UNIVERSITY OF CAMBRIDGE

Kyra Mozley
Murray Edwards College

Machine Learning for the Detection of Network Attacks

Computer Science Tripos - Part II

May 2020

Contents

1 Introduction **7**

1.1 Motivation 7

1.2 Previous Work 8

1.3 Challenges 8

1.4 Project Description 9

1.5 Overview of the Dissertation 9

2 Preparation **10**

2.1 Background Theory 10

2.1.1 Machine Learning Overview 10

2.1.2 Feature Selection 12

2.1.3 Machine Learning Algorithms 12

2.1.4 Evaluation Metrics 13

2.1.5 Networking 14

2.1.6 Network Attacks 15

2.2 Requirements Analysis 16

2.3 Choice of Tools 17

2.3.1 Programming Language 17

2.3.2 Development 17

2.3.3 Libraries 18

2.3.4 Dataset 18

2.4 Development Approach 19

2.5 Starting Point 19

2.6 Summary 19

3 Implementation **20**

3.1 Preprocessing . . . 20
3.2 Feature Selection . . . 22
3.3 Classification . . . 23
3.3.1 Implemented Models . . . 23
3.3.2 Implementing Classification . . . 24
3.4 Building the Intrusion Detection System . . . 25
3.4.1 Identifying a Flow . . . 26
3.4.2 Feature Extraction . . . 28
3.5 Generating Network Attacks . . . 28
3.5.1 Bot . . . 28
3.5.2 DoS and DDoS . . . 28
3.5.3 FTP and SSH Patator . . . 29
3.5.4 Probe . . . 29
3.5.5 Web Attack . . . 29
3.6 Repository Overview . . . 30
3.7 Summary . . . 30

4 Evaluation **31**

4.1 Evaluation of Different Learning Algorithms . . . 31
4.1.1 Initial Testing Results . . . 31
4.1.2 Model Refinement . . . 34
4.2 Final Testing Results . . . 40
4.3 Building the Intrusion Detection System . . . 42
4.4 Evaluation of the Intrusion Detection System . . . 43
4.4.1 Trained Network Attacks . . . 43
4.4.2 Unseen Network Attacks . . . 44
4.5 Summary . . . 45

5 Conclusion **46**

5.1 Success Criteria . . . 46
5.2 Lessons Learned . . . 46
5.3 Future Work . . . 47
5.4 Final Remarks . . . 47

A Full Results **52**

B Project Proposal **56**

List of Figures

3.1 Cumulative Feature Scores 22

3.2 KNN Precision, Recall and F1 Score for Different Values of K, of the Original Attack Labels on the Validation Dataset. 25

3.3 The IDS Pipeline 26

3.4 Device Set up for Attacks 28

3.5 Structure of Source Code Repository 30

4.1 Precision, Recall and Accuracy Scores for the Three Different Label Groupings, Across all the Different Supervised Algorithms, Tested on the Validation Dataset. 32

4.2 PCA of Actual Labels (left) and Predicted Clusters (right) on the Original Labels of the Validation Dataset 33

4.3 F1 Scores of Decision Tree and K Nearest Neighbour 35

4.4 F1 Scores of Decision Tree and Random Forest 36

4.5 Confusion Matrix for Random Forest on Validation Dataset 38

4.6 The Effect on F1 Score from Changing the Number of Estimators Parameter 40

4.7 The Effect on Classification Time from Changing the Number of Estimators Parameter 40

4.8 Confusion Matrix of Random Forest on Testing Dataset 41

List of Tables

2.1 Confusion Matrix for Binary Classification 14

2.2 Different Layers, Uses and Common Protocols of the TCP/IP Stack 15

2.3 Requirements for my Project to be a Success 17

3.1 Distribution of Labels in the Dataset 20

3.2 Grouping of Original Attacks into more General Categories 21

3.3 Description of Features Chosen 23

4.1 Predicted Clustering Result (K=7) Against True Grouped Labels 34

4.2 Predicted Clustering (K=2) Result Against True Binary Labels 34

4.3 Training and Classification Times for Decision Tree and K Nearest Neighbours 36

4.4 Training and Classification Times for Decision Tree and K Nearest Neighbours 37

4.5 Grouping of Original Attacks into more General Categories 38

4.6 Final Labels and their Groupings from the Original Attacks 39

4.7 Precision, Recall and F1 Score Achieved for the Different Labels Compared to the Percentage of the Dataset that was that Label 42

4.8 Success of Detecting Seen Attacks 43

4.9 Success of Detecting Unseen Attacks 44

Chapter 1

Introduction

This dissertation aims to use machine learning approaches to detect network attacks, to analyse data on the network and classify in real-time whether or not a particular behaviour is benign or an attack. To achieve this goal, I shall evaluate several machine learning algorithms following the selection of a suitable dataset, and implement a machine learning-based intrusion detection system to handle DoS, brute force, and web type attacks.

1.1 Motivation

Historically, people have used intrusion detection systems (IDS) to protect their networks against adversaries.[1] This involves monitoring the network and detecting network attacks through the use of attack signatures; when the traffic matches a known predefined attack pattern, it raises that there has been an attack. Traditional IDS are useful in detecting known attacks but fail in the event of an unseen attack, leaving them hopeless, and the network vulnerable against zero-day exploits. Additionally, the emergence of new attack patterns necessitates the update of the signature database containing the definition of attacks.

An alternative and complementary solution is the application of machine learning approaches as the basis of attack detection. Machine learning use has rapidly grown over the past decade, with applications in many different areas such as health-care, product recommendation and email spam filtering.[2] A category of machine learning algorithms, supervised techniques, allow a system to 'learn' from past events to be able to predict future outcomes, making it perfect for use in detecting new network attacks that would go undetected by a traditional IDS. Another method, unsupervised learning, is also considered. This approach models normal network behaviour and raises alarms when anomalous instances are detected.[3] This project follows a supervised approach and makes use of labelled network data during training.

1.2 Previous Work

Applying machine learning to intrusion detection has been explored in various papers. *Machine Learning Approach to IDS: A Comprehensive Review*[4] provides a good overview of previous papers in this topic, the algorithms used, and their respective detection accuracy. The range of IDS accuracy for the papers listed in the survey was between 88.0% and 99.9%, with the median accuracy being 97.2%. The paper which had the highest accuracy[5] uses the KDD Cup 1999 dataset[6], partitions the data into two classes (binary or attack), and trains using a support vector machine. This review provides us with an idea of the values that we should expect to reach in this dissertation.

The majority of the papers referenced in the review use the KDD Cup 1999 dataset. It contains nearly five million data points, 41 extracted connection features, and 24 different types of attacks. Labelled attack data accounts for 80.3% of the dataset. Its size makes it very appealing as it contains a vast number and variety of attacks. However, it is now two decades old, making it outdated, irrelevant and not suitable to be used in this domain. Furthermore, it does not represent realistic network traffic since there is too little benign labelled data, so should not be used in systems aimed to be deployed in a real network environment.

1.3 Challenges

Using machine learning for intrusion detection poses some challenges which need to be addressed. Firstly, as previously hinted, there is a lack of availability of suitable datasets. Reasons behind this range from being out of date (like the KDD Cup 1999), not covering a wide range of different attacks, or are not made public due to the privacy concerns associated with sharing network data. To combat the issue of datasets containing personally identifiable information, they may be anonymised, but this can affect the quality of the data provided.[7]

Typical signature-based systems have to have their rules frequently updated to recognise new attacks. However, with a learning-based system, how do we know when we would need to update it with new training data? We do not want an attack to occur for us to realise our training data is now out of date. Furthermore, we need to consider the scalability of a proposed system. Servers have a higher volume of traffic than a single machine, and so the viability of the processing delays must be considered.

Ultimately, the nature of the problem leads it to be challenging to test. We postulate that using machine learning in an IDS could allow us to detect unseen attacks, but papers lack testing to see if that is the case. We are trying to prepare ourselves against something that we have not yet seen, and do not know what form the attack may take. Given that the cost of errors is much higher than other machine learning applications, we need to be

confident in our solution. The best we can do is test our IDS on attacks which it was not trained on and observe its success in detecting those, and remain hopeful that it will also be able to detect zero-day attacks.

1.4 Project Description

The central premise of this dissertation is to implement a machine learning-based intrusion detection system. To do so, we start by researching different learning algorithms which may be suitable for use in the project. We then find a suitable dataset to train the chosen algorithms on, and then test and evaluate these results to build a final model. Next, we implement a sniffer tool to extract the chosen features and integrate the model to create our IDS, which can classify traffic in real-time. Finally, we evaluate the performance of the IDS by simulating a variety of network attacks and observe the classification outcome.

1.5 Overview of the Dissertation

This dissertation is split into five different chapters:

- *Chapter One: Introduction:* In this chapter, I have presented the motivation behind using machine learning for intrusion detection, previous work that relates to the project, and the project description.
- *Chapter Two: Preparation:* I cover the background theory needed for this dissertation, the requirements for the project, choice of tools, and the developmental approach used.
- *Chapter Three: Implementation:* I provide details of the steps taken to implement the machine learning algorithms, build the IDS, and to perform the network attacks.
- *Chapter Four: Evaluation:* I discuss the results of the machine learning algorithms, and work towards building my final model. I then implement this into my IDS and evaluate its performance on simulated network attacks.
- *Chapter Five: Conclusion:* I review the work completed, giving final remarks on the success, as well as possible future work.

Chapter 2

Preparation

In this chapter, I discuss the relevant background knowledge needed for this dissertation, what steps were required to make the project a success, my choice of tools, and the developmental approach I took.

2.1 Background Theory

Throughout this dissertation, I refer to different machine learning models and various terminology related to how to evaluate them. Here, I briefly provide an overview of machine learning, how each of the different algorithms works, and how to calculate the relevant metrics needed. I also present the relevant networking theory and discuss the various types of network attacks that the model will be able to detect.

2.1.1 Machine Learning Overview

Machine learning refers to algorithms that can automatically describe data with little or no human involvement.[8] The workflow for a machine learning project consists of preparing the inputs (features) for the model, completing the training, and finally, conducting testing of the model to see how well it performs. The input is usually obtained from some dataset relevant to the problem.

Features are measurements used to describe the object we wish to analyse. They may be predefined by the dataset already or may require extraction. It is important to select features which are relevant to the problem domain.[9] A **feature vector** is an n-dimensional array, where n is the number of features chosen. Many algorithms take this feature vector as input to the classifier.

In training, we take the feature vectors from the training dataset and fit the data to a model. Once we have our machine learning model, we can then begin to make predictions

as we pass it new feature vectors, where the new data is either from the evaluation/test dataset or extracted from a real-life environment.

Evaluating the dataset allows us to refine our model further, while testing can provide us with our final results as to how well it performs.

Types of Machine Learning

There exist different types of learning. In **supervised learning**, we associate a feature vector in training with a label, such that the model learns how to predict a correct label based on the given feature vectors. Hence, when the system is deployed and passed with new feature labels, it can predict what the corresponding label is.[8]

Learning from an unlabelled dataset is **unsupervised learning**. Here we use the training dataset to try to find patterns or groupings that exist in the data, and when passed a new feature vector, it returns which group it belonged to.[8]

Semi-supervised, reinforcement and deep learning are other types of learning approaches. Omitted are the details of how they work as they are not used in this project.

Machine Learning Tasks

Machine learning solves many different types of problems; we can group these problems into the following tasks:[10]

- **Classification** outputs a predicted label from some predefined set.
- **Regression** outputs a predicted real-valued numerical result.
- **Clustering** groups objects such that objects within the same cluster have similar properties compared to that of other clusters. There may or may not be a predefined number of clusters to fit the training data to.

Both classification and clustering are types of approaches which are suitable for this project. Classification aims to predict a category and assign the labels benign or attack. While clustering aims to divide the data into groups that are similar to each other. Here, we cluster the data into normal and anomalous cases.

Challenges

Machine learning is not without its challenges. The first main challenge lies in choosing a useful dataset. It is vital that the dataset is relevant to the task at hand and is sufficient in size so that the model has enough data to learn from. If performing supervised learning,

it adds the requirement to find labelled data, which may be hard to find, or we may manually assign labels which can be time-consuming.

Overfitting is another prominent issue in machine learning; it occurs when your model does not generalise well to new data. It can appear when you refine your model too much that it also has learnt all of the noise in the dataset, meaning that although the performances seem to be making improvements, it performs worse when you come to test it on new data.[9]

In the context of this project, further challenges arise, as previously introduced in Section 1.3. The data may contain personally identifiable information, such as IP addresses, so it is essential that it is anonymised, or the traffic simulated such that it contains no personal data. Furthermore, another issue faced is that there exists much variability in what is benign data, making it difficult for the model to learn what is 'normal' traffic, suggesting it may struggle to detect new attacks. Lastly, the cost of miss-classification is high. If the IDS flags benign traffic as attacks too often, it can result in the system being unusable. While if it classifies attacks as benign, this can cause severe damage since we are not registering that there has been an attack, leaving the network compromised. Thus, the system must have high performance.

2.1.2 Feature Selection

Feature selection is essential in any machine learning model. It should be the case that the features included are those who improve the accuracy of the model.[11] Feature selection allows us to reduce the number of data points, which in turn allows it to train faster and reduces the chance of overfitting.

2.1.3 Machine Learning Algorithms

Provided is an introduction to the different machine learning models that are referred to throughout this dissertation.

Support Vector Machine

A support vector machine (SVM) is a supervised, linear classifier. They create a hyperplane, and we aim to find the maximum-margin hyperplane which divides our feature vectors of one class from the set of feature vectors of another class. This division is a decision boundary to classify the data points.[12]

Decision Tree

Decision trees are a supervised method, often used for classification. To predict the class label for a given input, we begin at the root node and chose the successor node based on the value of a specific feature. We repeat this until we reach a leaf node, which defines the predicted class of this data.[12]

Naive Bayes

The Naive Bayes algorithm is a supervised simple probabilistic classifier based on applying Bayes' theorem. We assume that features are independent of each other, known as the naive assumption. To classify an input, we pick the class with the highest posterior probability, $P(y|\mathbf{x})$, where $\mathbf{x}$ is the feature vector and y is the class.[15]

K Nearest Neighbours

The K nearest neighbours (K-NN) algorithm is a supervised algorithm that works by assuming that data points of the same class will exist nearby. The K refers to the number of neighbours we look at the classification of to determine the class of input data.[16]

K Means Clustering

K means cluster is an unsupervised clustering algorithm that groups objects based on their feature values into K disjoint clusters (where K is a positive integer specifying the number of clusters). It clusters data based on the Euclidean distance between the data, classifying objects into the same cluster if they have similar feature values.[17]

2.1.4 Evaluation Metrics

We use metrics to judge the quality of the different machine learning algorithms that we have presented. We explore the appropriate metrics for this project.

