A Study of Machine Learning classifiers for Botnet Traffic Detection with an Imbalanced Dataset

Project Report

Group 1005

Aalborg University
Electronics and IT

Electronics and IT
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
A Study of Machine Learning classifiers for Botnet Traffic Detection with an Imbalanced Dataset

Theme:
Thesis

Project Period:
Spring Semester 2024

Project Group:
1005

Participant(s):
Kamilla Andersen-Otte
Georgi Toshkov Bolgurov

Supervisor(s):
Ashutosh Dhar Dwivedi

Page Numbers: 90

Date of Completion:
May 30, 2024

Abstract:

This thesis investigates the use of machine learning models to classify botnet traffic within an Internet of Things (IoT) network. Given the increasing prevalence of IoT devices in our society, their limited computational power makes them a vulnerable target for botnet exploitation, making advanced detection mechanisms that can adapt to evolving threats with minimal false positives necessary. Traditional methods of network security often fail to adapt to the dynamicaly adjusting nature of botnet attacks, making machine learning, with its ability to learn and detect patterns in data a more effective solution. This study refines and enhances machine learning models specifically tailored for IoT botnet detection by tackling three key questions: ensuring proper pattern recognition through training on imbalanced datasets, optimizing machine learning models for botnet traffic detection in IoT networks, and identifying the most influential network traffic features for detecting potential botnet activities. Five machine learning models - decision tree, random forest, Gaussian Naive Bayes, XGBoost, and a voting classifier - were trained on the BoT-IoT dataset sample, with the feature sets being selected based on both feature correlation and forward and backward selection methods. Addressing the dataset imbalance, different techniques were employed to balancing the classes. The trained models were also tested on a newly sampled dataset to provide performance validation. The results indicated that the voting vlassifier, combining decision tree and XGBoost on an oversampled dataset, achieved the most favorable performance.

Contents

Preface

Aalborg University May 30, 2024

Here is the preface. You should put your signatures at the end of the preface.

Kamilla Andersen-Otte
kande22@student.aau.dk

Georgi Toshkov Bolgurov
gbolgu22@student.aau.dk

Chapter 1

Practical Methods and Approaches

This chapter will outline the methods used for gathering and analyzing the data in this thesis to provide an overview of the process and agenda behind the development of the project and solution.

Information gathering

The information gathering for this thesis was initiated with a literature review of the state of the art within the field of botnets, IoT and machine learning. The literature was found through Google Scholar and Aalborg University's PRIMO and the search was based around the following keywords and their combinations:

- **"botnet"**
- **"detection"**
- **"identification"**
- **"IoT"**
- **"network"**
- **"traffic"**
- **"machine learning"**

Each type of chosen literature represents a contemporary analysis of botnets, some of them specifically within IoT networks, with some of the literary works also providing suggested solutions as to how they can be detected through the employment of machine learning.

Information processing

The data used for the experimental part of this thesis builds on the theory gathered from the literature review. The dataset was chosen by exploring the state-of-the-art literary works within the field of botnet detection, many of which utilized the BoT-IoT dataset from UNSW Canberra.[1] The classifiers used for the initial test cases in this thesis were also assembled from the literature review.

Python was used for training and testing all the models on the dataset, by using the following relevant libraries:

- **xgboost** for the deployment of the XGBoost classifier [2]
- **scikit-learn** for deploying the remaining models, for preprocessing of the data, performance metrics and for the feature selection methods [3]
- **time** for calculating the training time of the classifiers [4]
- **matplotlib** for plotting Pearson's correlation coefficients into a heatmap and visualising the decision tree[5]
- **seaborn** for visualizing the heatmap [6]
- **pandas** for loading the dataset files [7]
- **numpy** for handling arrays of data, such as saving the dataset in an array and then splitting it into testing and training sets [8]
- **pickle** for saving the trained models [4]
- **math** for certain mathematical tasks, such as calculating the Shannon entropy [4]
- **imblearn** for oversampling and undersampling of the dataset [9]

Overview of thesis structure

Chapter 2 will introduce the topic of botnets and how they relate to IoT networks, laying the groundwork for the problem formulation. The rest of the chapter will delve deeper into the background of the issue and draw on literary sources from the state of the art to extract the necessary tools for botnet detection.

Chapter 3 will further examine the theory, construction, and limitations of these tools to discard the ones that are not relevant.

Chapter 4 will describe the experiments conducted using the previously selected tools.

The theoretical approach behind the test cases will be explained with mathematical expressions, and the practicalities of the chosen parameters will also be described. The results from the experiments will be presented.

Chapter 5 will interpret the results by exploring their purpose, alignment, and any limitations of the experiments. The chapter describes how they could be transferred to a real-life detection system and possible future directions.

Chapter 6 is the conclusion, where the problem formulation will be addressed and the final solution for a botnet detection method will be presented.

Chapter 2

Introduction & Background

2.1 Introduction

Botnets are collections of compromised devices infected with malware and connected over the internet for an entity to exploit for malicious purposes [10]. Each compromised device in the botnet and the entity controlling them is the botmaster. Botnets increasingly disrupt our digital society through activities like spam, data theft, denial of service attacks, and malware propagation.[11]

A fundamental element of any botnet is its communication strategy[12]. Traditionally, botnets have used Internet Relay Chat (IRC) for coordination, with bots linking to an IRC server[13]. The botmaster in control of the botnet uses command and control (C&C) channels on this platform to manage the network of infected machines, sending commands and updates to maintain control over long periods.

While IRC for many years has been the dominant method, the adoption of HTTP-based communication has also become common[12]. Both IRC and HTTP-based botnets are based on a centralized structure, which makes them vulnerable to a single point of failure through the control server.

In recent years, there has been a shift towards more decentralized Peer-to-Peer (P2P) architectures. P2P botnets are constructed without a single point of failure that can be easily targeted, unlike IRC and HTTP. This decentralized approach allows the bot master to disseminate commands across the network, complicating the destruction of the botnet[14]. Modern botnets also often use various encryption methods[15], further complicating detection efforts. This combination of decentralization and encryption in newer botnets makes it challenging to fully assess their size and influence, thereby escalating the risks and challenges faced by cybersecurity measures.

Previously, the threat of botnets was primarily confined to home and office computers. However, this has changed with the exponential growth of the Internet of Things (IoT) sector, connecting millions of devices daily [16]. The now widespread use of IoT devices throughout most modern environments is rapidly growing, from fully automated factories

to smart cities. IoT devices offer both convenience and potential attack vectors[17]. While the connectivity offered by various IoT devices has immense benefits, it must be recognized that most prioritize functionality over security. The lack of encryption and usually outdated software leaves them vulnerable to malicious attacks [18]. For instance, a 2023 report by Securelist highlighted a significant increase in brute-force attempts targeting IoT devices, with 97.91% of these attempts targeting the Telnet protocol—widely used by various IoT devices [19], underscoring the need for research into detection solutions in this field.

This thesis focuses on detecting botnet traffic on an IoT network, addressing the detection of traffic in the attack phase of the botnet life-cycle. The machine learning algorithms employed will be decision tree, random forest, XGBoost, and Naive Bayes, as well as a voting classifier combining XGBoost with decision tree and Naive Bayes. The models will be trained on a 5% sample of the BoT-IoT dataset and tested on an additional dataset sampled for the purpose of this project, as well as oversampled and undersampled variations. Emphasis will be placed on dataset analysis and feature engineering, examining how imbalanced datasets impact machine learning algorithms and addressing this imbalance, along with how to optimize the classification through feature selection. For this, different filter and wrapper methods will be explored and implemented. Our work aims to provide clarity on multiple subject matters in the field, as outlined in the next section containing the problem formulation.

2.1.1 Problem formulation

In this thesis, we address the issue of detecting botnet attacks in IoT environments, of which the exploitation of the vulnerabilities in these connected devices is an increasing problem in our digitized society. The wide use of IoT technologies presents unique challenges in maintaining security measures, highlighting the need for advanced detection mechanisms that can adapt to evolving threats with minimal false positives and false negatives. Our study seeks to refine, enhance and validate the use of machine learning models specifically tailored for IoT botnet detection by answering the following key questions:

1. **How can we ensure proper pattern-recognition through training of machine learning models when handling imbalanced datasets, in a botnet detection scenario where attack traffic significantly outnumbers normal traffic?**
    - Here, we consider various machine learning algorithms and their configurations to determine which models provide the best trade-off between detection precision and computational efficiency, as well as how we can validate the success of the performance evaluations.
2. **What are the most effective strategies to optimise machine learning models for botnet traffic detection in IoT networks?**

- This question explores methods and techniques for adapting the learning algorithms to improve the model performance when handling imbalanced data from an IoT network, thus preventing bias towards the majority class.

3. **Which network traffic features are most influential in identifying potential botnet activities, and how can these features be systematically evaluated and selected?**

    - This involves describing and analysing network traffic features based on their predictive power and relevance to identifying malicious activities, potentially leading to more streamlined and focused detection models.

By addressing these questions, our research aims to develop a framework that enhances the detection of botnets in IoT networks with more precision and less false predictions, thereby contributing to more secure IoT environments.

2.2 Background

Due to the IoT systems being a both young and booming field, the devices are susceptible to suffer from numerous security issues. Some of the problems that are often represented within IoT security happen because of the rapidly growing market, where the manufacturers might bring the products to market before they have implemented sufficient security measures[20]. In addition to this, IoT devices are often small with limited processing power, which means that they might not have the computing resources to constantly update their firmware and run robust security protocols[21]. They are also physically exposed to security threats, since they are often placed in order to be accessible for the user, but in return is also more vulnerable to be tampered with - in addition to this, they are placed directly in the hands of often uneducated end users that do not have sufficient security awareness to know the benefits of changing the default password and not neglecting updates to the device[22].

All of these vulnerabilities, in addition to the sheer volume of IoT devices existing worldwide, make them prime targets for botnets. Cybercriminals can exploit these weaknesses by using automated scripts to scan and compromise large numbers of devices[23]. Significant incidents, like the Mirai botnet, demonstrated how default credentials in IoT devices could be leveraged to create massive botnets capable of launching large-scale DDoS attacks[24]. The Mirai malware targets IoT devices by scanning the internet for devices with default login credentials, infecting them, and using them to launch coordinated attacks. This botnet notably disrupted major internet services, highlighting the severe impact of insecure IoT devices[23].

Modern botnets have evolved from relying on centralized C&C servers to adopting decentralized, P2P architectures[25]. P2P botnets are established through P2P transfer pro-

tocols where the devices in the botnet are able to communicate directly with each other and therefore do not rely on a single server for C&C. This means that the botmaster does not need a centralized server to issue commands to the zombies but can instead use any of the zombies in the botnet to do this. This enhances resilience by eliminating single points of failure, making disruption more challenging. Additionally, botnets may employ evasion techniques that requires more sophisticated detection mechanisms, for instance through encrypted communication channels or domain generation algorithms that generate a large number of domain names for C&C servers, making it difficult to predict and block malicious domains[10].

Traditional detection methods, like signature-based and heuristic-based approaches, often fall short against the more sophisticated botnets. Detection techniques that are based on recognizing signatures of the botnet traffic may struggle with polymorphic and zero-day attacks, while heuristic methods that identify botnet traffic based on defined rules or behavior analysis may result in a large amount of benign traffic being flagged as suspicious [26]. Furthermore, the encryption and obfuscation strategies employed by modern botnets may obscure the malicious activities, complicating detection based on heuristics [27].

Nazir et al. [28] conduct a comprehensive review of the state-of-the-art within IoT botnet detection. Machine learning algorithms are mentioned as one approach for automatically identifying potential threats, which is an increasingly popular approach as they can analyze large amounts of data and learn from and adapt to novel types of data. This is an innovative and likely solution for the growing landscape within IoT, and using ML algorithms will be beneficial for scalability, real-time detection, and accuracy in identifying and analyzing IoT botnets.

Machine learning offers a promising solution for botnet detection by analyzing large volumes of network traffic to identify the patterns that are indicative of botnet activity. Machine learning algorithms can learn from both labeled and unlabeled data, making them adaptable to new and evolving threats. For supervised machine learning, the model is trained on data where the desired output to be predicted is already known, while for unsupervised, the model finds the structures in the data without human intervention. Supervised learning generally produces better performance; however, unsupervised learning may be more appropriate for experimenting with training a model on unknown data.

By integrating the problem of botnets within the field of IoT networks with supervised machine learning detection methods, this study aims to explore enhancing IoT network security through the development of robust and accurate botnet detection systems, contributing to a more secure digital ecosystem.

2.3 State of the art

After establishing the background for exploring botnets and analysing their traffic for the purpose of finding a robust method for detecting them, this section will focus on the examination of current identification solutions. The goal is to gain insights into specific advancements and challenges within this field and examine state-of-the-art methodologies used for detecting and mitigating botnet traffic, exploring both their effectiveness and limitations.

Alhowaide, Alsmadi and Tang [29] explored how to mitigate threats within the realm of IoT devices through intrusion detection systems based on machine learning. They lean on feature selection as a method for handling the often large amount of dimensional data needed for analysis when looking into solutions in this field, since effective feature selection reduces the volume and variety of the data by removing any redundant features while keeping the ones necessary for receiving satisfactory results. They tested multiple filter methods for feature selection on multiple datasets, both with regular network traffic and specifically IoT traffic, and the performances of the differently selected feature sets were measured. The results were based around the elapsed time of each filter method, how much they reduced the dataset in percentage, how many features were chosen, as well as the F-Score and ROC-AUC when training and testing five different machine learning models with the feature sets. The models were Bernoulli Naive Bayes, decision tree, k-nearest neighbor (KNN), Gaussian Naive Bayes, and random forest, where Bernoulli had the broadest distribution of performances depending on the feature selection method and random forest and K-Nearest Neighbor generally had the best performance metrics.

