Creating a ML model based on network activity to detect attacks from a malicious web server

Morgan Rogers

C1725806

MSc Cybersecurity Dissertation

Supervisor: Amir Javed

Cardiff University

School of Computer Science and Informatics

Table of Contents

Abstract

Malware has been a constant threat to network security, which has only grown within the backdrop of the Covid-19 pandemic. The prevalence of teleworking has provided new challenges to cybersecurity, in addition to techniques to deploy malicious payloads have become more sophisticated with Advanced Persistent Threats such as nation-states (or state-sponsored groups) participating in cyber attacks. Companies, governments, and NGOs with proper risk management strategies will likely implement solutions such as intrusion detection systems (IDS) or intrusion prevention systems (IPS) in an attempt to mitigate the risk of malware infecting their networks, however, some choose not to implement them, or some not properly maintaining these systems. IDS / IPS can be limited in effectiveness, not having its desired effect. IDS / IPS can regularly give too many false negatives or false positive results in regards to detecting malicious network traffic. Machine learning can provide a solution as a lightweight and accurate method for detecting malicious activity within network traffic. Models can be trained using a dataset, or multiple datasets, of existing network traffic packets containing malware activity that can determine malicious activity in current network traffic. This project will look at four different machine learning classification models that can detect malware in network traffic, evaluating and comparing the performance of each model to determine the most suitable algorithm for detecting malicious network activity.

Acknowledgements

Firstly, I would like to thank Amir Javed, my project supervisor, for his guidance during this project. This project was my first ever endeavour into machine learning and his support during this has been second to none and I cannot thank him enough for his help. Secondly, I would like to thank my family, Philip, Sally-Anne, Cerys, Emily, Ryan, Evan and Ioan. These past few months have not been the easiest and I couldn't have completed this dissertation without you. Your loving support has really meant the world to me, and I hope to keep doing you proud in the future. Finally, I would like to thank my mates for helping me out when times were tough. It has been a long while but the feeling to finally step out onto the rugby pitch with you lot or watch a Swans game together is something that I cannot quite put into words, so I'll just say thank you so much.

Chapter 1: Introduction

In today’s climate, organisations are increasingly motivated to tackle breaches in their networks. In 2020, UK businesses paid around £800 million to firms towards addressing cyber security issues [1] [2] with costs predicting to rise to $270 billion worldwide to tackle cybercrime in 2025 [3]. The adage that ‘time is money’ applies to cyber security with network uptime essential to business operations, with lack of availability equalling lost profits. In addition, cyber attacks can result in a loss of reputation for businesses and consequentially, customer’s trust within a business can deteriorate, also meaning lost profits. Moreover, organisations must navigate legal obligations such as the General Data Protection Regulation (GDPR) and the Data Protection Act (2018) which holds UK organisations responsible for protecting consumer data. The maximum fine for violating these laws is £17.5 million or 4% of the violating company’s income, whichever equates to a higher value [4]. This fine is far greater than the previous maximum fine of £500,000 for infringing on the previous legislation, the Data Protection Act (1998) [5], highlighting the larger financial incentive to protect networks now than ever before.

While the importance of network security has been recognised by organisations in recent years, the challenges to address security issues have escalated due to the COVID-19 pandemic. Organisations had to adapt working practices which saw many employees working at home for the first time. Directly after the first national lockdown in March 2020, around 40% of the UK’s workforce worked remotely the following month with the pandemic as the primary reason [6]. This left numerous organisations scrambling to improve network infrastructure and permitting employees to use personal devices to complete work. Additionally, the prevalence of video conferencing software such as Zoom meant that organisations had a greater exposure to cyber threats via a larger attack surface as well as a difficulty to adhere to security practices due to the reduced monitoring of employee activity by network administrators. For these reasons, the pandemic has led to the growth of ransomware. Ransomware attacks are carried out by utilising phishing, a social engineering method of attack that deceives users into accessing seemingly genuine links that are instead malicious. Commonly, COVID-related content is included within phishing attacks to cause sudden stress to the victim resulting in a higher chance of success. According to a Tessian survey, 47% of employees in the technology industry have been victim to a phishing email whilst working [7]. A successful phishing attack allows the perpetrator to intrude on the victim's network and infect it with malware. At network intrusion, the attacker can either deny access to the sensitive data using encryption thus making the system unavailable to the organisation or threatening to leak the sensitive data. The attacker will demand that the company pay a fee, usually via untraceable channels such as cryptocurrency, for the return of the unencrypted data, hence the name ransomware. The National Cyber Security Centre reported a three-fold increase of ransomware attacks in the UK in 2020 compared to 2019 [8]. A recent example is an attack on Irish healthcare systems in May 2021 where data was encrypted and stolen resulting in the closure of some health centres [9]. A $20 million ransom was demanded but not paid before most of the data was recovered.

Overall, the UK saw an increase of 31% in cybercrime at the start of the pandemic, costing businesses £6.2 million [10]. Increasingly, threat actors involved in this cybercrime are Advanced Persistent Threats (APTs) that are affiliated to nation states. APTs use sophisticated methods to infiltrate networks, typically spending extensive periods of time present on systems remaining undetected to discern a network fully before executing an attack. MITRE ATT&CK have identified 122 APT groups globally sponsored by governments such as China, Russia, and North Korea [11]. For example, the aforementioned attack on the Irish health service was believed to be carried out by a Russian APT group named Wizard Spider [12]. Organisations can take precautions to mitigate network intrusions and reduce the risk of becoming victims of cybercrime from APTs and cyber criminals. Most organisations use anti-malware tools and firewalls to limit damage from cyber attacks, while some also utilise IDS or IPS software. Both systems look at incoming and outgoing network traffic at the endpoint. An IDS will alert the administrator to malicious traffic to aid a response, whereas an IPS will attempt to block the malicious traffic before it reaches the network. It achieves this by comparing incoming network packets to known malware, and if they are similar, the software will output a positive result. However, these methods may not be adequate to defend networks against malware. Using machine learning is becoming increasingly popular and could be an effective solution to reduce the risk from malware. This project will look to use algorithms to predict whether incoming traffic is malicious or benign and the malware type. This is achieved using previous network data to help the algorithms learn to distinguish malicious packets from network traffic.

1.1: Intended Audience

This report is intended for cybersecurity students with an interest in machine learning. In addition, this report will interest professionals and researchers working within the cybersecurity sector looking to employ machine learning solutions for network intrusion detection.

1.2: Motivation

The motivation to carry out this project stems from the ineffectiveness from IDS / IPS software. The issue with the software is that they are prone to overkill when detecting intrusions meaning that benign traffic can wrongly be identified as malicious, known as false positives [13]. A higher false positive rate will result in wasted time for network administrators and consequently, less time to address genuine malware entering the system and a lack of confidence in the system. Additionally, an organisation using an IPS will see genuine traffic blocked from the network. Coupled with the rise of ransomware in 2020-21, the need for accurate malware detection in networks has been amplified. Machine learning algorithms with proper training could possibly reduce the false positive rate to predict future malicious activity, in a quicker time than traditional network IDS / IPS.

1.3: Aims and Objectives

This project aims to use several machine learning models to detect malicious network activity within a dataset containing network packets from a malicious webserver and compare the performance of each model using performance metrics. The project will look at four different machine learning models that have been created for the solution. These models are K-Nearest Neighbour (kNN), Support Vector Machines (SVM), Naïve Bayes, and Random Forest, all of which are supervised classification algorithms with the ability to classify data within a testing dataset, after being trained using a training dataset. A successful project will mean that these algorithms should have the ability to distinguish between malicious and benign data packets as well as identifying the type of attack within each packet. In addition, a successful project will produce meaningful results with the algorithms scoring well in effectiveness of classifying data as well as the speed in which the algorithms do so.

1.4: Structure

Chapter 1 introduces the report, discusses the aims and objectives of the project, the motivation for it and the audience it is intended for. Chapter 2 examines the related work that exists around this topic, as well as describing the problem that the project aims to address with an extended background explanation of machine learning. Chapter 3 will discuss the data used and the methodology which will be carried out in this project. Chapter 4 describes the implementation of the machine learning models, detailing its development. Chapter 5 evaluates the results from the solution and analyses the performance of the four models. Chapter 6 draws the report to a conclusion with a discussion about the results of the project, as well as the improvements that could be made should work on the project be extended in the future. Finally, Chapter 7 is about the reflection on the lessons I personally learned whilst undertaking this project.

Chapter 2: Research and Background Material

2.1: Existing Work

A review of existing work was carried out using Google Scholar and Cardiff University LibrarySearch to find similar articles of work related to this field of study.

The first research paper reviewed is Anthi et al. (2019), an article authored by personnel of Cardiff University. This looks at utilising a machine learning solution for an IDS for smart home appliances [14]. The article uses a variety of popular smart home devices and records the incoming and outgoing traffic from the devices on the network, with some traffic packets being malicious. The article evaluates the Naïve Bayes and SVM algorithms, among other classification algorithms to determine between malicious and benign traffic, in addition to attempting to classify the attack type on the device. This article, while related to this project, is specialised in detecting malicious traffic from Internet of Things (IoT) devices on an IoT testbed as opposed to a web server which this project specialises in.

Secondly, Almseidin et al. (2017) looks to evaluate different machine learning models for an IDS, similar to this project [15]. The paper used a knowledge discovery in databases (KDD) dataset containing different types of attacks, then using several supervised machine learning classification algorithms, including Naïve Bayes and Random Forest which will also be evaluated in this report, to determine whether TCP/IP data packets within the dataset. Of the seven algorithms evaluated, Naïve Bayes ranked 5th in True Positive, False Negative and Precision rates, and 4th in its False Positive rate. Additionally, it scored next to last in Root Mean Squared Error and 5th in Accuracy rate, highlighting the potential inefficiency of the Naïve Bayes algorithm. This is contrasted to Random Forest that scored highly in performance metrics, highlighting its suggested effectiveness in identifying malicious packets using binary classification. However, the attacks within the dataset of this study were heavily weighted towards Denial of Service attacks. Of the 4,898,431 packets in the dataset, Denial of Service attacks accounted for 79% of them. To effectively evaluate the performance of machine learning algorithms, the attacks within the dataset should be more varied in order to simulate network attacks that would concern an IDS, which this project seeks to do.

Finally, the last piece of related work reviewed is Halimaa A. and Sundarakantham (2019) also suggest implementing machine learning for an IDS which evaluated the classification of malicious activity from an NSL-KDD dataset using both Naïve Bayes and SVM [16]. Similar to the aforementioned research paper, Naïve Bayes scored poorly in the measured performance metrics, both accuracy and misclassification rates compared to its counterparts, with SVM recording 93.85% accuracy whereas Naïve Bayes scored an accuracy rate of around 71%. Where the study falls short compared to this report is that the dataset has a sample size of just 19,000 data instances which is not suffice enough to accurately measure the performance of a machine learning algorithm for an IDS because this software used in a commercial capacity would need to manage a more substantial volume of traffic data. In addition, this paper, like the previous paper only implements binary classification and does not attempt to classify the attack type, something that will be address in this project.

2.2: Background of Machine Learning

Machine learning is a field within computer science, related to artificial intelligence. It uses algorithms to determine trends within data by learning using statistics in a method similar to the way that humans learn, albeit with more computational power. Using a subset of a data sample, an algorithm is effectively trained from previous data and should possess the ability to make accurate predictions on new data that is input into the algorithm, with the algorithm becoming more accurate and efficient as it learns from more data input.

Once the problem is identified, the first step of a machine learning solution is to collect the initial dataset. The data type is dependent on the identified problem and can come from static sources such as databases or spreadsheets or collected in real-time such as network traffic. After data gathering, data is often messy and needs to be cleaned to match a format suitable for the algorithm, known as preprocessing. From the dataset, the data is separated into two groups, with a larger proportion of data being assigned into the 'training set' so that the model can learn patterns in the data. After training, the accuracy of the algorithm is evaluated using the smaller part of the dataset, the 'testing set'. Finally, once performance has been evaluated, the model should be adjusted to improve its accuracy. New data must then be gathered, and the process repeats again.

One technique of machine learning is to use supervised machine learning. Supervised learning makes use of data that is initially labelled when input into an algorithm and then labels can be predicted and output by the algorithm [17]. The solution looks to find patterns within input features and classify it using an output label. Some knowledge on what the correct output labels should be are required, which is determined using the training set. Using this knowledge from previous training data features and labels, it can predict output labels for the testing set which can then be measured in terms of performance by comparing the prediction to what the correct output should actually be.

There are two types of supervised machine learning models, Classification and Regression. This project will look at classification machine learning algorithms. Classification algorithms look to separate data points into categories, known as classes, depending on features that the data point possesses [18]. Classification algorithms can either be binary, such as ‘True’ or ‘False’ classes, or multi-class, such as ‘Red’, ‘Green’, ‘Blue’ and ‘Yellow’. Some algorithms could also be multi-labelled where there can be more than one output label can be assigned to a data point. Regression algorithms look to find patterns between multiple variables and predict numerical values based on the input variables [18]. This can be used for purposes such as forecasting the stock market, predicting the future price of stocks. The process of machine learning is described in the figure below.

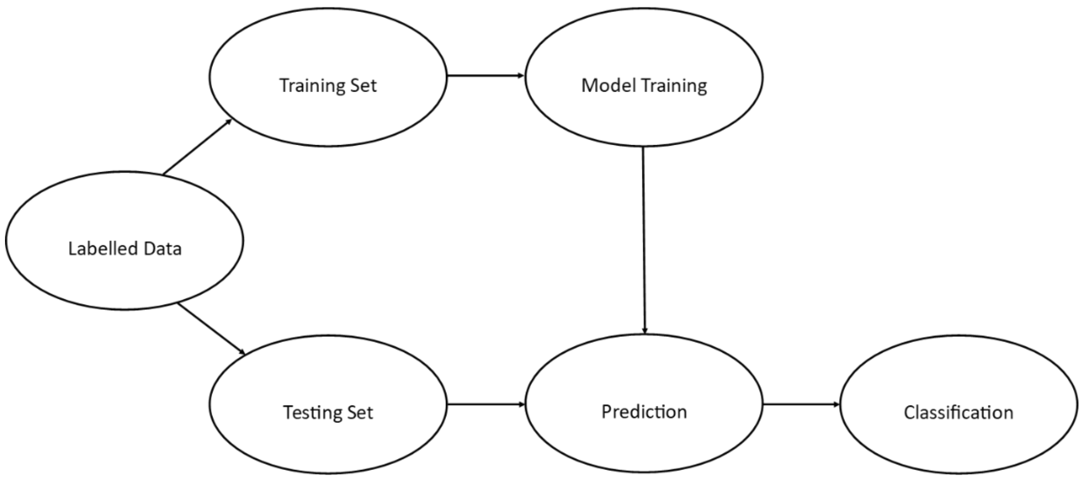


Figure 2.1 - Supervised machine learning process

The other technique within machine learning is unsupervised machine learning. These algorithms essentially discover patterns from unlabelled input training data [17]. The algorithm works without the need for humans to mediate but can be deemed less accurate as the performance cannot be measured properly. This will not be implemented in this project.

