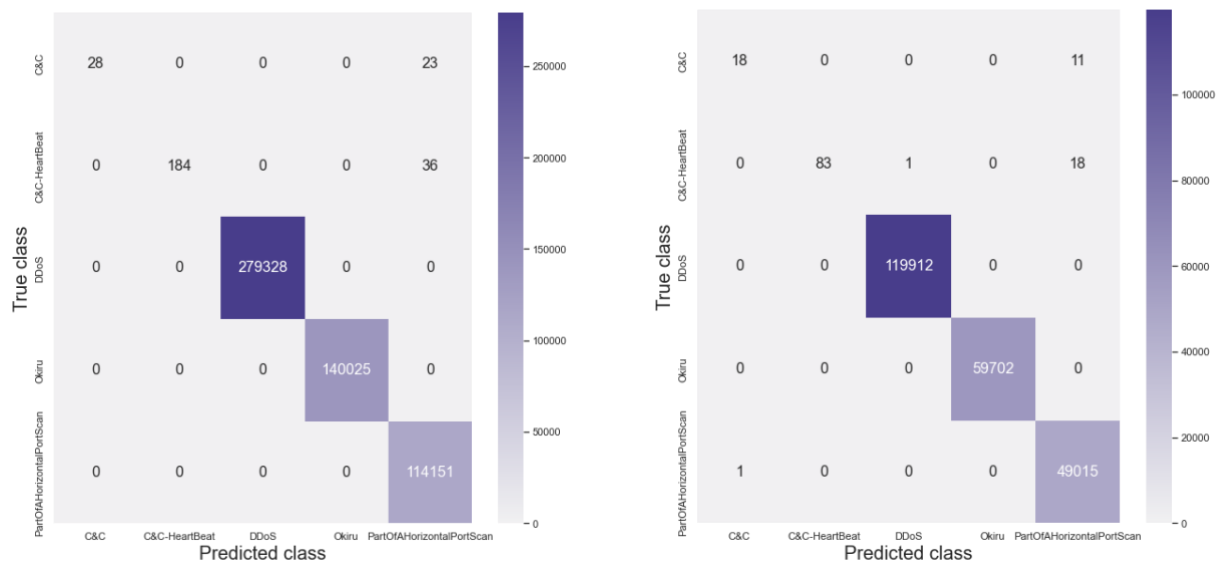


Figure 5.24 Confusion matrix for random forest (left-train/right-test)

Recall: Figure 5.25 shows the model's Recall for different classes. C&C attacks scored a Recall of 94.74% for the test set whereas it is 100% for the train set. Although there is only one sample that was misclassified as a C&C attack, as shown in Figure 5.24, the data imbalance resulted in a low Recall for the said class. As previously mentioned, C&C and C&C-HeartBeat are mostly misclassified as port scans; hence, they increase the False Negative rate for ports scan. However, as can be seen in Figure 5.25 ports scan scored a Recall of 99.95% and 99.94% for the train and test sets respectively. The rate of False Negatives did not seem to affect the Recall for this class due to the high proportion of port scans in the dataset. The model scored an overall Recall of 99.99% and 96.94% for the train and test sets respectively indicating a low False Negative rate generally.