A **confusion matrix** allows us to visualise the performance of the classification algorithm; it is a table whose results are divided into actual and predicted classes.[18] Table 2.1 shows a confusion matrix for a binary classification algorithm.

Accuracy is the number correct classifications, divided by the total number of classifications.

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

Although this may seem like a simple metric to use, it can fail to describe the true performance if we have highly imbalanced classes. [18]

	Predicted Attack	Predicted Benign
Truly Attack	True Positive (TP)	False Negative (FN)
Truly Benign	False Positive (FP)	True Negative (TN)

Table 2.1: Confusion Matrix for Binary Classification

We define **precision** as the ratio of the number of class predictions that truly were that class, over the number that was predicted that class.[12]

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

A low precision indicates that we have a large number of false positives in our results. Classifying truly benign data as an attack too often, the false-positive case, can render the system unusable, a situation we want to avoid.

Recall is the ratio of the number of class predictions that truly were that class, over the number of true occurrences of that class.

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

A low recall indicates that we have a large number of false negatives in our results. As previously mentioned, we wish to avoid classifying attacks as benign; this is the false-negative case and would lead to undetected network attacks. So we must ensure the recall value of our system is as close to 100% as possible to avoid this.

The **F1 score** allows us to combine the precision and recall values, and is their harmonic mean.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

By combining both the precision and recall into a single metric, it provides a balance between the two values. Since precision and recall values are important to consider in this context, we use the F1 score to evaluate the performance of our classifier.

2.1.5 Networking

The different attacks that we look at in this dissertation are network attacks; so we provide a brief overview of common protocols and theory related to networking.

We can use an abstraction of layers to describe how devices communicate across a network. Divided into four layers, the **TCP/IP model** describes how to handle and transfer data, where each layer provides a specific service. A layer uses information received from the layer below and provides information to the one above. It provides information as to how data to be sent across the network should be dealt with.[19] Table 2.2 shows the four

layers of the stack, a description of their purpose and common protocols associated with that layer. The attacks we look at in this dissertation mostly occur in the application and transport layers.

Layer	Function	Protcols
Application Layer	Interacts with the application program	FTP, HTTP, SMTP
Transport Layer	Transports data across the network	TCP, UDP
Network Layer	Defines the routing of the packets in the network	IP, ICMP
Link Layer	Describes how to physically send data throught the network	Ethernet, ARP

Table 2.2: Different Layers, Uses and Common Protocols of the TCP/IP Stack

The two most known protocols at the transport layer are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP ensures reliable, in-order delivery of packets by establishing a connection between the client and server.[20] The connection is created using the TCP three-way handshake. On the other hand, UDP is a connectionless protocol; this makes it simpler than TCP but also means it is less reliable.

The protocols in the application layer allow a user to interact and offer many different functionalities. Telnet and SSH (Secure Shell) provide a remote terminal, FTP (File Transfer Protocol) and TFTP (Trivial File Transfer Protocol) offer file transfer, and SMTP (Simple Mail Transfer Protocol) offers email support, to name a few.

2.1.6 Network Attacks

We provide a high-level classification overview of the different types of network attacks that are common today.[21]

Botnet

A botnet is a collection of compromised devices (bots) that are connected together and controlled by the adversary to perform malicious tasks. They are used to perform DDoS attacks, send spam emails, distribute further malware, and many other cybercrimes.[22]

Brute Force

The attacker aims to gain unauthorised access to information by repeated trial-and-error. Often, the attacker brute forces passwords and login credentials, but may also try to discover hidden web pages on a site.[7]

Denial of Service

A Denial of Service (DoS) attack is where an adversary attempts to overwhelm the victim, making it unavailable for its legitimate users. There are many different types of DoS attacks, occurring all across the different layers of the network stack, but most commonly the application or transport layer.

A Distributed Denial of Service (DDoS) attack also aims to paralyse the victim machine. It floods the victim machine with a large volume of traffic from multiple machines. To achieve a large number of devices, and hence more volume of data, the DDoS attack makes use of compromised hosts (bots).[23]

Web Attack

Vulnerabilities for web attacks exist mainly in the application layer, and likely occur from poor programming practices.[24] Code injection attacks occur when the attacker 'injects' code into an application, which can lead them to gain access to information they should not be able to have, or worse, the system itself. At the time of writing, The Open Web Application Security Project (OWASP)[25] lists injection as the top web application security risk. Cross-site scripting (XSS) occurs when an application includes untrusted data without proper validation or escaping; this can allow the attacker to deface sites, hijack user sessions, or redirect them to malicious sites.[33]

2.2 Requirements Analysis

The success criteria of this project were defined as (for further explanation, see Appendix B for the project proposal)

- Implemented at least one machine learning algorithm and evaluated its precision on a proportion of the dataset reserved for testing.
- Successfully developed an interface for the intrusion detection system, which can sniff and monitor traffic and alert on a suspected attack.
- Evaluated the IDS detection accuracy when tested on simulated network traffic.

Using these, we can construct the tasks that need to be completed, Table 2.3 shows these alongside their priority, difficulty and risk class. Priority rankings were assigned based on how important the step was in terms of making progress (i.e. do future tasks rely on this step). The difficulty was assigned based on whether I currently knew how to perform the step. Deciding the risk level was a combination of current knowledge as to how to complete the task, and how important the task was to the overall success criteria.

No.	Task	Priority	Difficulty	Risk
1	Find a suitable dataset	High	Medium	High
2	Preprocess the dataset	Medium	Low	Low
3	Perform feature selection on the dataset	Medium	Medium	Low
4	Perform the chosen five machine learning algorithms on the data, and evaluate using the validation set	High	High	Medium
5	Compare the different models to create the final model	Medium	Low	Low
6	Evaluate the final performance using the test set	Medium	Low	High
7	Build a system that can extract the chosen features from real-time traffic	High	High	High
8	Combine the machine learning model from task 5 and system from task 7 to create the intrusion detection system	Medium	High	Medium
9	Evaluate the performance of the IDS created in task 8 by simulating a variety of network attacks	Medium	High	High

Table 2.3: Requirements for my Project to be a Success

2.3 Choice of Tools

Deciding the right tools to use is essential for any project as it affects the difficulty of implementation and availability of other tools or libraries that may be required.

2.3.1 Programming Language

Python seemed the most appropriate language of choice for the project; its use is common for machine learning programs due to the availability of many libraries, which makes implementing easier. Although it meant that I had to spend some time at the start familiarising myself with the language, it was the right choice for this project.

2.3.2 Development

For the initial learning part of the project, I used a Jupyter Notebook[1]. Jupyter Notebooks allow for efficient testing since you can run the cells individually, and so do not have to run the whole program each time you make a change. It is beneficial when performing machine learning since you do not have to run the pre-processing each time. Also, the notebook provides visual outputs so that you can see graphs or tables directly.

For building the detection system, the IDE of choice was JetBrains' PyCharm[2]. PyCharm offers many useful tools such as auto-complete, refactoring and version control.

[1] jupyter.org [Accessed 6 May 2020]
[2] jetbrains.com/pycharm [Accessed 6 May 2020]

I used Git for version control, with a private repository on GitHub. As well as having the benefit of providing a back up of my code in the event of a hardware failure, it also allows you to revert to earlier versions in case I make too many erroneous changes. Furthermore, all files were automatically saved to Dropbox.

For reference, all testing was performed on my Apple MacBook Pro (quad-core Intel i7-8559U, 16GB RAM).

2.3.3 Libraries

I used NumPy[3] and Pandas[4] when pre-processing the dataset. NumPy allows for multidimensional arrays and can perform fast mathematical operations on them. Pandas provides us with the data frame structure, allowing us to convert easily to and from CSV files. Both Numpy and Pandas operate efficiently on large data.

For the learning phase of the project, I decided to use the Scikit-learn[5] library, which is an open-source library that works with NumPy and Pandas. It offers a wide range of tools and is simple to use, offering everything required for the learning in this project. It was not in the scope of this project to write the algorithms from scratch, as I did not require the fine-grained control gained from doing so. Hence, Scikit-learn's library was efficient and suitable for use.

Lastly, to retrieve information from the packets in my IDS, I used the Scapy[6] library. The library has tools to be able to send, sniff, and dissect packets.

2.3.4 Dataset

We require a dataset to provide us with an input to the learning algorithm, and further data to test the success of the model on.

When analysing network traffic, you can either look at flow-level or packet-level data. The flow-level analysis provides an overview of activity on the network, where a flow is a stream of data between a source and destination using a specific protocol. Packet-level data, on the other hand, captures the actual packets used in the network layer, and can, therefore, provide more in-depth analysis since it obtained the actual payloads. We look at flow-level data in this project. Although packet-level provides more data than flow-level, it would likely be far too large to efficiently monitor, hence flow-level is more appropriate for use here.

[3] numpy.org [Accessed 5 May 2020]
[4] pandas.pydata.org [Accessed 5 May 2020]
[5] scikit-learn.org [Accessed 5 May 2020]
[6] scapy.net [Accessed 5 May 2020]

I am using the CICIDS2017 dataset. It is a labelled dataset, containing nearly 3 million data-points, 80 flow-based features and 14 different types of common attacks. This dataset was the outcome of a research paper[7] to create a useful, reliable dataset for attacks. It aimed to solve the issues that the KDD99 dataset (and other accessible datasets) presented. I believe it was an appropriate dataset to use since it was labelled and offered a wide attack diversity across several different protocols. They decided which attacks to include based on the most popular listed in the 2016 McAfee report[21], resulting in relevant and up-to-date attack data.

2.4 Development Approach

The project was split into three main areas of development; initial machine learning and evaluation, building the IDS interface, and final evaluation. I performed these tasks in sequential order, all of which used an iterative development approach. An iterative approach allows you to receive feedback about the system consistently. For example, in the first section, an iterative workflow allowed me to incrementally make changes to my work in an attempt to improve the performance of the classifier.

2.5 Starting Point

I had a theoretical understanding of Machine Learning from Part IB *Machine Learning and Real-world Data*, as well as some general reading I had done. Other Tripos courses such as Part IB *Security* and Part IB *Computer Networking* also provided useful background knowledge in the area.
In terms of programming experience, I had some experience in Python and had used it for basic tasks, but never for a large project, machine learning or in a networking context. Hence, I had to gain an understanding of the libraries that I needed to use for the project.

2.6 Summary

In this chapter, I have discussed the relevant background knowledge required for this dissertation, my success criteria and project requirements, how I planned to implement the project and my starting point.

Chapter 3

Implementation

In this chapter, I present the initial steps taken to get the dataset ready for training, and how I implemented the chosen machine learning algorithms. As well as the development of the intrusion detection system, and how to perform a variety of network attacks.

3.1 Preprocessing

Preprocessing is the first step in implementation. It allows the raw data extracted from the dataset to be transformed through a series of steps such as deleting redundant records and normalising data, to get the data into the form required for learning. The data for the dataset was collected over five days, and consists of 80.3% normal traffic, with the remaining 19.7% being the fourteen types of attacks.

Table 3.1 shows the distribution of the different attacks in this dataset; it is clear that there is not sufficient data to train for Heartbleed, SQL Injection, or Infiltration. So we must drop these entries from the dataset.

Label	Entries
Benign	2,273,097
DoS Hulk	231,073
Port Scan	158,930
DDoS	128,027
DoS GoldenEye	10,293
FTP-Patator	7,938
SSH-Patator	5,897
DoS Slowloris	5,796
DoS Slowhttptest	5,499
Bot	1,966
Web Attack: Brute Force	1,507
Web Attack: XSS	652
Infiltration	36
Web Attack: SQL Injection	21
Heartbleed	11

Table 3.1: Distribution of Labels in the Dataset

As well as the original attack labels provided, I grouped the remaining eleven labels into more general categories using the mapping given in Table 3.2. I also assigned a binary (benign or attack) label to

Botnet	Bot
Brute Force	FTP-Patator, SSH-Patator
DDoS	DDoS
DoS	DoS GoldenEye, DoS Hulk, DoS Slowhttptest, DoS Slowloris
Probe	Port Scan
Web Attack	Web Attack: Brute Force, Web Attack: XSS

Table 3.2: Grouping of Original Attacks into more General Categories

each record. By evaluating each model on the 3 different grouping options, we can see if some types of attacks are more straightforward to predict than others. Or perhaps, if some models are better at predicting certain types of attacks than others, and whether grouping by the similarity of attack helps it to generalise well.

Datasets occasionally contain missing values; this can occur from errors in recording or extracting the features.[27] To deal with missing values in the dataset, I decided to drop any rows that contained NaN, Null or Inf values, as the dataset is large enough this has almost no effect on the results.

The next step was to split the dataset into training, validation, and testing datasets. The training dataset is what we use to train the model; this is the data that it learns from. Next, the validation dataset is what we use to perform initial testing and tuning of the model on, while the test dataset is what we use to evaluate the performance after we have created the final model. I chose to split the dataset using a 60:20:20 ratio to training: validation: testing. Furthermore, because the dataset was unbalanced, we perform a **stratified split** of the labels. This ensures that the datasets produced maintain the same proportions of classes as in the original, as opposed to random sampling, which splits the data randomly. Stratified sampling allows us to avoid the case were random sampling may not include enough instances of minority classes in the training dataset.

It became apparent that there are some redundant columns which I then dropped from the dataset since they were providing no useful information towards classification.

The last step in preprocessing is normalisation. Normalisation is essential since the scale of the feature values differs; some are between $[0, \infty]$ while others are between $[0, 1]$. So, by bringing all the features within the same range, we ensure that they contribute an equal amount towards the classification. I performed **min-max normalisation** using `scikit-learn`'s library. Min-max normalisation re-scales all of the features into the [0,1] range, using the following formula

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where x is the original value.