The field of machine learning has grown rapidly within the past two decades with the evolution of computer systems possessing larger computational power and access to large amounts of data via the Internet. However, machine learning was first theorised in the 1940s with the first program capable of learning was credited to American computer scientist Arthur Samuel in the 1950s. Whilst working at IBM, Samuel created a program with the ability to play the board game draughts (known as checkers in the United States) that would determine the 'best' move to play depending on the current state of the game. The more games that the program 'played' the better it became at choosing the 'best' move by learning from moves made during previous games and comparing them with the calculated chances of winning with other moves in a process called "rote learning" [19]. His paper on the subject includes the first ever use of the term "machine learning".

Nowadays, machine learning algorithms are used within a wide array of industries such as finance for detecting fraudulent transactions, and automatic trading on the stock market [20]; in healthcare to help medical personnel to diagnose patients with potential ailments in less time and more accuracy, as well as to aid drug research and development [21]; and manufacturing for improving product yield and thus profit/hour by optimising the output of each person and machine on a production line [22].

2.3: Project Machine Learning Models

This project will look at four machine learning models which will be explained in the following sections.

2.3.1: K-Nearest Neighbour (kNN) Algorithm

kNN is a simple algorithm used for classification and regression that works by classifying a data instance based on the classes of existing data points around the instance. It works by finding the distance between a new data point and the previously trained data, selecting the k number of previous data points and predicts the class of the new data point by 'voting'. This means that from the k number of previous data points, the majority class of this subset is assigned to the new data instance [23]. For example, if the K number = 3, the 3 closest neighbours to the new instance are checked. If two of the neighbours are classified as 'blue' and the other is classified as 'red', 'blue' is the majority class and is assigned to the new instance. It is important to determine the best K value for the optimal accuracy of the algorithm. A k value that is too low can be too specific, known as overfitting, whereas a k value that is too high can be too general, known as underfitting, either of which are not good for prediction [24]. An example of classifying a new data instance in a kNN algorithm is illustrated below. Note that the new instance would be labelled as Class B if k = 3 but Class A if k = 7.

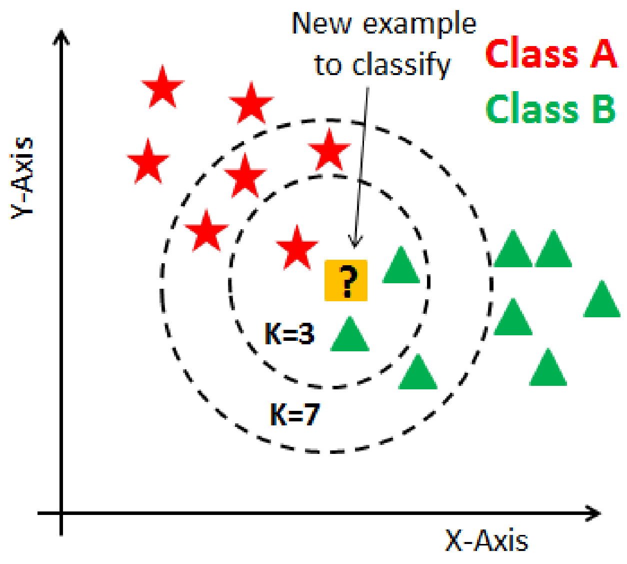


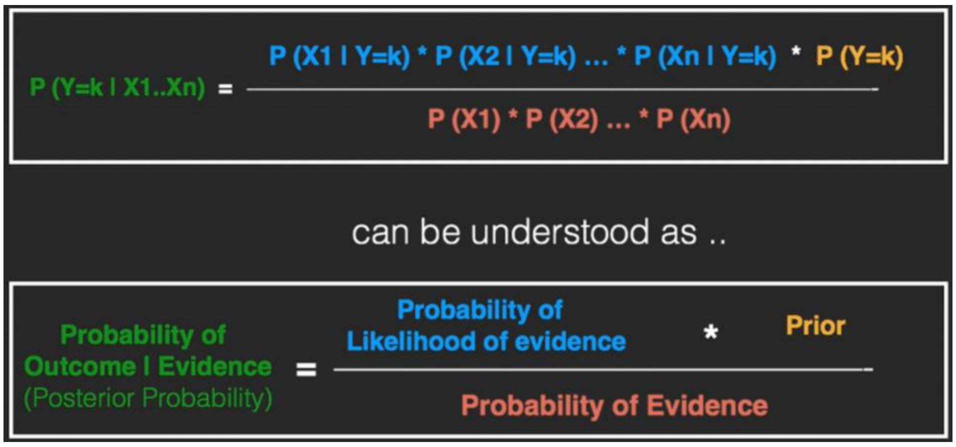
Figure 2.2 - kNN algorithm classifying new data [39]

2.3.2: Naïve Bayes Algorithm

Naïve Bayes is a classification algorithm that utilises probability to predict classes. It uses Bayes' theorem for calculating conditional probability, i.e., finding the probability of an outcome with the prior knowledge of probabilities for other outcomes. This algorithm predicts the classes of a new data point based on its features; however, it assumes that the features are equal and independent to each, hence why the algorithm is deemed "naïve" [25]. Within multi-class classification, the algorithm calculates the probability of each class being the output from the given set of features. The class with the largest probability calculation is the class that is assigned to the new data point. The probability of a class is determined during the training phase given the prior knowledge of a class pertaining to particular features. A description of Baye’s theorem and the formula for the Naïve Bayes can be found below.

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

Figure 2.5 - Bayes' theorem [40]



$$P(Y{=}k \mid X1..Xn) = \frac{P(X1 \mid Y{=}k) * P(X2 \mid Y{=}k) \ldots * P(Xn \mid Y{=}k) * P(Y{=}k)}{P(X1) * P(X2) \ldots * P(Xn)}$$

can be understood as ..

$$\text{Probability of Outcome | Evidence (Posterior Probability)} = \frac{\text{Probability of Likelihood of evidence} * \text{Prior}}{\text{Probability of Evidence}}$$

Figure 2.8 - Naive Bayes formula [25]

The probability of the outcome (Y) of a data instance being the class (k) with the features ($X_1 \ldots X_n$), the Posterior probability, is equal to the multiplied probabilities of each feature resulting in an outcome of class, k, multiplied by the probability of an outcome of class, k, based on overall previous data. This is divided by the multiplied probabilities that each feature, x, appears within an instance.

2.3.3: Support Vector Machines (SVM) Algorithm

The next model that will be implemented in this project is SVM. It can be used for classification and regression. SVM algorithms work by separating a dataset into two or more sections on a graph based on identified classes. The classes are separated by ‘hyperplanes’, which are linear lines on a two-dimensional graph or a subspace on an n-dimensional graph where n > 2, in any shape that best divides the classes [26]. Support vectors are the individual data instances that are located nearest to the hyperplane. The distance between the nearest support vector from each of two classes and the hyperplane is called the 'margin'. A hyperplane with the largest possible margin during training is desired to make data classification more accurate. If a support vector was taken off the graph, the hyperplane would also be required to move. The further a data point is from the hyperplane, the greater confidence we can have in the data point having the correct classification. However, outliers can occur within a graph which should be ignored [27]. A simple diagram that describes the SVM algorithm on a two-dimensional graph can be found below.

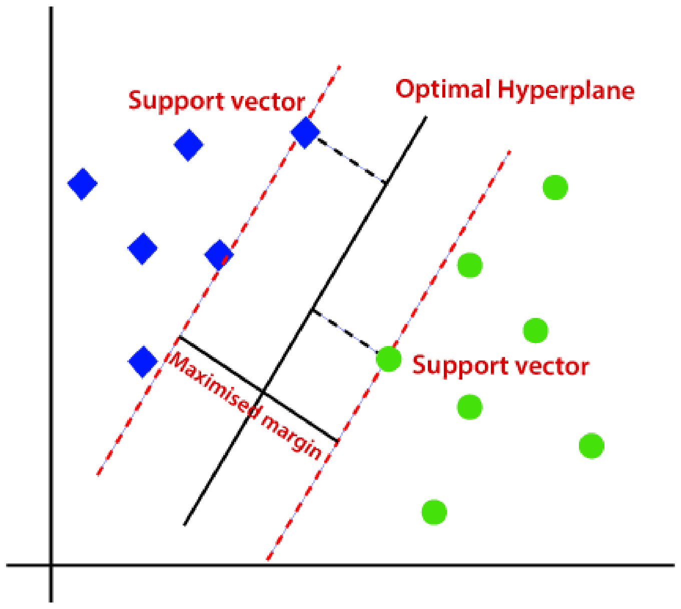


Figure 2.9 - SVM algorithm diagram [41]

2.3.4: Random Forest Algorithm

The Random Forest algorithm utilises individual component decision trees for classification [28]. A decision tree works by splitting data into two nodes based on certain parameters, e.g., whether a value for a column is above or below 2. The data will be continuously split using these decision nodes until the tree reaches the last node, known as the leaf. The leaf displays the final outcome predicted by the decision tree. A random forest will then use the multiple decision trees in order to choose which class to predict, known as ensemble learning. The prediction for each tree counts as a vote, and the prediction with the majority of votes will be determined as the classifier's prediction for each data instance. Random Forest is a reliable algorithm due to the notion that "A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models" [28].

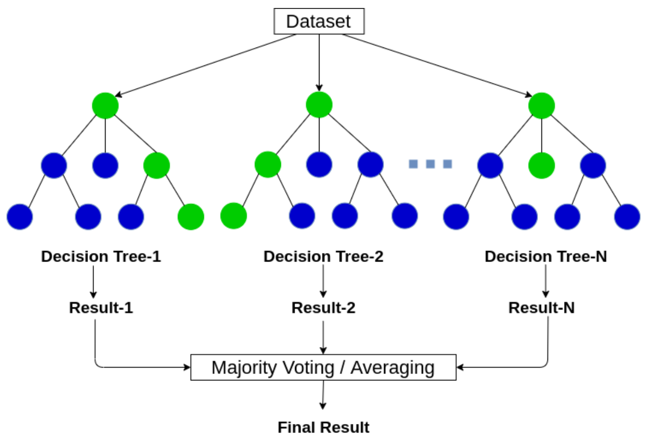


Figure 2.10 - Random Forest Diagram [44]

Note: The initial plan for this project was to only to implement kNN, Naïve Bayes and SVM algorithms. However, during the preprocessing stage, it was desired to include Random Forest to evaluate how popular machine learning algorithms with different types of classification would perform for these tasks. Naïve Bayes was selected as a probabilistic model, kNN as a distance-based model, SVM, while also distance-based, could act nonlinearly so Random Forest served as a tree-based classification model which would have been excluded in the solution originally.

2.4: Performance Metrics

In order to evaluate a model correctly, we must be able to measure its performance and evaluate it amongst the other models. The metrics used in this project to measure algorithm performance is outlined below.

When predicting data using a machine learning classification algorithm, there are four possible results that can exist in terms of correctness:

True Positive - the correct prediction that a data instance is within a specific class.

True Negative - the correct prediction that a data instance is not within a specific class.

False Positive - the incorrect prediction that a data instance is within a specific class.

False Negative - the incorrect prediction that a data instance is not within a specific class.

The results of these outcomes can be measured using a confusion matrix, which is generated on test data during the evaluation phase [29]. It compares the algorithm's output to the actual result to give one of the outputs listed above. A confusion matrix diagram for a binary classification algorithm is located below.

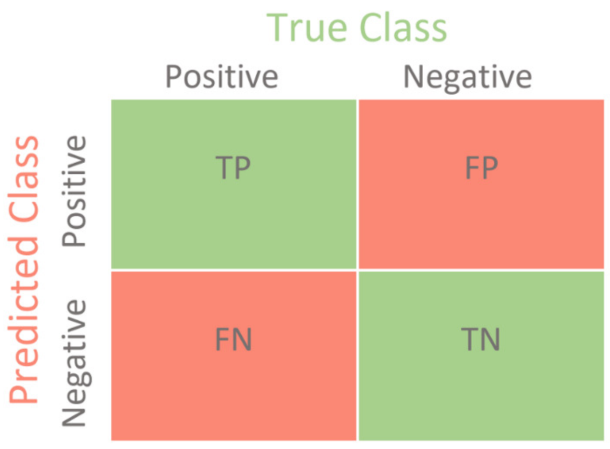


Figure 2.8 - Binary Classification Confusion Matrix [42]

This matrix can be extended for multi-class classification algorithms with the class names replacing the positive and negative labels on the matrix, an example of which can be seen below.

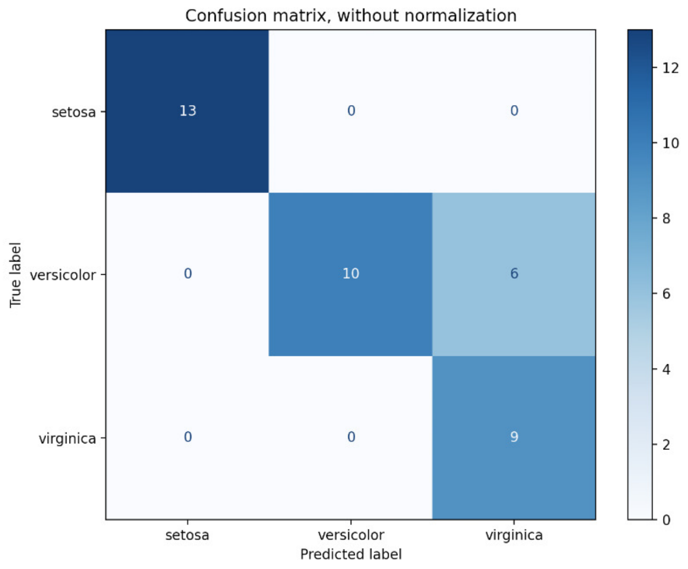


Figure 2.9 - Multi-Class Confusion Matrix [43]

While the matrix has no positive or negative labels, it is still possible to determine the outcome. In the example in Figure 2.9, for the 'virginica' class, the True Positive value is 9; the True Negative value is 23 (13 + 10); the False Positive value is 0 and the False Negative value is 6 and the False Negative value is 0.

Secondly, an algorithm can be measured in terms of accuracy. The accuracy is the rate of correct classifications on the test data, essentially it is the number of correct predictions divided by the total number of predictions [29]. It is calculated by:

$$Accuracy = \frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ Positives + False\ Negatives}$$

Another metric used to measure an algorithm is its precision. The precision is the rate of correct classifications of test data belonging to a particular class, i.e., the number of correct predictions to a particular class divided by the total number of predictions for a particular class [29]. It is calculated by:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

Additionally, an algorithm can be measured by its recall. The recall is the rate of correct classifications of test data belonging to a particular class compared to the amount of data that actually belongs to the class, i.e., the number of correct predictions to a particular class divided by the total number of data points that belong to that particular class [29]. It is calculated by:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

Accuracy is a useful metric but can be an impractical measure when the dataset used is disproportionate towards a certain class. Precision and Recall are beneficial metrics when the dataset is disproportionate towards a certain class [30].

It is convenient to merge the Precision and Recall into a single value because an algorithm can have some compromise between the two. This is measured by the f-score. An f-score of an algorithm with a value lower than 1 is weighted towards Precision and a value above 1 is weighted towards Recall [31]. It is calculated by:

$$f - score = \frac{2\ \times Precision\ \times Recall}{Precision + Recall}$$

Finally, because the machine learning models need to be quick at determining malicious and benign traffic, and attack types, the time taken for the model to provide results will also be measured.