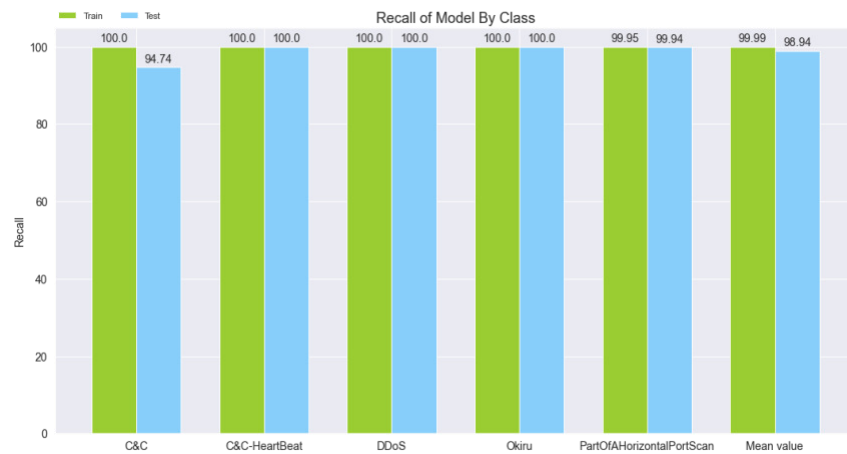


Figure 5.25 Recall of random forest

F1-score: The F1-score is the harmonic mean of Precision and Recall. As can be seen in Figure 5.26, the model's mean F1-score is 92.39% for the train set and 92.94% for the test set. A high F1-score indicates low False Positives and False Negatives simultaneously. Since C&C and C&C-HeartBeat scored low Precision and Recall, the overall precision, Recall and F1 score were negatively impacted.

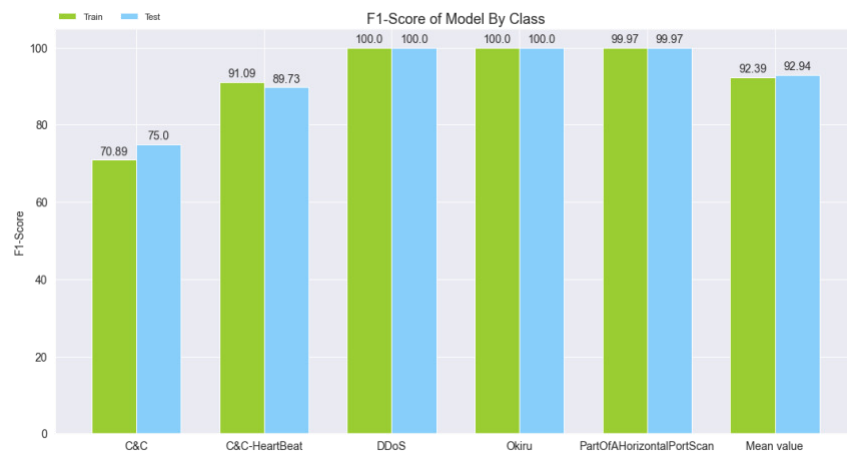


Figure 5.26 F1-Score of random forest

A summary of the model performance by class is illustrated in Table 5.8.

Table 5.8 summary of classifier performance

	Accuracy - Train	Accuracy - Test	AUC ROC - Train	AUC ROC - Test	Precision - Train	Precision - Test	Recall - Train	Recall - Test	F1-Score - Train	F1-Score - Test
C&C	-	-	-	-	54.90	62.07	100.00	94.74	70.89	75.00
C&C-HeartBeat	-	-	-	-	83.64	81.37	100.00	100.00	91.09	89.73
DDoS	-	-	-	-	100.00	100.00	100.00	100.00	100.00	100.00
Okiru	-	-	-	-	100.00	100.00	100.00	100.00	100.00	100.00
PartOfAHorizontalPortScan	-	-	-	-	100.00	100.00	99.95	99.94	99.97	99.97
Mean value	99.995579	99.994579	99.723451	99.700058	87.71	88.69	99.99	98.94	92.39	92.94

5.4.2 Dashboard – malicious traffic clustering and classification

The second tab on the dashboard displays elements related to clustering and classification. After the Autoencoder classifies traffic as benign or malicious, K-means and Random Forest will further cluster and classify the malicious traffic according to its type. As can be seen in Figure 5.27, clustering results are displayed as a chart, and it shows the malware groups where each cluster is coloured differently. However, based on the clusters alone, the type of malware cannot be determined. That is why a supervised model is included. Using a pie chart, the system displays the malware types and their distribution within the traffic that were fed to the system.



Figure 5.27 classification and clustering tab

5.5 Summary of the results

Autoencoder: The Autoencoder was trained on benign traffic only with a batch size of 32 for 100 epochs. When tested with malicious traffic the model yielded high reconstruction error since it can only correctly predict data similar to what it was trained on. Therefore, the reconstruction error was used as a threshold to distinguish between malicious and benign traffic where data samples that yield a reconstruction error higher than the threshold are considered malicious. The model was tested, and performance metrics were calculated for different threshold values. The best performance metrics were achieved by setting the threshold to 0.000742.