Alothman, Alkasassbeh and Baddar [30] aim for a multiclass approach where they not only distinguish between benign and botnet traffic, but also identify the type of botnet traffic. They preprocess the data by oversampling with Synthetic Minority Oversampling Technique (SMOTE) and train three different classifiers: decision tree, random forest, and multi-layer perception with the BoT-IoT dataset. This is done both with the complete feature set, as well as with two selected feature sets based on Pearson's Correlation Coefficient and Relief-F. The performance evaluation results were based on which classifier reached the highest accuracy and F-score with which feature set. In addition to this, they evaluated the false negative rate for both binary classification of benign versus botnet traffic, as well as for the specific categories and subcategories of botnet traffic, to measure which type of classification resulted in most of the botnet traffic being undetected. The results showed that random forest and decision tree were superior in achieving the best scores for both binary and multiclass classification.

Saad et al. [31] specifically focus on the detection of P2P botnets, which have the added functionality of decentralisation, where the botmaster can use any of the bots to distribute

commands to other bots, making the botnet more difficult to shut down. They train five different machine learning models: support vector machine (SVM), KNN, Gaussian-based classifier, Naive Bayes, and artificial neural network (ANN), and extract the features for training based on whether they are useful for linking specific types of network traffic or if they can identify hosts with similar patterns. They classified the traffic into three different classes and evaluated the models on training speed, prediction speed, the ratio of true predictions, and the ratio of total errors. They concluded that none of the models managed to satisfy all of the requirements. SVM, KNN, and ANN performed well for the detection of botnets; however, both the ANN and SVM were deemed unsuitable for online detection due to the training and classification time.

Leevy et al. [32] analyze specifically the BoT-IoT dataset in order to find the minimum number of features for binary classification with the decision tree classifier to provide a simplified approach for handling the large amount of data. In addition to this, they generated the feature importance in order to choose the features that caused the biggest reduction in impurity, thereby increasing the accuracy, and the results showed that the top three features (1: destination ports, 2: source to destination byte count and 3: transaction state) were sufficient for receiving near-perfect results, where the evaluation metrics they used for the performance of the models were AUC, the F-score and the AUPRC.

Pokhrel, Abbas and Aryal [33] aim to detect specifically DDoS attack traffic in an IoT network. The machine learning models used where KNN, Gaussian Naive Bayes and multi-layer perception ANN. They used the BoT-IoT dataset, where they applied feature selection by setting a F-score threshold value and oversampled the minority class with SMOTE. The models were evaluated with the accuracy score and ROC-AUC, where KNN showed the most stable results with both the imbalanced dataset and the oversampled version.

Venu, Kumar and Rao [34] explore how four different machine learning models, KNN, Naive Bayes, random forest and logistic regression, manages to detect botnets in three different datasets, CTU-13, CICIDS2017 and IoT-23. Feature selection was performed by using logistic regression to extract the top 10 features from each data set, based on what features that have the strongest associations with the target variable. The results were evaluated on accuracy, precision, recall and F1-score, where random forest got a perfect score on all three datasets.

Kim et al. [35] aim to analyze the performance of seven different machine learning models, as well as two deep learning models. The machine learning models were Naive Bayes, KNN, logistic regresssion, decision tree and random forest. They were trained on the N-BaIoT dataset and tested with both binary and multiclass classification, which was evaluated with the F1-score. The results showed that all models except logistic regression got

satisfactory scores on the binary, while both logistic regression, KNN and Naive Bayes showed low performance results on the multiclass - leaving decision tree and random forest as the best classifiers in this study.

Alshamkhany et al. [36] explore the application of classic machine learning models for detecting botnet attacks. Their study utilizes the UNSW-NB15 dataset and employs feature selection through principal component analysis, Chi-squared and ANOVA. Decision tree, Naive Bayes, KNN and SSVM are trained and tested, where the key findings show that decision trees achieve 100% performance evaluation in precision, recall and F-score after applying feature selection.

Guerra-Manzanares, Bahsi and Nomm [37] delve into different types of feature selection methods for improving machine learning models for detecting botnet traffic in IoT networks. They explore Fisher's score and Pearson's correlation coefficient along with sequential forward feature selection and sequential backward feature selection, evaluating them separately and combined. They used KNN and random forest for multi-class classification and the results were evaluated with the F-score, showing that combining filter and wrapper methods for a hybrid feature selection does improve the F-score as opposed to just using filter methods, while it also reduces some of the computational complexity of wrapper methods. The best results were received when using Fisher's score with random forest, or combining Fisher's score with the wrapper methods for KNN.

Lefoane et al. [38] research how to apply feature selection for removing redundant features with the aim of making botnet detection methods more efficient. In this study, the feature selection is based on the feature value frequency in each of the features as represented in a dataset for binary classification of benign and botnet traffic, in order to remove the features with the most noise based on a threshold value. After this, they use decision tree, logistic regression, and SVM for classification and evaluation of their performance, where they measured the true positive rate, the false positive rate, the precision, F-score, and the overall success rate, which is the proportion of all the correctly classified instances to the total amount of instances. The results showed that the proposed feature selection method does result in improved performance across all of the evaluation metrics, with logistic regression and decision tree achieving the overall best scores.

Kalakoti, Nomm and Bahsi [39] explore how to minimise feature sets for machine learning by applying either filter or wrapper method feature selection - namely Pearson's correlation coefficient, Fisher's score, mutual information, and ANOVA for filter methods, while for wrapper methods they used recursive feature elimination, sequential forward feature selection, and sequential backward feature selection. These feature selection methods were applied to two different datasets, N-BaIoT and MedBIoT. The classification formulations were both binary for benign and botnet traffic, as well as multiclass for predicting multi-

ple other attributes such as the specific attack type, the malware type or the botnet phase. For predicting these target variables, four different machine learning models were used - decision tree, random forest, KNN, and extra tree classifier. They were evaluated based on the F1-score, and the highest detection rate with the least time to classify was achieved by the decision tree with sequential backward selection for both binary and multiclass classifications. In general, the wrapper methods were more effective in finding optimal feature sets for each classification.

Al-Sarem et al. [40] study how to maximise the efficiency of machine learning intrusion detection systems through feature selection on the N-BaIoT dataset. The proposed method is an aggregated version of mutual information, principal component analysis, and ANOVA, and the models used for classification are random forest, XGBoost, Gaussian Naive Bayes, KNN, SVM, and logistic regression. The models predicted both binary classification for benign or TCP attack traffic, and multiclass classification for benign, Bashlite, and Mirai traffic instances. The evaluation of the performance of the models was done with precision, recall, and F1-score, where the best results were yielded with mutual information feature selection for binary classification, and XGBoost and KNN generally achieved the best scores.

State of the art key findings

This section evaluated state-of-the-art literary works within the topic of botnet identification. It identified the models, datasets, feature selection methods and evaluation techniques employed in various studies within the topic of botnet detection, providing valuable insights into potential directions for our own further research. The literature showed tendencies between the machine learning models used for classification, in all cases multiple models were used either separately for comparison or combined to make a hybrid model. In addition to this, favorable results were often received when utilizing feature selection methods to remove redundant features or emphasize features with strong correlation to the target variable. Guerra-Manzanares, Bahsi & Nomm chose to combine filter and wrapper methods with favorable results, while the other works compared feature selection techniques. The evaluation metrics were most often based on accuracy, precision, recall and/or the F1-score.

Table 2.1 will show an overview of the notable findings from the literature that we will consider for the further progress of this project, in terms of the different machine learning models used and their performances.

When exploring the suitable datasets for training a model to identify IoT botnets, numerous datasets appeared in the state of the art, one of them being BoT-IoT. The Bot-IoT dataset was created by UNSW Canberra and incorporates both benign and botnet traffic with more than 72 million records - as the name suggests, it consists of IoT network traffic,

including multiple different types of botnet attacks, such as (D)DoS, keylogging and data exfiltration, in addition to benign traffic[1]. This dataset has been referenced in numerous of the literary works in this chapter, which prompts us to further explore the performance and resilience of models that are trained with the BoT-IoT dataset. We will be using the BoT-IoT dataset for this project to assess its performance and explore opportunities for enhancement.

Table 2.1: The best performing machine learning models from the literature review

Model	Sources that used it	Results
KNN	Pokhrel, Abbas & Aryal [33]	92.1 Accuracy & 92.2 ROCAUC
	Al-Sarem et al. [40]	98.28 Accuracy
	Guerra-Manzanares, Bahsi & Nomm [37]	Over 99.9 accuracy when using Fisher's score + SFFS
	Alhowaide, Alsmadi & Tang [29]	2nd best performance across feature selection methods
Decision Tree	Kalakoti, Nomm & Bahsi [39]	Highest detection rate & lowest time to classify
	Alshamkhany et al. [36]	100 Precision, Recall & F-score
	Lefoane et al. [38]	99.9 Overall Success Rate & F-score
	Kim et al. [35]	100 Precision, Recall & F1-score for binary
	Leevy et al. [32]	100 F1-score & AUPRC
	Alothman, Alkasassbech & Baddar [30]	96.0 Accuracy in binary & 93.0 in multiclass
Random Forest	Venu, Kumar & Rao [34]	100 Accuracy, Precision Recall & F1-score
	Alothman, Alkasassbech & Baddar [30]	96.3 Accuracy in binary & 93.0 in multiclass
	Guerra-Manzanares, Bahsi & Nomm [37]	Over 99.9 accuracy when using Fisher's score
	Alhowaide, Alsmadi & Tang[29]	Best performance across feature selection methods
	Kim et al. [35]	100 Precision, Recall & F1-score for binary
Naive Bayes	Kim et al. [35]	100 Precision, Recall & F1-score for binary
XGBoost	Al-Sarem et al. [40]	99.19 Accuracy
SVM	Saad et al. [31]	Highest detection rate & highest training time
Logistic Regression	Lefoane et al. [38]	100 Overall Success Rate & F-score

Chapter 3

Theory

After having explored the state of the art within botnet identification, understanding the underlying theories behind its models and techniques becomes essential. The machine learning models that were highlighted in the previous chapter, as well as methods for feature selection, will be further examined in this chapter. The goal of adding feature selection is to identify which features in the dataset that will be most informative for achieving an optimal result when applying the model.

3.1 Rationale for choice of machine learning models

The selection of algorithms for any machine learning task is critical and should be based on the specific characteristics of the data, the computational resources available, and the desired accuracy and interpretation of the model[41]. In this section, we will discuss the models from last chapter and choose which ones to use for our experiments based on their characteristics, strength and weaknesses.

For the analysis of the BoT-IoT dataset, which is both large and highly imbalanced, the chosen models should be well-suited to handle the complexities of network traffic data, which includes handling large volumes of data containing numerous features with a wide range of feature values across the data set.

The models chosen for further analysis as based on the literature review is KNN, SVM, decision tree, random forest, XGBoost, logistic regression and Naive Bayes. This section will describe their characteristics, highlight their differences and explore their practical implementation for detection of botnet traffic through a large-scale dataset such as BoT-IoT.

Naive Bayes

Naive Bayes is based on Bayes theorem, which describes the probability of an event based on already acquired knowledge about conditions that might be related to this event. Naive

Bayes assumes that each feature will contribute independently, and therefore will not affect the presence of other features, even though this independence might not hold in real-world data, where they might actually be dependent - this is why it is called "naive". Since the model assumes that the presence of each feature is independent of the others, the probability is calculated for each class. To make a prediction, the class with the highest conditional probability is selected[42].

Naive Bayes is fast in making predictions, since it only computes the probabilities of the features for each class and then chooses the class with the highest probability, which is a significant advantage in real-time detection systems. It works under the assumption of feature independence, which might not always hold in real-life but provides a simplified assumption for the computation. It requires a smaller amount of training data for estimation, which makes it suitable for quickly adapting to changes in attack behaviors, which is appropriate dynamic environments such as IoT networks. While not directly applicable here, its effectiveness in text classification tasks demonstrates its robustness in handling diverse types of data, supporting its use in network traffic classification where metadata also often behaves independently, such as duration, source port and packet rates[43].

K-Nearest Neighbors

The KNN algorithm classifies data points based on the "nearest neighbours". If we visualize our existing data as points in a graph – depending on how they cluster, we can then classify them on that. If we want to classify a new data point, we will do that based on what its nearest neighbours in the graph are. This means that KNN is a way to say that we do not need to compare the new data to classify with the whole graph, but just the K nearest neighbours, where K is whichever number we choose.

KNN showed notably good performance in multiple of the literary works - however, it is very computationally costly and therefore not recommended for use with large datasets[44].

Support Vector Machine

For SVM, the goal is to find the best line that divides different categories in your data - this line is called the decision boundary. The decision boundary should maximize the margin, which is the distance between the line and the nearest data points of any class. By maximizing this margin, the SVM will find the best possible separation between classes. The support vectors are the specific points that are the closest to this line and will therefore be the points that decide how the line is positioned to find the best possible separation.

As already denoted in table 2.1 in the past chapter, SVM suffers from high training time, since finding the line that maximizes the margin between classes is a complex problem[45].

Decision tree

The decision tree algorithm breaks the dataset down into smaller and smaller subsets based on different attributes - similar to a flow chart where the branches are the decisions made based on the features represented by the nodes. The tree starts with a root node representing the entire dataset. The model will examine all features and selects the one that best splits the dataset into two subsets that are as homogeneous as possible with respect to the target variable. The dataset will be split into smaller subset and this continues recursively until it is stopped or it reaches a point where its confident enough to make a final decision or prediction[46].