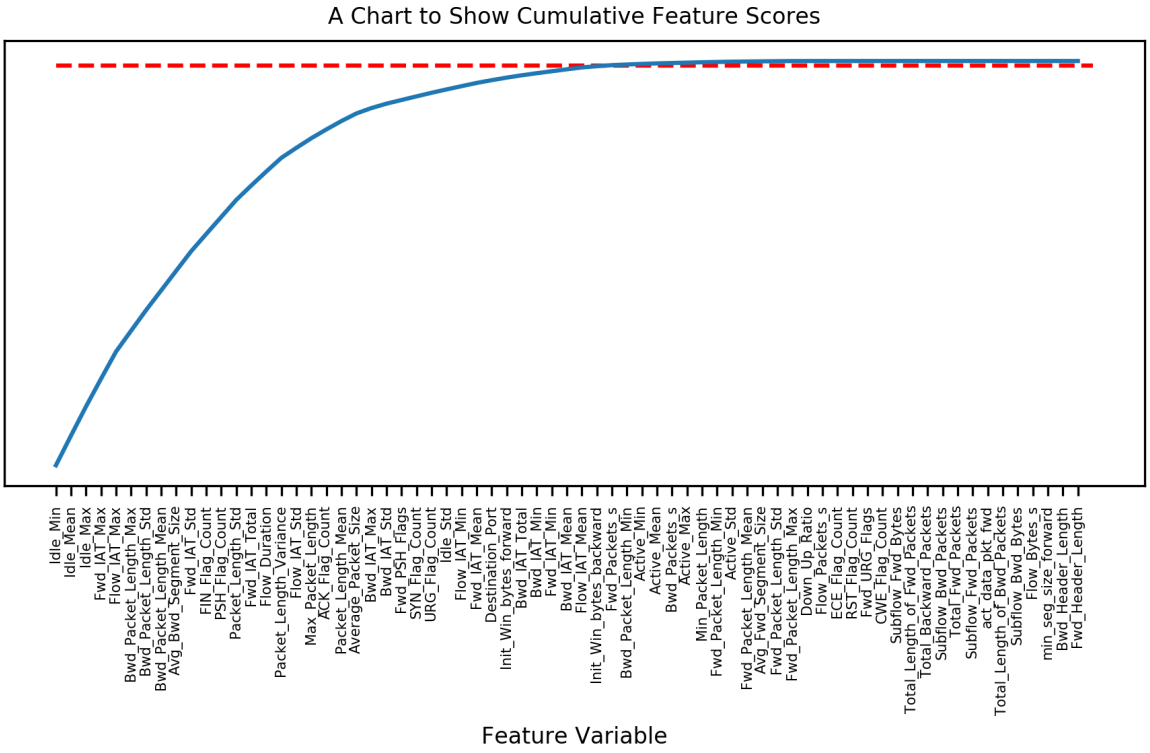

Figure 3.1: Cumulative Feature Scores

3.2 Feature Selection

As discussed in Section 2.1.2, feature selection allows us to reduce the dimensionality of our data, and select the features which are relevant to the classification. We apply the **Chi-Squared** test to the training data which tests the independence of a feature variable with the class label. If the feature variable and class label are independent, then the feature variable is not relevant in deciding the class label, and so we can discard it. The `SelectKBest` function from the `scikit-learn` library allows us to apply the chi-squared algorithm, and then select the top 'K' most dependent features for our dataset.

Figure 3.1 shows the cumulative scores for the sorted feature scores produced by the libraries chi-squared algorithm. The red line represents 99%, meaning that the features which lie above that point are adding very little information, and so we can remove them. This gives us a value of `k=40` to provide to the `SelectKBest` function. Table 3.3 lists the 40 chosen features, with a description of each. Most of the features chosen relate to either the statistics of the packet length sizes, the flags used in the flow (if TCP), the inter-arrival time between packets, or the amount of time the flow spent idle.

Feature	Description	Feature	Description
dest_port	Destination port number	fwd_PHS_flags	PSH (push) flag count (forward direction)
flow_duration	Duration of flow in microseconds	fwd_packets_s	Number of forward packets per second
bwd_packet_len_max	Maximum packet length (backward direction)	max_packet_len	Maximum packet length
bwd_packet_len_min	Minimum packet length (backward direction)	packet_len_mean	Mean packet length
bwd_packet_len_mean	Mean packet length (backward direction)	packet_len_std	Standard deviation of packet length
bwd_packet_len_std	Packet length standard deviation (backward direction)	packet_len_var	Packet length variance
flow_IAT_mean	Mean packet inter-arrival time	FIN_flag_count	FIN (finished) flag count
flow_IAT_std	Standard deviation of packet inter-arrival time	SYN_flag_count	SYN (synchronisation) flag count
flow_IAT_max	Maximum packet inter-arrival time	PSH_flag_count	PSH (push) flag count
flow_IAT_min	Minimum packet inter-arrival time	ACK_flag_count	ACK (acknowledgement) flag count
fwd_IAT_total	Total packet inter-arrival time (forward direction)	URG_flag_count	URG (urgent) flag count
fwd_IAT_mean	Mean packet inter-arrival time (forward direction)	avg_packet_size	Average size of a packet
fwd_IAT_std	Standard deviation of packet inter-arrival time (forward direction)	avg_bwd_segment_size	Average size (backward direction)
fwd_IAT_max	Maximum packet inter-arrival time (forward direction)	init_win_bytes_forward	Number of bytes sent in the initial window (forward direction)
fwd_IAT_min	Minimum packet inter-arrival time (forward direction)	init_win_bytes_backward	Number of bytes sent in the initial window (backward direction)
bwd_IAT_total	Total packet inter-arrival time (backward direction)	active_min	Minimum time a flow was active before becoming idle
bwd_IAT_mean	Mean packet inter-arrival time (backward direction)	idle_mean	Mean time a flow was idle before becoming active
bwd_IAT_std	Standard deviation of packet inter-arrival time (backward direction)	idle_std	Standard deviation of time a flow was idle before becoming active
bwd_IAT_max	Maximum packet inter-arrival time (backward direction)	idle_max	Maximum time flow idle before becoming active
bwd_IAT_min	Minimum packet inter-arrival time (backward direction)	idle_min	Minimum time flow idle before becoming active

Table 3.3: Description of Features Chosen

3.3 Classification

We have preprocessed the data and transformed it into the selected features. Now, we can fit our training data to the classifier.

3.3.1 Implemented Models

For the chosen five learning algorithms I have implemented, I provide a brief reason as to why I believed they were suitable for use in this project.

Support Vector Machine

I chose to implement an SVM since a paper referenced in my project proposals starting point[13] (see Appendix B) evaluated network intrusion detection using an SVM and achieved a detection accuracy of 88.03% on the UNSW-NB15[14] dataset. SVMs create a decision boundary to discriminate between the classes, and so will divide the hyper-plane into the different benign and attack class labels.

Decision Tree

The decision tree is a popular choice for classification problems and is why I decided it was appropriate to use here. They are easy to implement and can work with large data. However, they may suffer from overfitting. To create our decision tree, we fit the dataset, which creates decisions to be made at each node. This is performed by randomly splitting the dataset and assessing the split using the **Gini impurity**. Formally, the

Gini impurity measures the frequency of which a randomly chosen element from this split would be incorrectly labelled if it were randomly labelled according to the distribution of labels in this split.[28] Gini impurity ranges between zero, when all elements belong to the same class, and reaches the maximum value when the values are equally distributed across all of the classes. We aim to minimise the Gini impurity, such that this split perfectly separates the class.

Naive Bayes

Naive Bayes is a simple and commonly applied machine learning algorithm. I am trying to classify network traffic as normal or malicious, and so it seems appropriate to try a Bayes classifier. As mentioned in Section 2.1.3, naive Bayes makes the naive assumption that the features are independent of each other. However, this may not hold with the set of features chosen, and so might result in poor results.

K Nearest Neighbour

K-NN classifies using a majority voting on nearby points, so if we assume that similar types of attacks feature vectors are nearby to each other, it should produce a good performance. Also, no training time is needed; instead, it stores the training data in memory to be used at test time. However, this means it is computationally intensive.

K Means Clustering

Clustering allows us to divide the data into similar groupings, such that data points within the same cluster are similar. Hence, by implementing clustering, we aim to exploit the differences between benign and adverse traffic in the hope that the algorithm assigns them to different clusters. However, unlike all of the supervised methods we have previously discussed, K means clustering is unsupervised, meaning that there are no labels for it to predict. As a result, we have lost the notion of what is a 'correct' result. Instead, we can try to map the clusters to the different label groupings to see if there is a correlation.

3.3.2 Implementing Classification

For each of the algorithms mentioned, we train and evaluate them three times, each with the different label classes (original labels, grouped labels, and binary labels) and compare how each performs using metrics discussed in Section 2.1.4.

The first three models were straightforward to implement, while the last two required choosing the K parameter. To choose the value of K for the K-NN, we can evaluate the algorithm on the validation dataset across different values of K, and see which performs

Figure 3.2: KNN Precision, Recall and F1 Score for Different Values of K, of the Original Attack Labels on the Validation Dataset.

best. Figure 3.2 shows the precision, recall and F1 scores for odd K values up to 50. K is odd to avoid ties when deciding the class. After looking at the graph, we decide to set `K=7`. Although precision is highest when `K=13`, we also need a high recall, therefore we go with the highest F1 score, which is when K is set to 7.

Deciding the value of K for K means cluster was more natural since the problem predefined it; the number of clusters is the same as the number of labels in the dataset since we want the clusters to represent the different classification labels.

Principal component analysis (PCA) can be used to visualise the output of the K means cluster algorithm. PCA transforms high dimensional data into lower dimensions, allowing us to see the clusters easier.[29]

The next chapter discusses decisions made when evaluating these algorithms, and so the implementation of the final model is omitted here and instead discussed in Section 4.1.

3.4 Building the Intrusion Detection System

The pipeline given in Figure 3.3 shows an overview of how the system works. As before, we start by preprocessing the dataset. We extract our forty chosen features and their

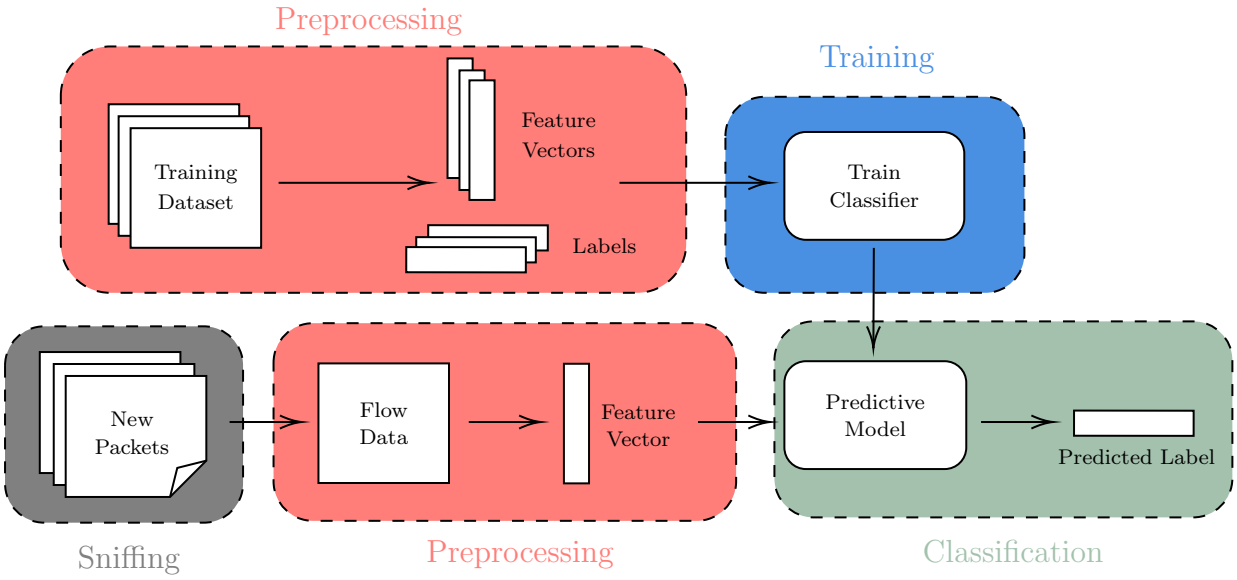

Figure 3.3: The IDS Pipeline

respective class labels and train our classifier, whose details are overlooked here and discussed in Section 4.3. This time, we use the whole dataset to train the classifier since we do not need to hold out any for testing and validation. Once training of the model is complete, the system starts analysing the incoming traffic packet by packet. This traffic may either be live, using the machines network interface or from a supplied `.pcap` file. A `.pcap` file contains packet data created during network capture and allows us to analyse previously collected data. The program then identifies the bidirectional flows (see 3.4.1). Once a flow is terminated, we preprocess the data to convert our raw data into the appropriate input for our predictive model (see 3.4.2). This input is then passed to the model, which outputs the predicted label of this flow, and alerts the user if it is not benign.

If we are analysing live data, the IDS continues to do so indefinitely, or if classifying a `.pcap` file it terminates the program once it has finished going through all the flows.

3.4.1 Identifying a Flow

A flow is identified by its flow ID, given by the following tuple:

```
(source_ip, destination_ip, source_port, destination_port, protocol)
```

The first packet in the flow is assumed to be in the forward direction, and so defines the source and destination directions, making a future packet with these values switched a packet in the backwards direction. There are only two transport layer protocols that we consider here; UDP and TCP. A UDP flow terminates on time out, given here as 600

seconds. A TCP flow, as well as time out, may also terminate via the usual connection tear down (FIN flag), or reset (RST flag).

When a new packet is received, we have to check whether it belongs to an existing flow, and if not, create a new flow instance. If the packet belongs to an existing flow, we must check if it was a terminating packet. We store all current flows in a dictionary, with the keys being the flow-ID, and the value is the reference to the flow object. Below is a code snippet, displaying how to identify a flow. Assume the existence of a packet class and a flow class, containing the relevant setters and getters for each.

```
if packet.getFwdID() in current_flows.keys():
    # Packet is a forward packet in an existing flow
    flow = current_flows[packet.getFwdID()]

    # Check for timeout
    if (packet.getTimestamp() - flow.getFlowStartTime()) > FlowTimeout:
        # Flow timed out. Terminate and classify it
        classify(flow.terminated())
        # Delete flow from current flows
        del current_flows[packet.getFwdID()]

        # Create new flow for the new packet
        flow = Flow(packet)
        current_flows[packet.getFwdID()] = flow

    # Check for FIN or RST flags
    elif packet.getFINFlag() or packet.getRSTFlag():
        # Add packet to flow
        flow.new(packet, 'fwd')
        # Terminate and classify flow
        classify(flow.terminated())
        # Delete flow from current flows
        del current_flows[packet.getFwdID()]

    else:
        flow.new(packet, 'fwd')

elif packet.getBwdID() in current_flow.keys():
    # Packet is a backward packet in an existing flow
    # Do same as for fwd, replacing with getBwdID and 'bwd' where
    appropriate ...

else:
    # Packet belongs to a new flow
    flow = Flow(packet)
    current_flows[packet.getFwdID()] = flow
```

3.4.2 Feature Extraction

Once a flow has terminated, we can then extract all the needed features. The `scapy` library provides tools to be able to retrieve quantities such as a packets payload, header length, flags and the like. That, combined with Pythons `statistics` library allowed me to calculate all of the features. `flow.terminated()` prompts the calculation of these features and returns the feature vector, which can then be passed to the classifier to receive the prediction.