Clustering with K-means: Since the K-means is an unsupervised learning algorithm, it cannot be simply evaluated using classification performance metrics. Hence, different evaluation methods were used to evaluate the model. According to the evaluation result, the model successfully clustered port scans, Okiru botnets and DDoS attacks. Yet, it failed to group C&C and C&C-HeartBeat in separate clusters. This was justified by the imbalance of data which was caused by the low proportion of these two classes in the data set.

Classification with Random Forest: The Random Forest model was trained and tested with malicious traffic only. The model scored an overall Accuracy of 99.99%. However, the model failed to classify C&C and C&C-HeartBeat correctly and they were mostly classified as port scans instead. Consequently, the rate of False Positive for the C&C and C&C-HeartBeat classes and the False Negatives for the port scans class increased. In addition to the imbalance in the

data classes and the inadequate amount of data, the result was justified by the fact that the C&C and C&C-HeartBeat samples within the used dataset had similar features to port scans.

5.6 Limitations

Due to the time constraints of this project, there were some obstacles and challenges faced and implementation decisions had to be made to ensure the successful achievement of the main aims of this project. One limitation is the dataset size and class distribution within it. Prior to the training of the Autoencoder, the dataset was not only split into train and test sets, but malicious samples were also removed from the train set limiting the number of samples for the Autoencoder to learn from. As previously mentioned, the distribution of malware classes within the dataset was significantly imbalanced. There was an overabundance of DDoS attacks with respect to the rest of the malware categories which influenced the training and testing of K-means and Random Forest negatively. Another challenge concerns the labelling process of the dataset. Since the dataset was manually labelled there is a possibility that incorrect or erroneous labelling had occurred; thus, jeopardizing the validity of the results and potentially the entire scheme. Furthermore, the recentness of the training dataset is an important aspect. Malware evolves on a daily basis; hence the prototype's practical use can only be evaluated in real time. Finally, the project focused on building and evaluating only three machine learning models that were chosen based on popularity within the malware detection field. Each model was implemented with the intention to solve a specific problem and achieve a certain goal. However, due to the time limit, no comparison with other models or algorithms that could be used to solve the same problem was included. Consequently, the discussion of the results and evaluation were limited to only three machine learning models.

CHAPTER 6

Future Work

The aim of this project was to implement three machine learning models and experiment with two malware detection concepts followed by an implementation of a simulation of a potential malware detection system. The research results indicate that the proposed solution has the capability to solve the proposed problem. However, there are more avenues to be explored that could improve the results achieved in this project and further contribute to the IoT security field, precisely, the field of malware detection. This section will discuss possible approaches that could enhance the quality of the research

6.1 Anomaly detection with an Autoencoder

Due to the significant computational cost required to build and train an Autoencoder, the model was trained with only 290,463 samples limiting the learning opportunities for the Autoencoder. Hence, increasing the size of the dataset would significantly increase the learning quality. Furthermore, no hyperparameter tuning was conducted since the main goal was to build a simple model and test the hypothesis using it. The values for certain parameters were chosen only because they were the widely accepted default values. Although the results indicate the validity of the hypothesis, building a more advanced model could improve the performance quality. Hence, investing time in testing the model with different parameters in order to find the optimal parameters that would minimise the reconstruction error could be a potential development strategy.

6.2 Malware Clustering and classification

As previously mentioned, there was an imbalance distribution of the malware classes within the malicious samples. Due to the time constraint, it was not possible to solve this issue

properly and the dataset was used as it is. This issue could have been addressed and solved using cross-validation and k-folds cross validation methods rather than the traditional train-test split method. Furthermore, K-means and random forest were utilised to classify and categorise malicious traffic based on the attack type. These algorithms were chosen based on popularity and results of previously conducted research on the same problem. Therefore, experimenting with different clustering and classification algorithms such as Naive Bayes, Support Vector Machine and DBSCAN, could increase the validity of this research and potentially find better models that could achieve the same goals with better performance metrics. Moreover, K-means was evaluated using a variety of performance metrics to find the optimal number of clusters excluding silhouette analysis. The method was avoided in this project due to its high computational cost, especially for large datasets. Silhouette analysis, when compared to the Elbow approach, can be used to evaluate the separation distance between the generated clusters. It also offers the added benefit of detecting outliers in a cluster. However, it can be used to further improve the clustering quality of the model.