Decision trees are advantageous for their ease of interpretation as they provide clear visualization of the decision-making process, which can help with understanding the feature importance in network security. They are capable of handling both numerical and categorical data and unlike SVM or KNN, decision trees do not require defining a hyperplane or calculating distance metrics, which can become an obstacle for the computational resources in high-dimensional spaces, such as in network traffic data[47].

Random forest

Random forest is called an ensemble learning method - meaning that it aggregates the predictions from multiple models, specifically multiple decision trees. It builds multiple decision trees and merges them together during training - it will make a final prediction and for classification, it will output the most frequently predicted class by the individual trees. Specifically for random forest, the ensembling technique is called bagging, where the focus is on building new models independently of each other and averaging the ensemble of the independent models[48].

As an ensemble of decision trees, random forest mitigates the risk of overfitting associated with individual decision trees, making it robust across various datasets. It can provide a less biased performance through averaging multiple trees, reducing the variance of predictions. Each tree is built on a random subset of features, making the model more diverse and less likely to bias towards specific features. However, the process of building multiple trees is more costly than building just one and therefore random forest will often have a higher training time than decision tree[49].

XGBoost

XGBoost stands for eXtreme Gradient Boosting and similar to random forest, XGBoost is also bases its predictions on an ensemble consisting of multiple trees. The difference lies in that XGBoost iteratively builds the trees for correcting the errors that the previous tree made in its predictions. For XGBoost, the ensembling technique is called boosting and

focuses on building the new models sequentially of each other to give them the chance to correct mistakes iteratively, as opposed to the parallel nature of random forest.

XGBoost can use multiple CPU cores at the same time, which will speed up the training process. It can also be tailored to specific problems since the parameters allows for the user to define the loss functions and evaluation metrics. In addition to this, it reduces overfitting due to its built-in regularization techniques[50].

Logistic regression

Logistic regression is another classification algorithm used mostly for binary classification. The algorithm will assign weights to each feature according to how dominant they are in making the prediction - to do this, a sigmoid function is used to combine the features and their weights and deliver a value between 0 and 1. This value will denote the probability of what class a data instance should belong to and the prediction will be made based on a threshold.

Logistic regression is easy to implement and understand through the statistical coefficients that decides the predictions. It does not require a lot of computational resources, however it also assumes independence between the features which is often not true, similarly to Naive Bayes. In addition to this, it is mostly efficient for binary classifications and since it assumes a linear relationship between features it may not capture any complex relationships represented in the data[51].

Comparative analysis

For the selection of appropriate algorithms for the BoT-IoT dataset, a comparative analysis was conducted to evaluate the strengths and weaknesses of each algorithm. The objective was to evaluate multiple models for the purpose of detecting botnet traffic and therefore no specific number of models were to be discarded before moving on to the experiments. The goal was instead to exclude any of the previously mentioned models that could preemptively be proven to not be suitable for the nature of this project. For instance, the characteristics of the BoT-IoT dataset — large scale, many features, and severe class imbalance — require models that can efficiently process vast amounts of data with high accuracy and reasonable computational cost, which some of the models may not be suitable for [52, 53]. In addition to this, an IoT network usually consists of devices with limited computing resources; because of these considerations, we are prioritising machine learning models that can do efficient classification on large datasets with a viable training time. The following considerations were made regarding possible disadvantages of the previously mentioned models as presented by the literature review:

- **Computational efficiency:** Both SVM and KNN are computationally demanding with large datasets. SVM requires extensive grid searching for optimal hyperpa-

rameters, and KNN suffers from having to compute the distance to every single data point in the dataset, which is impractical with over 72 million traffic instances[54, 55].

- **Scalability and real-time applicability:** Given the need for real-time analysis in network security, algorithms that can be easily scaled and parallelized across multiple processors are preferred. For instance, XGBoost and random forest offer built-in methods for parallelization of the training of multiple trees, whereas KNN and SVM generally do not scale as well with increasing data size[50, 48].

The decision to exclude certain models from the experiments was based on the following strategic considerations: the computational cost and lack of real-time applicability of SVM and KNN are unfitting for this project. For the purpose of realistic results, the size of the dataset is not negotiable and therefore these models were not appropriate. Therefore, we decided to proceed with Naive Bayes, decision tree, random forest, XGBoost, and logistic regression to evaluate their performance. As mentioned previously, XGBoost specifies a custom-defined learning objective. Referring back to chapter 2, Al-Sarem et al. [40] did not specify which learning objective was used for their results, therefore we have chosen to implement logistic regression for the learning objective of XGBoost instead of separately. By integrating logistic regression within the ensemble learning of XGBoost, we allow the logistic regression to serve as a baseline model that will benefit the iterative ensemble technique of XGBoost.

Each of the chosen models allows for both binary and multi-class classification of data. This means that if we want to classify network traffic, we can classify it as either benign or botnet when using binary classification; but if we use multi-class classification, we can classify the network traffic into more than two different classes, for instance DDoS, DoS, keylogging, and benign. This can be useful if we want to predict exactly what type of attack the botnet traffic can be classified as. For this project, we will focus on binary classification, since the imbalance in the dataset that we want to explore is already prevalent within the binary classes.

3.2 Feature selection

As presented in the literature review in the previous chapter, feature selection is a method for improving the performance of a machine learning model. Feature selection is the process of systematically choosing the most relevant subset of the full feature set based on certain criteria, with the goal of finding the most relevant features for predicting the target variable. In other words, we want to determine the most optimal set of features that will allow us to construct a machine learning model for, in this specific case, detecting botnet traffic in an IoT network. The criteria used to choose this feature set can be based on statistics, which are known as filter methods, or it can be based on optimising the performance of the given model at training time, which are known as wrapper methods.

One reason for performing feature selection is to avoid overfitting, which occurs when the model is trained "too well" with the data and therefore may capture any noise in the data instead of learning the actual patterns we are looking for to classify the data correctly. If this happens, the model may not make the right predictions when introduced to new data. Feature selection techniques can mitigate overfitting by reducing the complexity of the model and focusing on the most informative features[56]. In addition to this, a smaller feature set may reduce the training time. Since the BoT-IoT dataset has more than 30 features, we assume that some features may be more informative than others and that it should be possible to reduce the dimensionality of the input space. Also, the model will require less training time and computational resources, which will be beneficial since the BoT-IoT dataset is large.

Filter methods

Filter methods for feature selection are usually the computationally cheapest option, since the features are evaluated based on statistical measures to determine their relevance. This means that the features are evaluated independently of the machine learning model we are planning to use. This method is generally used to perform an initial "filtering" of the features in the dataset before training the model[56].

Referring back to section 2.3, one of the most widely used methods for feature selection when using filter methods is Pearson's correlation coefficient. It measures the linear correlation between two sets of data—where a set of data could be one feature in a dataset, so this would be the linear correlation for each pair of features in a dataset. If the correlation between two features is perfectly linear, this means that if one variable changes, the other variable will also change by a constant amount, and the relationship between the two variables, or features, will therefore appear as a straight line if plotted on a graph. The Pearson correlation coefficient ranges from 1 to -1, where 1 represents a perfectly linear relationship between two variables, and 0 means that there is no linear relationship and therefore no correlation between the two variables. A value of -1 indicates a perfect negative linear correlation, meaning that if one variable increases, the other will decrease proportionally.

A reason for excluding features with too much linear correlation is that it may be difficult for the model to distinguish the individual effect of each feature, thus not properly recognising the patterns in the data if the features are too highly correlated. In addition to this, features that have a high linear correlation with other features may provide mostly redundant information that does not improve the training of the model. By removing the redundant feature, it will both decrease the training time of the model and minimise the risk of overfitting the model to any redundant data represented in the highly correlated features[56].

We will proceed with Pearson's correlation since it is widely used in the state of the art literary works, as seen in the previous chapter. In addition to this, we want to measure the correlation between the features and the target variable, similar to Venu, Kumar and Rao[34]. This can also be done with Pearson's - we would just have to measure the correlation between each feature and the target variable instead of just measuring the correlations between each pair of features. However, another filter method for feature selection that was represented in the literary works in the last chapter is ANOVA. ANOVA stands for Analysis of Variance and the purpose is not to identify any linear correlations in the feature set, but differs from Pearson's in that it calculates the correlation between the features and the target variable based on how the feature values are represented for both groups within the target variable (the groups would in this case be attack and normal traffic). It does this by assessing if the variance between normal and attack traffic can be explained by the variance between the values of the features in the dataset[57].

Since the redundancy in the dataset that can be expressed with linear correlation is already handled with Pearson's, we found it more insightful to use ANOVA for measuring how the values of the target variable are represented across the feature set. This can capture more complex patterns that are not linear in categorical target variables while portraying how different ranges of feature values impact the final prediction, providing a deeper understanding of the relationship between the feature values and the target variable.

Wrapper methods

As mentioned previously, wrapper methods actually take the performance of the given machine learning model into account in order to decide which features are the most informative and relevant. A search algorithm will be used for exploring the different feature combinations for finding the subset that will optimize the model at training time, which will make it more computationally expensive than the filter methods that simply calculates a score and where the selected feature set will be based on the threshold as set by the individual interpreting the results.

Feature selection using wrapper methods can be done by exploring all possible subsets of features with the model, but also by starting with an empty set of features and then iteratively adding the feature that makes the best contribution to model performance – this is called forward feature selection. We can also do backward feature selection, in which we include all features in the model from the beginning and then iteratively remove the feature that makes the smallest decrease to the model performance. Both of these methods will iteratively run until a specified number of features is met or there is no significant enhancement to the model. Exhaustive feature selection will evaluate all possible feature subsets, making it very computationally costly but very accurate in finding the most op-

timal subset of features for the model[56]. Therefore, we will be using forward feature selection and backward feature selection.

3.3 BoT-IoT dataset

TThe BoT-IoT dataset was created by Koroniotis et al.[1] in the Cyber Range Lab at UNSW Canberra with multiple virtual machines (VMs) for producing both benign and botnet traffic, as well as simulated IoT sensors. These were simulated using the Node-red tool, which is installed on Ubuntu VMs and creates JavaScript code for mimicking IoT device and sensor behaviour. The resulting simulated IoT devices consisted of a weather station, a smart fridge, motion-activated lights, a thermostat, and a garage door, generating normal IoT traffic. Ostinato, a traffic generator tool, was used for generating additional normal traffic within the network. The attacks were performed by four Kali Linux machines, representing bots launching different types of network attacks[1].

The features were extracted with Argus and new, additional features were generated. These were created to capture additional patterns over time within the network traffic—each additionally created feature is based on analysing and measuring different aspects of the traffic within a window consisting of 100 connections of the collected data points. These additional features are only included in the 5% sample of the dataset. The features and their descriptions are denoted in the table below.

As described in the table, *mean*, *stddev*, *sum*, *min* and *max* are features indicating attributes about aggregated records. Koroniotis et al.[1] do not provide further explanation on what the term "aggregated records" represent, but after consulting the Argus documentation we found that the aggregation of records can be based on different criteria, such as the protocol and/or the destination port, in order to generate statistics based on the chosen criteria, such as mean duration and standard deviation for the records with a specific protocol and state to generate data that is useful for detecting specific attacks[58].

Extracted features	Description	Example values
pkSeqID	The unique identifier assigned to each record in the traffic sequence	Unique int from 1, 2, 3, ..., 73370443
stime	The start time of the record measured in Unix time	float
flgs	Indicates various properties and attributes of the network flow states	e: Ethernet encapsulated flow s: Src loss/ retransmissions U: unknown IP options set
flgs_number	Numerical representation of flgs	1: e, 6: eU, 2: e s,
proto	The transaction protocol	udp, tcp, arp
proto_number	Numerical representation of proto	1: tcp, 2: arp, 3: udp
saddr	Source IP address	192.168.100.148
sport	Source port number	int
daddr	Destination IP address	192.168.100.6
dport	Destination port number	int
pkts	Number of packets in the transaction	int
bytes	Number of bytes in the transaction	int
state	Reports the basic state of the transaction, depending on the protocol	RST: reset CON: connected INT: initial
state_number	Numerical representation of state	1: RST, 2: CON, 4: INT
ltime	The end time of the record measured in Unix time	float
seq	Another unique identifier assigned to each record in the traffic sequence	Unique int from 1, 2, 3, ..., 73370443
dur	Total duration of the record	float
mean	Indicates the average, or typical, duration of aggregated records	float
stddev	Measure how the aggregated records are spread aroung the mean – a higher deviation will indicate greater variability in the durations of the records	int
sum	Represents the total combined durations of the aggregated records	float
min	The shortest duration among the aggregated records	float
max	The longest duration among the aggregated records	float
spkts	Indicates the number of packets sent from the source to the destination	int
dpkts	Indicates the number of packets sent from the destination to the source	int
sbytes	Indicates the number of bytes transmitted from the source to the destination in the transaction	int
dbytes	Indicates the number of bytes transmitted from the destination to the source in the transaction	int
rate	Packets per second in the transaction	float
srate	Packets per second in the transcation specifically from the source	float
drate	Packets per second in the transcation specifically from the destination	float
attack	Indicates whether the record is normal traffic or attack traffic	0: normal traffic 1: attack traffic
category	Category of traffic	DoS, Theft
subcategory	Subcategory of traffic	TCP, UDP, Keylogging

Figure 3.1: Descriptions of the extracted features in the BoT-IoT dataset [1]

Generated features	Description	Example values
TnBPSrcIP	Total number of bytes per source IP	int
TnBPDstIP	Total number of bytes per destination IP	int
TnP_PSrcIP	Total number of packets per source IP	int
TnP_PDstIP	Total number of packets per destination IP	int
TnP_PerProto	Total number of packets per protocol	int
TnP_Per_Dport	Total number of packets per destination port	int
AR_P_Proto_P_SrcIP	Average number of packets transmitted per time unit for each possible combination of protocol and source IP, where the time unit is calculated by dividing the total number of packets transmitted for each combination by the total duration of which they were sent	float
AR_P_Proto_P_DstIP	Average number of packets transmitted per time unit for each combination of protocol and destination IP	float
N_IN_Conn_P_DstIP	Number of inbound connections per source IP	int
N_IN_Conn_P_SrcIP	Number of inbound connections per destination IP	int
AR_P_Proto_P_Sport	Average number of packets transmitted per time unit for each combination of protocol and source port	float
AR_P_Proto_P_Dport	Average number of packets transmitted per time unit for each combination of protocol and destination port	float
Pkts_P_State_P_Protocol_P_DestIP	Packet count based on each combination of state, protocol and destination IP	int
Pkts_P_State_P_Protocol_P_SrcIP	Packet count based on each combination of state, protocol and source IP	int

Figure 3.2: List of generated features in the 5% BoT-IoT dataset [1]

The scenarios that were generated with the virtual machines and simulated IoT devices to generate the traffic in the dataset included probing attacks in the form of scanning, denial of service and distributed denial of service attacks with UDP, TCP, and HTTP, and information theft attacks with either keylogging or data theft. In addition to this, benign traffic was generated. In total, benign traffic constituted 9543 of the instances in the full dataset, while the attack traffic constituted more than 73.3 million instances. Out of the attack traffic, the majority consisted of DoS traffic with more than 71.5 million instances, while the probing attacks contributed more than 1.8 million instances, and the information theft constituted 1587 instances. This obviously makes for a highly imbalanced dataset, whether the goal is binary classification with the benign/attack feature or multiclass classification with the specific types of attacks. As mentioned above, the dataset includes an already extracted 5% sample of the entire dataset. This sample is the first dataset that we will be using for this project.