3.5 Generating Network Attacks

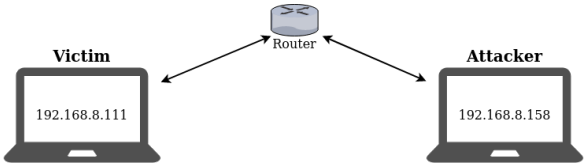

Figure 3.4: Device Set up for Attacks

To test the IDS works on real-time traffic, we need to test its ability to detect the attacks it was trained on. To do this, we need to generate network attacks, targeted at the machine running the IDS and evaluate its ability to detect these. Figure 3.4 shows the two devices, the victim and attacker, which are on the same network. The attacker is running Kali Linux, a Linux distribution that is preinstalled with penetration testing tools.[30] We use these tools to generate the desired network attacks and attack our victim machine. For each of the different attacks, I present how I generated these to test the IDS implementation.

3.5.1 Bot

We use Ares[1], which is a Python-based botnet and backdoor. It offers tools such as keylogging, remote shell, and file transfers. We host the command and control server on our Kali attacker machine and launch the agent on our compromised victim.

3.5.2 DoS and DDoS

Four different types of DoS attacks exist in this dataset. To perform GoldenEye, Hulk and Slowloris, we clone their respective git repositories[2][3][4], and run the scripts on the attacker machine, providing the victims URL as a parameter. The final DoS attack, Slowhttptest[5], is preinstalled in Kali, and as before, provided with the target URL to launch the attack.

[1] github.com/sweetsoftware/Ares [Accessed 30 Apr. 2020]
[2] github.com/jseidl/GoldenEye [Accessed 13 Apr. 2020]
[3] github.com/grafov/hulk [Accessed 13 Apr. 2020]
[4] github.com/gkbrk/slowloris [Accessed 18 Apr. 2020]
[5] tools.kali.org/stress-testing/slowhttptest [Accessed 12 Apr. 2020]

A Distributed DoS (DDoS) attack requires multiple devices (bots) to be able to overwhelm the target machine. As I would not personally have permission to use enough devices to create the amount of traffic required for a DDoS attack, I decided to use available PCAP files for DDoS attacks found online. I chose to use data from the IDS2012[31] dataset, specifically the data captured on Tuesday 15th June, which contains DDoS attacks created in a similar way to the original dataset.

3.5.3 FTP and SSH Patator

FTP and SHH Patator are both brute force attacks on a server, in which we make many repeated guesses at a password to gain unauthorised access to the system. Here, we use the Patator package[6] to perform both attacks. To perform brute force, the program is supplied with a dictionary of passwords to try; we use the rockyou password dictionary (preinstalled on Kali) which contains over 14 million unique passwords. We run both FTP and SSH servers on the victim machine, and then launch the attack from the attackers' machine.

3.5.4 Probe

A probe attack aims to discover weaknesses or vulnerabilities in a system.[32] The probe attack we perform uses Nmap[7](Network Mapper) to perform a port scan on the victim machine.

3.5.5 Web Attack

To perform the various web attacks, we host the Damn Vulnerable Web App (DVWA)[8] on our victim machine. DVWA is a PHP/MySQL based web application that is susceptible to many common web attacks. To perform brute force, we first use Burp Suite[9] offered by Kali to intercept the HTTP requests so that we can see the form they take, such as the login parameters and the cookies. Once we have this data, we use THC-Hydra[10] to brute force the login.

We use XSSer[11] to perform the cross-site scripting attacks. XSSer is a framework which can detect and exploit web vulnerabilities. We pass it the URL for the DVWA on our victim machine, along with the required cookies, and the program automates the testing for XSS.

[6] tools.kali.org/password-attacks/patator [Accessed 3 Apr. 2020]
[7] nmap.org [Accessed 3 Apr. 2020]
[8] dvwa.co.uk [Accessed 12 Apr. 2020]
[9] tools.kali.org/web-applications/burpsuite [Accessed 11 Apr. 2020]
[10] tools.kali.org/password-attacks/hydra [Accessed 4 May 2020]
[11] xsser.03c8.net [Accessed 29 Apr. 2020]

3.6 Repository Overview

The first phase of the project (machine learning phase) was conducted solely in a Jupyter Notebook and would read in the `.csv` dataset files stored in the `MachineLearningCVE` directory.

Building the IDS required more substantial software development. Figure 3.5 provides an overview of the structure of the repository.

The `main.py` method allows the user to decide whether they would like to sniff live traffic or parse a `.pcap` file. We train the classifier in `train.py`, and analyse new packets in the `sniff.py` method.

The class `PacketInfo.py` describes a packet and contains setters and getters for the information required from a packet, such as its flags or header length. The `Flow.py` class describes the bidirectional flow, whilst `FlowFeature.py` contains the setters and getters for the feature vector variables. The (feature vector, label) pairs are written to the `output_logs.csv` so that they are available to be reviewed at a later time.

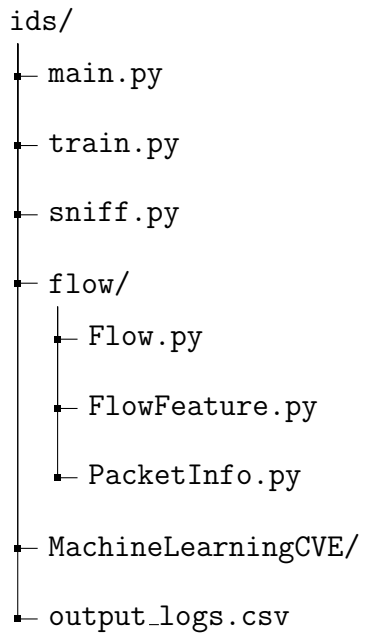

Figure 3.5: Structure of Source Code Repository

3.7 Summary

In this chapter, I have discussed the various steps involved in the machine learning pipeline; preprocessing, feature selection, training and prediction. I have also laid out how I built the intrusion detection system and how to generate the attacks required to test it. In the next chapter, I provide details about choosing and implementing the final model, as well as the evaluation.

Chapter 4

Evaluation

In this chapter, I discuss the results from implementing five chosen machine learning algorithms on the dataset, how I used these to design the final model, and the success of the model integrated into the intrusion detection system.

4.1 Evaluation of Different Learning Algorithms

Using the evaluation metrics discussed in Section 2.1.4, I compare the different learning approaches, allowing me to design the final model. Appendix A offers full results from my experiments.

4.1.1 Initial Testing Results

We perform initial testing on the proportion of the dataset held out for validation. We can split the evaluation of the algorithms into the supervised approaches and the unsupervised approach since they have different evaluation approaches.

Supervised Algorithms

For the chosen supervised algorithms, we can compare their metrics directly.

Figure 4.1 shows the precision, recall and accuracy that the supervised algorithms achieved. This graph reveals why accuracy itself does not provide enough information about the performance of an algorithm; naive Bayes on the original labels achieved an accuracy of 86.45%, yet had a recall of 20.76%. This recall value would mean that we are only finding the true positive attacks a fifth of the time, which indicates that naive Bayes is not suitable for use in an IDS.

The graph shows us that precision and recall rise as the attack groupings get less specific, giving the binary case the highest performance. However, we want to be able to provide

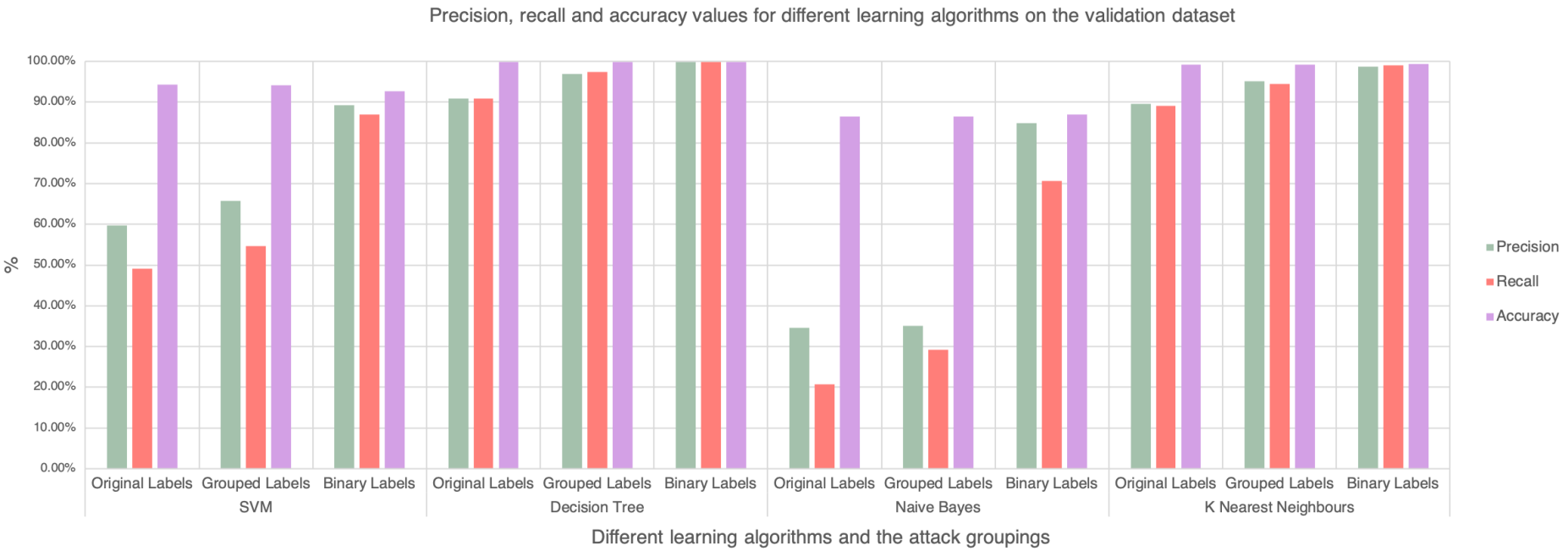

Figure 4.1: Precision, Recall and Accuracy Scores for the Three Different Label Groupings, Across all the Different Supervised Algorithms, Tested on the Validation Dataset.

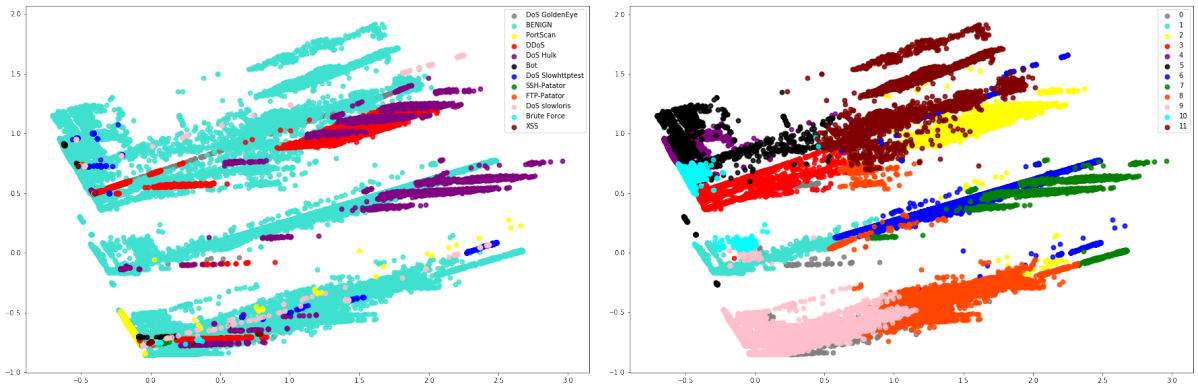

Figure 4.2: PCA of Actual Labels (left) and Predicted Clusters (right) on the Original Labels of the Validation Dataset

more information than just 'attack'; hence there is a trade-off between the specific class provided and the result.

We can see that naive Bayes has the worst performance rates of any algorithm across all the three different groupings. The reason that naive Bayes likely performed so bad in comparison due to the other algorithms is due to the naive assumption we made; we assumed that all the features are independent, which is not necessarily valid in this case. For example, `Packet_Length_Variance` and `Packet_Length_Std` are features that are not independent. Hence the assumption fails, and we achieve poor results.

SVM was chosen for consideration due to its use in a paper[13] found in my initial research which achieved a detection accuracy of 88.03% on the UNSW-NB15[14] dataset. Although I used a different dataset, I achieved a detection accuracy of 94.28% on the 12 different labels, hence outperforming their classifier.

From the graph, it is clear that the decision tree and K-NN classifiers are suitable to use within the IDS since they achieve high precision and recall, and so we investigate them further in Section 4.1.2.

Unsupervised Algorithms

Unsupervised methods lack the well-defined evaluation metrics that supervised methods have due to the lack of correct labels provided to assess the results. However, as we are working with a labelled dataset, we can see if the clusters obtained from the K means clustering algorithm match the attack labels. Also, we can use principal component analysis (PCA) to compare the output of the clustering algorithm visually.

Figure 4.2 shows the result of PCA on both the actual labels and the predicted clusters on the validation data. It is clear from these graphs that our prediction lacks the correct structure that the actual labels hold, indicating poor performance.

	0	1	2	3	4	5	6
Benign	**200,622**	153	84,357	101,962	34,051	24,325	8,795
Botnet	0	0	**236**	154	1	0	0
Brute Force	0	0	**1,366**	606	0	795	0
DDoS	0	4,218	**11,675**	9,712	0	0	0
DoS	1,656	**18,399**	5,309	13,839	407	358	10,374
Probe	1	0	**31,727**	12	9	0	12
Web Attack	0	0	**396**	31	4	0	0

Table 4.1: Predicted Clustering Result (K=7) Against True Grouped Labels

In the supervised models, we saw the trend that the grouped labels usually performed better than the original labels, with the binary labels performing best. However, the grouped labels also lack significant structure when compared with the correct labelling. Table 4.1 shows the result of the clustering algorithm against the actual (grouped) labels for the data. The clustering is still not significant. In bold, for each attack, is the cluster value which contains the maximum number of records of that attack. The results reveal that cluster 0 is likely to be benign data and cluster 1 DoS attacks. The remaining attacks all seem to belong to cluster 2, alongside a fifth of the benign data. Overall, there is no clear result from the original labelled data, nor the grouped labels.

Lastly, the binary classification shows no signs of improvement either. The results are given in Table 4.2. We conclude here that K means clustering is not suitable for use as it has not produced any promising results.

	0	1
Benign	38,857	**415,408**
Attack	33,468	**77,829**

Table 4.2: Predicted Clustering (K=2) Result Against True Binary Labels

4.1.2 Model Refinement

After initial analysis of the five different implemented models, we now begin to look more in depth at the well performing ones, allowing us to finalise details of our model.