2.5: Introduction into IDS

IDS work by inspecting incoming and outgoing traffic from a network and flagging any malicious activity that it discovers. There are two types of IDS: host-based and network-based [32]. Host-based looks at the traffic flow from all the different devices connected to the network, it looks at evaluating the current state of each system with previously recorded states and flagging the change in environment as a possible intrusion. Network-based IDS looks at endpoints on a network and analysing the entire traffic flow, usually a large amount, into the network and comparing the data with previously known traffic containing malware. When the current traffic data matches the malicious traffic data, a flag will be output by the IDS. This project would work by implementing the machine learning algorithms within a network-based IDS.

Chapter 3: Data and Methodology

3.1: System Overview

The system will receive network traffic from packet capture file. The algorithm will identify whether a data packet is malicious or not based on distinct features of the packet. The algorithm will classify the packet as malicious or benign depending on this identification. From the data packets that have been identified as malicious, the algorithm will then identify the type of attack that has been perpetrated within the packet.

3.2: Data Collection and Usage

An important part within the early stages of a machine learning workflow is the collection of data to train and test the algorithms. However, the project supervisor Amir Javed provided the dataset prior to undertaking this project. This was achieved by simulating traffic within a cyber range, a virtual environment that allows malware to be executed without the risk of it infecting the network. This traffic contained both malicious and benign packets to simulate actual traffic that an organisation's network would handle thus building the dataset. This traffic was recorded using the tcpdump tool, and output into a packet capture (pcap) file for solution use. To access the pcap file, a packet sniffing and analysing software called Wireshark can be used to view all packets in the dataset within a Graphical User Interface and allows for packets to be filtered by different parameters. The original pcap file contained 617290 data packets in total from 30 minutes of packet capture at a total size of 807MB. However, not all these packets are relevant and must be removed accordingly during the preprocessing phase. In addition, it is necessary to shorten the dataset due to limitations in processing power. The project is being undertaken using a personal laptop so the dataset size must be limited as it would not be viable to wait a substantial time to process such a large dataset. The pcap file will be exported to a CSV file for preprocessing as it can be used to easily manipulate data into a dataframe as data values are separated by a comma thus allowing simple distinction between data.

3.3: Attack Classes

There are four different attack types present within the dataset:

Ransomware - attempts to block access to data on the victim network by encrypting it until a ransom is paid to the perpetrator.

Trojan Horse - a piece of software that makes the victim believe that they have downloaded genuine software when in fact, it is malware that they have installed. Typically, used to gain backdoor entry onto a computer system or a network.

Downloader - a program that is discretely installed onto a network like a trojan horse but remains idle until a remote connection is established with the perpetrator so that further malware can be installed.

Miner - software that disguises itself on a victim's network to siphon off computing resources for wrongful purposes.

The machine learning models will look to classify all malicious packets into these attack types. It must be noted that malicious data packets can be of more than one attack type which is a challenge that the models must be equipped to deal with during implementation.

3.4: Requirement Specification

This requirement specification will define the conditions needed for the solution to effectively serve its purpose. For the solution to work effectively the solution should be:

Reliable - The solution should produce accurate results that can be measured. The amount of incorrect output should be kept to an absolute minimum.

Quick - The solution should provide output within a relatively short amount of time to mimic real-time detection of malicious traffic within a network.

Lightweight - The solution should not exhaust resources such as system storage or computational power so that it can be utilised by organisations with limited infrastructure.

Usable - The solution should provide an output that is intuitive to understand thus providing no delay to personnel responding to a network breach.

3.5: Project Methodology

There were two project methodologies considered for implementation, Agile and Waterfall. The Agile methodology separates a project into smaller tasks. Work on these tasks is continuous in a short amount of time per round of production with editing occurring incrementally within each repetition until full completion. The other methodology for consideration was Waterfall. This is a sequential methodology that works by separating phases of development into linear steps. Each phase is completed until moving onto the subsequent step, and so on, until full project completion [33]. The Waterfall methodology has been chosen for this project. Whilst it is less flexible than Agile, it is much simpler to define goals needed to achieve completion within each phase and to set deadlines for this completion. Agile is more suited to team projects whereas this solution will be undertaken by a single person. Additionally, due to the short timeframe of this dissertation, deadlines need to be much more explicit which Agile does not allow for as much as Waterfall.

The project implementation has been divided into the following sections:

Data Preprocessing - The data received from the data collection phase is cleaned so that it can be used efficiently within the machine learning models. Datasets can be messy with large amounts of unneeded data as well as missing values that must be dealt with for successful machine learning. The amount of data will also be trimmed down during this phase as aforementioned in the Data Collection and Usage section of this chapter.

Model Training - The four machine models are developed sequentially. Each model is trained using a subset of the newly processed dataset so that it can deduce patterns within the data. Patterns discovered can be used to identify data packets that are malicious in the testing phase, as well as the attack being executed by the packet. Models may need to be trained again under different parameters in order to get better performance.

Model Testing - The remainder of the subset is used to test the models. The now trained model will predict whether data is benign or malicious. From the malicious packets, the model will predict the attack type from the features of the packet.

Evaluation - The results of each model will be calculated using performance metrics. These metrics will be compared to each other to determine which machine learning model is most suitable for detecting malicious attacks from network activity and identifying the attack type.

3.6: Programming Language and Machine Learning Libraries

This will outline the software that will be utilised while developing the machine learning models for this this project. The programming language being used for development is Python. Python is a popular high-level programming language that includes built-in data structures, has syntax resembling natural language and is relatively fault-tolerant compared to other language. The version of the Python software that will be used during this project is 3.7.7. Additionally, Python provides library modules for specific functionality depending on the project, including for machine learning. The machine learning library packages (including its version number) that will be used during this project are listed and described below.

Sklearn - This library supports tools for machine learning in Python. It provides supervised and unsupervised machine learning algorithms. Also known as Scikit-learn. Version: 0.24.2.

NumPy - This library supports a selection of mathematical operations in Python. Version: 1.21.2.

SciPy - This library supports scientific computing functions within Python. Version: 1.7.1.

Pandas - This library provides tools that manipulates data especially with the use of data frames. Version: 1.3.3.

Matplotlib - This library supports data representation such as plotting. Version: 3.4.3.

Seaborn - A Python library used for data visualisation. Used for outputting confusion matrices in this project. Version: 0.10.1.

Chapter 4: Implementation

This section will explain the actions taken to create this solution. As aforementioned in Chapter 2, the solution was completed using the Python programming language together with the Scikit-learn library for its machine learning tools. Another piece of software was utilised for development, that has been previously unmentioned, called Jupyter Notebook. This software allows for the writing and execution of Python code using a web application. Each Python script is contained within a ‘notebook’ document. Each notebook splits code within cells that can be executed as standalone code, that helps to test individual part of implementation, or in conjunction with other cells to run as a coherent script. This notebook can then be exported as a .py file that combines each cell into a script. Jupyter Notebook was chosen due to previous familiarity from previous studies in addition to the ease of use, especially in regard to data visualisation tools that can be used within the software.

4.1: Data Preprocessing

Data collected from the original acquisition phase is most often not suitable for machine learning. This is due to the data being ‘messy’, usually with data instances containing missing values or additional unnecessary noise that can worsen a machine learning model’s performance [34]. This is best described in the computing concept garbage in, garbage out, which means that bad quality data being input into a program will produce bad results because of the unfaulty nature of the data being used. This highlights the need to clean data before developing the models as this will produce more precise output in a shorter amount of time. The data preprocessing phase was the most time-extensive part of the Implementation phase as the dataset was difficult to clean.

To begin, the pcap file must be converted into a CSV file to allow for data cleaning and manipulation in this phase. This can be achieved using Wireshark which has the feature to export the packet data from a pcap file into a CSV file with data values formatted within rows and columns. The CSV file can be loaded in Python using the read_csv function in preparation for machine learning. The input data is inserted into a dataframe using the Pandas library to easily visualise the data like a spreadsheet.

The features used in the input dataset were selected as they were deemed to provide enough relevant information for the model to learn patterns with this information applying to all data instances, regardless of the transmission protocol used or attack type employed. The original dataset included the source and destination IP addresses for every packet; however, these were removed because they can be used to bias predictions. These features can be used as unique identifiers the model can use during learning to assign an output to a specific IP address, thus overfitting the model. E.g., if the model learns that a lot of traffic from the source 192.168.9.101 IP address is malicious, it will predict traffic from that IP address is malicious. This may produce excellent performance results for this particular dataset but will likely perform poorly when introduced to another dataset as the model is dependent on these identifiers for prediction instead of effectively identifying patterns between features. Some other features will be removed before the final implementation of the machine learning models but this will be discussed later in the report in the feature selection section.

The input features in the dataset are as follows:

Protocol - The network protocol used to transmit the data from source to destination.

Length - The size of total raw data within a captured data packet.

Source Port - The port at which the network connection begins, and data packets are outgoing.

Dest Port - The port at which the network connection finishes, and data packets are incoming.

Flags_[ACK] - A flag signifying that a data packet has been received by one IP address from another.

Flags_[FIN, ACK] - A flag acknowledging that one IP address received a connection termination request from another IP address.

Flags_[FIN, PSH, ACK] - A flag acknowledging the previous packet was received, notifying to push the transmitted data to the relevant application and terminate the connection with the other IP address.

Flags_[PSH, ACK] - A flag acknowledging that the previous packet was received and the other IP address should push the transmitted data to the relevant application

Flags_[RST, ACK] - A flag acknowledging that the previous packet was received and, typically, the connection will be closed unilaterally by the source IP.

Flags_[RST] - A flag that typically notifies the destination IP that the connection will be closed unilaterally by the source IP.

Flags_[SYN, ACK] - A flag acknowledging the destination IP that the previous synchronisation packet has been received and sends a synchronisation packet with its initial sequence number.

Flags_[SYN] - A flag notifying the destination IP of its initial sequence number, typically signifying the beginning of a TCP connection.

TTL - The Time to Live value is the time, in seconds, that the packet should stay on a network before being ceasing to exist.

Seq No. - A value that shows how much data, in bytes, that has been sent within a TCP stream.

Ack No. - A value that shows how much data, in bytes, that has been received within a TCP stream, acknowledging that the data has been received.

Calc. Win. Size - The amount of space the source IP can receive from the other IP in the connection.

TCP Length - The size of the packet payload, in bytes. This is the length minus the header length.

Stream Index - The number of the TCP stream. 0 = first stream, 1 = second stream and so on.

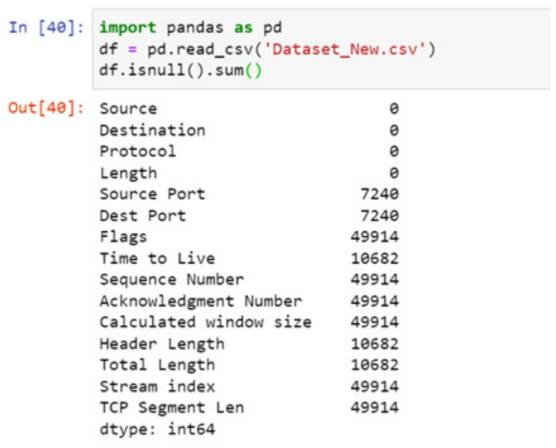
TCP Seg. Length - A value of data sent, in bytes, in the segment by the source IP.

In addition, the dataset contains the output labels:

Malicious - Identifies whether a packet is benign or part of a network attack.

Attack Type - The type of attack that a malicious packet is perpetrating.

The first step for cleaning the data is to handle all missing values present in the dataset. Missing values can be dealt with by using the mean, median or mode to fill null values within a data instance. For example, if the median is used, the median value of a feature is calculated from every data instance and when an instance contains a null value for that feature, it is replaced with the median value. This is the same method if mean or mode was used instead. Another method of handling missing values is to delete the rows that contain null values. The latter approach was taken for two reasons. Firstly, some information is specific to a particular protocol, for example, one protocol could typically use larger packets than another so it would not be suitable to use the median for packets using protocols where the size would be much larger or smaller than the median. Secondly, the approach of removing rows is only suitable when the dataset is large enough. As the dataset contains over 600,000 packets, this is large enough that removing rows will not be a hindrance to model performance so was deemed the suitable method for handling missing values, mitigating against potential outliers in the input data. A function, isnull.sum provides a count of null values within each column of the dataset as seen below in Figure 4.1.



```
In [40]: import pandas as pd
         df = pd.read_csv('Dataset_New.csv')
         df.isnull().sum()

Out[40]: Source                        0
         Destination                   0
         Protocol                      0
         Length                        0
         Source Port                7240
         Dest Port                  7240
         Flags                     49914
         Time to Live              10682
         Sequence Number           49914
         Acknowledgment Number     49914
         Calculated window size    49914
         Header Length             10682
         Total Length              10682
         Stream index              49914
         TCP Segment Len           49914
         dtype: int64
```

Figure 4.1 - Null Values in Dataset

It was determined that from the 33 protocols present in the dataset, only 18 provide data for every column, i.e., no null values. The data instances containing one of the remaining 15 protocols were then deleted from the dataset as they did not provide enough information for machine learning. This is shown in Figure 4.2 below. After row deletion, there were no remaining null values and the size of the dataset was reduced to 567298 packets, only an 8.1% reduction from the original dataset.