6.3 Dashboard

This solution was developed for the purposes of Scientific Research to examine if the proposed strategy is appropriate for addressing the project's problem. Due to the limited time frame and the computational cost, the major goal of this research was to investigate the proposed solution and provide a good starting point rather than a fully functional software. Hence, no machine learning training or models was present in the backend of the dashboard. However, using the outcomes of this research, an actual user interface that integrates the three models can be implemented and used as a fully functioning malware detection system for IoT devices.

CHAPTER 7

Conclusion

The aim of the project was to develop machine learning models that can be integrated to build a malware detection system for IoT devices. It was never the intention to aim for a specific result or baseline, but rather a proof of concept that integrating these models can be used for malware detection. An Autoencoder, K-means and random forest models were successfully developed. The results suggest that an Autoencoder trained with only benign traffic yields a higher reconstruction error when tested with malicious traffic than benign traffic. Therefore, the reconstruction error can be used as a threshold value to distinguish between malicious and benign traffic. The model was tested with a range of threshold values and setting it at 0.000742 resulted in the highest Accuracy (89%), F1-score (91%) and AUC ROC (89%). Following the successful implementation of the Autoencoder, K-means and Random Forest models were built to further classify and categorise traffic, classified by the Autoencoder as malicious, based on the different types of attacks. Both models showed better performance on Okiru botnets, port scans and DDoS attacks whereas they failed to cluster and classify C&C and C&C-HeartBeat attacks and they were often classified as port scans. This was justified by the fact that C&C and C&C-HeartBeat attacks were significantly low in proportion in comparison to the rest of the malware classes within the dataset in addition to the fact these attacks have similar features to the port scans within the dataset. Nonetheless, it did not impact the overall performance significantly; yet, imperfections could have been avoided by using better data splitting methods. In conclusion, the results indicate that these models have the ability to detect malicious traffic and further classify them based on attack types. Hence, with further improvements, such as hyperparameter tuning and better sampling of the dataset, they can be utilised to build a fully functioning malware detection system for IoT devices.

CHAPTER 8

Reflection on Learning

The final year project has been the most challenging academic task so far. During this project, I became aware of the considerable level of discipline and dedication required to complete it.

This project piqued my interest because it was related to machine learning. Despite the fact that I had not previously had the opportunity to study this topic in depth, I was eager to learn more about it. I am usually motivated by new challenges because I believe they allow me to grow and educate myself. Another important motivation for selecting a Data Science-related project was my desire to learn more about this field of computer science before deciding if it is something I want to pursue further at a higher level. This experience motivated me to consider applying for a postgraduate degree in Data Science.

Although I had minimal experience in creating Machine Learning models and almost no knowledge of the term Artificial Neural Networks prior to this project, I was able to expand my knowledge in the field of Machine Learning and Deep Learning while working on this project. During this project, I focused on building an Artificial Neural Network and two machine learning models. I was able to build, test and evaluate these models while also exploring different approaches and models. However, I feel like I have not experimented enough with the concept Artificial Neural Networks and that there is so much more to learn about Deep Learning models. I believe that this experience would be significantly helpful if I decide to pursue a Data Science profession. Although in this project I focused on the Data Science aspect of the solution, I was able to improve my technical and coding skills while developing the dashboard. I experimented with multiple frameworks for creating Graphical user interfaces and simple web applications

before deciding to implement the final software with Dash. This allowed me to recognise my strengths and weaknesses in terms of coding and what aspects I should be focusing on for the future.

The timeline specified in the initial report assisted me in staying on track. As someone who has previously worked with an Agile methodology, it was logical for me to try to adopt comparable ideas to this project. Many people prefer to read through extensive documentation before implementing and testing, but I have discovered that starting the implementation as soon as possible and learning as I go is the most suitable approach for me. This way I would be able to start developing earlier while also gaining a deeper understanding of the subject. However, there were occasions when I had to sacrifice a significant amount of my personal time in order to meet the project's objectives. Task prioritising and time management were two of the most essential skills I learned while developing this project. When the workload began to increase, I had to prioritise the tasks in order for the project to continue successfully. Furthermore, because the entire project was dependent on a single person, time management and self-motivation were critical. Furthermore, I was able to strengthen my self-discipline by completing such a large project on time. Rather than rushing to finish work close to the deadline, I made sure to work consistently throughout the semester. Consequently, I was left with enough time near the end of the semester to work on the report without feeling rushed.