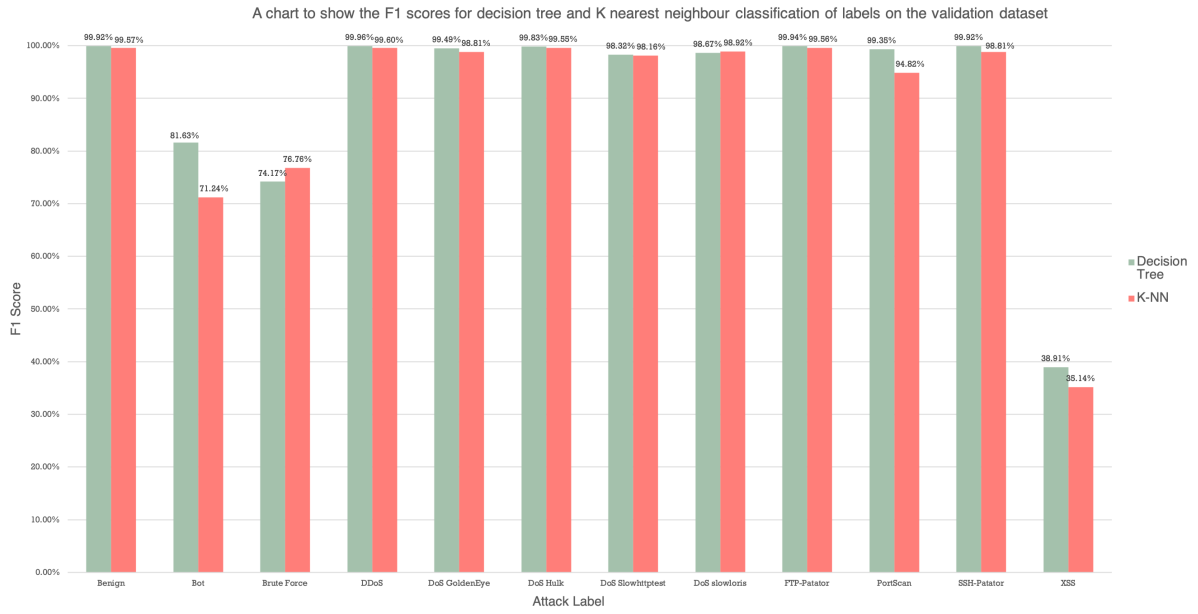

Figure 4.3: F1 Scores of Decision Tree and K Nearest Neighbour

Decision Tree vs K Nearest Neighbours

Due to the success of decision trees and K-NN in the previous subsections, we now compare the two algorithms further. Firstly, we can look at how they perform in classifying the attacks individually. Figure 4.3 shows the F1 scores for both decision tree and K-NN on each of the different labels. Decision tree offers a higher F1 score than K-NN in all the attack labels other than brute force and DoS slowloris, meaning that decision tree is the best performing algorithm out of all the five considered so far.

It is apparent from Figure 4.3 that some types of attacks may be easier to detect than others. Cross-site scripting (XSS) performs very poorly for both the algorithms, while bot and brute force also perform slightly worse than the rest of the labels.

Identifying XSS may not perform well under the chosen features since we are looking at flow-based features, which exist in the transport and network layer of the network stack, while features that identify XSS may be in the application layer. Using machine learning to identify XSS has been done before[33], using naive Bayes, decision tree and SVM to achieve a precision of 98%, 99% and 99% respectively. They extracted URL based features, such as the presence of special characters and request for cookies, as well as JavaScript-based features like the number of script functions and the number of references to a JavaScript file for example. Hence, it could be possible to achieve better results for XSS if we instead had different features.

As well as the evaluation metrics, we also need to compare the time taken for training and classification. The IDS classifies in real-time hence the classification time must be fast. On the other hand, training time is less important since we can train the system before it goes live, although the user sees this as wasted time. Table 4.3 shows the training and

	Decision Tree			K-NN		
	Original	Grouped	Binary	Original	Grouped	Binary
Training Time (s)	105.93	100.01	92.33	2,306.96	2,507.44	2,457.47
Classification Time (s)	0.73	0.23	0.16	1,963.92	2,064.91	2,008.67

Table 4.3: Training and Classification Times for Decision Tree and K Nearest Neighbours

classification time for both decision tree and K-NN. We see that decision tree takes far less time in both cases. The large difference in times arises because the K-NN algorithm has a query complexity of $O(dn^2)$[34] while the decision tree is $O(\log n)$[35], for n samples in d dimensions, and since n is large (1.7 million) the quadratic run time of K-NN is not feasible.

Decision Tree vs Random Forest

From the five different algorithms we have tested, it is apparent that the most suitable for use on the chosen dataset and the selected attacks is the decision tree. Due to its success, we suggest considering a **random forest classifier**. The random forest is a collection of decision trees, building each tree by randomly sampling the features and the training data, and using majority voting amongst the trees to assign the class label.[36] As a result, they tend to outperform decision trees since they overcome the overfitting problem that decision trees may suffer from and are less sensitive to outlier data.

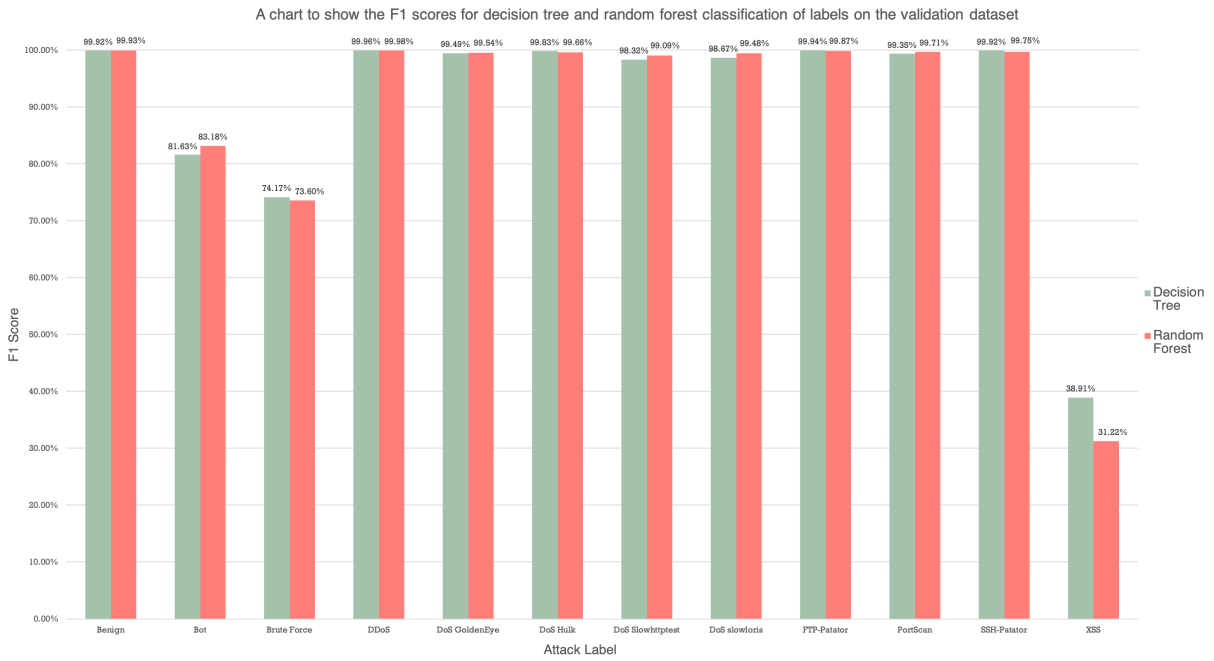

Figure 4.4: F1 Scores of Decision Tree and Random Forest

Figure 4.4 shows the F1 scores that decision tree and random forest achieved in classifying

	Decision Tree			Random Forest		
	Original	Grouped	Binary	Original	Grouped	Binary
Training Time (s)	105.93	100.01	92.33	77.36	78.26	70.19
Classification Time (s)	0.73	0.23	0.16	0.99	0.90	0.65

Table 4.4: Training and Classification Times for Decision Tree and K Nearest Neighbours

the original labels on the validation dataset. We can see that they achieve similar results, which we would expect given that the random forest is a collection of decision trees. The random forest has a higher F1 score than the decision tree in 7 out of the 12 labels for the graph presented. It also had higher F1 scores in 5 out of the 7 grouped labels, and in both of the binary labels.

As before, we compare the training and classification times for both algorithms, given in Table 4.4. Unlike the K-NN case, they are now in the same order of magnitude. We see that the random forest trained slightly faster, while the decision tree had a faster classification. Nonetheless, these times are not different enough to significantly advantage either model.

Overall, the random forest was the most appropriate model to use going forward due to the results it achieved on the validation data. Unfortunately, none of the attacks performed better at classifying the labels where it offered lower results (bot, brute force, XSS). If that was the case, we could have combined the models with having a hybrid model and implementing a voting rule to classify the traffic.

Final Grouping of Labels

We need to consider what labels we want our classifier to predict. Throughout the evaluation in the last sections, we considered three different options; the original 12 from the dataset, our custom 7 which grouped the 12 labels by type of attack, or the binary label case. We saw that in general, the binary case achieved the highest F1 scores and accuracy. However, I rule this out as a potential option due to wanting to make the system provide a useful output. We want to be able to tell the user what type of attack occurred, rather than just 'attack'. As a reminder to the reader, Table 4.5 shows the grouping proposed in Section 3.1.

Despite the low F1 scores brute force and XSS achieved in the last subsection for the random forest, when they are combined into the web attack label, it achieves an F1 score of 98.83%, compared to the previous 73.60% and 31.22% respectively. As we can see in Figure 4.5, the reason this occurs is we class 89 of the 130 XSS attacks as brute force. Similarly, 69 of the 301 brute force attacks are incorrectly classed as XSS. Hence, when grouped under one label, these errors go away, and we correctly classify 424 of the total 431

Botnet	Bot
Brute Force	FTP-Patator, SSH-Patator
DDoS	DDoS
DoS	DoS GoldenEye, DoS Hulk, DoS Slowhttptest, DoS Slowloris
Probe	Port Scan
Web Attack	Web Attack: Brute Force, Web Attack: XSS

Table 4.5: Grouping of Original Attacks into more General Categories

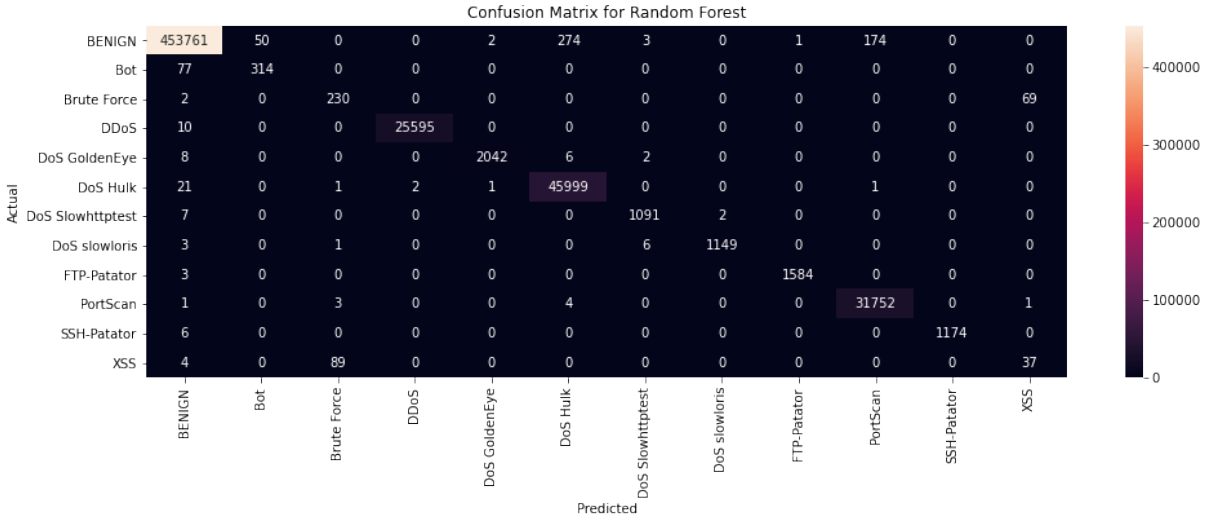

Figure 4.5: Confusion Matrix for Random Forest on Validation Dataset

web attack labels. This further hints that we have not selected features specific enough to identify web attacks; we have enough to know that a web type attack occurred but do not have enough information to differentiate on type of attack.

Although the four different types of DoS attacks already performed well, having scores above 99% for precision and recall, we want to group these into one category. Since there exist many different types of DoS attack, it makes sense to generalise to that case, rather than fit new ones into one of the four we have trained on.

I initially proposed grouping FTP-Patator and SSH-Patator into one label since they are both types of brute force. However, they both performed exceptionally well on the random forest classifier when considered individually, achieving F1 scores of 99.87% and 99.75% respectively. When looking at the confusion matrix, the only miss-classification either attack had was to a benign label, so we do not group these in the final grouping, and instead, consider them as two separate cases.

There are no other relevant or credible groupings to consider, and so we have decided upon the final grouping of labels, given in Table 4.6.

Benign	Benign
Botnet	Bot
DDoS	DDoS
DoS	DoS GoldenEye, DoS Hulk, DoS Slowhttptest, DoS Slowloris
FTP-Patator	FTP-Patator
Probe	Port Scan
SSH-Patator	SSH-Patator
Web Attack	Web Attack: Brute Force, Web Attack: XSS

Table 4.6: Final Labels and their Groupings from the Original Attacks

Optimising Parameters

Machine learning models also take a collection of parameters as input, these values are known as **hyperparameters** and control the behaviour of the model (such as the depth of the tree in decision trees and random forests). We can tune these hyperparameters to find the values which yield the optimal results for our problem.[37]

For our model, a random forest, listed are the parameters which we will tune, and a description of them according to the `scikit-learn` library:[38]

- `n_estimators`: The number of trees in the forest
- `max_depth`: The maximum depth of the tree
- `min_samples_split`: The minimum number of samples required to split a decision node
- `min_samples_leaf`: The minimum number of samples required to be a leaf node
- `max_features`: The number of features to consider when looking for the best split
- `bootstrap`: Whether bootstrap samples are used when building trees. If false, the whole dataset is used to build each tree

First, we consider the number of estimators parameter. This controls the number of decision trees that we evaluate in our ensemble. Figure 4.6 displays the F1 scores for different values of the `n_estimators` parameter for the random forest on the validation dataset. The highest peak is when `n_estimators=800`. However, Figure 4.7 shows why we do not choose this as the value for the number of estimators. Time taken for classification increases linearly with the number of estimators, so setting `n_estimators=800` would incur a classification time of around 50 seconds. This is not feasible in our solution;

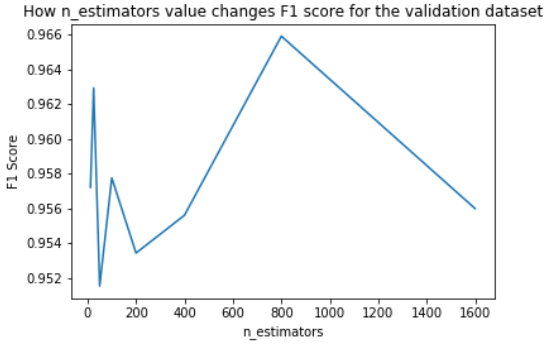

Figure 4.6: The Effect on F1 Score from Changing the Number of Estimators Parameter

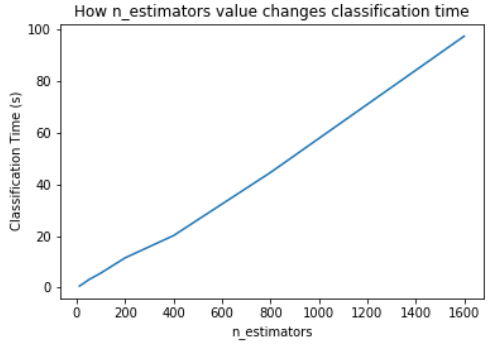

Figure 4.7: The Effect on Classification Time from Changing the Number of Estimators Parameter

we need classification to be fast since we are classifying in real-time, and so despite the potential of a better performing model, we instead set `n_estimators=25` which is the second maximum. Setting it to 25 will allow us to classify 32 times faster at a sacrifice of a 0.3% decrease in the F1 score, which is an appropriate trade off to make in this case.