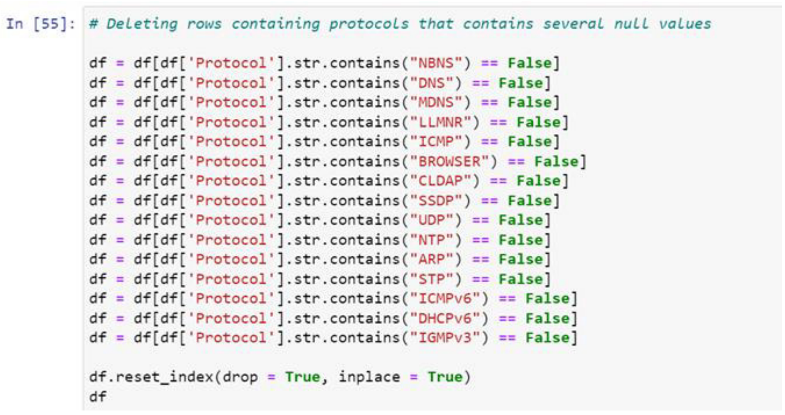

```
In [55]: # Deleting rows containing protocols that contains several null values

         df = df[df['Protocol'].str.contains("NBNS") == False]
         df = df[df['Protocol'].str.contains("DNS") == False]
         df = df[df['Protocol'].str.contains("MDNS") == False]
         df = df[df['Protocol'].str.contains("LLMNR") == False]
         df = df[df['Protocol'].str.contains("ICMP") == False]
         df = df[df['Protocol'].str.contains("BROWSER") == False]
         df = df[df['Protocol'].str.contains("CLDAP") == False]
         df = df[df['Protocol'].str.contains("SSDP") == False]
         df = df[df['Protocol'].str.contains("UDP") == False]
         df = df[df['Protocol'].str.contains("NTP") == False]
         df = df[df['Protocol'].str.contains("ARP") == False]
         df = df[df['Protocol'].str.contains("STP") == False]
         df = df[df['Protocol'].str.contains("ICMPv6") == False]
         df = df[df['Protocol'].str.contains("DHCPv6") == False]
         df = df[df['Protocol'].str.contains("IGMPv3") == False]

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

Figure 4.2 - Row deletion of particular protocols

Initially, it was predicted that time constraints would mean that the attack type classification would not be able to be completed so only the binary classification of predicting malicious or benign packets would be achieved. As a result, it was assumed that all inbound and outbound traffic from/to a specified list of IP addresses would be labelled malicious and the rest would be benign. This meant that the number of malicious packets present were 478070, over 84% of the dataset, highlighting that the dataset was weighted heavily towards malicious packets. A list was provided by the project supervisor that specified which IP addresses used in the cyber range were malicious and benign hosts. IP addresses with the network ID of 192.168.6 were designated 'tartarus' and addresses with the network ID of 192.168.9 were designated 'gehenna'. Tartarus addresses were malicious while gehenna addresses were benign. There were 10 tartarus hosts with address 192.168.6.101 assigned 'tartarus-001' through to 192.168.6.110 assigned 'tartarus-110'. Additionally, there were 5 gehenna hosts with 192.168.9.101 assigned 'gehenna-001' through to 'gehenna-005' for 192.168.9.105. So initially, any packets coming from or being received by the tartarus addresses were labelled as malicious in the malicious/benign column with the gehenna addresses labelled as benign.

Later, when it was realised that the attack type classification could be achieved within the project timeframe, the method for labelling packets in the dataset as malicious or benign was altered using another resource from the project supervisor. This resource was a list of attacks carried out using the cyber range during the data acquisition phase, the attack samples list. The list included a description of the type of attack that was carried out, the timestamp from when the attack was carried out relative to the start of the capture, and the

IP address that carried out the attack, under its tartarus designation. The list included more information, but it is not relevant to discuss for this project. From the list, the malicious packets could be mapped by comparing the timestamp from the list with the timestamp of the packets coming to or from the perpetrating IP address in the packet capture file. In order to know how many packets needed to be labelled malicious after an attack, a reasonable time interval was required to determine the duration of attacks, allowing for a definitive cut-off point for labelling a packet as malicious. Real attacks on organisation networks are commonly concluded after a short period of time to reduce the chance of the attacker(s) being detected intruding in the network [35]. The duration interval ultimately decided was two minutes which is not particularly too short or too long. This means that the new criteria for a packet being labelled malicious was now defined. From the time that an attack begins, provided in the attack samples list, every packet in the dataset containing the attacking IP address as the source or the destination IP address with a timestamp of no more than two minutes from the attack start time would be labelled malicious with the remaining packets labelled benign. As timestamps were not included within the dataset, it was required to refer back to the original packet capture file using Wireshark to compare the timestamps. The attack type was also labelled in Attack Type column at this point using the attack samples list which will be discussed more later. The reasoning for changing the method in which packets were labelled malicious was to better represent 'normal' traffic that an organisation would experience. While traffic from malicious IP addresses can be present within a network, not every single packet would be malicious so by labelling malicious packets this way, the dataset is more realistic to a real-world scenario.

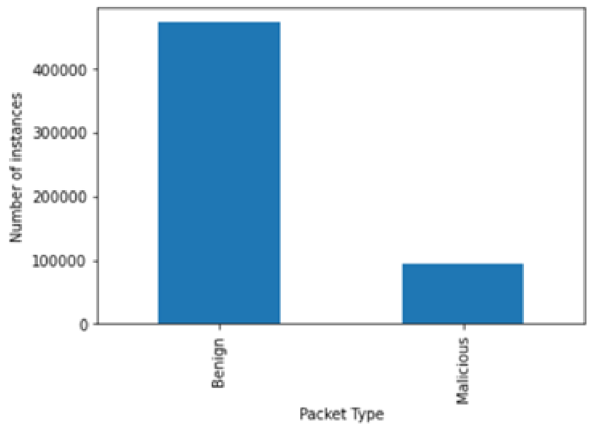


Figure 4.3 - Malicious/Benign Split

Figure 4.3 shows the split of the dataset between malicious and benign packets. Contrary to the original dataset, the chart shows that the data has a large disparity the opposite way with the majority of data instances being benign not malicious. However, this is not an issue as this simulated dataset is now more realistic to a real-world scenario where an organisation would see more normal network activity rather than malicious on any given day.

For implementing an attack type classification model, a subset of the dataset used for the binary classification was used for input data. This is because classifying attack types only applies to malicious packets so it would be illogical and inefficient to include the benign packets within the input data, whilst also affecting model accuracy. For this reason, the dataset used for this classification consists only of the malicious packets in the binary classification dataset.

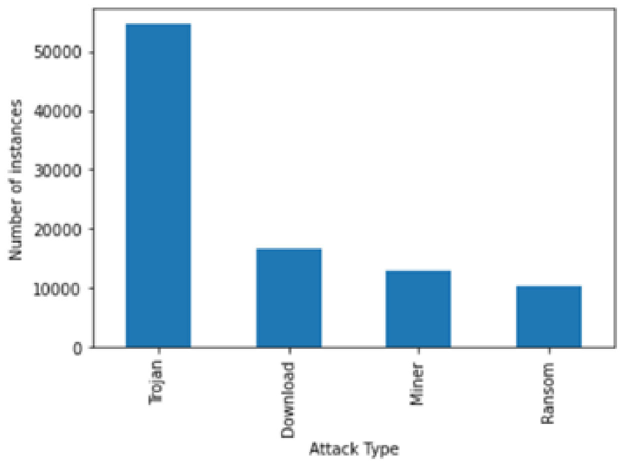


Figure 4.4 - Attack Type Split

Figure 4.4 shows the split in the number of data instances between the four attack types present within the dataset. We can see that the dataset is quite small with only 94,331 data packets and that the data is heavily weighted towards trojan attacks compared to the other attacks. It should be noted that all the malicious packets contained a trojan horse. However, some packets contained an attack in addition to a trojan horse coupled with it. For the purposes of this classification, where a data instance contains a trojan coupled with another attack, the instance will be labelled as only the additional attack in the Attack Type column of the dataset and not as a trojan. As the dataset is small, the attack type classification is a proof of concept that the selected machine learning models can distinguish between attack types. While multilabel classification could be used, meaning the attack types could be labelled as both, it makes more sense to use a multiclass classification approach to use distinct classes due to the limited sample size. In addition, this approach avoids distorting the effectiveness of the models by skewing the data imbalance further towards trojan attacks.

As previously mentioned, machine learning models can only interpret numerical data so the categorical data present in the dataset must be converted from string type into integer type by using encoding. Prior to the dataset including separate flag columns, it contained the flag description in one column using string, e.g., “[RST, ACK]”. Therefore, the flags needed to be encoded into separate columns where the flag that applies to the packet contains a value of 1 and the rest of the flag columns contain a value of 0. This was achieved by using the get_dummies function within the Pandas library while also creating the separate columns. Next, the protocol column contained the values as string that needed to be encoded. This was done by using the LabelEncoder function included in Scikit-learn’s preprocessing module. This would convert all the categorical data in the column into numerical data with each unique protocol assigned a unique integer to identify it. The integers were assigned to protocols ascendingly based on alphabetical order. For the purposes of aiding others understand the dataset in the future, the protocols and their assigned values have been listed below:

0 = DCERPC; 1 = DRSUAPI; 2 = EPM; 3 = HTTP; 4 = HTTP/JSON; 5 = KRB5; 6 = LDAP; 7 = NBSS; 8 = OCSP; 9 = RPC_NETLOGON; 10 = SAMR; 11 = SMB; 12 = SMB2; 13 = SSLv3; 14 = TCP; 15 = TLSv1; 16 = TLSv1.1; 17 = TLSv1.2.

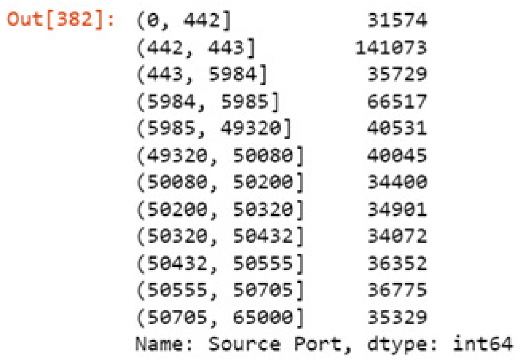
Moreover, the Malicious/Benign and Attack Type labels both needed to be encoded as both contained categorical data. The malicious/benign label was encoded by simply using the LabelEncoder function where benign data instances where assigned a value of 0 and malicious given a 1 value. Initially, to encode the Attack Type label, the get_dummies function to assign a 1 value to the relevant attack type and 0s to the others, with a column for each attack type. However, when evaluating the performance of the models for multiclass classification, the performance metrics showed the model underperforming from what was expected, as well as a longer time of completion. It was hypothesised that the extra data from creating new columns via get_dummies was the cause of this as the models must learn more data values, consequently meaning a longer time to process. The method to encode this data was then changed to using the LabelEncoder so that a value would be assigned to each attack type instead, which did improve performance. Again, for reader convenience, the integers assigned to each attack type are as follows:

0 = Trojan; 1 = Downloader; 2 = Miner; 3 = Ransomware.

The next stage of preprocessing was to employ binning on the Source and Dest Port columns. This looks to place data values within specific 'bins' by grouping values together. The bins are divided and have predetermined edges which signify a range of values. If the column of a data instance has a value within a bin's edges, the value will be placed in that bin. Each bin is assigned an integer which is used to represent it. Binning is not a requirement during preprocessing but was deemed necessary for this scenario because the dataset included 1783 unique port numbers ranging from port 80 to port 64702. Binning, in this case, improved accuracy due to the reduction in the range of values, resulting in less likelihood of outliers in the data and decrease the time taken for the model learn and make predictions because the model is only needs to process a reduced number of distinct values. It was decided that 12 bins would be used with the bin edges manually selected using fixed-width binning. The edges of each bin and its assigned bin number are as follows:

0 = [0, 442]; 1 = [442, 443]; 2 = [443, 5984]; 3 = [5984, 5985]; 4 = [5985, 49320]; 5 = [49320, 50080]; 6 = [50080, 50200]; 7 = [50200, 50320]; 8 = [50320, 50432]; 9 = [50432, 50555];
10 = [50555, 50705]; 11 = [50705, 65000].

These bin edges were chosen to produce approximately evenly sized bins, attempting to get around 37,000 data instances per bin plus/minus some thousands. Evenly sized bins are more efficient for models as the data is not weighted towards a particular bin value so the model has enough data to be able for training when it must process a particular bin. However some ports, namely port 443 and port 5985, are so prevalent within the dataset that these two ports consume an entire bin to themselves. Port 443 is a port typically used for secure HTTP traffic and is so frequent that its bin is around 4 times the size of the other bins. Port 5985 typically sees HTTP traffic and its bin is almost twice the size of the other bins. The bin sizes can be viewed in Figure 4.5 below.

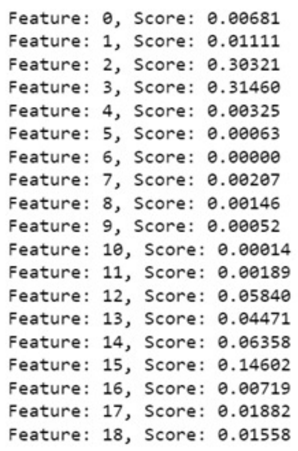


```
Out[382]: (0, 442]           31574
          (442, 443]        141073
          (443, 5984]        35729
          (5984, 5985]       66517
          (5985, 49320]      40531
          (49320, 50080]     40045
          (50080, 50200]     34400
          (50200, 50320]     34901
          (50320, 50432]     34072
          (50432, 50555]     36352
          (50555, 50705]     36775
          (50705, 65000]     35329
          Name: Source Port, dtype: int64
```

Figure 4.5 - Bin Sizes

The penultimate part of the preprocessing phase is the scaling of data. Similar to binning, scaling also is not a requirement for machine learning but has the potential to improve model performance. Scaling helps especially for distance-based predictive models as a large distance between values can drastically alter the predictions made by a model [36]. There are two methods of scaling, standardisation and normalisation in which the former was decided to be used. Standardisation works by centering around a mean value. The mean is the center point and is given a value of 0. The range of values in the dataset have values around 0, both positive and negative with its magnitude based on the standard deviation of values in the dataset.

Finally, the remaining stage of preprocessing was feature selection. Originally, feature selection was not to be implemented but that changed during the model development stage to see possible improvements in algorithm performance. Feature selection works by assigning weights to each feature depending on its importance, the more important a feature is, the larger weighting score it receives. The features with the smallest values can then be removed from the dataset for all algorithms. This aims to reduce the amount of data that needs to be learned, improving algorithm speed, in addition to reducing the complexity and variance in the dataset thus improving performance [37]. This stage can be repeated until a desired number of features are remaining. Random Forest provides functionality to perform feature selection within scikit-learn with feature_importances_ as displayed in Figure 4.6 below.