In conclusion, the most beneficial aspect of working on this project has been the expansion of my knowledge in a subject that I am highly interested in researching. Most importantly, I was able to learn a lot from this project while enjoying it rather than working on it for the sake of the degree. I am optimistic that I will apply my newly acquired knowledge and expertise in the near future.

CHAPTER 9

References

- [1] Analytics Vidhya. 2019. K Means Clustering | K Means Clustering Algorithm in Python. [online] Available at: <<https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-K-means-clustering/>> [Accessed 12 March 2022].
- [2] Anthi, E., Williams, L., Slowinska, M., Theodorakopoulos, G. and Burnap, P., 2019. A Supervised Intrusion Detection System for Smart Home IoT Devices. IEEE Internet of Things Journal, 6(5), pp.9042-9053.
- [3] Austin, M., 2021. IoT Malicious Traffic Classification Using Machine Learning.
- [4] Awad, M. and Khanna, R., 2015. Efficient learning machines. [New York, NY]: Apress.
- [5] Bains, J., Goyal, R., Kopanati, H. and Savaram, B., 2021. Machine learning-IoT 23. Concordia University of Edmonton.
- [6] Bajaj, A., 2022. Performance Metrics in Machine Learning [Complete Guide] - neptune.ai. [online] neptune.ai. Available at: <<https://neptune.ai/blog/performance-metrics-in-machine-learning-complete-guide>> [Accessed 8 May 2022].
- [7] Berry, M., Mohamed, A. and Yap, B., 2020. Supervised and Unsupervised Learning for Data Science.
- [8] Binieli, M., 2022. Machine learning: an introduction to mean squared error and regression lines. [online] freeCodeCamp. Available at: <<https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-line-c7dde9a26b93/>> [Accessed 21 March 2022].
- [9] Breiman, L., 2001. Machine Learning, 45(1), pp.5-32.
- [10] Brownlee, J., 2014. A Gentle Introduction to Scikit-Learn. [online] Machine Learning Mastery. Available at: <<https://machinelearningmastery.com/a-gentle-introduction-to-scikit-learn-a-python-machine-learning-library/>> [Accessed 12 April 2022].
- [11] CloudxLab. 2022. NumPy and Pandas Tutorial - Data Analysis with Python. [online] Available at: <<https://cloudxlab.com/blog/numpy-pandas-introduction/>> [Accessed 16 March 2022].
- [12] Dertat, A., 2022. Applied Deep Learning - Part 3: Autoencoders. [online] Medium. Available at: <<https://towardsdatascience.com/applied-deep-learning-part-3-Autoencoders-1c083af4d798>> [Accessed 17 March 2022].
- [13] Docs.zeeq.org. 2022. base/protocols/conn/main.zeeq — Book of Zeek (git/master). [online] Available at: <<https://docs.zeeq.org/en/master/scripts/base/protocols/conn/main.zeeq.html>> [Accessed 8 February 2022].
- [14] Education, I., 2020. What is Random Forest?. [online] IBM.com. Available at: <<https://www.ibm.com/cloud/learn/random-forest>> [Accessed 8 May 2022].
- [15] EduPristine. 2022. Machine Learning Methods: K-means Clustering Algorithm. [online] Available at: <<https://www.edupristine.com/blog/beyond-K-means>> [Accessed 12 March 2022].
- [16] Gandhi, R., 2018. K-means Clustering — Introduction to Machine Learning Algorithms. [online] Medium. Available at: <<https://towardsdatascience.com/K-means-clustering-introduction-to-machine-learning-algorithms-c96bf0d5d57a>> [Accessed 12 March 2022].
- [17] Garcia, F. and Muga, F., 2016. Random Forest for Malware Classification. Ateneo de Manila University.
- [18] GeeksforGeeks. 2021. Difference Between Matplotlib VS Seaborn. [online] Available at: <<https://www.geeksforgeeks.org/difference-between-matplotlib-vs-seaborn/>> [Accessed 10 March 2022].