To optimise the remaining hyperparameters, we perform a randomised search. This search strategy creates a grid of different values for each hyperparameter. It then performs training and testing on random combinations of these hyperparameter values, providing each combination with a score. We use `RandomizedSearchCV` from the `scikit-learn` library to implement parameter optimisation, and then call `best_params_` to return the parameter setting that gave the best results on the hold out data. This provides us with our final parameter results:

```
{   'n_estimators' : 25, 'min_samples_split': 5, 'min_samples_leaf' : 1,
    'max_features' : 20, 'max_depth' : 200, 'bootstrap' : True
}
```

4.2 Final Testing Results

We performed the final testing on the proportion of the dataset held out for testing. Using a random forest classifier, we achieved a final average F1 score of 97.59% and an accuracy of 99.91%. The confusion matrix in Figure 4.8 provides us with visual results of our performance.

Botnet is the lowest-performing attack, with some of its labels being classified as benign. We speculate the reasoning for this misclassification is due to the different operational phases of a botnet. Botnets communicate with a command and control server, which can launch a variety of commands. Hence it is likely that our algorithm has learnt to detect botnet when malicious network traffic is produced due to this botnet operation (such as

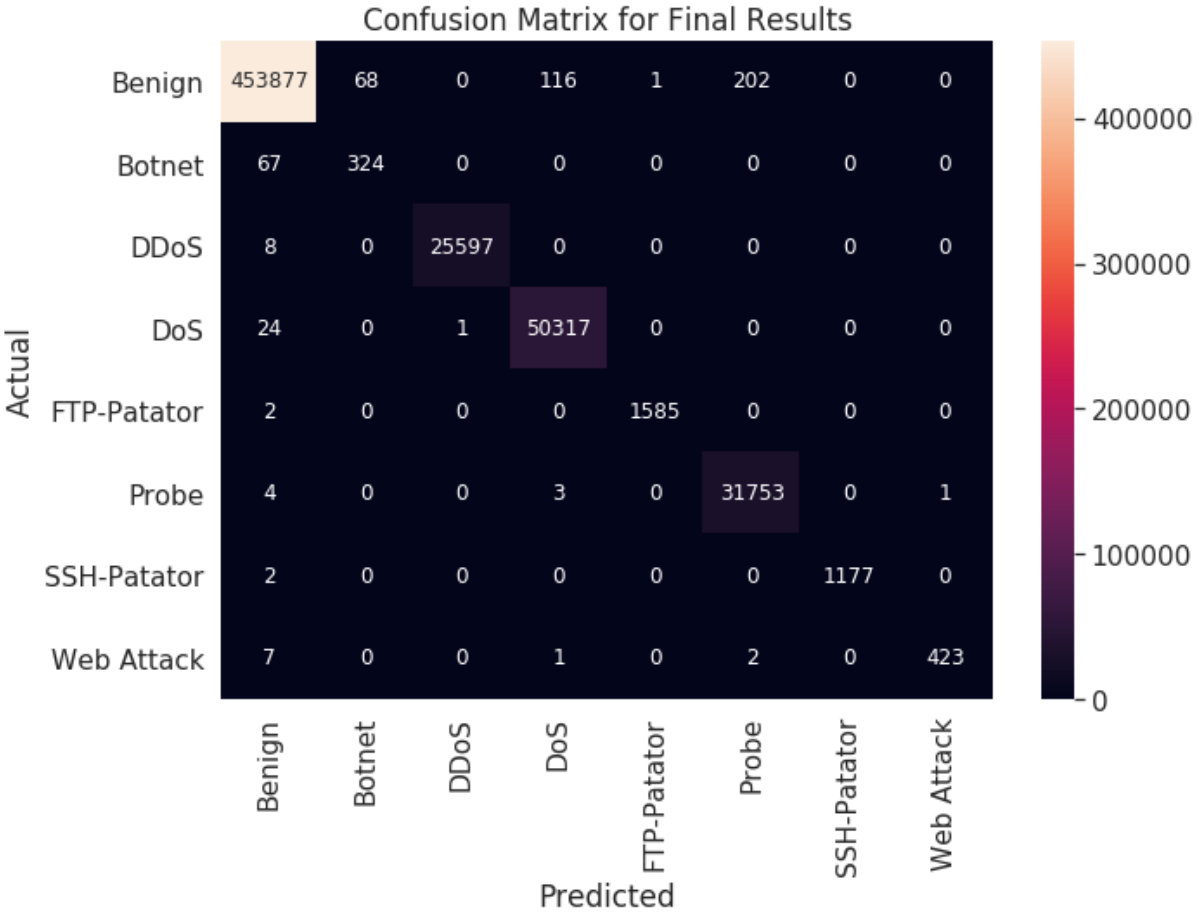

Figure 4.8: Confusion Matrix of Random Forest on Testing Dataset

being used in a DDoS attack). However, bots may also sit idle, or the adversary can limit the intensity of the attacks to disguise as 'normal' traffic, and this is probably where our model is classifying botnet traffic as benign.[39]

Table 4.7 compares the results of the testing with the percentage of the dataset that label occupied. We see the two lowest scoring labels, botnet and web attack, each account for less than 0.1% of the dataset. The lack of data for these two attack labels may be why they are the only attacks to have an F1 score less than 99%.

The benign case had a precision of 99.97%. The precision measures the false positives, which in the case of the IDS would be benign traffic classified as an attack. The high precision for the benign case shows that the system is usable and will not classify all the users normal data as attacks. Similarly, the benign case achieved a recall of 99.91%. The recall measured the false negatives, the case where we classify attacks as benign. This is the situation we wanted to avoid since it means we have not identified a network attack and is a big problem. However, the 99.91% recall leaves us confident in classifying traffic appropriately.

In Section 1.2, we discussed the results of a survey[4] that compared various machine learning approaches to IDS. The range of accuracy achieved was between 88.0% and

Label	Precision (%)	Recall (%)	F1 Score (%)	% of Training Data
Benign	99.97	99.91	99.94	80.32
Botnet	82.65	82.86	82.76	0.07
DDos	100.00	99.97	99.98	4.53
DoS	99.76	99.95	99.86	8.90
FTP-Patator	99.94	99.87	99.90	0.28
Probe	99.36	99.97	99.67	5.62
SSH-Patator	100.00	99.83	99.91	0.21
Web Attack	99.76	97.69	98.71	0.08
Average	97.68	97.51	97.59	12.50

Table 4.7: Precision, Recall and F1 Score Achieved for the Different Labels Compared to the Percentage of the Dataset that was that Label

99.9%, with the median accuracy being 97.2%. We exceeded this median value, achieving an accuracy of 99.9%, matching that of the highest accuracy in the survey. Unfortunately, the survey does not include precision, recall or F1 metrics so we cannot compare our values. The paper in the review that achieved the highest accuracy only considered attacks in the benign case. In contrast, we have achieved the same accuracy and provided more specificity about the attack.

Overall, we can be confident that our system can detect the types of attacks it was trained on. An average precision of 97.68% and recall of 97.51% ensures that our system can differentiate between benign and adverse traffic with minimal errors. Section 4.4.2 evaluates the IDS ability to detect unseen attacks, which we use to judge the success of the systems ability to detect zero-day exploits.

4.3 Building the Intrusion Detection System

We now implement the machine learning model into our IDS. As discussed in Section 3.4, we preprocess the dataset, extract our features, and train the classifier on the whole dataset. We use the random forest classifier provided in the `scikit-learn` library, which then classifies network traffic into one of the eight different labels, as given in Table 4.6. If the algorithm predicts a flow to be anything other than benign, the interface will output the predicted attack label, as well as write all feature vectors and predicted classifications to an output log file.

4.4 Evaluation of the Intrusion Detection System

To test the intrusion detection system, we cannot use CSV files as before since we need network data to be able to analyse the flows. Instead, we generate network attacks onto a victim machine that is running the IDS, and assess its success based on its ability to detect non-benign flows, and whether or not it assigns the correct attack label.

4.4.1 Trained Network Attacks

Firstly, we assess the IDS on the attacks that it has been trained to detect, using techniques discussed in Section 3.5 to generate these attacks.

Table 4.8 shows the success of the system to predict the attacks correctly. Unlike before, when we were testing the dataset, we do not have a quantitative way to analyse the success of the detection. Therefore, we provide a tick if there was at least one occurrence of that label, alongside the benign background traffic. The star indicates that it also predicted another label alongside the attack. Both web attack brute force and XSS had occurrences of DoS labels in their classification (10 Dos, 157 Web Attack and 3 DoS, 19 Web Attack labels respectively.)

Attack	Predicted
Botnet	✓
DDoS IDS2012 PCAP[31]	✓
DoS: Golden Eye	✓
DoS: Hulk	✓
DoS: Slowhttptest	✓
DoS: Slowloris	✓
FTP-Patator	✓
Probe	✓
SSH-Patator	✓
Web Attack: Brute Force	✓*
Web Attack: XSS	✓*

Table 4.8: Success of Detecting Seen Attacks

These results are promising and show that the model can sufficiently detect the attacks it has been trained on. While there were errors in detecting web attacks, these errors are rather small, and we have discussed previously the reason why web attacks perform worse. Firstly, due to only occupying 0.08% of the dataset, and secondly, the features chosen may not relate well to identifying web attacks.

We note the success of correctly detecting the instance of the Ares botnet running on our victim machine, despite the low recall scores botnet achieved in the last subsection. As expressed in the last subsection, its poor recall rate was likely due to times the botnet was not performing malicious attacks (or was doing so at moderate volume). Here, our bot is operational, launching commands given from the command and control server, explaining our improved ability to detect it.

4.4.2 Unseen Network Attacks

We discussed in Section 1.3 that one of the challenges in building a machine learning IDS to detect new attacks was that it is hard to test how well it can truly work in new cases. Here, we apply other common attacks which the model has not been trained on, to see if it is also able to detect them. Success in detecting these hints towards the ability to defend against zero-day exploits. Using a mix of available PCAP files, and generating our own attacks, we test our IDS. Table 4.9 shows the results of detecting different unseen attacks.

The Botnet 2014 dataset[40] is from the same lab as the original dataset we used to train; it contains 16 different botnet attacks compared to the one instance of botnet we trained on. The second botnet PCAP[41] contains an instance of botnet Neris and is obtained from a part of a malware capture project.

Attack	Predicted
Botnet 2014 Dataset PCAP[40]	✓*
Botnet Neris PCAP[41]	✓
DoS Apache killer	✓
DoS R.U.D.Y	✓
DoS Slow Read	✓
DoS SYN FLOOD	✓
MySQL Brute Force	✓
SMB Brute Force	✗
Telnet Brute Force	✓
Web Directory Traversal	✓
Web SQL Injection	✗

Table 4.9: Success of Detecting Unseen Attacks

The classification of the first botnet PCAP, unfortunately, contained a handful of DoS labels. The second botnet PCAP only contained botnet instances alongside the benign background traffic. Overall, there were many instances of botnet detected so we can be confident in our ability to detect botnet, albeit a few DoS misclassifications, especially considering it offered the lowest F1 scores seen in our testing in Section 4.2.

Our IDS detected all of the DoS attacks that were generated. This shows that the performance of the original model on the DoS types attacks it was trained on has generalised well to never seen before ones. DoS attacks did make up a very high proportion of the dataset, so the exposure to the vast amount of records has translated well to the new attacks.

MySQL, SMB (Server Message Block) and Telnet brute force are all performed using the patator package as we did for SSH and FTP-Patator. MySQL and Telnet brute force were classified as SSH-Patator, while SMB was not recognised as an attack. We hypothesise the reason why SMB was the only protocol tested to fail at detecting brute force is due to the use case of SMB. SMB is used to connect Windows machines within the same LAN,

as well as aid the discovery of printers on the network. These are cases our dataset has not dealt with. Ultimately, our IDS aimed to protect against outside threats (i.e. not on our LAN), and so we do not worry about the failure to detect SMB brute force.

Here we realise that we should have combined FTP-Patator and SSH-Patator into one attack. We previously focused on generating the highest results and offering specificity about the type of attack when choosing the groupings of attacks in Section 4.1.2, but this has removed some of the essential generality that is required here. Brute forcing Telnet and MySQL was not an instance of SSH-Patator, but it is the closest attack type in our available labels. Hence, if instead, we grouped FTP and SSH into a general brute force category, that would apply better to these types of attacks.

Web directory traversal attack aims to gain unauthorised access to files outside of the web root folder by using a series of `../` on file references. We use the built-in DotDot-Pwn package[1] in Kali to attack the DVWA running on the victim machine. This was successfully classified as a web attack by the IDS. To perform SQL injection against the DVWA, we use Kali's Sqlmap[2] tool, unfortunately, the IDS only outputted benign labels during this attack. Yet again, this is likely due to the features chosen not being relevant to application-layer attacks.

We conclude that the IDS is sufficient in detecting attacks that it has been exclusively trained on, and offers some generalisation to unseen attacks. We note, in particular its success on DoS type attacks. Furthermore, despite the original low recall score for botnet, our system can detect new types of botnet in the PCAPs supplied. More rigorous testing and a wider selection of attacks would allow further analysis into the proposed IDS success. However, as previously mentioned, complete coverage of attacks is impossible given that we can never know the form of the zero-day exploit we are trying to protect against.