```
Feature: 0, Score: 0.00681
Feature: 1, Score: 0.01111
Feature: 2, Score: 0.30321
Feature: 3, Score: 0.31460
Feature: 4, Score: 0.00325
Feature: 5, Score: 0.00063
Feature: 6, Score: 0.00000
Feature: 7, Score: 0.00207
Feature: 8, Score: 0.00146
Feature: 9, Score: 0.00052
Feature: 10, Score: 0.00014
Feature: 11, Score: 0.00189
Feature: 12, Score: 0.05840
Feature: 13, Score: 0.04471
Feature: 14, Score: 0.06358
Feature: 15, Score: 0.14602
Feature: 16, Score: 0.00719
Feature: 17, Score: 0.01882
Feature: 18, Score: 0.01558
```

Figure 4.6 - Feature Importances Output

Two passes of feature selection were used for this stage. To understand its effectiveness in improving the speed, some preliminary testing was performed. One example was that using the original dataset for binary classification, the kNN algorithm took over 20 minutes to process before it was terminated early. After the first iteration of feature selection, the time taken to process for kNN diminished significantly. In the first iteration, it was determined that the features to be removed were the Flags_[FIN, ACK], Flags_[FIN, PSH, ACK], Flags_[PSH, ACK], Flags_[RST, ACK], Flags_[RST] and Flags_[SYN, ACK] columns. This equated to a reduction in over 3.4 million data values which explains the large decrease in processing time between the two kNN algorithm tests. The second iteration, the remaining flag columns, Flags_[ACK] and Flags_[SYN], were removed as well as the Time to Live column to produce the final datasets for binary and multiclass classification respectively. Feature selection was performed separately for both classification types, however, the same result transpired on both occasions.

4.2: Training and Testing Split

Both datasets used for the binary classification and multiclass classification must be separated into a training set and a testing set as aforementioned in Chapter 2. Scikit-learn provides functionality for splitting the data using train_test_split within the model_selection module in scikit-learn. This function requires parameters for the input features and labels and how large the testing set will be as a ratio from 0.0 to 1.0. In addition, a random_state parameter was utilised. This is used to maintain the same training and testing sets throughout development and implementation. Without the random_state parameter, every time the models are run, new sets are generated. A random number generator is used to determine which data instances are placed in each set so by explicitly defining a seed for the generator, it will produce the same sets each time. This was implemented for ease of measuring performance. As the models were not run simultaneously, it would not be a fair comparison if the data used for learning were different for each model. The training/testing split for machine learning can be set to whatever is desired by the developer and can often depend on the problem that the solution is attempting to address. For the binary classification, a split of 70%/30% for training/testing split was chosen. This was chosen because of the requirements specified for this solution. It is important that malicious traffic is detected as quickly as possible in this scenario. The more training data that can be utilised by the machine learning model, the more accurate it will be, but this also means that more time will transpire to learn from this data. With such a tradeoff between speed and accuracy, speed was prioritised at the expense of some accuracy due to the need to produce quick results for this problem. The sacrifice in accuracy is an insignificant one so it is not too concerning because, while the split has decreased the weighting towards the training set, the dataset is still large enough to provide a sufficient training set. This is contrary to the rationale of deciding the training/testing split for the multiclass classification. Because the dataset is small, there is an increased need to have a larger training set in order for the models to be accurate. As the models need to process less data from this limited sample size, the model will take less time to execute so it is logical to prioritise accuracy in the speed/accuracy tradeoff, sacrificing some speed of completion. For this reason, the training/testing split decided for multiclass classification was 80%/20%.

4.3: Model Development

The complete code for each model has been appended to the end of the report in Appendix A. The method of developing each model followed a similar pattern. When a model's code is executed, the first step is to start by using the time function in Python to get a timestamp of the execution start time. The machine learning begins by creating the classifier algorithm using a scikit-learn function from the model's respective module within the library. The data is then trained by utilising the training data, with the features and labels being used in an attempt to deduce patterns, achieved by using the fit() method in scikit-learn. The model will then attempt to predict the labels on the testing set using the predict() method in scikit-learn based on the information it has learned. Upon completion of the predictions, the time function is applied again to create another timestamp of the execution finish time. These timestamps are used by subtracting the finishing timestamp from the starting timestamp to retrieve a total duration for model processing which will be used to compare performance. Finally, the performance metrics are then calculated and output, this will be elaborated on later in the chapter. Further explanation will now be provided for the implementation of each model.

4.3.1: Naïve Bayes

The Naïve Bayes classifier was the first machine learning model implemented for this project. Due to the naïve nature of the model as well as its reputation for being reliable, the performance of Naïve Bayes served as a baseline standard for comparison with the other models. This model was simple to develop using the naïve_bayes module in scikit-learn. A Gaussian algorithm was used for binary classification, implemented by using GaussianNB within the naïve_bayes module. For the multiclass classification, a multinomial algorithm was implemented by using MultinomialNB in the naïve_bayes module in scikit-learn as it was easier to implement than using a Gaussian algorithm, in addition to providing better results. Neither algorithm required any parameters upon creation.

The Naïve Bayes model was expected to be the quickest algorithm but the lowest-scoring algorithm in terms of performance metrics amongst the four being implemented but still performing reliably prior to development of the models.

4.3.2: Random Forest

The Random Forest classifier was next to be implemented. To create the model, the RandomForestClassifier algorithm, part of the ensemble module in scikit-learn, was used for both binary and multiclass classification. This required a parameter, n_estimators, which determined the number of trees utilised by the algorithm for machine learning. It is important that the optimal number of trees are chosen as this can affect speed and performance. Analysis was undertaken to determine the ideal number of trees as seen below. Note: the accuracy and time scores in the following graphs and the subsequent graphs for kNN neighbour scores were calculated using a different random state in train_test_split, i.e., different training/testing sets, so the scores displayed will be different to the final scores seen in Chapter 5. Also, the top graph of each figure displays predictions on the training set with the bottom graph displaying the predictions on the testing set.

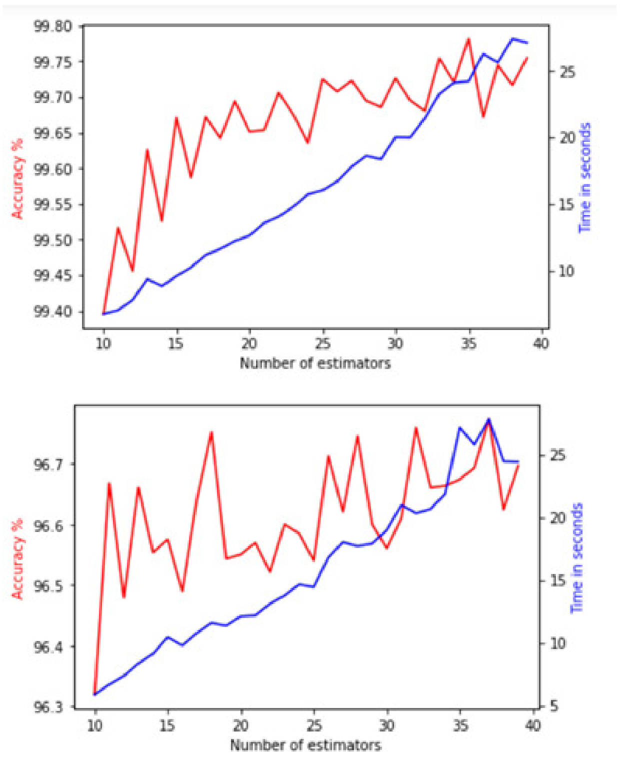


Figure 4.7 - RF number of estimators performance (Binary)

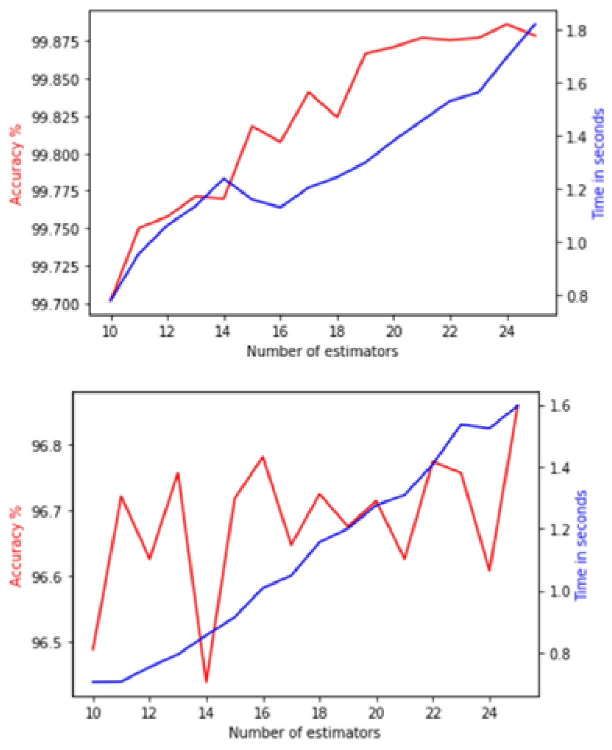


Figure 4.8 - RF number of estimators performance (Multiclass)

For analysis, the prediction accuracy on the training data is included as well as the testing data to ensure that a particular parameter does not cause a model to overfit by correlating the two. Initially, the accuracy and time taken to process was compared using the number of trees between 1 and 101 in 10 increments, i.e., 1, 11, 21, …, 101 for both Random Forest classification types. This was a guide to narrow down the optimal number of trees. For the binary classification, the optimal range was found between 10 and 40. For multiclass classification, this range was between 10 and 25. The general trend for both is that as the number of estimators increases, the accuracy tends to increase but the time taken to execute increases also. This means that there is a speed/accuracy tradeoff, sacrificing one to improve the other. As specified in the Training and Testing Split section, this problem requires quick detection of malicious activity, so it was decided to forfeit a negligible amount of accuracy to implement a quicker solution. It was determined that the optimal number of trees for binary classification was 18 and the optimal amount for multiclass classification was 13. The difference in estimators is understandable as generally the larger the dataset, the more trees are needed for an effective Random Forest algorithm.

It was predicted that Random Forest would perform excellently but be quite slow to process compared to the other models prior to development.

4.3.3: k-Nearest Neighbour

The next model to be implemented was the k-Nearest Neighbour classifier. This model was created by using KNeighborsClassifier, part of the neighbors module in scikit-learn, which was used for the binary and attack type classification. This classifier also required a parameter, n_neighbors, which determined how many neighbors will participate in the voting to assign a class to a new data instance. Similar to the Random Forest implementation, it is also important to determine the optimal number of neighbours as this affects speed and performance. For this reason, analysis was performed again to identify the optimal number as illustrated below.

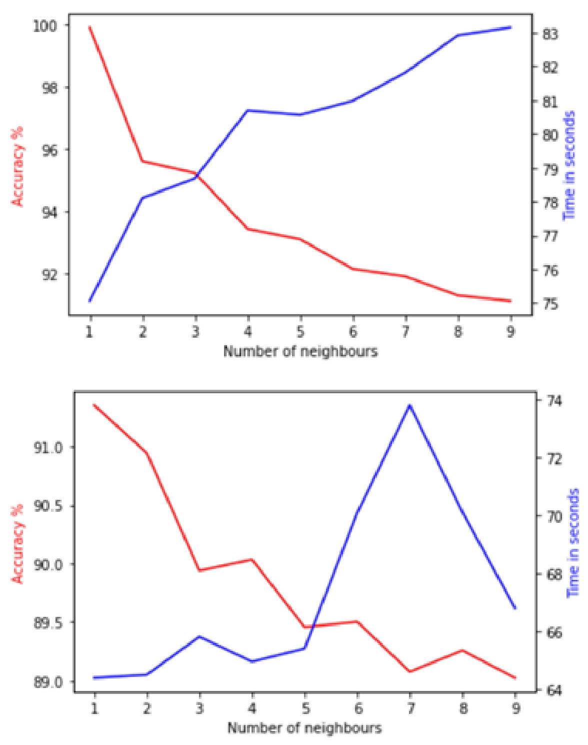


Figure 4.9 - kNN number of neighbours performance (Binary)

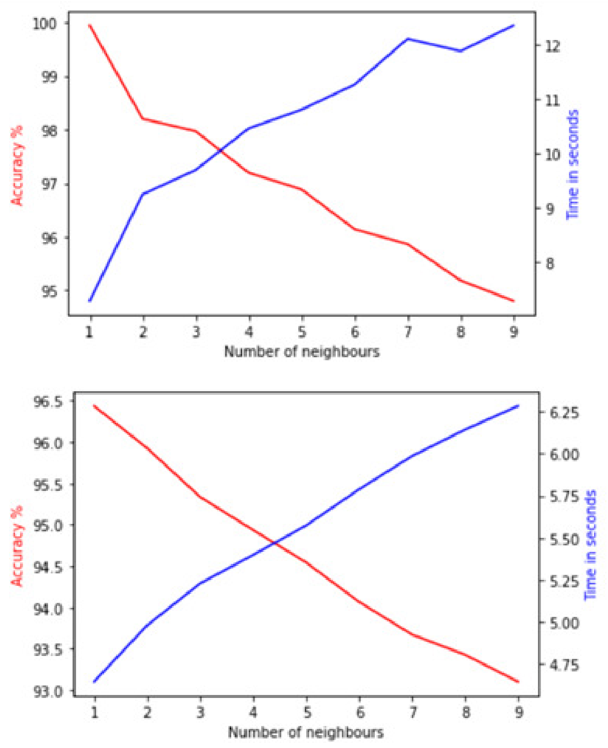


Figure 4.10 - kNN number of neighbours performance (Multiclass)

From the above, we can view two general trends for both binary and multiclass classification using kNN. Firstly, the greater the number of neighbours used by the algorithm, the slower the algorithm is. Secondly, accuracy declines with the more neighbours used by the algorithm. So, we must use a lower number of neighbours for the model for speed and accuracy. However, it is not recommended to use a value too low as this can lead to model overfitting, highlighted by the 100% accuracy scored for both classification models using 1 nearest neighbour when predicting the training set. Eventually, 4 nearest neighbours were selected for voting in the kNN model for both binary and multiclass classification.

k-Nearest Neighbour was predicted to perform well but would also take a substantial amount of time to reach completion prior to development.

4.3.4: Support Vector Machines

The final model to be developed was Support Vector Machines. The model was implemented by svm.SVC as part of the svm module in scikit-learn used for both binary and multiclass classification. The SVM classifier required a kernel parameter, specifying which kernel would be used in the algorithm. A kernel is a way that data is transformed for use by the algorithm. Three were considered and tested prior to the final implementation, linear, a polynomial kernel, and a Radial Basis Function (RBF) kernel. With prior knowledge that SVM models require long training times, analysis to determine which kernel to use was carried out using a small training set, which would not be used for final implementation as it would have provided unreliable results. As SVM takes a substantial amount of time so it was not viable timewise to use larger datasets in analysis. The linear kernel underperformed in contrast to the other kernels whereas the polynomial and RBF kernels had comparable performance metrics. However, the polynomial kernel took less time to complete processing compared to the RBF kernel so that consequently was the kernel of choice.

SVM was predicted to perform the best of the models in terms of performance metrics, however, it was also hypothesised that it would also take the longest amount of time to execute prior to development.

4.4: Performance Metrics

In order to ascertain which machine learning model is best, performance metrics need to be implemented on conclusion of the model execution. Scikit-learn provides functionality for calculating performance metrics in its metrics module. Firstly, a confusion matrix can be generated using confusion_matrix from the true labels of the dataset and the labels predicted by the model, which are also the parameters for each of the metrics used in this project. To output the confusion matrix, Seaborn has been used to visualise the data using its heatmap representation. This uses colours to help users interpret results, with a darker colour meaning a larger proportion of a particular result in the dataset. The matrix displays the true labels on the Y axis and the predicted labels on the X axis and the results are output as a percentage of the dataset that has been predicted to a class relative to the data's actual class, i.e., if the number of predicted benign packets that are actually malicious packets (False Negatives) equates to 2% of the testing set then 2% will be output in the confusion matrix for False Negative. The metrics module also provides functions for the other performance metrics with are listed as follows:

Accuracy: accuracy_score; Precision: precision_score; Recall: recall_score; f-measure: f1_score. As previously mentioned, the execution time is calculated by subtracting the finishing timestamp from the starting timestamp that were generated using the time Python function before and after the model was executed.

Chapter 5: Results Evaluation

The summary of results from the calculated performance metrics are as follows:

Metric	Binary Classification				Multiclass Classification			
	NB	RF	kNN	SVM	NB	RF	kNN	SVM
Accuracy (%)	78.66	96.51	96.12	N/A	13.89	96.82	95.57	N/A
Precision (%)	22.07	91.18	92.89	N/A	35.88	96.80	95.60	N/A
Recall (%)	11.11	87.47	83.07	N/A	13.89	96.82	95.57	N/A
f-measure (%)	14.78	89.29	87.70	N/A	8.93	96.81	95.54	N/A
Time (sec.)	0.261	12.05	256.1	>30 mins	0.054	1.29	2.02	>30 mins

As illustrated in the above table, the SVM algorithm as predicted would take the longest amount of time to execute but, unexpectedly, did not produce any results after 30 minutes. As an indicator as to how SVM may have performed, a test run showed that the algorithm scored 97% accuracy and high-90s in the remaining performance metrics for binary classification. However, this was using a smaller training set in test_train_split and still took over 2 hours to complete. As speed is a main requirement for this solution, waiting longer than 30 minutes for output, using the standard training/testing set split, is not viable as this delays the first response to network intrusions so for the results evaluation, SVM will be disregarded henceforth as no meaningful results could be obtained from the algorithm.

Firstly, the accuracy of the classifiers. For binary classification, Naïve Bayes scored fairly at 78.66%, however it was outperformed by the other two algorithms. Random Forest and k-Nearest Neighbour scored comparably with the former slightly edging out in accuracy scoring 0.31% better. For multiclass classification, Naïve Bayes the algorithm performed very poorly in accuracy at a rate under 15%. On the contrary, Random Forest and k-Nearest Neighbour scored excellently in accuracy, over 95% for both with the former slightly outscoring the latter, with 96.82% and 95.57% in accuracy rates respectively.

Random Forest performed exactly as predicted scoring very well. In addition, k-Nearest Neighbour performed well as predicted but exceeded expectations with accuracy results in the mid-90s for multiclass classification. However, whilst Naïve Bayes performed adequately for binary classification in line with prediction, it fell short of expectations for multiclass classification scoring poorly. It is notable that the algorithms performed better in binary classification than multiclass classification for accuracy metrics. A graphical comparison can be seen below in Figure 5.1. (Note: The blue bars in the graph represent the binary classification scores for each respective algorithm, with the red bars representing the multiclass classification scores. This applies for the subsequent graphs for each performance metric in this report).

Accuracy Ranking: 1st: Random Forest; 2nd: k-Nearest Neighbour; 3rd: Naïve Bayes

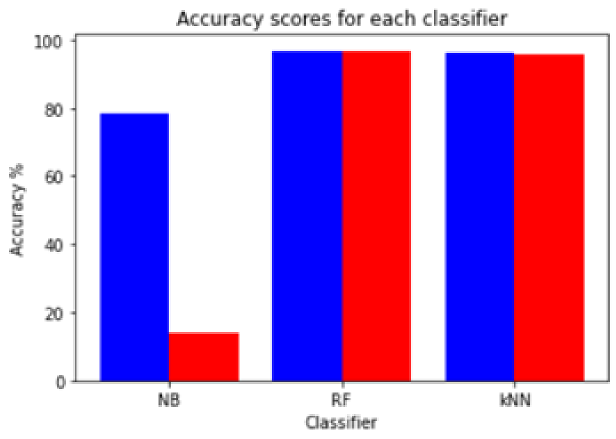


Figure 5.1 - Accuracy scores for each classifier of both classification types

Secondly is the evaluation of the precision scores. In binary classification, Naïve Bayes scored low in precision metrics at 22.07%. The other two classifiers scored considerably better. Both algorithms scored similarly in the low 90s for precision, but unlike the accuracy scores, k-Nearest Neighbour scored better than Random Forest at 92.89% and 91.18%. In multiclass classification, Naïve Bayes performed better than it did for binary classification but still scored below par for precision at 22.07%. In contrast, Random Forest and k-Nearest Neighbour both improved on their binary classification score, measuring at 96.80% and 95.6% respectively, with Random Forest edging out once again.

Random Forest performed on par with prior prediction. In addition, k-Nearest Neighbour again slightly surpassed expectations with its precision metrics. On the other hand, Naïve Bayes scored much lower than anticipated for precision. Contrasted to the accuracy metrics, the algorithms performed stronger for multiclass classification compared to binary classification. A bar chart illustrating the precision metrics for the three algorithms is displayed below in Figure 5.2.

Precision Rankings: 1st: k-Nearest Neighbour; 2nd: Random Forest; 3rd: Naïve Bayes.

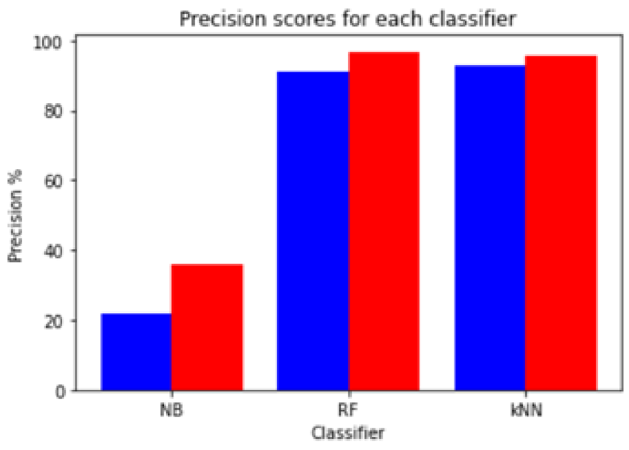


Figure 5.2 - Precision scores for each classifier of both classification types

Next, the recall scores of the classifiers. For binary classification, Naïve Bayes scored even worse in recall at a rate of 11.11%. Once again, the other algorithms measured better, Random Forest scoring 87.47% considerably outperforming k-Nearest Neighbour at 83.07%. Multiclass classification served slightly better than binary classification for Naïve Bayes, but still abysmally, scoring 13.89%. Random Forest and k-Nearest Neighbour saw a large improvement measuring at 96.82% and 95.57% respectively; Random Forest outscoring k-Nearest Neighbour again but with a smaller separation between the two values compared to binary classification.

Random Forest performed predictably for multiclass classification but underperformed in line with expectations for binary classification for recall. Surprisingly, k-Nearest Neighbour transcended expectations for multiclass classification but then scored below what was expected for binary classification. Again, Naïve Bayes scored well below its anticipated results for recall. Similar to the precision metrics, the algorithms scored better for multiclass classification compared to binary classification for recall metrics. The recall scores for each algorithm are displayed below in graphical form in Figure 5.3.

Recall Rankings: 1st: Random Forest; 2nd: k-Nearest Neighbour; 3rd: Naïve Bayes

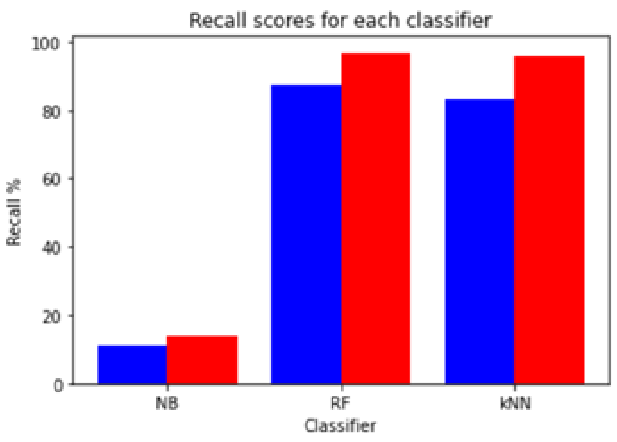


Figure 5.3 - Recall scores for each classifier of both classification types

Penultimately is the evaluation of the f-measure scores. Unexpectedly for Naïve Bayes considering the previous metric evaluations, the f-measure scores for both binary classification and multiclass classification were very low, scoring 14.78% and 8.93% respectively. Moreover, the remaining two classifiers were comparable in f-measure scores with Random Forest outperforming k-Nearest Neighbor for both classification types. For binary classification, Random Forest measured at 89.29% vs. k-Nearest Neighbour measuring at 87.70% whereas for multiclass classification, Random Forest scored 96.81% to k-Nearest Neighbour's 95.54% for f-measure.

Random Forest measured slightly below expectations for f-measure for binary classification and on par with multiclass classification. k-Nearest Neighbour performed oppositely; it scored in line with expectations for binary classification but exceeded anticipated predictions for multiclass classification. In addition, Naïve Bayes scored inferiorly to what was expected for f-measure. In tandem with the precision and recall metrics, the algorithms performance for multiclass classification exceeded their performance for binary classification in f-measure scores. The increased performance score for f-measure in multiclass classification, as well as precision and recall scores, compared to binary classification may suggest that the malicious attacks dataset is more unfairly distributed than the binary classification dataset as these metrics typically score better than accuracy when the dataset is more imbalanced [38]. The f-measure metrics are graphically displayed below in Figure 5.4.

f-measure Ranking: 1st: Random Forest; 2nd: k-Nearest Neighbour; 3rd: Naïve Bayes.

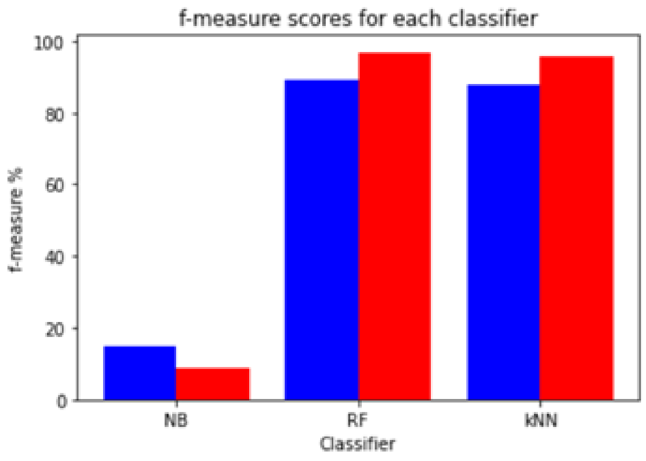


Figure 5.4 - f-measure scores for each classifier of both classification types

Finally, the time measurement evaluation. Naïve Bayes, as expected, was the fastest performing classifier for both classification types. For binary classification, Naïve Bayes performed around 46 times faster than the next best-performing classifier, Random Forest, and over 980 times faster than its slowest counterpart, k-Nearest Neighbour. The multiclass classification performance was even quicker, with Naïve Bayes almost 24 times quicker than Random Forest and over 37 times faster than k-Nearest Neighbour. Random Forest performed quicker than anticipated for both classifications, the middle-ranked algorithm scored over 21 times quicker than k-Nearest Neighbour in binary classification and for multiclass classification, it scored over 1.5 times faster than k-Nearest Neighbour. K-Nearest Neighbour performed in line with expectations for binary classification but exceeded expectations incredibly in speed for multiclass classification, with almost a 127 times improvement compared to binary classification. It is notable that the multiclass classification was quicker to complete than binary for all classifiers but is understandable as it is working with smaller datasets, however, it is interesting to see the substantial decrease in times between the two classification types.

Time Ranking: 1st: Naïve Bayes; 2nd: Random Forest; 3rd: k-Nearest Neighbour.

The confusion matrices for each classifier in each classification type can be found in Appendix B at the end of this report.

Chapter 6: Conclusion

This project set out to evaluate machine learning algorithms to detect malicious data in network traffic which has been successfully achieved. The primary goal was to identify between malicious and benign packets within a dataset which was completed, in addition to a secondary aim to identify attack types from the malicious dataset. Of the three original algorithms to be implemented, two were able to produce results that could be evaluated, as well as a late addition in Random Forest, to achieve three completed algorithms to solve the proposed problem.

The recommended machine learning algorithm to use is Random Forest, which is suitable for binary classification of network traffic as well as the multiclass classification of malicious traffic packets. To summarise the results, Naïve Bayes was exceptionally quick to classify but its performance metrics scored far too low to effectively work in an IDS. On the contrary, the k-Nearest Neighbour classifier produced brilliant results but took far too long to complete the binary classification, taking over 4 minutes to complete. Random Forest scored remarkably in performance metrics, ranking first in all except one performance metric, while slightly underperforming in recall score for binary classification. The good performance is supplemented with a quick processing time, 12 seconds for binary classification and even less for multiclass classification, showing Random Forest to be a very reliable algorithm for detecting malicious network traffic. The speed of this classifier would allow network administrators and first responders to quickly detect malware incoming to an organisation’s network, enabling a faster response, which this solution looked to address.

Especially as a novice to machine learning prior to this project, I believe this project has been a resounding success in producing desired output and providing a suitable solution to the outlined problem. However, the project is not without its deficiencies. Firstly, the lack of results produced from the SVM model. It is disappointing to not be able to produce results for this and while it was anticipated that this algorithm would take a considerable amount of time to execute, if there was more time available, analysis would be undertaken to attempt to reduce the algorithm’s complexity in order for it to produce suitable results. Secondly, the algorithms should be validated using previously unseen data to determine whether algorithms are overfitting or whether they work generally to any packet capture sample.

Next, if there were less time constraint, the low performance metrics for the Naïve Bayes model would be investigated. It performed well below what was predicted and while there is one hypothesis on why it performed so abysmally, analysis could be performed to implement changes to improve performance. My theory is that the dataset being biased towards one class, skewed the results for Naïve Bayes as it calculated the probability towards one class more than others as illustrated in its confusion matrices. However, this is just a theory and would warrant further examination to confirm. This theory does highlight another pitfall of the project, that the datasets are imbalanced, which is more relevant to attack type classification. This was identified as pitfall in another research paper that was discussed in related work in Chapter 2, but I must be honest in identifying this as a pitfall in this project also. With more time, I would like to have completed data acquisition personally for several reasons. Firstly, it would provide an opportunity to develop my skills in using a cyber range to simulate network attacks. Secondly, I would oversee the number of data instances for each attack and the balance between these attacks. Finally, I would implement more ransomware attacks in the data as the recent rise in ransomware attacks was the main motivation for undertaking this project.

I hope that this dissertation will support individuals or organisations in an academic and/or practical sense in looking to implement machine learning algorithms to determine malware in network traffic, particularly in Intrusion Detection Systems.

6.1: Future Work

To extend this project in the future, I would look to provide functionality so that the machine learning models would work on much larger volumes of traffic in real-time which would be much more realistic to the conditions of a large organisation using an IDS. Additionally, I would look to implement more machine learning algorithms, and possibly deep learning such as using Neural Networks, to evaluate the effectiveness of a wider range of possible solutions for an IDS. This would mean possibly using different machine learning software such as TensorFlow as Scikit-learn typically is not used for deep learning, which could lead to comparisons between different software in which would possibly perform better for individual algorithms. Extending from this point, a further solution could look to implement unsupervised learning algorithms as this project only implements supervised models. Unsupervised models would be a good idea for this topic as some malware breaching a network could be previously unknown, i.e., zero-day attacks, where it would be impossible to be formerly labelled so an unsupervised model would be more equipped in distinguishing possible unidentified malware. Penultimately, a wider range of attacks could be implemented in the dataset to evaluate how well the models would perform on the new attacks present as well as how accurate the models would be in distinguishing the correct attack type if there was a larger abundance of possible malware that can attack the network. Finally, to extend the project in future, the classifiers should support multilabel classification, a deficiency in this project, as network attacks are likely to use several different types of malware simultaneously that would need to be classified in an IDS.

Chapter 7: Reflection on Learning

Prior to the writing of this report, during the project selection phase, I understood that this dissertation would serve as a precursor to my later career in terms of required skillset. Therefore, I wanted to choose a project that would challenge me and develop additional skills to my repertoire gained from my previous studies. I was attracted to a project concerning machine learning due to my genuine interest in the topic, in part, because of its growing prevalence within the field of cybersecurity, however, I had never undertaken any machine learning projects before this dissertation. By selecting a project in a topic in which I had limited knowledge, I metaphorically threw myself into the deep end. This required extra diligence and a greater emphasis on research before any implementation could begin and while I could have chosen a more familiar topic, I do not regret pushing myself and I am proud to deliver this dissertation. I believe that the completion of this project has not only provided me with a more enhanced knowledge of machine learning, in addition to the workflow needed to implement models, network security and IDS but also softer skills that will be necessary when working within a professional environment such as adaptability, researching and problem solving.

During research, there was an abundance of online material that was available to me. Whilst learning about machine learning principles I was able to utilise textbooks and other materials using Cardiff LibrarySearch as well as tutorials to equip me with the knowledge to complete this project, in addition to a plethora of web articles and research papers from Google Scholar to support points made within the report. Project management was adequate overall, however, when unforeseen circumstances within my personal life resulted in delays, it was imperative to adapt self-imposed deadlines accordingly. In hindsight, I should have designated extra time to allow for these disruptions when mapping out the schedule of the project. I felt that I spent a slightly excessive amount of time on the research phase which consequently meant that I had less time than desired to tackle the latter stages of this project which signifies the need to improve upon my time management skills. Additionally, I refrained from contacting the project supervisor during times of slow progress which I understand I must improve upon hereafter by asking for help when required.

On the contrary, I think that my approach to problem solving developed well during this dissertation. I found that splitting one big problem into several smaller problems and addressing them individually was much more manageable in completing tasks. Also, I discovered later in the project that it was simpler to write sections of the report simultaneously whilst completing tasks as opposed to writing up following a task completion. The initial method meant that I had to remember everything I performed which was not time-efficient rather than writing a first draft concurrently with the work I was carrying out, then editing the draft later. Moreover, this report provided the opportunity to develop my referencing skills, a previous weakness of mine. Whilst I recognise that this project has skimmed what is capable with machine learning, I believe it has allowed me to build a good foundation in my education and has put me in good stead to possibly pursue it within a professional capacity in the future.

References

[1] S. O'Dea, “Volume of investment of United Kingdom business in cyber security sector 2020, by business size,” Statista, 4 August 2021. [Online]. Available: https://www.statista.com/statistics/586580/average-investment-in-cyber-security-by-united-kingdom-uk-businesses/#statisticContainer. [Accessed 16 August 2021].

[2] Department for Digital, Culture, Media & Sport, “Record year for UK’s £8.9bn cyber security sector,” GOV.UK, 18 February 2021. [Online]. Available: https://www.gov.uk/government/news/record-year-for-uks-89bn-cyber-security-sector. [Accessed 16 August 2021].

[3] D. Jose, “Cybersecurity spending must rise,” HSBC, 12 May 2021. [Online]. Available: https://www.gbm.hsbc.com/insights/global-research/cybersecurity-spending-must-rise. [Accessed 16 August 2021].

[4] Information Commisioner's Office, “Penalties,” Information Commisioner's Office, 23 April 2021. [Online]. Available: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-law-enforcement-processing/penalties/. [Accessed 16 August 2021].

[5] D. Walker, “What is the Data Protection Act 1998?,” ITPro, 12 April 2021. [Online]. Available: https://www.itpro.co.uk/data-protection/28085/what-is-the-data-protection-act-1998. [Accessed 16 August 2021].

[6] A. Cameron, “Coronavirus and homeworking in the UK: April 2020,” Office for National Statistics, 8 July 2020. [Online]. Available: https://www.ons.gov.uk/employmentandlabourmarket/peopleinwork/employmentandemployeetypes/bulletins/coronavirusandhomeworkingintheuk/april2020. [Accessed 16 August 2021].

[7] T. Sadler and J. Hancock, “Why We Click: The Psychology Behind Phishing Scams and How to Avoid Being Hacked,” Tessian, 7 September 2020. [Online]. Available: https://www.tessian.com/blog/why-we-click-on-phishing-scams/. [Accessed 16 August 2021].

[8] National Cyber Security Centre, “NCSC defends UK from more than 700 cyber attacks while supporting national pandemic response,” National Cyber Security Centre, 3 November 2020. [Online]. Available: https://www.ncsc.gov.uk/news/ncsc-defends-uk-700-cyber-attack-national-pandemic. [Accessed 16 August 2021].

[9] Kaspersky, “Irish health service hit by ransomware,” Kaspersky Daily, 14 May 2021. [Online]. Available: https://www.kaspersky.co.uk/blog/irish-health-service-ransomware/22768/. [Accessed 16 August 2021].

[10] Security Magazine, “UK sees a 31% increase in cyber crime amid the pandemic,” Security Magazine, 23 October 2020. [Online]. Available: https://www.securitymagazine.com/articles/93722-uk-sees-a-31-increase-in-cyber-crime-amid-the-pandemic. [Accessed 16 August 2021].

[11] MITRE ATT&CK, “Groups,” MITRE ATT&CK, n.d.. [Online]. Available: https://attack.mitre.org/groups/. [Accessed 17 August 2021].

[12] A. Scroxton, “Conti ransomware syndicate behind attack on Irish health service,” ComputerWeekly.com, 17 May 2021. [Online]. Available: https://www.computerweekly.com/news/252500905/Conti-ransomware-syndicate-behind-attack-on-Irish-health-service. [Accessed 17 August 2021].

[13] B. Violino, “Security tools' effectiveness hampered by false positives,” CSO, 2 November 2015. [Online]. Available: https://www.csoonline.com/article/2998839/security-tools-effectiveness-hampered-by-false-positives.html. [Accessed 18 October 2021].

[14] E. Anthi, L. Williams, M. Słowińska, G. Theodorakopoulos and P. Burnap, “A Supervised Intrusion Detection System for Smart Home IoT Devices,” *IEEE Internet of Things Journal,* vol. 6, no. 5, pp. 9042-53, 2019.

[15] M. Almseidin, M. Alzubi, S. Kovacs and M. Alkasassbeh, “Evaluation of machine learning algorithms for intrusion detection system,” in *International Symposium on Intelligent Systems and Informatics (SISY)*, Subotica, 2017.

[16] A. H. A. and K. Sundarakantham, “Machine Learning Based Intrusion Detection System,” in *International Conference on Trends in Electronics and Informatics (ICEI)*, Tirunelveli, 2019.

[17] J. Delua, “Supervised vs. Unsupervised Learning: What’s the Difference?,” IBM, 12 March 2021. [Online]. Available: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning. [Accessed 22 October 2021].

[18] J. Brownlee, “Difference Between Classification and Regression in Machine Learning,” Machine Learning Mastery, 11 December 2017. [Online]. Available: https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/. [Accessed 22 October 2021].

[19] A. L. Samuel, “Some Studies in Machine Learning Using the Game of Checkers,” *IBM Journal of Research and Development,* vol. 3, no. 3, pp. 210-229, 1959.

[20] C. Trinidad, “What is Machine Learning (in Finance)?,” Corporate Finance Institute, n.d.. [Online]. Available: https://corporatefinanceinstitute.com/resources/knowledge/other/machine-learning-in-finance/. [Accessed 20 August 2021].

[21] M. Thomas, “15 Examples of Machine Learning in Healthcare That Are Revolutionizing Medicine,” built in, 9 April 2020. [Online]. Available: https://builtin.com/artificial-intelligence/machine-learning-healthcare. [Accessed 20 August 2021].

[22] L. Columbus, “10 Ways Machine Learning Is Revolutionizing Manufacturing In 2019,” Forbes, 11 August 2019. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2019/08/11/10-ways-machine-learning-is-revolutionizing-manufacturing-in-2019/?sh=7b0fa4de2b40. [Accessed 20 August 2021].

[23] A. Christopher, “K-Nearest Neighbor,” Medium, 2 February 2021. [Online]. Available: https://medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4. [Accessed 30 August 2021].

[24] S. Yıldırım, “K-Nearest Neighbors (kNN) — Explained,” Towards Data Science, 29 February 2020. [Online]. Available: https://towardsdatascience.com/k-nearest-neighbors-knn-explained-cbc31849a7e3. [Accessed 30 August 2021].

[25] S. Prabhakaran, “How Naive Bayes Algorithm Works? (with example and full code),” Machine Learning Plus, 4 November 2018. [Online]. Available: https://www.machinelearningplus.com/predictive-modeling/how-naive-bayes-algorithm-works-with-example-and-full-code/. [Accessed 31 August 2021].

[26] N. Bambrick, “Support Vector Machines: A Simple Explanation,” KDnuggets, July 2016. [Online]. Available: https://www.kdnuggets.com/2016/07/support-vector-machines-simple-explanation.html. [Accessed 21 October 2021].

[27] S. Ray, “Understanding Support Vector Machine(SVM) algorithm from examples (along with code),” Analytics Vidyha, 13 September 2017. [Online]. Available: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/. [Accessed 1 September 2021].

[28] T. Yiu, “Understanding Random Forest,” Medium, 12 June 2019. [Online]. Available: https://towardsdatascience.com/understanding-random-forest-58381e0602d2. [Accessed 24 October 2021].

[29] J. Jordan, “Evaluating a machine learning model.,” Jeremy Jordan, 21 July 2017. [Online]. Available: https://www.jeremyjordan.me/evaluating-a-machine-learning-model/. [Accessed 2 September 2021].

[30] J. Brownlee, “8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset,” Machine Learning Mastery, 15 August 2020. [Online]. Available: https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/. [Accessed 22 September 2021].

[31] J. Brownlee, “How to Calculate Precision, Recall, and F-Measure for Imbalanced Classification,” Machine Learning Mastery, 3 January 2020. [Online]. Available: https://machinelearningmastery.com/precision-recall-and-f-measure-for-imbalanced-classification/. [Accessed 3 September 2021].

[32] DNSstuff, “What Is an Intrusion Detection System? Latest Types and Tools,” DNSstuff, 18 October 2019. [Online]. Available: https://www.dnsstuff.com/intrusion-detection-system#what-is-a-network-intrusion-detection-system. [Accessed 27 October 2021].

[33] S. Ferguson, “The most popular project management methodologies: A comparison,” Tech Radar, 28 October 2020. [Online]. Available: https://www.techradar.com/uk/news/the-most-popular-project-management-methodologies-a-comparison. [Accessed 22 September 2021].

[34] Ciklum, “Garbage In, Garbage Out: How to Prepare Your Data Set for Machine Learning,” Ciklum, 11 January 2019. [Online]. Available: https://www.ciklum.com/blog/garbage-in-garbage-out-how-to-prepare-your-data-set-for-machine-learning/. [Accessed 25 October 2021].

[35] L. Tung, “This is how long hackers will hide in your network before deploying ransomware or being spotted,” ZDNet, 19 May 2021. [Online]. Available: https://www.zdnet.com/article/this-is-

how-long-hackers-will-spend-in-your-network-before-deploying-ransomware-or-being-spotted/. [Accessed 26 October 2021].

[36] P. Sharma, "Why is scaling required in KNN and K-Means?," Medium, 25 August 2019. [Online]. Available: https://medium.com/analytics-vidhya/why-is-scaling-required-in-knn-and-k-means-8129e4d88ed7. [Accessed 26 October 2021].

[37] "ntroduction to Feature Selection methods with an example (or how to select the right variables?)," Analytics Vidhya, 1 December 2016. [Online]. Available: https://www.analyticsvidhya.com/blog/2016/12/introduction-to-feature-selection-methods-with-an-example-or-how-to-select-the-right-variables/. [Accessed 26 October 2021].

[38] R. Deshpande, "Precision and Recall in Machine Learning," Medium, 27 June 2020. [Online]. Available: https://medium.com/analytics-vidhya/precision-and-recall-in-machine-learning-c8a1b9638eeb. [Accessed 21 October 2021].

[39] S. A. Gotke, "Most Popular Distance Metrics Used in KNN and When to Use Them," KDnuggets, 11 November 2020. [Online]. Available: https://www.kdnuggets.com/2020/11/most-popular-distance-metrics-knn.html. [Accessed 30 August 2021].

[40] D. Soni, "What is Bayes Rule?," Towards Data Science, 10 May 2018. [Online]. Available: https://towardsdatascience.com/what-is-bayes-rule-bb6598d8a2fd. [Accessed 31 August 2021].

[41] D. Agarwal, "Introduction to SVM(Support Vector Machine) Along with Python Code," Analytics Vidhya, 26 April 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/04/insight-into-svm-support-vector-machine-along-with-code/. [Accessed 2 September 2021].

[42] J. Mohajon, "Confusion Matrix for Your Multi-Class Machine Learning Model," Towards Data Science, 29 May 2020. [Online]. Available: https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826. [Accessed 2 September 2021].

[43] S. Loukas, "Multi-class Classification: Extracting Performance Metrics From The Confusion Matrix," Towards Data Science, 19 June 2020. [Online]. Available: https://towardsdatascience.com/multi-class-classification-extracting-performance-metrics-from-the-confusion-matrix-b379b427a872. [Accessed 2 September 2021].

[44] H. Ampadu, "Random Forests Understanding," AI Pool, 1 May 2021. [Online]. Available: https://ai-pool.com/a/s/random-forests-understanding. [Accessed 24 October 2021].

Appendix A:

The complete Python code for this project:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sklearn
import seaborn as sn
import time
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Importing the dataset into a dataframe
data = pd.read_csv('Binary Dataset.csv')

X = data.iloc[:, 0: 10].values
Y = data.iloc[:, -1].values

# Splitting the dataset into testing and training subsets
training_X, testing_X, training_Y, testing_Y = train_test_split(X, Y, test_size = 0.3, random_state = 42)

# Scaling the data
sc = StandardScaler()
sc.fit(training_X, testing_X)
training_X_std = sc.transform(training_X)
testing_X_std = sc.transform(testing_X)

# Naive Bayes - Binary Classification
from sklearn.naive_bayes import GaussianNB

# Generate starting timestamp
start_time = time.time()

# Creating the classifier
NB_model = GaussianNB()
# Classifier is trained with training set
NB_model.fit(training_X_std, training_Y)

# Predicting testing set
predict_Y = NB_model.predict(testing_X_std)

finish_time = time.time()

# Creating the confusion matrix and performance metrics
cm = confusion_matrix(testing_Y, predict_Y)
acc = accuracy_score(testing_Y, predict_Y)
pre = precision_score(testing_Y, predict_Y)
rec = recall_score(testing_Y, predict_Y)
f_mea = f1_score(testing_Y, predict_Y)
process_time = finish_time - start_time

# Outputting confusion matrix
tick_labels = ["Benign", "Malicious"]
ax = sn.heatmap(cm / np.sum(cm), annot = True, fmt = '.2%',
            xticklabels = tick_labels, yticklabels = tick_labels, cmap = 'Blues')
ax.set_title('Confusion Matrix for Naive Bayes classifier');
ax.set_xlabel('Predicted')
ax.set_ylabel('True')
plt.show()

# Displaying perfomance metrics
print("Accuracy score: " + str(acc * 100) + "%")
print("Precision score: " + str(pre * 100) + "%")
print("Recall score: " + str(rec * 100) + "%")
print("f-measure score: " + str(f_mea * 100) + "%")
print("Processing time: " + str(process_time) + " seconds")

# Random Forest - Binary Classification
from sklearn.ensemble import RandomForestClassifier

start_time = time.time()

RF_model = RandomForestClassifier(n_estimators = 18)

RF_model.fit(training_X_std, training_Y)

predict_Y = RF_model.predict(testing_X_std)

finish_time = time.time()
```

```
cm = confusion_matrix(testing_Y, predict_Y)
acc = accuracy_score(testing_Y, predict_Y)
pre = precision_score(testing_Y, predict_Y)
rec = recall_score(testing_Y, predict_Y)
f_mea = f1_score(testing_Y, predict_Y)
process_time = finish_time - start_time

tick_labels = ["Benign", "Malicious"]
ax = sn.heatmap(cm / np.sum(cm), annot = True, fmt = '.2%',
           xticklabels = tick_labels, yticklabels = tick_labels, cmap = 'Blues')
ax.set_title('Confusion Matrix for Random Forest classifier');
ax.set_xlabel('Predicted')
ax.set_ylabel('True')
plt.show()

print("Accuracy score: " + str(acc * 100) + "%")
print("Precision score: " + str(pre * 100) + "%")
print("Recall score: " + str(rec * 100) + "%")
print("f-measure score: " + str(f_mea * 100) + "%")
print("Processing time: " + str(process_time) + " seconds")

# k Nearest Neighbours - Binary Classification
from sklearn.neighbors import KNeighborsClassifier

start_time = time.time()

kNN_model = KNeighborsClassifier(n_neighbors = 4)

kNN_model.fit(training_X_std, training_Y)

predict_Y = kNN_model.predict(testing_X_std)

finish_time = time.time()

cm = confusion_matrix(testing_Y, predict_Y)
acc = accuracy_score(testing_Y, predict_Y)
pre = precision_score(testing_Y, predict_Y)
rec = recall_score(testing_Y, predict_Y)
f_mea = f1_score(testing_Y, predict_Y)
process_time = finish_time - start_time



tick_labels = ["Benign", "Malicious"]
ax = sn.heatmap(cm / np.sum(cm), annot = True, fmt = '.2%',
           xticklabels = tick_labels, yticklabels = tick_labels, cmap = 'Blues')
ax.set_title('Confusion Matrix for KNN classifier');
ax.set_xlabel('Predicted')
ax.set_ylabel('True')
plt.show()

print("Accuracy score: " + str(acc * 100) + "%")
print("Precision score: " + str(pre * 100) + "%")
print("Recall score: " + str(rec * 100) + "%")
print("f-measure score: " + str(f_mea * 100) + "%")
print("Processing time: " + str(process_time) + " seconds")

# Support Vector Machines - Binary Classification
from sklearn import svm

start_time = time.time()

SVM_model = svm.SVC(kernel='poly')

SVM_model.fit(training_X_std, training_Y)

predict_Y = SVM_model.predict(testing_X_std)

finish_time = time.time()

cm = confusion_matrix(testing_Y, predict_Y)
acc = accuracy_score(testing_Y, predict_Y)
pre = precision_score(testing_Y, predict_Y)
rec = recall_score(testing_Y, predict_Y)
f_mea = f1_score(testing_Y, predict_Y)
process_time = finish_time - start_time
```