- [19] GeeksforGeeks. 2022. Clustering in Machine Learning. [online] Available at: <<https://www.geeksforgeeks.org/clustering-in-machine-learning/?ref=lbp>> [Accessed 19 February 2022].
- [20] GeeksforGeeks. 2022. ML | Determine the optimal value of K in K-means Clustering. [online] Available at: <<https://www.geeksforgeeks.org/ml-determine-the-optimal-value-of-k-in-K-means-clustering/>> [Accessed 28 February 2022].
- [21] GeeksforGeeks. 2022. Supervised and Unsupervised learning. [online] Available at: <<https://www.geeksforgeeks.org/supervised-unsupervised-learning/>> [Accessed 10 February 2022].
- [22] Girgin, S., 2019. K-means Clustering Model in 6 Steps with Python. [online] Medium. Available at: <<https://medium.com/@sametgirgin/K-means-clustering-model-in-6-steps-with-python-dfe95e5a5fac>> [Accessed 14 March 2022].
- [23] Goodfellow, I., Bengio, Y. and Courville, A., 2017. Deep learning. Cambridge, MA: The MIT Press, pp.493-523.
- [24] Gray, S., 2020. What is the Agile Methodology in Software Development?. [online] Medium. Available at: <<https://serenagray2451.medium.com/what-is-the-agile-methodology-in-software-development-c93023a7eb85>> [Accessed 16 March 2022].
- [25] HaddadPajouh, H., Dehghantanha, A., Khayami, R. and Choo, K., 2018. A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting. Future Generation Computer Systems, 85, pp.88-96.
- [26] Hasan, M., Islam, M., Zarif, M. and Hashem, M., 2019. Attack and anomaly detection in IoT sensors in IoT sites using machine learning approaches. Internet of Things, 7, p.100059.
- [27] Hegde, M., Kepnang, G., Al Mazroei, M., Chavis, J. and Watkins, L., 2020. Identification of Botnet Activity in IoT Network Traffic Using Machine Learning. 2020 International Conference on Intelligent Data Science Technologies and Applications (IDSTA).
- [28] Hubens, N., 2022. Deep inside: Autoencoders. [online] Medium. Available at: <<https://towardsdatascience.com/deep-inside-Autoencoders-7e41f319999f>> [Accessed 25 March 2022].
- [29] Hussain, F., Hussain, R., Hassan, S. and Hossain, E., 2020. Machine Learning in IoT Security: Current Solutions and Future Challenges. IEEE Communications Surveys & Tutorials, 22(3), pp.1686-1721.
- [30] IBM Education. 2022. What is Deep Learning?. [online] Available at: <<https://www.ibm.com/cloud/learn/deep-learning>> [Accessed 19 February 2022].
- [31] Jain, A. and Dubes, R., 1988. Algorithms for clustering data. Englewood Cliffs, NJ: Prentice Hall.
- [32] Jo, T., 2021. Machine learning foundations. Springer International Publishing.
- [33] Jupyter.org. 2022. Project Jupyter. [online] Available at: <<https://jupyter.org>> [Accessed 8 May 2022].
- [34] Kaspersky. 2022. Kaspersky Cyber Security Solutions for Home & Business. [online] Available at: <<https://www.kaspersky.co.uk>> [Accessed 9 December 2021].
- [35] Kelleher, J., 2019. Deep learning.
- [36] Keras. 2022. Keras documentation: About Keras. [online] Available at: <<https://keras.io/about/>> [Accessed 14 April 2022].
- [37] Kim, J., Shim, M., Hong, S., Shin, Y. and Choi, E., 2020. Intelligent Detection of IoT Botnets Using Machine Learning and Deep Learning. Applied Sciences, 10(19), p.7009.
- [38] Malathi, C. and Padmaja, I., 2021. Identification of cyber attacks using machine learning in smart IoT networks. Materials Today: Proceedings.
- [39] Meng, Y., Liang, J., Cao, F. and He, Y., 2018. A new distance with derivative information for functional K-means clustering algorithm. Information Sciences, 463-464, pp.166-185.