4.5 Summary

This chapter provided the evaluation of different machine learning algorithms as we worked towards developing our final model, the random forest classifier, which we use in our IDS. We then evaluated the learning-based IDS on its ability to recognise a selection of generated network attacks.

[1] tools.kali.org/information-gathering/dotdotpwn [Accessed 30 Apr. 2020]
[2] tools.kali.org/vulnerability-analysis/sqlmap [Accessed 30 Apr. 2020]

Chapter 5

Conclusion

In this chapter, I provide a summary of the work conducted, a reflection of the project, and potential areas for future work.

5.1 Success Criteria

In Section 2.2 I said that this project would be a success if I had

- Implemented at least one machine learning algorithm and evaluated its precision on a proportion of the dataset reserved for testing.
- Successfully developed an interface for the intrusion detection system, which can sniff and monitor traffic and alert on a suspected attack.
- Evaluated the IDS detection accuracy when tested on simulated network traffic.

I have achieved all of these points, and thus the project was a success. Section's 3.3, 4.1, and 4.2 saw me implement and evaluate different machine learning models as well as test the final model. Section 3.4 discussed the development of the IDS and how it monitors the traffic to detect flows; we then integrated the learning model into it in Section 4.3. Lastly, in Section 4.4. we evaluated the IDS on simulated network traffic, on both seen and unseen attacks.

5.2 Lessons Learned

In carrying out this project, I furthered my understanding of machine learning algorithms and the steps required to implement them. The project allowed me to explore different types of network attacks and learn how to perform them. I acquired knowledge of Python

and the libraries used (Numpy, Pandas, Scikit-Learn and Scapy). Ultimately, I gained invaluable skills in software development, independent research and time management.

If I were to do the project again, the main change I would implement is in choosing the features. I would perform feature selection, taking into account each type of attack individually. DoS attacks account for almost half of the attack data, compared to the 0.35% of the attack data which botnet occupies for example. Hence, when scoring features, those who identify DoS attacks outweigh those who identify botnet. Instead, we could select the top five features for each different attack, and combine these to form the feature vector. We would likely achieve considerable performance gains on the attacks which occupy a smaller proportion of the dataset, without sacrificing much of the performance for the higher frequency attacks.

5.3 Future Work

Any future work should focus on improving the performance of the IDS. I list possible extensions for the project which may increase its success.

Deep Learning: Deep learning is a type of machine learning which makes use of neural networks. The machine trains itself to be able to perform a task, in this case, identify attacks. The Machine Learning for IDS survey discussed in Section 1.2 included several approaches which used neural networks, achieving promising results.

Feature Selection: We suggested that the reason detecting attacks such as XSS did not achieve as high performance as other attacks were due to the features selected. Hence, further work could also incorporate application-level data to generate more relevant features in the hope to improve the performance of botnet and web attacks.

Redundancy: We could also implement a signature-based method, which we check before passing to the classifier, such that any attacks which match a known pattern are correctly classified. While this would not improve the systems ability to detect unseen attacks, it would improve the overall performance.

5.4 Final Remarks

This dissertation has explored the use of machine learning in an intrusion detection system. The results achieved points towards correctness, but full test coverage is impossible by definition of the problem. Overall, we can be content in our proposed model, a random forest classifier which achieved 97.59% F1 Score and 99.91% accuracy. These results are comparable to the higher end of recorded values in previous related papers. We have also shown the systems ability to detect previously unseen attacks, and discussed reasons for those it was unable to detect, and hinted towards future work to address this.

Bibliography

[1] A. Borkar, A. Donode and A. Kumari, "A survey on Intrusion Detection System (IDS) and Internal Intrusion Detection and protection system (IIDPS)," 2017 International Conference on Inventive Computing and Informatics (ICICI), Coimbatore, 2017, pp. 949-953.

[2] Das, Sumit, Aritra Dey, Akash Pal and Nabamita Roy. "Applications of Artificial Intelligence in Machine Learning: Review and Prospect." 2015 International Journal of Computer Applications 115, 2015, pp. 31-41.

[3] S. Suthaharan, "An Iterative Ellipsoid-based Anomaly Detection Technique for Intrusion Detection Systems," 2012 Proceedings of IEEE Southeastcon, Orlando, FL, 2012, pp. 1-6.

[4] Kunal and M. Dua, "Machine Learning Approach to IDS: A Comprehensive Review," 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 2019, pp. 117-121.

[5] Weijun li, Zhenyu Liu, "A Method of SVM with Normalization in Intrusion Detection", Procedia Environmental Sciences, Volume 11, Part A, 2011, pp. 256-262, ISSN 1878-0296.

[6] KDD Cup 1999 Dataset: kdd.ics.uci.edu/databases/kddcup99/kddcup99 [Accessed 24 Feb. 2020]

[7] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization", 4th International Conference on Information Systems Security and Privacy (ICISSP), Portugal, January 2018.

[8] Watt, Jeremy, Reza Borhani, and Aggelos K. Katsaggelos. "Introduction to Machine Learning." Chapter. In Machine Learning Refined: Foundations, Algorithms, and Applications, 2nd ed., pp 1–18. Cambridge: Cambridge University Press, 2020. doi:10.1017/9781108690935.003.

[9] Girish Chandrashekar, Ferat Sahin, "A Survey on Feature Selection Methods," Computers Electrical Engineering, Volume 40, Issue 1, 2014, pp 16-28, ISSN 0045-7906.

[10] Flach, Peter, "Machine Learning: The Art and Science of Algorithms That Make Sense of Data", Cambridge: Cambridge University Press, 2012. doi:10.1017/CBO9780511973000.

[11] Clarence Chio, David Freeman, "Machine Learning and Security: Protecting Systems with Data and Algorithms", 1st ed., O'Reilly, 2018.

[12] Shalev-Shwartz, Shai, and Shai Ben-David, "Understanding Machine Learning: From Theory to Algorithms", Cambridge: Cambridge University Press, 2014. doi:10.1017/CBO9781107298019.

[13] Md Nasimuzzaman Chowdhury, Ken Ferens and Mike Ferens, "Network Intrusion Detection Using Machine Learning", International Conference on Security and Management, 2016.

[14] Moustafa, Nour, and Jill Slay. "UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 network data set)."Military Communications and Information Systems Conference (MilCIS), 2015. IEEE, 2015.

[15] scikit-learn.org/stable/modules/naive_bayes [Accessed 27 Feb. 2020]

[16] Zaki, Mohammed J., and Wagner Meira, Jr. "Probabilistic Classification." Chapter. In Data Mining and Machine Learning: Fundamental Concepts and Algorithms, 2nd ed., pp 469–82. Cambridge: Cambridge University Press, 2020. doi:10.1017/9781108564175.023.

[17] Zaki, Mohammed J., and Wagner Meira, Jr. "Representative-Based Clustering." Chapter. In Data Mining and Machine Learning: Fundamental Concepts and Algorithms, 2nd ed., pp 334–63. Cambridge: Cambridge University Press, 2020. doi:10.1017/9781108564175.017.

[18] Watt, Jeremy, Reza Borhani, and Aggelos K. Katsaggelos, "Linear Two-Class Classification." Chapter. In Machine Learning Refined: Foundations, Algorithms, and Applications, 2nd ed., pp 125–73. Cambridge: Cambridge University Press, 2020. doi:10.1017/9781108690935.010.

[19] Panwar, Shivendra S., Shiwen Mao, Jeong-dong Ryoo, and Yihan Li. "TCP/IP Overview." Chapter. In TCP/IP Essentials: A Lab-Based Approach, pp 1–25. Cambridge: Cambridge University Press, 2004. doi:10.1017/CBO9781139167246.003.

[20] Panwar, Shivendra S., Shiwen Mao, Jeong-dong Ryoo, and Yihan Li. "TCP Study." Chapter. In TCP/IP Essentials: A Lab-Based Approach, pp 111–33. Cambridge: Cambridge University Press, 2004. doi:10.1017/CBO9781139167246.009.

[21] https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2016.pdf p 50 [Accessed 19 Apr. 2020]

[22] S. Chen, Y. Chen and W. Tzeng, "Effective Botnet Detection Through Neural Networks on Convolutional Features," 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), New York, NY, 2018, pp. 372-378.

[23] M. A. Saleh and A. Abdul Manaf, "Optimal specifications for a protective framework against HTTP-based DoS and DDoS attacks," 2014 International Symposium on Biometrics and Security Technologies (ISBAST), Kuala Lumpur, 2014, pp. 263-267.

[24] P. Sinha, V. K. Jha, A. K. Rai and B. Bhushan, "Security Vulnerabilities, Attacks and Countermeasures in Wireless Sensor Networks at Various Layers of OSI Reference Model: A Survey," 2017 International Conference on Signal Processing and Communication (ICSPC), Coimbatore, 2017, pp. 288-293.

[25] owasp.org/www-project-top-ten/ [Accessed 18 Apr. 2020]

[26] https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A7-Cross-Site_Scripting_(XSS) [Accessed 18 Apr. 2020]

[27] Shichao Zhang, Z. Qin, C. X. Ling and S. Sheng, ""Missing is useful": Missing Values in Cost-Sensitive Decision Trees", in IEEE Transactions on Knowledge and Data Engineering, vol. 17, no. 12, pp. 1689-1693, Dec. 2005.

[28] T. Zhi, H. Luo and Y. Liu, "A Gini Impurity-Based Interest Flooding Attack Defence Mechanism in NDN," in IEEE Communications Letters, vol. 22, no. 3, pp. 538-541, March 2018.

[29] Ding, Chris and He, Xiaofeng, "K-Means Clustering via Principal Component Analysis", Proceedings of the Twenty-First International Conference on Machine Learning, Association for Computing Machinery, 2004.

[30] kali.org [Accessed 9 Apr. 2020]

[31] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, Ali A. Ghorbani, "Toward Developing a Systematic Approach to Generate Benchmark Datasets for Intrusion Detection", Computers Security, Volume 31, Issue 3, May 2012, Pages 357-374, ISSN 0167-4048, 10.1016/j.cose.2011.12.012.

[32] Sazzadul Hoque, Mohammad. "An Implementation of Intrusion Detection System Using Genetic Algorithm." International Journal of Network Security Its Applications 4., pp 109–120, 2012.

[33] Vishnu, B. and Kp, Jevitha, "Prediction of Cross-Site Scripting Attack Using Machine Learning Algorithms", pp 1-5,2014. doi:10.1145/2660859.2660969.

[34] scikit-learn.org/stable/modules/neighbors [Accessed 30 Mar. 2020]

[35] scikit-learn.org/stable/modules/tree.html [Accessed 30 Mar. 2020]

[36] Ali, Jehad and Khan, Rehanullah and Ahmad, Nasir and Maqsood, Imran, "Random Forests and Decision Trees", International Journal of Computer Science Issues(IJCSI), Vol. 9, 2012

[37] Probst, Philipp, Marvin N. Wright, and Anne-Laure Boulesteix."Hyperparameters and Tuning Strategies for Random Forest." Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 9, no. 3 (2019): e1301.

[38] scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier [Accessed 7 May 2020]

[39] Matija Stevanovic, Jens M. Pedersen, "On the Use of Machine Learning for Identifying Botnet Network Traffic.", Journal of Cyber Security and Mobility, Vol. 4, pp 1-32 2015.

[40] Beigi, Elaheh Biglar, et al. "Towards Effective Feature Selection in Machine Learning-based Botnet Detection Approaches." Communications and Network Security (CNS), 2014 IEEE Conference on. IEEE, 2014.

[41] mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-42/ [Accessed 30 Apr. 2020]

Appendix A

Full Results

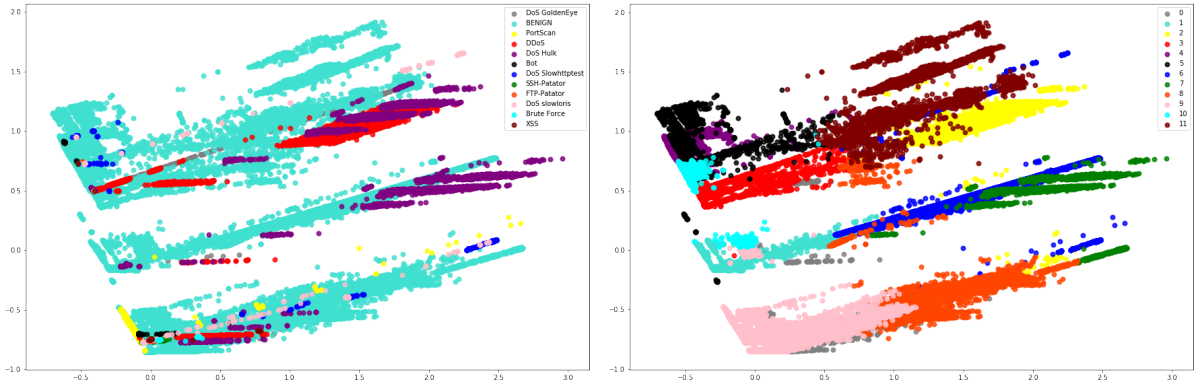

Figure A.1: PCA of Actual Labels (left) and Predicted Clusters (right) on the Original Labels of the Validation Dataset

Cluster	0	1	2	3	4	5	6	7	8	9	10	11
Benign	2,939	**200,212**	131	28,099	48,945	22,094	6,314	151	34,010	81,328	24,970	5,072
Bot	3	0	0	3	151	0	0	0	1	**233**	0	0
Brute Force	1	0	0	0	28	0	0	0	0	**272**	0	0
DDoS	**11,784**	0	4,216	9,601	0	0	0	0	0	4	0	0
DoS GoldenEye	**1,010**	0	61	340	13	0	151	0	7	476	0	0
DoS Hulk	2,796	1,652	**17,937**	13,399	39	0	10	10,148	0	38	0	6
DoS Slowhttptest	0	0	0	18	56	191	194	0	46	**595**	0	0
DoS Slowloris	0	0	0	1	14	160	214	0	363	**400**	0	7
FTP-Patator	0	0	0	4	2	**791**	0	0	0	790	0	0
PortScan	20	1	0	10	2	0	21	0	0	**31,707**	0	0
SSH-Patator	0	0	0	2	**598**	4	0	0	0	576	0	0
XSS	1	0	0	1	2	0	0	0	4	**122**	0	0