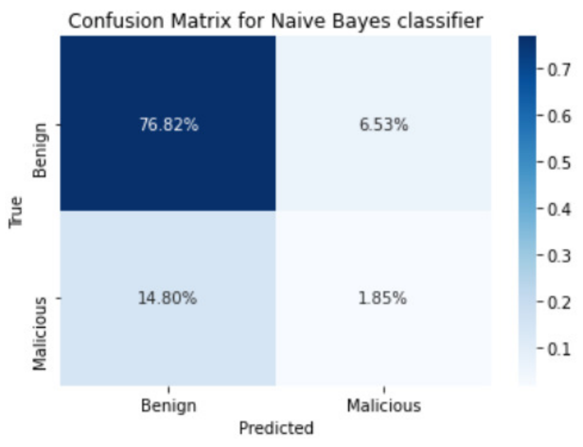
```
tick_labels = ["Benign", "Malicious"]
ax = sn.heatmap(cm / np.sum(cm), annot = True, fmt = '.2%',
           xticklabels = tick_labels, yticklabels = tick_labels, cmap = 'Blues')
ax.set_title('Confusion Matrix for SVM classifier');
ax.set_xlabel('Predicted')
ax.set_ylabel('True')
plt.show()

print("Accuracy score: " + str(acc * 100) + "%")
print("Precision score: " + str(pre * 100) + "%")
print("Recall score: " + str(rec * 100) + "%")
print("f-measure score: " + str(f_mea * 100) + "%")
print("Processing time: " + str(process_time) + " seconds")

# Changing dataset
data = pd.read_csv('Malicious Dataset.csv')

X = data.iloc[:, 0: 10].values
Y = data.iloc[:, -1].values

# Splitting the dataset into testing and training subsets
training_X, testing_X, training_Y, testing_Y = train_test_split(X, Y, test_size = 0.3, random_state = 42)

# Scaling the data
sc = StandardScaler()
sc.fit(training_X, testing_X)
training_X_std = sc.transform(training_X)
testing_X_std = sc.transform(testing_X)

# Naive Bayes - Multiclass Classification
from sklearn.naive_bayes import MultinomialNB

start_time = time.time()

NB_model = MultinomialNB()

NB_model.fit(training_X, training_Y)

predict_Y = NB_model.predict(testing_X)
```

```
finish_time = time.time()

cm = confusion_matrix(testing_Y, predict_Y)
acc = accuracy_score(testing_Y, predict_Y)
pre = precision_score(testing_Y, predict_Y, average = 'weighted')
rec = recall_score(testing_Y, predict_Y, average = 'weighted')
f_mea = f1_score(testing_Y, predict_Y, average = 'weighted')
process_time = finish_time - start_time

tick_labels = ["Trojan", "Downloader", "Miner", "Ransom"]
ax = sn.heatmap(cm / np.sum(cm), annot = True, fmt = '.2%',
           xticklabels = tick_labels, yticklabels = tick_labels, cmap = 'Blues')
ax.set_title('Confusion Matrix for Naive Bayes classifier');
ax.set_xlabel('Predicted')
ax.set_ylabel('True')
plt.show()

print("Accuracy score: " + str(acc * 100) + "%")
print("Precision score: " + str(pre * 100) + "%")
print("Recall score: " + str(rec * 100) + "%")
print("f-measure score: " + str(f_mea * 100) + "%")
print("Processing time: " + str(process_time) + " seconds")

# Random Forest - Multiclass Classification

start_time = time.time()

RF_model = RandomForestClassifier(n_estimators = 13)

RF_model.fit(training_X_std, training_Y)

predict_Y = RF_model.predict(testing_X_std)

finish_time = time.time()
```