Data preprocessing

To prepare the 5% BoT-IoT dataset for effective machine learning analysis, several preprocessing steps were undertaken. These steps were crucial in ensuring that the data

was suitable for training machine learning models to classify network traffic behaviours accurately.

The primary objective of using the BoT-IoT dataset was to develop a predictive model capable of identifying different types of network traffic, specifically distinguishing between normal and attack traffic. This setup defines a classification problem where the type of traffic is the target variable that we want to predict. This is demonstrated as a binary classification problem, where the target variable is categorised into two classes, attack traffic and normal traffic. This can also be represented with a binary encoding of 0 and 1, where 0 represents normal traffic as the Negative class and 1 represents attack traffic as the Positive class.

The dataset was divided into training and validation sets to evaluate the model's performance accurately. The splitting was conducted with an 80:20 ratio, where 80% of the data was used for training the models, and 20% was reserved for testing. This division helps in ensuring that the model has a substantial amount of data for learning while also retaining a separate subset for prediction.

In addition to this, the dataset was encoded into numerical values instead of strings to process and interpret the data as the same data type.

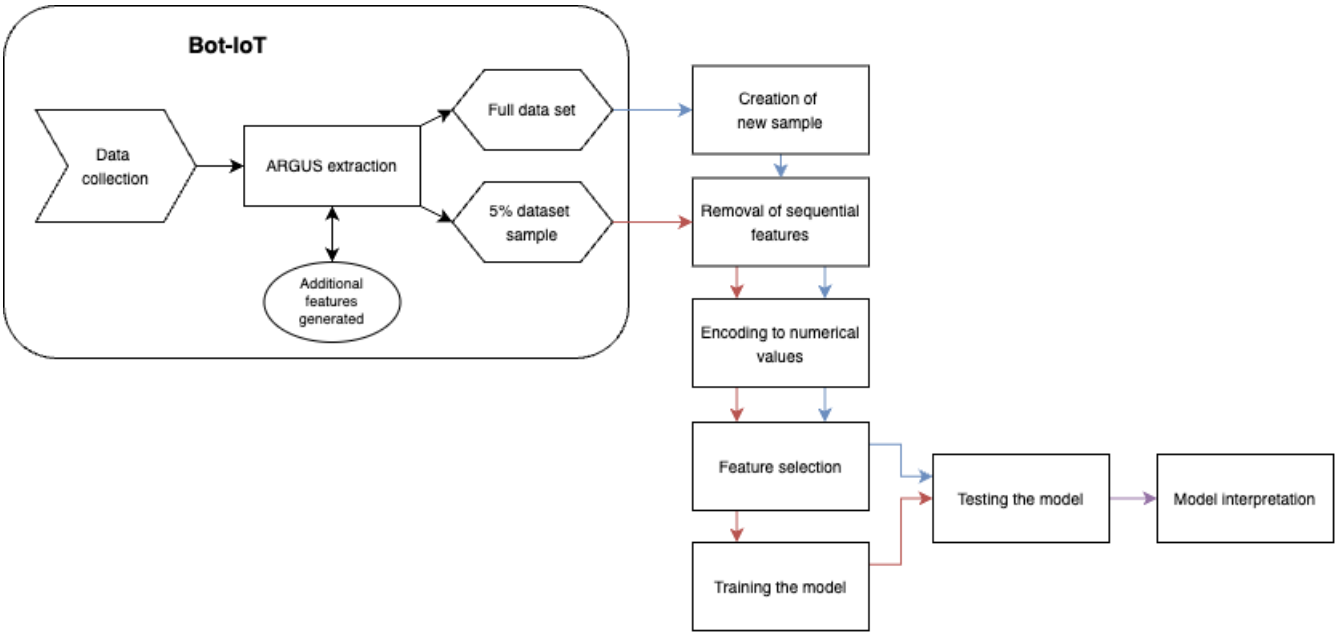


Figure 3.3: Workflow diagram visualizing the steps from data collection to our final interpretation

In table 3.5, the complete process of this project is shown. The creation of the BoT-IoT dataset is shown, including data collection as well as the usage of ARGUS to extract the features and generate additional features, which is then composed into the full dataset consisting of all the generated network traffic instances, as well as the smaller version of

the dataset, as generated by the original creators, consisting of 5% of the dataset with some extra generated features. For the full dataset, we create a new sample that is slightly larger than the 5% dataset with a larger proportion of normal traffic to attack traffic. Both of these are encoded into numerical values for more seamless processing and go through multiple rounds of feature selection. The models are trained on 80% of the 5% dataset, then tested on the remaining 20% of the set, and the trained model is then tested additionally on the new sample of the full dataset that was generated for the sake of additional validation in this project. Lastly, the results will be interpreted.

Dataset samples for this project

As mentioned above, a new sample was created for validation of the trained models. The original 5% dataset as sampled by Koroniotis et al.[1] will be denoted *D1*, and the second dataset that we sampled from the full BoT-IoT set will be denoted *D2* in the rest of the report. D1 contains 477 instances of normal traffic and more than 3.6 million instances of total traffic, making the proportion of normal traffic 0.01% of the total traffic. For D2, we have 7003 normal traffic instances and 4 million traffic instances in total, making the proportion of normal traffic 0.18% of the total traffic.

	D1	**D2**
Normal traffic	477	7003
Attack traffic	3668045	3992997
Total traffic	3668522	4000000

Figure 3.4: Proportions of the types of traffic classifications in the datasets

The reason for creating a new sample of the full dataset is that D1 has an even lower amount of normal traffic, leaving us with queries about how well the model is learning and recognising the patterns of the data and whether it will be able to subsequently identify botnet traffic when introduced to new traffic instances. The aspect of introducing new data to a model trained for botnet detection was not represented in any of the literary works in section 2.3, and that is why we want to explore the introduction of new data.

In a real-life scenario, a trained model would not be of much use if it were only able to predict the standard 20% testing data of the original dataset that it was also trained on. In addition to this, due to D1 being so imbalanced, the accuracy score can be very high since the majority class is so dominant compared to the minority class. The skewed nature of a dataset like BoT-IoT can lead to a phenomenon known as the "majority class trap", where the majority class, which in this case is the attack traffic, is so abundant that it overwhelms the model's learning process. This leads to the evaluation of model performance using metrics such as overall accuracy becoming misleading in the context of class imbalance. A high overall accuracy can mask the model's inability to generalise to unseen benign data points. Therefore, a more nuanced evaluation approach is crucial, and this is why

we have chosen to also focus on F1-score and the confusion matrix to provide a more comprehensive picture of the model's performance across both classes.

Evaluation metrics

Before moving on to the experiments, we will take a look at metrics for the evaluation of the machine learning models in order to ensure an informed choice of the evaluation metrics that we choose for measuring the performance of our own test cases.

The **confusion matrix** is a table showing how a model's predictions compare to the actual classes, illustrating where it got confused. Visualised in the confusion matrix is the **True Positive rate** (TP) and **True Negative rate** (TN) denoting how many instances of each class were correctly predicted along with the **False Negative rate** (FN) and the **False Positive rate** (FP), representing the proportion of the instances that were incorrectly classified as belonging to the wrong class. If a Positive instance were incorrectly classified, this would be a False Negative, since it would falsely be predicted as being Negative.

True Negative	False Positives
False Negatives	True Positive

Figure 3.5: Confusion matrix

The table above visualises the confusion matrix and how exactly it presents the summary of the predictions. For this project, the Positive class is the attack traffic class and the Negative class is the normal traffic class. So when we refer to TP, this will mean correctly classified attack traffic, visualised in the bottom right corner of the confusion matrix, and TN will mean correctly classified normal traffic, visualised in the upper left corner of the confusion matrix.

Classification accuracy Accuracy measures the amount of correct predictions to the total amount of predictions that the model has made. Therefore, in a very imbalanced dataset such as BoT-IoT, the accuracy can be high even though there is a low classification rate for just the minority class. This can be expressed as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.1}$$

Precision measures the proportion of how many of the positive predictions are true positives—meaning how many of the traffic instances that were classified as attack traffic were actually attack traffic. This is calculated as:

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

Recall measures the proportion of true positive instances among all positive instances—meaning how many of the attack traffic instances were actually predicted correctly. This can be calculated as:

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

F1-score combines the recall and precision scores and measures how many correct predictions were made in total. When a dataset is imbalanced, accuracy is not always the best performance metric to use as mentioned above. Precision describes how good the model is at making accurate attack traffic predictions, while recall describes how good the model is in general at making accurate predictions for both types of traffic instances. Aiming to increase precision, thus the number of correct attack predictions, may lead to low recall since the model will become more conservative and only predict an instance as attack traffic if it is very confident—this will lead to some false negatives. On the other hand, if we want to increase recall, making as many attack predictions as possible, it will be less conservative to ensure that it will not miss any attack traffic instances, which will lead to some false positives.

The F1-score computes the harmonic mean of recall and precision. The harmonic mean places more weight on the lower of the two values in order to also capture the changes made in the minority class. This can be expressed as:

$$F1 - score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3.4}$$

The F1-score will primarily be used for evaluating the performance of the models in the next section, as it will capture the balance between precision and recall. In addition to this, the confusion matrix will be used for gaining insights into the predictions of the models[59].

Chapter 4

Experiments

For the experimental part of this project, the goal is to evaluate the different machine learning algorithms that were chosen in the last chapter due to their prevalence in the state of the art as well as their suitability for a large dataset like BoT-IoT, and to evaluate their performance metrics in order to make informed choices on using machine learning for botnet detection.

4.1 Model parameters

For finding an optimal solution for using machine learning to identify botnet traffic, we decided to focus on four different machine learning models: decision tree, Naive Bayes, random forest, and XGBoost, specifically with logistic regression as the learning task, since these were all represented in section 2.3 with good performance evaluations.

For the initial experiments, the *attack*, *category*, and *subcategory* features are dropped, as these are the features that denote whether we are dealing with normal or botnet traffic, which is what we want the models to predict.

Attack denotes whether the traffic is attack traffic or not (1: attack, 0: normal), *category* further categorises the traffic into specific attacks (DoS, DDoS, Reconnaissance, Theft), and *subcategory* categorises the specific attacks into even more specifics—for example, DoS can be subcategorised as HTTP, UDP, or TCP. Each of these features is an evident choice for a target variable depending on what the goal is for the model to predict, but we chose to focus on binary classification, where *attack* is the only binary variable. However, as explained previously, *category* and *subcategory* would be appropriate for multi-class classification. Therefore, the target variable for all experiments in this report is *attack*.

For each of the models, the following parameters were selected as chosen by a grid search:

Model	Python implementation	Parameter configuration	Description
Naïve Bayes	GaussianNB	var_smoothing=1	A value based on the largest feature vvariance, that is added to features to ensure it is never zero or excessively small to “smooth” the variance
Decision Tree	DecisionTreeClassifier	min_samples_leaf=5	Controls the minimum number of samples required to be at a leaf node for it to split further
		class_weight=”balanced”	Address class imbalance issues, so the minority class will have more importance during training
		min_samples_split=10	The minimum number of samples required to split an internal node
		max_depth=40	Sets the limit for the maximum depth level that the tree can grow to
Random Forest	RandomForestClassifier	n_estimators=400	The number of individual estimators that will be included in the ensemble
		min_samples_leaf=5	Controls the minimum number of samples required to be at a leaf node for it to split further
		class_weight=”balanced”	Address class imbalance issues, so the minority class will have more importance during training
		min_samples_split=10	The minimum number of samples required to split an internal node
		max_depth=40	Sets the limit for the maximum depth level that the tree can grow to
XGBoost	XGBoostClassifier	objective="binary:logistic"	Indicates that the model is trained for binary classification using logistic regression as the base learner
		eval_metric="logloss"	The model's performance will be evaluated based on the logarithmic loss
		n_estimators=400	The number of individual estimators that will be included in the ensemble
		max_depth=40	Sets the limit for the maximum depth level that the tree can grow to
		scale_pos_weight=1	Multiplies the weight of the minority class samples by this value
		gamma=0.1	Requires a minimum reduction in the loss function to split a node further
		learning_rate=0.1	The size of the steps taken when iteratively minimizing the loss function
		reg_alpha=0.01	Adds L1 regularization by penalizing large coefficients
		reg_lamba=0.01	Adds L2 regularization by penalizing large weights

Figure 4.1: The parameters that were chosen for the Python implementations of the models [2][3]

For the Naive Bayes, decision tree and random forest, the classifiers from the scikit-learn library was used and the XGBoostClassifier from the XGBoost library was used. This chapter will elaborate on the mathematical foundations of the models, in order to elaborate on and clarify the model descriptions in the last chapter.