Table A.1: Predicted Clustering Result (K=12) Against True Original Labels

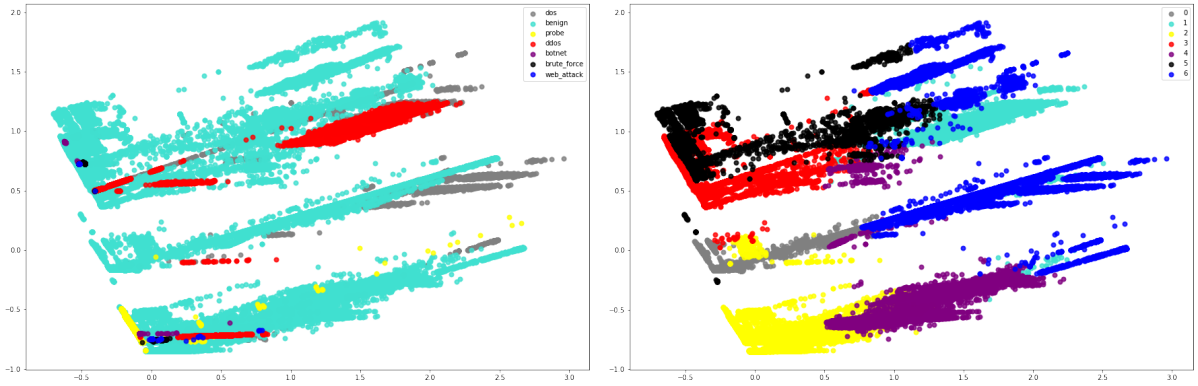

Figure A.2: PCA of Actual Labels (left) and Predicted Clusters (right) on the Grouped Labels of the Validation Dataset

Cluster	0	1	2	3	4	5	6
Benign	**200,622**	153	84,357	101,962	34,051	24,325	8,795
Botnet	0	0	**236**	154	1	0	0
Brute Force	0	0	**1,366**	606	0	795	0
DDoS	0	4,218	**11,675**	9,712	0	0	0
DoS	1,656	**18,399**	5,309	13,839	407	358	10,374
Probe	1	0	**31,727**	12	9	0	12
Web Attack	0	0	**396**	31	4	0	0

Table A.2: Predicted Clustering Result (K=7) Against True Grouped Labels

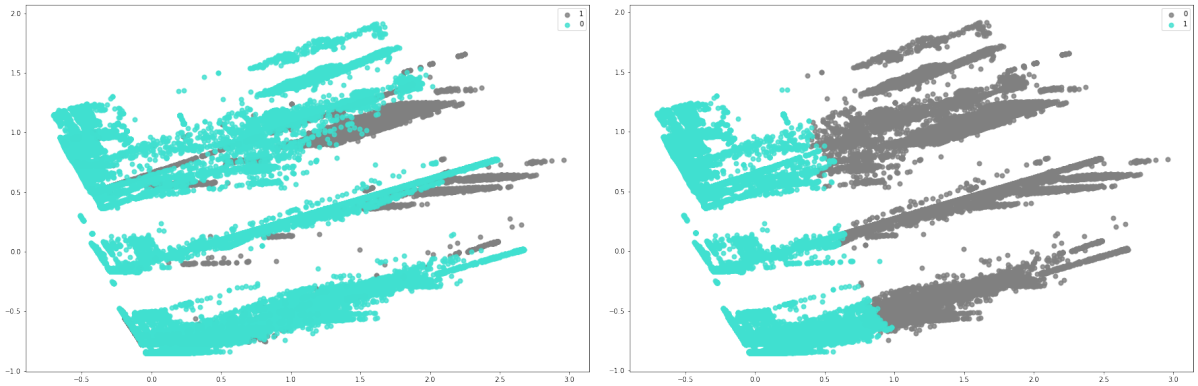

Figure A.3: PCA of Actual Labels (left) and Predicted Clusters (right) on the Binary Labels of the Validation Dataset

Cluster	0	1
Benign	38,857	**415,408**
Attack	33,468	**77,829**

Table A.3: Predicted Clustering (K=2) Result Against True Binary Labels

	SVM			Decision Tree			Naive Bayes			KNN			Random Forest		
	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
Benign	96.87	97.51	97.19	99.91	99.92	99.92	87.74	98.54	92.83	99.55	99.40	99.48	99.91	99.89	99.90
Bot	0.00	0.00	0.00	80.15	82.61	81.36	0.00	0.00	0.00	82.05	57.29	67.47	81.90	46.29	59.15
Brute Force	0.00	0.00	0.00	72.52	72.52	72.52	0.00	0.00	0.00	73.77	89.40	80.84	75.97	77.48	76.72
DDoS	98.61	82.92	90.08	99.96	99.97	99.97	77.17	49.57	60.36	99.69	99.30	99.49	99.98	99.94	99.96
DoS GoldenEye	81.66	69.21	74.92	99.86	99.17	99.42	88.73	6.12	11.45	98.07	98.59	98.33	99.80	99.13	99.46
DoS Hulk	98.04	86.00	91.63	99.86	99.87	99.86	66.22	61.37	71.24	99.34	99.64	99.49	99.39	99.66	99.52
DoS Slowhttptest	81.12	75.07	77.98	98.72	98.36	98.54	66.21	17.83	28.10	97.57	98.64	98.10	99.45	98.81	99.61
DoS Slowloris	93.16	49.35	64.52	98.21	99.40	98.80	4.80	20.19	7.76	98.13	99.40	98.76	99.94	99.87	99.91
FTP-Patator	100.00	50.41	67.03	99.87	99.94	99.91	0.00	0.00	0.00	99.75	99.75	99.75	99.41	99.97	99.69
PortScan	74.75	99.29	85.29	99.39	99.18	99.29	15.57	0.19	0.37	992.51	94.70	93.59	99.41	99.97	99.69
SSH-Patator	0.00	0.00	0.00	99.75	99.83	99.79	0.00	0.00	0.00	98.22	98.47	98.35	1.00	99.41	99.70
XSS	0.00	0.00	0.00	36.51	35.11	35.80	0.00	0.00	0.00	52.73	22.14	31.18	45.13	38.93	41.80

Table A.4: Full Results for Supervised Methods on the Validation Dataset by Label

	SVM			Decision Tree			Naive Bayes			KNN			Random Forest		
	Original	Grouped	Binary	Original	Grouped	Binary	Original	Grouped	Binary	Original	Grouped	Binary	Original	Grouped	Binary
Precision (%)	59.65	65.76	89.23	90.82	96.89	99.80	34.51	35.01	84.81	89.64	95.07	98.74	90.77	98.08	99.80
Recall (%)	49.08	54.70	87.05	90.87	97.35	99.80	20.76	29.26	70.64	89.11	94.43	99.06	90.15	96.58	99.85
F1 Score (%)	52.68	56.92	88.08	90.84	97.12	99.80	22.42	31.21	74.66	89.24	94.72	98.80	90.42	97.28	99.83
Accuracy (%)	94.28	94.19	92.67	99.83	99.87	99.87	86.45	86.55	87.02	99.24	99.27	99.30	99.85	99.91	99.89
Training Time (s)	453.01	343.23	121.45	105.93	100.01	92.33	11.47	7.07	0.60	2306.97	2507.44	2457.47	77.36	78.26	70.19
Classification Time (s)	0.20	0.10	0.03	0.73	0.23	0.16	0.13	0.11	0.06	1963.92	2064.91	2008.67	0.99	0.90	0.65

Table A.5: Summary of Results for Supervised Methods on the Validation Dataset

Appendix B

Project Proposal

Computer Science Tripos Part II - Project Proposal

Machine Learning for the Detection of Network Attacks

Kyra Mozley

25th October 2019

Introduction

Historically, people have implemented Intrusion Detection Systems (IDS) to protect their network, these rely on signature-based methods of known attacks to detect network anomalies[1]. This leaves them helpless in the face of new, never seen before attacks. As the cyber threat landscape changes we must change the way that we are detecting attacks.

The aim of this project is to implement my own IDS using a machine learning approach. The concept of machine learning has been around for many years, but its use has grown rapidly in the last decade or so. It allows a system to 'learn' from past events without being explicitly programmed, which is what makes it perfect for the detection of network attacks that would slip past a traditional IDS.

I will be evaluating a variety of different machine learning algorithms, from a combination of supervised and unsupervised approaches to see which has the best overall performance. I will then select and implement the model with the best performance in my IDS. It is

[1] `http://www.ijsrp.org/research_paper_jul2012/ijsrp-july-2012-105.pdf`

likely that specific models will provide better performance for specific attacks. In such cases I aim to create an ensemble model to ensure the best overall detection rate across all attack types.

Starting Point

I have a theoretical understanding of some machine learning algorithms from Part IB *Machine Learning and Real-world Data* as well as some general background reading. Furthermore, Part IB courses in *Security* and *Computer Networking* have also provided useful background knowledge. I have some basic experience with Python, but have never performed machine learning or built a complex project with it.

Much research already exists on applying machine learning for network anomaly detection[2]. For example, Darktrace[3] is a proprietary software which uses AI to build a pattern of a network allowing it to understand what is considered 'normal activity' so it can identify and respond to threats in real time.

The UNSW-NB15 dataset[4], created by the Australian Centre for Cyber Security has been used in previous papers[5] on this topic. It contains two million record entries, nine types of attacks (Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode and Worms) and has a total of 49 features. The paper referenced used Support Vector Machines as their learning algorithm on this dataset, and so this will be the first one I implement.

Substance and Structure of the Project

Research: I will start my work with background research on the algorithms, including both supervised and unsupervised methods, used in machine learning based detection. This will allow me to gain an understanding of the algorithms, and what methods are appropriate for detecting which attacks. Lastly, I will need to gain further experience using Python, mainly for a machine learning and data handling approach.

Find and preprocess data: I will find a suitable recent network traffic dataset which contains a mix of normal traffic and attacks. It must be recent since if the dataset is outdated then it may not reflect current attacks, and so will be trained on attacks that may not occur as often today. Furthermore, as I plan to use a mix of supervised and unsupervised approaches, I will need labelled and unlabelled data. The UNSW-NB15

[2] `https://personal.utdallas.edu/~muratk/courses/dmsec_files/oakland10-ml.pdf`
[3] `https://www.darktrace.com/en/`
[4] `https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-NB15-Datasets/`
[5] `http://worldcomp-proceedings.com/proc/p2016/SAM3540.pdf`

dataset is sufficient, but it only contains 9 attacks; I hope I can find an appropriate dataset with more attacks than this.

Feature Selection: Feature selection allows the machine learning algorithm to train faster, reduces overfitting and (provided the right features are chosen) improves accuracy. I will apply a filter method to select the features from the dataset that I will use.

Evaluate different machine learning algorithms: I will use the Python Scikit-learn[6] library to train a range of supervised and unsupervised machine learning algorithms and compare relevant metrics (such as precision and recall) to identify the strengths and weaknesses of each.

Build an interface: Once I have decided how to implement the machine learning aspect, I will build a tool to integrate the network sniffer and the machine learning model to provide the functionality to alert when network attacks are detected.

Final Evaluation: I will need to evaluate the performance of my IDS. I will simulate network attacks alongside normal traffic and evaluate it using standard metrics.

Success Criterion

The project will be a success if I have:

- Implemented at least one machine learning algorithm and evaluated its precision on a proportion of the dataset reserved for testing.
- Successfully developed an interface for the intrusion detection system, which can sniff and monitor traffic and alert on a suspected attack.
- Evaluated the IDS detection accuracy when tested on simulated network traffic.

Possible Extensions

Extensions are focused on improving the accuracy of the IDS.

Redundancy: I could also implement a signature based method, which gets checked first so that way it will always detect known attacks, improving the precision of the system. This will also need a way of remotely being updated as new signatures become available.

Deep Learning: Whilst machine learning makes decisions based on what it has learnt in training, deep learning continuously learns using a neural network. Implementing a deep learning approach will likely yield greater performance, since it can evolve over time, but would require more compute power.

[6] scikit-learn.org

Training Data: The accuracy of a system relies heavily on the data it is trained on. Retraining the IDS on another appropriate dataset, or perhaps merging datasets if it allows, could lead to an overall better performance.

Timetable and Milestones

1. **24th October - 6th November: Research**
   I will research into the various types of machine learning algorithms and decide which ones I will implement in my evaluation. In addition, I will also gain the relevant experience in Python that will be needed to undergo this project.
   Milestone: Deliver a write up explaining the machine learning algorithms I will implement to my supervisor.

2. **7th November - 20th November: Preparation**
   I will find a relevant dataset that is suitable for use in this project, clean and preprocess it ready for the next step. In addition, I will perform feature selection, using a filter approach.
   Milestone: I have agreed with my supervisor the features that I will use for the next step.

3. **21st November - 4th December: Implementation**
   Using the data, I will train and then test the various machine learning algorithms I had chosen and calculate the relevant metrics of each to compare performance.
   Milestone: First success criterion met.

4. **5th December - 18th December: Implementation**
   I will implement the sniffer tool. Integrate the best performing machine learning model with the sniffer tool to provide a command line IDS system.
   Milestone: Sniffer tool integration with IDS passes simple test case; notifies of attack when fed with the datasets test attack data.

5. **19th December - 1st January: Christmas Holiday**

6. **2nd January - 15th January: Implementation**
   Contingency weeks, perhaps work on any extensions that seem suitable.

7. **16th January - 29th January: Evaluation**
   Evaluate the IDS. Also prepare my progress report.
   Milestone: Completed progress report and presentation (deadline 31st Jan).

8. **30th January - 12th February: Evaluation**
   Perform final evaluation of the IDS by simulating network attacks and traffic.
   Milestone: All success criteria met.

9. **13th February - 26th February: Write Up**
   Write first draft of introduction and preparation chapters, and give to supervisor.

10. **27th February - 11th March: Write Up**
    Write first draft of implementation chapter, and give to supervisor.

11. **12th March - 25th March: Write Up**
    Write first draft of evaluation and conclusion chapters, and give to supervisor.
    Milestone: First draft of dissertation complete.

12. **26th March - 8th April: Write Up**
    Reread and edit dissertation, sending it to my supervisor and director of studies for feedback.

13. **9th April - 22nd April: Exam Revision**

14. **23rd April - 6th May: Write Up**
    Make any final changes based on feedback, prepare to submit all required work.
    Milestone: Dissertation complete and submitted (deadline 8th May).

Resource Declaration

I will be using my personal laptop for convenience; an Apple MacBook Pro (quad-core Intel i7-8559U, 16GB RAM) which dual boots macOS Catalina and Kali Linux. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

I will be using Github[7] with a private repository for revision control, ensuring I push any changes made, as well as regularly uploading all source code, data and the dissertation to Dropbox[8]. In event of hardware failure, I have a spare Lenovo Thinkpad X121e (dual-core Intel i3-2357M, 8GB RAM, Ubuntu 19.04), as well as the option to use the MCS workstations provided.

[7] Github.com
[8] Dropbox.com