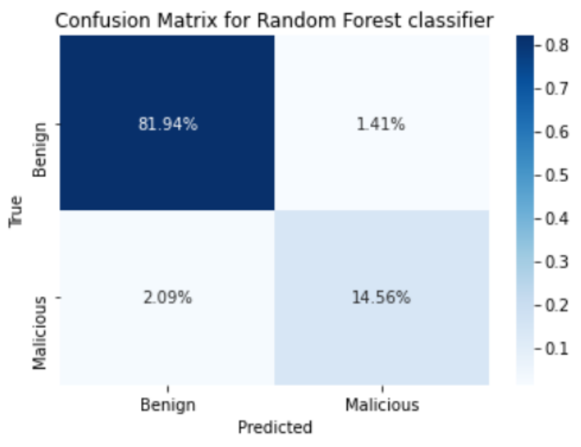
```
cm = confusion_matrix(testing_Y, predict_Y)
acc = accuracy_score(testing_Y, predict_Y)
pre = precision_score(testing_Y, predict_Y, average = 'weighted')
rec = recall_score(testing_Y, predict_Y, average = 'weighted')
f_mea = f1_score(testing_Y, predict_Y, average = 'weighted')
process_time = finish_time - start_time

tick_labels = ["Trojan", "Downloader", "Miner", "Ransom"]
ax = sn.heatmap(cm / np.sum(cm), annot = True, fmt = '.2%',
           xticklabels = tick_labels, yticklabels = tick_labels, cmap = 'Blues')
ax.set_title('Confusion Matrix for Random Forest classifier');
ax.set_xlabel('Predicted')
ax.set_ylabel('True')
plt.show()

print("Accuracy score: " + str(acc * 100) + "%")
print("Precision score: " + str(pre * 100) + "%")
print("Recall score: " + str(rec * 100) + "%")
print("f-measure score: " + str(f_mea * 100) + "%")
print("Processing time: " + str(process_time) + " seconds")

# k Nearest Neighbours - Multiclass Classification

start_time = time.time()

kNN_model = KNeighborsClassifier(n_neighbors = 4)

kNN_model.fit(training_X_std, training_Y)

predict_Y = kNN_model.predict(testing_X_std)

finish_time = time.time()

cm = confusion_matrix(testing_Y, predict_Y)
acc = accuracy_score(testing_Y, predict_Y)
pre = precision_score(testing_Y, predict_Y, average = 'weighted')
rec = recall_score(testing_Y, predict_Y, average = 'weighted')
f_mea = f1_score(testing_Y, predict_Y, average = 'weighted')
process_time = finish_time - start_time


  tick_labels = ["Trojan", "Downloader", "Miner", "Ransom"]
  ax = sn.heatmap(cm / np.sum(cm), annot = True, fmt = '.2%',
             xticklabels = tick_labels, yticklabels = tick_labels, cmap = 'Blues')
  ax.set_title('Confusion Matrix for KNN classifier');
  ax.set_xlabel('Predicted')
  ax.set_ylabel('True')
  plt.show()

  print("Accuracy score: " + str(acc * 100) + "%")
  print("Precision score: " + str(pre * 100) + "%")
  print("Recall score: " + str(rec * 100) + "%")
  print("f-measure score: " + str(f_mea * 100) + "%")
  print("Processing time: " + str(process_time) + " seconds")

  # Support Vector Machines - Multiclass Classification

  start_time = time.time()

  SVM_model = svm.SVC(kernel='poly')

  SVM_model.fit(training_X_std, training_Y)

  predict_Y = SVM_model.predict(testing_X_std)

  finish_time = time.time()

  cm = confusion_matrix(testing_Y, predict_Y)
  acc = accuracy_score(testing_Y, predict_Y)
  pre = precision_score(testing_Y, predict_Y, average = 'weighted')
  rec = recall_score(testing_Y, predict_Y, average = 'weighted')
  f_mea = f1_score(testing_Y, predict_Y, average = 'weighted')
  process_time = finish_time - start_time

  tick_labels = ["Trojan", "Downloader", "Miner", "Ransom"]
  ax = sn.heatmap(cm / np.sum(cm), annot = True, fmt = '.2%',
             xticklabels = tick_labels, yticklabels = tick_labels, cmap = 'Blues')
  ax.set_title('Confusion Matrix for SVM classifier');
  ax.set_xlabel('Predicted')
  ax.set_ylabel('True')
  plt.show()


 print("Accuracy score: " + str(acc * 100) + "%")
 print("Precision score: " + str(pre * 100) + "%")
 print("Recall score: " + str(rec * 100) + "%")
 print("f-measure score: " + str(f_mea * 100) + "%")
 print("Processing time: " + str(process_time) + " seconds")
```