Gaussian Naive Bayes

For Naive Bayes, the Gaussian Naive Bayes classifier was used. This is a variant of the Naive Bayes classifier specifically designed for classifying continuous features, and it uses a Gaussian distribution to model the likelihood of the features. This means that we assume each feature follows a normal distribution within each class. This was chosen over Bernoulli and multinomial Naive Bayes because Bernoulli assumes that all features are binary and would therefore not be able to handle the variance of the feature values, but only 1 and 0. For multinomial Naive Bayes, the feature values typically represent the frequency of the given feature. Since the features we are dealing with for traffic classification will mostly be represented as numerical values that can be any real number within a range, and therefore are continuous variables, we will be using Gaussian Naive Bayes.

Bayes' theorem is used to combine the likelihoods of observing specific feature values given each class:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)} \tag{4.1}$$

P(y | X) is the posterior probability of the target variable y given the feature set X, which is the measurement of how likely it is for a class label to be true given the observed features. *P(y)* is the prior probability of the target variable y, which is how likely it is for a class label to be true in the absence of any information about the features. This is where an imbalanced dataset might make an impression, since this prior probability can represent the proportion of the high amount of attack traffic with a high number, compared to the low amount of normal traffic where the prior probability of this class would have a much lower number. *P(X | y)* is the likelihood of observing any feature in X given the target variable y, while *P(X)* is the probability of observing a feature in X without considering y.

Since Gaussian Naive Bayes is based on the normal distribution, this means that for each feature X, the likelihood of observing feature X given the target variable y is calculated using the mean and the standard deviation for each class for each feature. This means that we would calculate the mean and standard deviation of both the attack traffic class and the normal traffic class for each feature. After this, the likelihoods of all features for each class are multiplied. The joint likelihoods for each class are then multiplied by the prior probability, which gives us the posterior probability for each class. The class with the highest posterior probability is the class to which the instance should be assigned[42].

The *var_smoothing* parameter calculates the variance in each feature in the dataset, representing the spread of the feature values around the mean. For the largest variance, this will be multiplied by the specified value, which in this case is 1. The largest feature variance multiplied will then be added to all features in the dataset to smooth this variance and ensure that no features have too small a variance, providing stability to the calculations[3].

Decision tree

The decision tree classifier does not have a formula similar to Naive Bayes, as it is based on binary decisions that partition the feature set X into subsets, with the aim of these subsets having similar values of the target variable y. This process continues until a set criterion for stopping is met, which is determined by the parameters *max_depth*, *min_samples_splits* and *min_samples_leaf*.

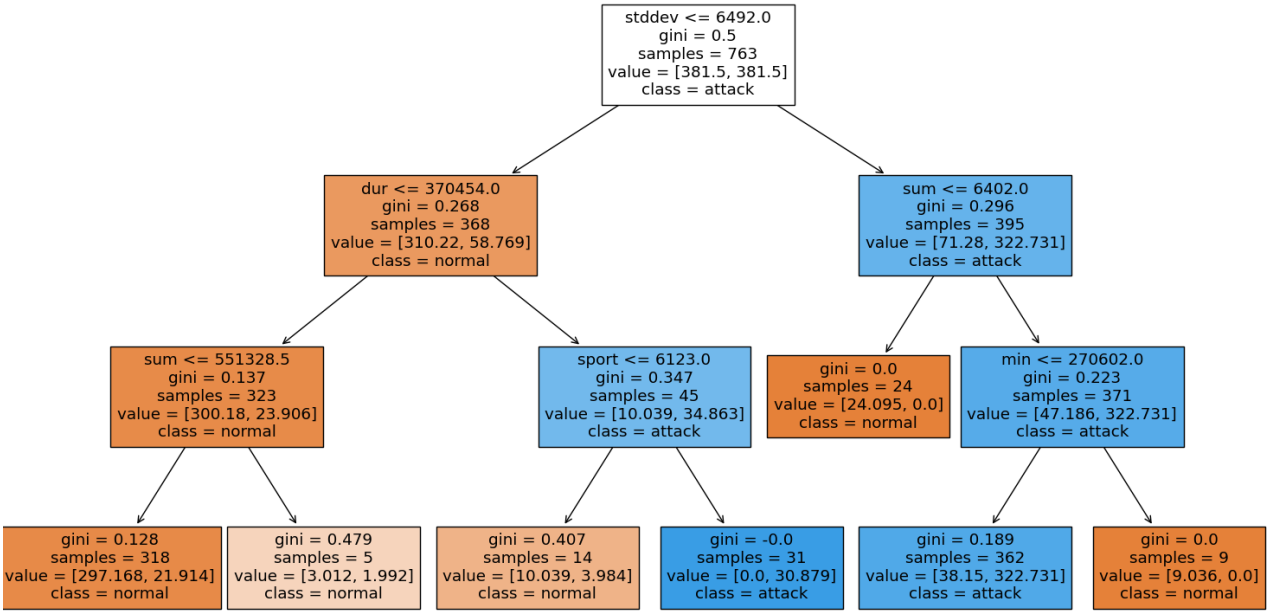


Figure 4.2: Decision tree visualized

The maximum depth, represented in scikit-learn with *max_depth*, sets a criterion that the tree can reach a specific maximum depth. For our test cases, this maximum depth is 40. This means that there can be a maximum of 40 levels of decision nodes descending from the root node. The decision tree visualised in figure 4.2 has a *max_depth* of 3[5].

The minimum number of samples required to split an internal node, represented by *min_samples_split*, is set to 10 for our test cases. This means that if the subset of dataset samples at an internal node is less than 10, it cannot be split further. The minimum number of dataset samples required to be at a leaf node, represented by *min_samples_leaf*, is set to 5. This means that the leaf nodes, which are the endpoints of the tree where the prediction is made, need to have at least 5 samples to base the decision on. If a split will cause any of the resulting leaf nodes to have fewer than 5 samples, the split will not be considered.

The *class_weight* parameter balances the classes by penalising a class with its weight compared to the other class. The weights for the classes are computed as

$$w_i = \frac{n}{kn_i} \tag{4.2}$$

where w_i denotes the weight for class i, n is the total number of dataset samples, k is the amount of classes - so for binary classification, that would be 2 and n_i is the number of dataset samples specifically belonging to class i[3, 46].

Random forest

For random forest, which consists of multiple decision trees, the explanations for *max_depth*, *min_samples_split*, *min_samples_leaf*, and *class_weight* are the same for each tree in the "forest". The number of trees that the random forest will consist of is represented by *n_estimators*—for our test cases, this is set to 400. Each of these estimators will be individually trained on a randomly sampled subset of the dataset during the training of the model. These subsets are not mutually exclusive; each record in the dataset can appear in multiple samples or not at all. When all 400 trees are trained on randomized samples, the random forest model will combine their predictions to make one final prediction. The final class prediction for a given record in the dataset is determined by considering the prediction of each tree as a vote for either normal or attack traffic. The traffic record will then be classified as the class label that has the highest number of "votes".

If we refer back to the visualised tree in 4.2, we can see that, according to one of the leaf nodes, the decision tree has predicted that traffic records with a feature value of 21.914 for *sum* belong to the class label 'normal traffic'. If we had a random forest with three additional estimators that predicted that traffic records with a feature value of 21.914 for *sum* belong to the class label 'attack traffic', then this would be the final prediction due to attack having the highest number of votes[3, 48].

XGBoost

The XGBoost model also consists of multiple trees with the same values for *max_depth*, *min_samples_split*, *min_samples_leaf*, and *n_estimators*. For XGBoost, the estimators are ensembled differently than for random forest. The estimators are built sequentially, with each new tree trained to correct the prediction errors made by any of the previous trees. This is where the *learning_rate* parameter comes into use—it controls the size of the step taken at each iteration in the direction that will minimise the loss function. The loss function quantifies the difference between the predicted values by the model and the true values in the dataset, measuring how well they match. The value of the *learning_rate* denotes the scaled contribution of each tree to the final prediction—meaning that when we set the value to 0.1, the contribution of an estimator will be scaled by 0.1 when it is added to the ensemble.

The XGBoost implementation for our experiments was specified to use logistic regression for binary classification as the objective function, meaning that the loss function used for optimisation when training the trees will be logistic loss. Because of this, XGBoost will adjust the parameters of the decision trees, such as when to split a node, to minimise the divergence between the predicted probabilities and the true binary labels in the training data. When using logistic regression with XGBoost, it creates trees that iteratively learn how to minimise the divergence between the predicted probabilities and the true binary

labels in the training data.

Returning to the loss function, this is defined by the *eval_metric* parameter. When we use logistic loss as the evaluation metric for evaluating the performance of the model, we want a lower value since this will indicate that the predictions are more aligned with the actual labels.

The *objective* parameter specifies how the model is trained. It is for specifying what the model should achieve during training—which is set to *binary:logistic*, since it is trying to predict binary classes and it is using logistic regression to make these predictions. Despite the name, it is in fact used for classification and not regression.

The binary logistic loss function is defined as:

$$\text{Log loss} = \frac{1}{N}\sum_{i=1}^{N} -(y_i \log(p_i) + (1 - y_i)\log(1 - p_i)) \tag{4.3}$$

where N is the total number of samples in the dataset, y_i is the true label of sample i—it will be 1 if the traffic record belongs to the positive class and 0 if it belongs to the negative class—while p_i is the predicted probability that the model has assigned the sample i to the positive class.

$(1 - y_i)$ will therefore be defined based on what the true label of the sample i is, acting as a switch that will turn off the contribution of $\log(1 - p_i)$ if the actual label of the sample is positive. However, if the true label of sample i is negative, $(1 - y_i)$ will be 1 and therefore allow $\log(1 - p_i)$ to contribute to the loss calculation.

This effectively means that the function is split into two terms, where $(y_i \log(p_i))$ calculates the loss associated with predicting the positive class and consists of the product of the actual label of the instance and the natural logarithm of the predicted probability that the instance is assigned to the positive class. The other term, $(1 - y_i)\log(1 - p_i)$, calculates the loss associated with predicting the negative class and consists of the product of the complement of the actual label and the natural logarithm of the complement of the predicted probability that the instance is assigned to the positive class. This is the term that will be switched off if the true label of the instance we are trying to predict is positive.

The *scale_pos_weight* denotes the weight for the positive class, which in this case is the attack traffic. This parameter can be used to deal with class imbalance. For our experiments, the parameter value is kept as the default, as this gave the best results in the grid search. However, the following formula can be used to calculate an appropriate value to combat imbalance between the negative and positive classes in the dataset:

$$scale\_pos\_weight = \frac{n}{p} \tag{4.4}$$

where n is the number of negative samples in the dataset and p is the number of positive samples in the dataset.

The regularisation parameters are represented by *gamma*, *reg_alpha*, and *reg_lambda*, where *gamma* denotes the threshold for the minimum loss reduction required to split a node further. This means that a split will only be made if the potential decrease in loss from splitting a node is equal to or larger than 0.1.

The *reg_alpha* and *reg_lambda* parameters are used for regularisation. Lasso regularisation is represented by *reg_alpha*, where the purpose is to penalise the coefficients of less important features so they will have less effect on the training of the model.

Ridge regularisation is represented by *reg_lambda*, which penalises the magnitude of the feature coefficients. This means that it will focus less on the features that are important, ensuring that the training of the model does not depend too heavily on just a few features[2].

4.2 Experiment setup

The experiments outlined in the subsequent sections were conducted using two different machines, each configured to ensure consistency across the tests while leveraging distinct system architectures. This section details the specifications of each machine.

Machine 1: MacBook Pro

Machine 1 is a MacBook Pro, selected for its robust performance and reliability in handling computational tasks. Below are the detailed specifications:

MacBook Specifications:

- **Processor:** Apple M2 Pro with a 10-core CPU
- **Memory:** 16GB Unified RAM
- **Storage:** 512GB SSD
- **Display:** 14.2-inch Liquid Retina XDR display with a resolution of 3024x1964
- **Graphics and Neural Engine:** 16-core GPU coupled with a 16-core Neural Engine
- **Operating System:** macOS Sonoma 14.0

Python Version: Python 3.10.4, which is the latest stable version compatible with macOS Sonoma as of the latest updates.

Machine 2: Ubuntu Virtual Machine

Machine 2 is a virtual machine hosted on an enterprise-grade server, chosen to evaluate the scalability and performance of the algorithms under a controlled, high-resource environment. The specifications are as follows:

VM Specifications:

- **Processor:** AMD EPYC with Instruction-Based Process Branching (IBPB)
- **CPU Details:** Family 23 model, with 16 physical cores
- **Memory:** 32GB RAM
- **Operating System:** Ubuntu Linux 22.04

Python Version: Python 3.11.0, the most recent release optimized for performance and compatibility with Ubuntu 22.04.