- [40] Mohajon, J., 2020. Confusion Matrix for Your Multi-Class Machine Learning Model. [online] Medium. Available at: <<https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826>> [Accessed 8 May 2022].
- [41] Muthu.co. 2022. Mathematics behind K-Mean Clustering algorithm. [online] Available at: <<https://muthu.co/mathematics-behind-k-mean-clustering-algorithm/>> [Accessed 13 March 2022].
- [42] Najafabadi, M., 2017. MACHINE LEARNING ALGORITHMS FOR THE ANALYSIS AND DETECTION OF NETWORK ATTACKS. Ph.D. Florida Atlantic University.
- [43] Oracle. 2022. What is the Internet of Things (IoT)?. [online] Available at: <<https://www.oracle.com/uk/internet-of-things/what-is-iot/>> [Accessed 9 December 2021].
- [44] scikit-learn. 2022. sklearn.cluster.KMeans. [online] Available at: <<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>> [Accessed 8 April 2022].
- [45] scikit-learn. 2022. sklearn.ensemble.RandomForestClassifier. [online] Available at: <<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>> [Accessed 17 April 2022].
- [46] Scribd. 2022. OU2 Difference Between ML DL AI. [online] Available at: <<https://www.scribd.com/document/491836312/OU2-Difference-Between-ML-DL-AI>> [Accessed 19 February 2022].
- [47] Sebastian Garcia, Agustin Parmisano, & Maria Jose Erquiaga. (2020). IoT-23: A labeled dataset with malicious and benign IoT network traffic (Version 1.0.0) [Data set]. Zenodo. <http://doi.org/10.5281/zenodo.4743746>
- [48] Sha, K., Wei, W., Andrew Yang, T., Wang, Z. and Shi, W., 2018. On security challenges and open issues in Internet of Things. Future Generation Computer Systems, 83, pp.326-337.
- [49] Shafiq, M., Tian, Z., Sun, Y., Du, X. and Guizani, M., 2020. Selection of effective machine learning algorithm and Bot-IoT attacks traffic identification for internet of things in smart city. Future Generation Computer Systems, 107, pp.433-442.
- [50] Shalev-Shwartz, S. and Ben-David, S., 2009. Understanding Machine Learning.
- [51] Sinaga, K. and Yang, M., 2020. Unsupervised K-means Clustering Algorithm. IEEE Access, 8, pp.80716-80727.
- [52] Sokolova, M. and Lapalme, G., 2009. A systematic analysis of performance measures for classification tasks. Information Processing & Management, 45(4), pp.427-437.
- [53] Stoian, N., 2020. Machine Learning for Anomaly Detection in IoT networks: Malware analysis on the IoT-23 Data set. BS. University of Twente.
- [54] TensorFlow. 2022. Writing your own callbacks | TensorFlow Core. [online] Available at: <https://www.tensorflow.org/guide/keras/custom_callback> [Accessed 23 April 2022].
- [55] Tomar, A., 2021. Dash for Beginners: Create Interactive Python Dashboards. [online] Medium. Available at: <<https://towardsdatascience.com/dash-for-beginners-create-interactive-python-dashboards-338bfc6b6ffa4>> [Accessed 12 April 2022].
- [56] Wei, H., 2020. How to measure clustering performances when there are no ground truth?. [online] Medium. Available at: <<https://medium.com/@haataa/how-to-measure-clustering-performances-when-there-are-no-ground-truth-db027e9a871c>> [Accessed 15 April 2022].
- [57] Wood, T., 2022. Random Forests. [online] DeepAI. Available at: <<https://deepai.org/machine-learning-glossary-and-terms/random-forest>> [Accessed 8 May 2022].

- [58] Yang, M., Chang-Chien, S. and Nataliani, Y., 2018. A Fully-Unsupervised Possibilistic C-Means Clustering Algorithm. *IEEE Access*, 6, pp.78308-78320.
- [59] Yıldırım, S., 2022. Evaluation Metrics for Clustering Models. [online] Medium. Available at: <<https://towardsdatascience.com/evaluation-metrics-for-clustering-models-5dde821dd6cd>> [Accessed 8 May 2022].
- [60] Zeadally, S. and Tsikerdekis, M., 2019. Securing Internet of Things (IoT) with machine learning. *International Journal of Communication Systems*, 33(1).
- [61] Zuccarelli, E., 2021. *Performance Metrics in Machine Learning—Part 3: Clustering*. [online] Medium. Available at: <<https://towardsdatascience.com/performance-metrics-in-machine-learning-part-3-clustering-d69550662dc6>> [Accessed 13 April 2022].