Appendix B:

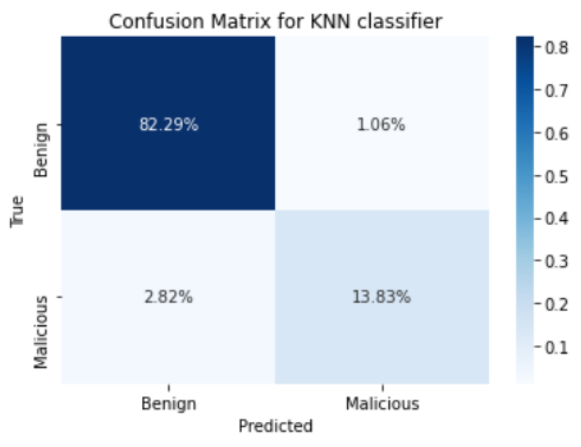
Confusion Matrix for Naïve Bayes - Binary Classification:



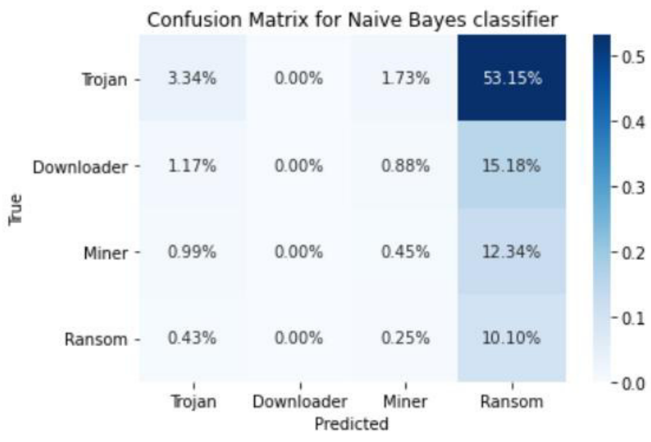
Confusion Matrix for Random Forest - Binary Classification:



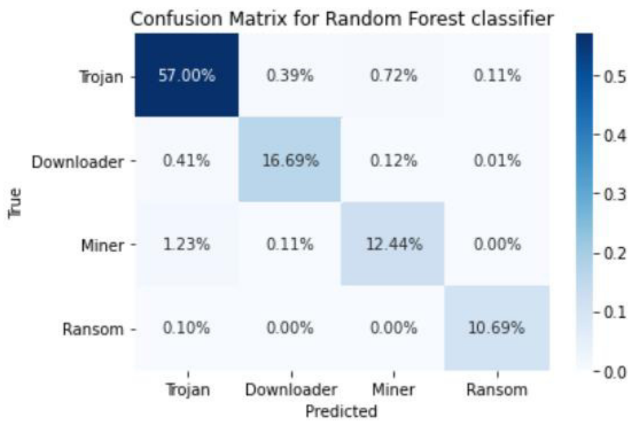
Confusion Matrix for k-Nearest Neighbours - Binary Classification:



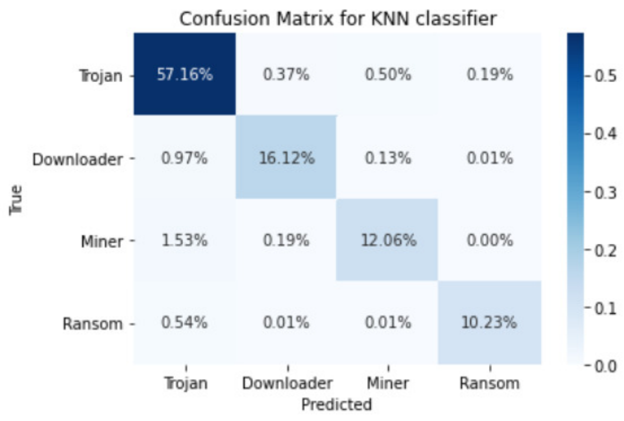
Confusion Matrix for Naïve Bayes - Multiclass Classification:



Confusion Matrix for Random Forest - Multiclass Classification:



Confusion Matrix for k-Nearest Neighbours - Multiclass Classification:


Confusion Matrix for KNN classifier
Trojan
Downloader
Miner
Ransom
True
Predicted