4.3 Feature selection

Koroniotis et al.[1] perform a type of filter feature selection on the dataset themselves by calculating an entropy score for each feature to measure the uniqueness of the information that the feature carries, and a correlation score for each feature to measure how unrelated they are to other features. The goal was to choose the features that have the least amount of redundant information and are minimally correlated with the other features.

Lefoane et al.[38] also state that some features specifically included in the BoT-IoT dataset may be too trivial to contribute significantly to the classification of botnet traffic. These can be features such as packet sequence ID, which is introduced for administrative purposes rather than providing information about the content of the traffic, or source port, which is generated arbitrarily by the network protocol and again does not carry information about the content of the traffic. By eliminating these features from the dataset, we will facilitate the next step for ranking the features that are genuinely relevant to our analysis.

In continuation of this, Lefoane et al.[38] also apply a type of filter method for feature selection by setting a threshold for the frequency of each unique value in a column compared to the total instances in the whole dataset. If any of the unique values in a column exceed this threshold, the column will be dropped as it is deemed redundant and less informative. However, for features such as packet sequence ID, a unique value will never exceed the threshold, as each unique value occurs only once—in this case, they will also need to be considered less informative.

These observations from the reviewed literature may help us in creating our own method for feature selection. As already mentioned above, the *attack*, *category*, and *subcategory* features are dropped from the dataset since they represent the values that we want to predict. We know that we do not want features where excessive uniqueness is added for administrative purposes, but we also do not want features characterised by a lot of redundancy. In addition to this, calculating the correlation between features may aid in figuring out how dependent the features are. If the features are less correlated, it means

they provide independent information, possibly making them more valuable for analysis since they will contain less redundant information.

Other than *pkSeqID*, as established above, some of the features that can be considered as being unique for administrative purposes are *stime*, the start time of the record, and *ltime*, the end time of the record, both represented in Unix time and therefore can be considered another type of sequence number that will be distinctive for each individually initiated network traffic record. Also, *seq* represents the Argus sequence number—Argus is a system for network monitoring that was used for generating the records for the dataset and extracting the information for the features from the original pcap files. The Argus sequence number is a unique identifier generated by Argus to identify each flow in the traffic, similar to *pkSeqID*. In addition to this, three features are represented twice, both textually (*flgs*, *proto*, *state*) and numerically (*flgs_number*, *proto_number*, *state_number*). Therefore, moving forward, we will only be using their numerical representations since the processing of the numerical representation will align with the other features in the dataset that are also numerical.

Selected features	Discarded features
flgs_number	pkSeqID
proto_number	stime
saddr	flgs
sport	proto
daddr	state
dport	seq
pkts	ltime
bytes	attack
state_number	category
dur	subcategory
mean	
stddev	
sum	
min	
max	
spkts	
dpkts	
dbytes	
rate	
srate	
TnBPSrcIP	
TnBPDstIP	
TnP_PSrcIP	
TnP_PDstIP	
TnP_PerProto	
TnP_Per_Dport	
AR_P_Proto_P_SrcIP	
AR_P_Proto_P_DstIP	
N_IN_Conn_P_SrcIP	
N_IN_Conn_P_DstIP	
AR_P_Proto_P_Sport	
AR_P_Proto_P_Dport	
Pkts_P_State_P_Protocol_P_DestIP	
Pkts_P_State_P_Protocol_P_SrcIP	

Figure 4.3: The features that are selected and discarded after the first round of feature selection

In table 4.3, we have presented the features that have so far been excluded from training any model for identifying botnets, as they were deemed too trivial and not informative enough about the traffic instances to provide meaningful contributions. The features that we have selected to move forward with for additional experimenting and further selection are also represented.

4.4 Entropy

For the remaining set of features, we want to calculate an entropy score in order to determine how random the values of the features are. We have already removed the features with excessive uniqueness, but as mentioned above, a feature with too much redundancy will be less informative. The Shannon entropy is calculated for each feature, as seen in the table below. We chose the Shannon entropy since it calculates the randomness of each feature independently of the other features, simplifying the comparison across the different

features. This independence of the calculation is especially beneficial as we assess and refine the feature set as an ongoing process. If we chose to calculate the joint entropy instead, this would measure the randomness while considering the combined outcomes of multiple variables, which may introduce more complexity and noise for the initial interpretation of the feature informativeness. The formula for Shannon entropy is as follows:

$$S(X) = -\sum_{i=1}^{n} P(x_i) \log_2(P(x_i)) \tag{4.5}$$

where *S(x)* is the Shannon entropy, $P(x_i)$ is the probability of a specific feature value i and the sum is taken over all the possible values of that feature[60].

A low entropy suggests that the data is more predictable, meaning there are fewer possible outcomes. A high entropy value indicates that the feature has greater uncertainty, resulting in a larger number of possible outcomes.

Feature	Entropy
Flgs_number	1.1553480730724173
Proto_number	1.0173582167483124
Saddr	2.0322471341245953
Sport	15.932449511584895
Daddr	1.5273824095534954
Dport	0.4544164404411461
Pkts	3.711864744005883
Bytes	4.687531473939456
State_number	1.492003364325404
Dur	18.000645461169633
Mean	16.577583394639973
Stddev	15.597379825428192
Sum	17.469981114744474
Min	8.334065739007425
Max	16.687829817216944
Spkts	3.7144189783925134
Dpkts	1.0620750046750762
Sbytes	4.505539218278007
Dbytes	1.070880118833997
Rate	15.377150936993345
Srate	15.045897508427254
Drate	1.330869794618918
TnBPSrcIP	9.934281852946318
TnBPDstIP	9.78452052117244
TnP_PSrcIP	8.540596282990968
TnP_PDstIP	8.47918664192442
TnP_PerProto	8.40084420328171
TnP_Per_Dport	8.35854842458128
AR_P_Proto_P_SrcIP	15.000969377228296
AR_P_Proto_P_DstIP	14.96551600803349
N_IN_Conn_P_DstIP	2.141347187827017
N_IN_Conn_P_SrcIP	4.1693408717025235
AR_P_Proto_P_Sport	15.45550858697274
AR_P_Proto_P_Dport	14.941668556040453
Pkts_P_State_P_Protocol_P_DestIP	8.496023348051104
Pkts_P_State_P_Protocol_P_SrcIP	8.464850255238538

Figure 4.4: The Shannon entropy scores of the features

Each feature's entropy is calculated by summing all possible values of the feature in the dataset and the probability that a given feature value will occur, and the base-2 logarithm

of this probability. The Shannon entropy uses a base-2 logarithm, and the unit of the calculated entropy is bits. The entropy value denotes the average number of bits required to encode the values of the feature. For example, since the entropy of *stddev* is approximately 15.597, this means that on average we need 15.597 bits to represent each value of that feature in the dataset. This ties in with the concept of randomness, as a higher entropy value represents more randomness and variability in the feature values. Therefore, if we were to encode the feature values into binary, we would need 15.597 bits on average to represent the range of possible feature values.

We will now set a threshold for how much unique information we want the features to provide. We aim to select the features with the most informativeness, and while there are no specific metrics for doing this, by examining the table above, we see a notable leap between the features with the lowest entropy (0-5) and those with a higher entropy (8-18). Therefore, we discard all features with an entropy below 5, leaving us with the following features:

Feature	Entropy
sport	15.932449511584895
dur	18.000645461169633
mean	16.577583394639973
stddev	15.597379825428192
sum	17.469981114744474
min	8.334065739007425
max	16.687829817216944
rate	15.377150936993345
srate	15.045897508427254
TnBPSrcIP	9.934281852946318
TnBPDstIP	9.78452052117244
TnP_PSrcIP	8.540596282990968
TnP_PDstIP	8.47918664192442
TnP_PerProto	8.40084420328171
TnP_Per_Dport	8.35854842458128
AR_P_Proto_P_SrcIP	15.000969377228296
AR_P_Proto_P_DstIP	14.96551600803349
AR_P_Proto_P_Sport	15.45550858697274
AR_P_Proto_P_Dport	14.941668556040453
Pkts_P_State_P_Protocol_P_DestIP	8.496023348051104
Pkts_P_State_P_Protocol_P_SrcIP	8.464850255238538

Figure 4.5: The features chosen based on the entropy threshold

Validation of results

To validate the results of the entropy-based feature selection, we created a new dataset sample (D2) based on the full version of the BoT-IoT dataset. The full version of the dataset is split into 74 files with around 500,000 traffic records in each, and we concatenated 4 of them to create D2. In total, D2 consisted of around 4 million records, of which 7,003 were normal traffic and the rest were attack traffic.

For the initial experiments, we trained and tested decision tree, random forest, XGBoost, and Naive Bayes models on 80% and 20% of D1, respectively. For this, D1 was oversampled, undersampled, and kept in its original imbalanced state. Then, the trained models were saved and tested on D2 to examine how the trained models would react to new data that they were not trained on. The purpose of this was to explore whether overfitting or underfitting is present in the trained models and, if it is, what methods and modifications may minimise it. Overfitting is when the trained model learns the dataset "too well", causing it to consider too much of the noise as a part of the pattern to recognize for the learning task, which will cause a low performance when testing the model with new data. Underfitting is when the model is not trained enough on the patterns of the data and therefore will not recognize the patterns when introduced to new data - also resulting in low performance.[61]

For the full dataset, the generated features as shown in 3.2 were not included and therefore were discarded for the validation, so the trained model would be compatible with the newly created dataset. This limits the scope of the testing results to the shared set of features between D1 and the full set, as shown in 3.1. The features that are valid for further filtering due to being represented in both D1 and D2 and having an entropy exceeding the threshold of 5 are shown in table 4.6.

Feature	Entropy
sport	15.932449511584895
dur	18.000645461169633
mean	16.577583394639973
stddev	15.597379825428192
sum	17.469981114744474
min	8.334065739007425
max	16.687829817216944
rate	15.377150936993345
srate	15.045897508427254

Figure 4.6: The features that are represented in both D1 and D2 and their entropy scores

However, we still assume that the intersection of features between D1 and D2 will give an indication of whether feature selection based on the Shannon entropy is a relevant metric for identifying botnet traffic and that it will provide a relevant perspective on how the trained models perform, since all the generated features are based on the same original set of features.

Since both D1 and D2 are highly imbalanced, there is a possibility of getting a high accuracy score without the normal traffic actually being predicted correctly, as it is the class with notably fewer records. As mentioned previously, we will not only look at the F1-score but also prioritise the confusion matrix and the number of TPs and TNs when checking for satisfactory results, to get a better understanding of the predictions made by the trained model. These metrics will provide insights into the classifier's ability to correctly identify negative and positive instances and thereby help us assess how effective

the trained model is in classifying the different types of traffic and not just how many accurate predictions it makes in total.

As mentioned above, we trained the models with the original, imbalanced version of D1. In addition to this, we trained the models with an undersampled version of D1, where the majority class containing attack traffic was undersampled with RandomUnderSampler. We also trained the models with two different oversampled versions of D1, where the minority class was oversampled with either Synthetic Minority Oversampling Technique (SMOTE) or RandomOverSampler. These were all implemented with the imbalanced-learn library in Python [9]. The purpose of this was to see if oversampling the minority class or undersampling the majority class would affect the predictions, especially when testing the model additionally with D2, which has a larger ratio of normal traffic, the minority class. By balancing the class distribution, the model may be able to learn the characteristics of the minority class better, thus leading to better performance.

The oversampling is done differently between RandomOverSampler and SMOTE, which is why we found it informative to explore both options. RandomOverSampler randomly replicates already existing records that belong to the minority class until the dataset is balanced, while SMOTE generates synthetic examples by choosing one record belonging to the minority class and then selecting the 5 nearest neighbours of that record. The newly generated record in the minority class will then be created with the feature values of these neighbours [62]. RandomUnderSampler works by simply selecting a subset of the records belonging to the majority class, with the same size as the minority class, and then discarding the rest of the majority class [63].

For the test cases in this section, the feature set that the models were trained on was either the full set or a selected subset based on Shannon entropy scores. We calculated the entropy scores based on the imbalanced dataset, and not the undersampled or oversampled datasets, to have more universal entropy scores that are based on the "original" D1 as generated by the creators. As seen in table 4.6, this left us with the following features after applying the previously established entropy threshold of 5: *sport*, *dur*, *mean*, *stddev*, *sum*, *min*, *max*, *rate*, and *srate*.

For all test cases, we will primarily focus on and discuss the performance evaluation of D2 unless otherwise specified, since the addition of the new dataset D2 for testing and additional validation of classical machine learning methods is the primary objective of these experiments.

	Decision Tree	
	Training and testing on D1	Testing the trained model on D2
All features, imbalanced	F1 score: 0.9999851771701787 Confusion Matrix: [[105 3] [8 733589]]	F1 score: 0.9750961704686825 Confusion Matrix: [[867 6136] [175295 3817702]]
All features, oversampling with SMOTE	F1 score: 0.9999952290661707 Confusion Matrix: [[733412 0] [7 733799]]	F1 score: 0.9637418061081674 Confusion Matrix: [[998 6005] [261329 3731668]]
All features, oversampling with RoS	F1 score: 0.9999993184380233 Confusion Matrix: [[734155 0] [1 733062]]	F1 score: 0.9762425121161508 Confusion Matrix: [[980 6023] [166644 3826353]]
All features, undersampling with RuS	F1 score: 0.9842949166632566 Confusion Matrix: [[92 1] [2 96]]	F1 score: 0.3299110395816039 Confusion Matrix: [[6590 413] [3202498 790499]]
Entropy-selected features, imbalanced	F1 score: 0.999876574312932 Confusion Matrix: [[83 19] [84 733519]]	F1 score: 0.8251251716574382 Confusion Matrix: [[3629 3374] [1177982 2815015]]
Entropy-selected features, SMOTE	F1 score: 0.9930239754685927 Confusion Matrix: [[732852 912] [9323 724131]]	F1 score: 0.7809926745400895 Confusion Matrix: [[5932 1071] [1426787 2566210]]
Entropy-selected features, RoS	F1 score: 0.9999611509681299 Confusion Matrix: [[733197 0] [57 733964]]	F1 score: 0.8175824881223365 Confusion Matrix: [[5942 1061] [1223200 2769797]]
Entropy-selected features, RuS	F1 score: 0.9685293660256077 Confusion Matrix: [[98 0] [6 87]]	F1 score: 0.6225019668436076 Confusion Matrix: [[4568 2435] [2182860 1810137]]

Figure 4.7: Decision Tree initial test results with full feature set and entropy-selected features

For the decision tree tests, the full feature set showed high F1-scores for all test cases except the undersampled one. However, the TN scores were very low. In comparison, the test cases with the entropy-selected feature sets showed a significant improvement in the TN scores in D2, especially for both of the oversampled test cases. However, the TP scores were lower, thereby resulting in lower F1-scores for the entropy-selected test cases.

	Random Forest	
	Training and testing on D1	Testing the trained model on D2
All features, imbalanced	F1 score: 0.9999959019138375 Confusion Matrix: [[109 2] [1 733593]]	F1 score: 0.9917834523198005 Confusion Matrix: [[2503 4500] [48153 3944844]]
All features, oversampling with SMOTE	F1 score: 0.9999972737520935 Confusion Matrix: [[733719 1] [3 733495]]	F1 score: 0.9858888322916516 Confusion Matrix: [[1733 5270] [92999 3899998]]
All features, oversampling with RoS	F1 score: 0.9999972737520986 Confusion Matrix: [[733174 0] [4 734040]]	F1 score: 0.9910017238268785 Confusion Matrix: [[3076 3927] [54998 3937999]]
All features, undersampling with RuS	F1 score: 1.0 Confusion Matrix: [[114 0] [0 77]]	F1 score: 0.6454639660109794 Confusion Matrix: [[6994 9] [2085368 1907629]]
Entropy-selected features, imbalanced	F1 score: 0.999875607940438 Confusion Matrix: [[57 21] [86 733541]]	F1 score: 0.9031053462133896 Confusion Matrix: [[1798 5205] [690679 3302318]]
Entropy-selected features, SMOTE	F1 score: 0.9999802347026642 Confusion Matrix: [[733703 10] [19 733486]]	F1 score: 0.8536298491944729 Confusion Matrix: [[5417 1586] [1009482 2983515]]
Entropy-selected features, RoS	F1 score: 0.9999679665879666 Confusion Matrix: [[732970 0] [47 734201]]	F1 score: 0.8765143622769105 Confusion Matrix: [[1914 5089] [864091 3128906]]
Entropy-selected features, RuS	F1 score: 0.9528563086167997 Confusion Matrix: [[94 3] [6 88]]	F1 score: 0.6477936133117894 Confusion Matrix: [[4574 2429] [2073996 1919001]]

Figure 4.8: Random forest initial test results with full feature set and entropy-selected features

For the random forest tests, the entropy-selected test cases had higher F1-scores than the decision tree. For the full set of features, both the TN and TP scores for D2 were higher than those for the decision tree. For both decision tree and random forest, the undersampled test cases showed relatively high TN scores but also the lowest TP scores so far.

	Naïve Bayes	
	Training and testing on tD1	Testing the trained model on D2
All features, imbalanced	F1 score: 0.9882121796323063 Confusion Matrix: [[87 14] [16887 716717]]	F1 score: 0.0019740962267912974 Confusion Matrix: [[6939 64] [3989057 3940]]
All features, oversampling with SMOTE	F1 score: 0.927904398982339 Confusion Matrix: [[697864 36228] [69496 663630]]	F1 score: 0.0020120215089110124 Confusion Matrix: [[6940 63] [3988981 4016]]
All features, oversampling with RoS	F1 score: 0.9299280183960642 Confusion Matrix: [[701879 32213] [70526 662600]]	F1 score: 0.0015773158650389954 Confusion Matrix: [[6947 56] [3989852 3145]]
All features, undersampling with RuS	F1 score: 0.9476439790575916 Confusion Matrix: [[86 5] [5 95]]	F1 score: 0.0022006174048920537 Confusion Matrix: [[6937 66] [3988603 4394]]
Entropy-selected features, imbalanced	F1 score: 0.9973546828099454 Confusion Matrix: [[9 92] [3580 730024]]	F1 score: 0.7636193642604113 Confusion Matrix: [[2920 4083] [1517325 2475672]]
Entropy-selected features, SMOTE	F1 score: 0.7874230737561143 Confusion Matrix: [[603101 130991] [180543 552583]]	F1 score: 0.3961097397711903 Confusion Matrix: [[6741 262] [3004656 988341]]
Entropy-selected features, RoS	F1 score: 0.7878863672769935 Confusion Matrix: [[605040 129052] [181756 551370]]	F1 score: 0.38266876670936556 Confusion Matrix: [[6802 201] [3046160 946837]]
Entropy-selected features, RuS	F1 score: 0.8220871865043942 Confusion Matrix: [[76 15] [19 81]]	F1 score: 0.38326172372355344 Confusion Matrix: [[6805 198] [3044345 948652]]

Figure 4.9: Naive Bayes initial test results with full feature set and entropy-selected features

The F1-scores for the full set with the Naive Bayes test cases were extremely low. This was due to very low TP scores with D2; however, these test cases all had very high TN scores. For the entropy-selected test cases, the TN scores were similarly high except for the imbalanced test case. In addition, the entropy-selected test cases showed a much higher TP score than the full set, although still quite low compared to both the decision tree and random forest.

	XGBoost	
	Training and testing on D1	Testing the trained model on D2
All features, imbalanced	F1 score: 1.0 Confusion Matrix: [[84 0] [0 733621]]	F1 score: 0.23057750386569176 Confusion Matrix: [[6938 65] [3471638 521359]]
All features, oversampling with SMOTE	F1 score: 1.0 Confusion Matrix: [[733070 0] [0 734148]]	F1 score: 0.6888053364989222 Confusion Matrix: [[6743 260] [1889674 2103323]]
All features, oversampling with RoS	F1 score: 1.0 Confusion Matrix: [[733389 0] [0 733829]]	F1 score: 0.6536408880568318 Confusion Matrix: [[6751 252] [2049319 1943678]]
All features, undersampling with RuS	F1 score: 1.0 Confusion Matrix: [[103 0] [0 88]]	F1 score: 6.020962641849415e-06 Confusion Matrix: [[6890 113] [3992997 0]]
Entropy-selected features, imbalanced	F1 score: 0.9999542534053423 Confusion Matrix: [[54 27] [4 733620]]	F1 score: 0.871312537002122 Confusion Matrix: [[2677 4326] [897647 3095350]]
Entropy-selected features, SMOTE	F1 score: 0.9926375079590982 Confusion Matrix: [[732222 674] [10128 724194]]	F1 score: 0.7536781208930275 Confusion Matrix: [[4934 2069] [1570343 2422654]]
Entropy-selected features, RoS	F1 score: 0.9976063382028812 Confusion Matrix: [[733919 0] [3512 729787]]	F1 score: 0.8134641080437001 Confusion Matrix: [[6000 1003] [1246785 2746212]]
Entropy-selected features, RuS	F1 score: 0.9528071220405522 Confusion Matrix: [[95 1] [8 87]]	F1 score: 0.6654655465657539 Confusion Matrix: [[6853 150] [1996630 1996367]]

Figure 4.10: XGBoost initial test results with full feature set and entropy-selected features

The XGBoost test cases showed the most diverse results. When oversampling the entropy-selected set with both RoS and SMOTE, the D2 results showed relatively high TN scores and TP scores similar to the decision tree test cases. When oversampling or undersampling the full set, the performance evaluation on D2 shows very high TN scores, but with low TP scores, where specifically the undersampled test case had a TP score of 0.

We can conclude that feature selection based on the Shannon entropy score of the features does not improve the F1-score for most of the test cases. However, it does improve the number of TP classifications for XGBoost on D2 and the number of TN for Naive Bayes and Decision Tree on D2, which is something we want to explore further. It also seems to prevent the model from overfitting to the majority class in some test cases, specifically for the decision tree as shown in 4.7.

We can also conclude that it seems beneficial to oversample or undersample the imbalanced D1 for some test cases. Specifically for XGBoost, the oversampled test cases balance out the TN and TP scores—still showing signs of overfitting to the minority class, but with a better TN ratio for the entropy-selected test cases. For Naive Bayes, the performance evaluations do not show a notable difference when oversampling or undersampling the dataset. However, for both decision tree and random forest, the TP scores in D2 are the highest when either oversampled or kept in its imbalanced state.

The performances of the decision tree, random forest, and XGBoost are near perfect when training them on D1 with the full feature set. However, the validation test cases with D2 suggest that they are overfitting. Therefore, the near-perfect performance results when testing on D1 are obviously not enough to define them as sufficient for botnet detection when introduced to new data.

On account of this, we will further explore the selection of appropriate features for the detection of botnet traffic in order to improve the performance of the models further.

4.5 Feature correlation

After testing and validating the initial feature selection based on entropy, we will measure whether the dependency between the features can affect the performance of the models. Two types of correlation dependency are measured: Pearson's correlation coefficient, which measures the correlation between the remaining features, and ANOVA, which measures the correlation between the remaining features and the target variable.

For Pearson's, if two features have a strong correlation, the information they provide will be less diverse and carry more redundant information. Therefore, we would preferably discard the features where the Pearson's correlation coefficient is high. For ANOVA, we want a higher score since this will represent larger differences between the features and therefore less redundancy.

Pearson's correlation coefficient

First, we will calculate Pearson's correlation coefficient. This was used in multiple literary works in section 2.3 and provides a way to identify the relationship between the features that is easy to understand and interpret. Pearson's correlation coefficient is calculated by finding the mean of feature X and the mean of feature Y, as well as the covariance between these features. The covariance denotes the relationship between the two features: if it is positive or negative, it will indicate a linear relationship, while a zero covariance will indicate that the relationship between the features is not linear. The mean of the features is then used to calculate the standard deviation σ of each feature, and Pearson's correlation coefficient is then calculated with this formula:[56]

$$P = \frac{cov(X, Y)}{\sigma_X \cdot \sigma_Y} \tag{4.6}$$

In table 4.11, the calculated correlation coefficients are represented between each pair of remaining features after applying the Shannon entropy threshold. In figure 4.12, a heat map generated with Matplotlib[5] and seaborn[6] is shown to visualise the correlations—a darker colour represents a strong negative correlation between two features, while a lighter colour represents a strong positive correlation between two features.

	sport	dur	mean	stddev	sum	min	max	rate	srate
sport	1.0	-0.05833	0.02049	-0.02067	0.0326	0.04275	1.36391e-05	0.034837	0.02055
dur	-0.05833	1.0	-0.15245	0.03927	-0.15304	-0.19579	-0.03048	-0.479484	-0.44508
mean	0.02049	-0.15245	1.0	0.36894	0.57793	0.71455	0.8492	0.52998	0.58766
stddev	-0.02067	0.03927	0.36894	1.0	0.67078	-0.30461	0.69134	0.14109	0.22598
sum	0.0326	-0.15304	0.57793	0.67078	1.0	0.18469	0.66948	0.22441	0.27249
min	0.04275	-0.19579	0.71455	-0.30461	0.18469	1.0	0.38415	0.38122	0.37831
max	1.36391e-05	-0.03048	0.8492	0.69134	0.66948	0.38415	1.0	0.43059	0.50082
rate	0.03483	-0.47948	0.52998	0.14109	0.22441	0.38122	0.43059	1.0	0.85665
srate	0.02055	-0.44508	0.58766	0.22598	0.27249	0.37831	0.50082	0.85665	1.0

Figure 4.11: Table with Pearson's correlation coefficients, showing how much each pair of features are correlated

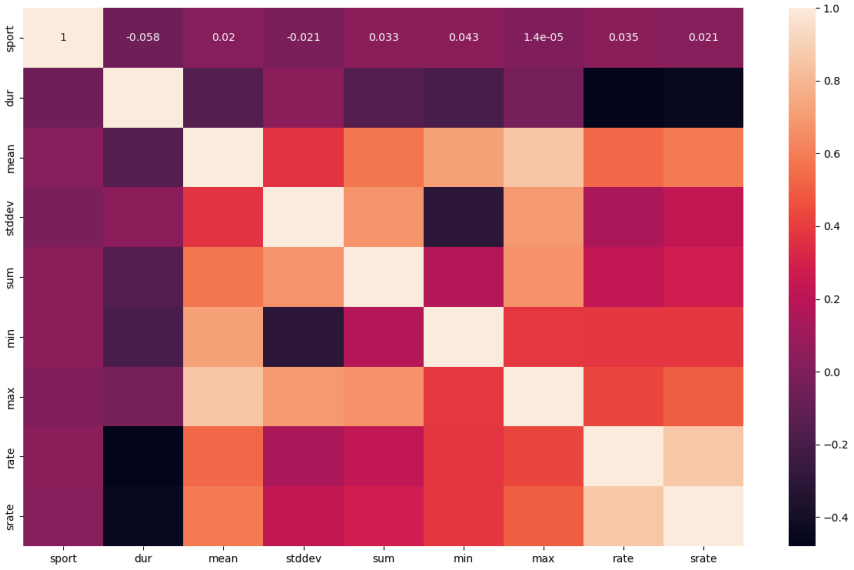


Figure 4.12: Heat map of correlation coefficients

We can define a high correlation as being represented by a positive or negative coefficient closer to 1.0. We can see that a number of the remaining features have a high correlation with another feature. However, as seen in the table, a feature may have a high correlation with one feature but a low correlation with another, so defining a feature and the information it carries as redundant based on its correlation coefficient is a more complex task. Therefore, we would like to get an overview of the average correlation coefficient for each feature in order to identify the features that stand out as being particularly correlated with other features. As mentioned above, the coefficient can be both negative and positive based on whether a positive or negative linear correlation is found between

the features, but a strong correlation will be represented by a number closer to either -1.0 or 1.0. Therefore, we calculate the mean of the absolute values of each feature's correlation coefficients, as shown in the table below.

Feature	Pearson's correlation coefficient
sport	0.13669628085251534
dur	0.2837733656851961
mean	0.5334716442388107
stddev	0.3847469395689922
sum	0.4206053081361638
min	0.3984578207409045
max	0.5062340891019292
rate	0.45314151744752784
srate	0.47639751400194924

Figure 4.13: The mean correlation coefficient for each feature

As seen in table 4.13, the mean of the absolute values of each feature's correlation coefficient ranges roughly between 0.1 to 0.55. While there is no definitive method for setting a threshold for when Pearson's correlation is too high, selecting the five features with the least amount of correlation would effectively establish the upper threshold to be 0.45 and above, where the correlation is considered too high and the features are deemed too redundant. This leaves us with the following five features: *dur*, *stddev*, *sum*, *sport* and *min*.

ANOVA

Analysis of Variance (ANOVA) measures the variability in the features of the dataset by taking the target variable and its categories — in this case, normal and attack — and then calculating the feature value variability within these categories for each feature. This variability is found by calculating the mean value of each feature in each group (the groups being normal traffic and attack traffic in our case). For example, the mean values of both *sport*, *dur*, *stddev*, and all other features would be calculated for both normal traffic and attack traffic. By using the mean value for a feature, ANOVA then calculates both the variability within the groups (e.g., how the values of *dur* vary within the attack traffic group) and the variability between groups (e.g., how the values of *dur* vary between the normal traffic group and the attack traffic group, and whether there is a significant difference in the distribution of *dur* values between the two groups).

The ratio between the within-group variability and the between-group variability is called the F-value. If this value is high, it indicates that the feature is relevant for predicting the target variable since the mean of the feature values differs significantly between the two class groups. This suggests less redundancy in the information provided by the feature across the different categorical groups.[57]

Feature	ANOVA score
sport	9.647656375969683
dur	489.2184927103964
mean	354.3894234831894
stddev	411.88838532046884
sum	481.8250533807435
min	11.733910410556188
max	426.7609547616382
rate	421.64867976751265
srate	161.58170071860138

Figure 4.14: The ANOVA score for each feature

The ANOVA scores range from 9 to 490. Similar to Pearson's, there is no definitive method for setting a threshold for the level of correlation, but opting to select the top five features would establish a threshold of 400 for the ANOVA correlation score. We consider this threshold to be appropriate, since the rest of the features have notably lower correlation scores below 400, resulting in the following five features being selected based on their ANOVA correlation: *dur*, *stddev*, *sum*, *max*, and *rate*.

Filter methods conclusion

To conclude on whether feature dependency affects the performance of the models, both for the training and testing on D1 and for testing the trained model on D2, the four models were trained first with the selected features from the ANOVA correlation score, then with selected features from Pearson's correlation coefficient, and lastly with the combined selected features from both ANOVA and Pearson's. For these experiments, the dataset was again employed in four different states: the imbalanced D1, as well as D1 oversampled with SMOTE, RoS, and undersampled with RuS.

The reason for testing both correlation-based feature sets individually as well as combined is that they measure correlation differently, as mentioned above. Pearson's measures the linear correlation between each pair of features, while ANOVA measures the correlation between each feature and the target variable based on the feature value variability. The two feature sets may prove to be complementary to each other and thereby improve robustness when training a model. Selecting features based on Pearson's correlation coefficient ensures that there are no features with excessive redundancy included, while selecting features based on ANOVA's correlation score ensures that the features have a more informative contribution to the target variable.

The time for calculating the Shannon entropy score, Pearson's coefficient, or the ANOVA score for the selected feature sets was all under 1 second, making all of them feasible and easily attainable methods for measuring the variance and dependency in the feature set, as seen in table 4.15.

Filter method	Calculation time
Shannon entropy	0.895564079284668s
Pearson's correlation	0.40062475204467773s
ANOVA score	0.17047715187072754s

Figure 4.15: The calculation time of the three different methods for filtering the features

This knowledge gives further purpose to using these filter methods as a computationally resourceful approach for restricting the feature set. However, to evaluate the usefulness of the selected feature sets, the model performances will now be studied.

	Decision Tree	
	Training and testing D1	Testing the trained model on D2
ANOVA, imbalanced	F1 score: 0.9861053095176893 Confusion Matrix: [[77 20] [19903 713705]]	F1 score: 0.8438601362435252 Confusion Matrix: [[1723 5280] [1065845 2927152]]
ANOVA, SMOTE	F1 score: 0.986447262075392 Confusion Matrix: [[734280 195] [19686 713057]]	F1 score: 0.8633903070509936 Confusion Matrix: [[4335 2668] [948533 3044464]]
ANOVA, RoS	F1 score: 0.9862885519188552 Confusion Matrix: [[733350 0] [20114 713754]]	F1 score: 0.9628608796713702 Confusion Matrix: [[1467 5536] [268396 3724601]]
ANOVA, RuS	F1 score: 0.9529157630173952 Confusion Matrix: [[89 1] [8 93]]	F1 score: 0.6561909628202977 Confusion Matrix: [[3002 4001] [2036160 1956837]]
Pearson's, imbalanced	F1 score: 0.9997766102062007 Confusion Matrix: [[56 38] [168 733443]]	F1 score: 0.97053851056698 Confusion Matrix: [[108 6895] [209189 3783808]]
Pearson's, SMOTE	F1 score: 0.9928446857200848 Confusion Matrix: [[733319 1053] [9445 723401]]	F1 score: 0.8509911375837702 Confusion Matrix: [[1332 5671] [1022436 2970561]]
Pearson's, RoS	F1 score: 0.9999516091006709 Confusion Matrix: [[733260 0] [71 733887]]	F1 score: 0.9859911743713771 Confusion Matrix: [[78 6925] [90140 3902857]]
Pearson's, RuS	F1 score: 0.9161475174040621 Confusion Matrix: [[92 6] [10 83]]	F1 score: 0.5992803317067447 Confusion Matrix: [[3963 3040] [2279083 1713914]]
ANOVA + Pearson's, imbalanced	F1 score: 0.9998567924670013 Confusion Matrix: [[69 22] [102 733512]]	F1 score: 0.9082319421858358 Confusion Matrix: [[2030 4973] [656493 3336504]]
ANOVA + Pearson's, SMOTE	F1 score: 0.9927064086313477 Confusion Matrix: [[731916 1383] [9318 724601]]	F1 score: 0.9621239724334711 Confusion Matrix: [[2112 4891] [274554 3718443]]
ANOVA + Pearson's, RoS	F1 score: 0.999953653785547 Confusion Matrix: [[733593 0] [68 733557]]	F1 score: 0.8923067239222628 Confusion Matrix: [[1666 5337] [761959 3231038]]
ANOVA + Pearson's, RuS	F1 score: 0.9369648295079087 Confusion Matrix: [[97 2] [10 82]]	F1 score: 0.5768580851159978 Confusion Matrix: [[4600 2403] [2369529 1623468]]

Figure 4.16: Decision Tree results, first trained + tested D1 and then tested on D2

The results showed that 30 out of 36 of the XGBoost, decision tree, and random forest test cases scored an F1-score of at least 0.96 when training and testing the model on the D1 dataset. The test cases for the Decision Tree are shown in figure 4.16. However, many of these models did not achieve similar F1-scores when tested on the D2 dataset. Among those that did, none reached a TN score suggesting that the model effectively identified the feature values characterizing normal traffic. As seen in table 4.16, only a few D2 test cases managed to get a TN score higher than the FP score.

	XGBoost	
	Training and testing on D1	Testing the trained model on D2
ANOVA, imbalanced	F1 score: 0.9998709935776086 Confusion Matrix: [[47 70] [11 733577]]	F1 score: 0.9948111446468023 Confusion Matrix: [[495 6508] [21424 3971573]]
ANOVA, SMOTE	F1 score: 0.9845727639924805 Confusion Matrix: [[733505 460] [22170 711083]]	F1 score: 0.893948067057244 Confusion Matrix: [[4532 2471] [753607 3239390]]
ANOVA, RoS	F1 score: 0.9854223575384129 Confusion Matrix: [[733694 0] [21384 712140]]	F1 score: 0.9615299424655342 Confusion Matrix: [[1533 5470] [278390 3714607]]
ANOVA, RuS	F1 score: 0.9633083637607164 Confusion Matrix: [[99 2] [5 85]]	F1 score: 0.6387326119479346 Confusion Matrix: [[3923 3080] [2113154 1879843]]
Pearson's, imbalanced	F1 score: 0.9998911024153655 Confusion Matrix: [[46 62] [6 733591]]	F1 score: 0.9807094116303392 Confusion Matrix: [[832 6171] [132099 3860898]]
Pearson's, SMOTE	F1 score: 0.9887603170559821 Confusion Matrix: [[731029 2109] [14381 719699]]	F1 score: 0.9156457008146917 Confusion Matrix: [[1110 5893] [605376 3387621]]
Pearson's, RoS	F1 score: 0.997590672759528 Confusion Matrix: [[732556 0] [3535 731127]]	F1 score: 0.8952433914070376 Confusion Matrix: [[1354 5649] [742419 3250578]]
Pearson's, RuS	F1 score: 0.93719344625268 Confusion Matrix: [[88 4] [8 91]]	F1 score: 0.7990684208020126 Confusion Matrix: [[2399 4604] [1325355 2667642]]
ANOVA + Pearson's, imbalanced	F1 score: 0.9999127853721882 Confusion Matrix: [[51 48] [9 733597]]	F1 score: 0.9940740497544868 Confusion Matrix: [[891 6112] [27894 3965103]]
ANOVA + Pearson's, SMOTE	F1 score: 0.9927145316968431 Confusion Matrix: [[732378 813] [9876 724151]]	F1 score: 0.8896275082080193 Confusion Matrix: [[2124 4879] [779838 3213159]]
ANOVA + Pearson's, RoS	F1 score: 0.9975470389712618 Confusion Matrix: [[734184 0] [3599 729435]]	F1 score: 0.9160410209527804 Confusion Matrix: [[2078 4925] [603532 3389465]]
ANOVA + Pearson's, RuS	F1 score: 0.9738435381723446 Confusion Matrix: [[90 0] [5 96]]	F1 score: 0.674391238271809 Confusion Matrix: [[4231 2772] [1954837 2038160]]

Figure 4.17: XGBoost results, first trained + tested D1 and then tested on D2

When training the XGBoost model with features chosen by ANOVA, Pearson's, or both, there was a significant improvement in accuracy scores compared to the initial XGBoost test cases. The test cases where D1 was undersampled showed the lowest accuracy for that model, while both the ANOVA-selected features and the combined ANOVA + Pearson's selected features yielded an F1-score of more than 0.99 when testing the trained model on D2, as shown in figure 4.17. Both of these test cases had some of the highest TP scores in this section, with 3,965,103 TP classifications and 27,894 FN classifications for the ANOVA + Pearson's, and 3,971,573 TP classifications and 21,424 FN classifications for the ANOVA test case on the imbalanced D1.

However, the number of TN classifications was subpar, with 891 TN classifications and 6,112 FP classifications for the ANOVA + Pearson's on the imbalanced D1, and 495 TN classifications and 6,508 FP classifications for the ANOVA on the imbalanced D1.

	Random Forest	
	Training and testing on D1	Testing the trained model on D2
ANOVA, imbalanced	F1 score: 0.9859259450573816 Confusion Matrix: [[82 15] [20164 713444]]	F1 score: 0.9682580173566927 Confusion Matrix: [[1357 5646] [227767 3765230]]
ANOVA, SMOTE	F1 score: 0.9864960034695784 Confusion Matrix: [[732892 129] [19681 714516]]	F1 score: 0.8794983637646442 Confusion Matrix: [[4201 2802] [846913 3146084]]
ANOVA, RoS	F1 score: 0.9862637157426787 Confusion Matrix: [[734463 0] [20150 712605]]	F1 score: 0.9666091072196248 Confusion Matrix: [[1436 5567] [240272 3752725]]
ANOVA, RuS	F1 score: 0.9738162388056782 Confusion Matrix: [[97 2] [3 89]]	F1 score: 0.6070759144755515 Confusion Matrix: [[4543 2460] [2247304 1745693]]
Pearson's, imbalanced	F1 score: 0.9998214960117587 Confusion Matrix: [[54 54] [86 733511]]	F1 score: 0.9967835295759959 Confusion Matrix: [[16 6987] [4780 3988217]]
Pearson's, SMOTE	F1 score: 0.9929953496110602 Confusion Matrix: [[732598 709] [9568 724343]]	F1 score: 0.9342503272414411 Confusion Matrix: [[1132 5871] [476039 3516958]]
Pearson's, RoS	F1 score: 0.9999502459748045 Confusion Matrix: [[733843 0] [73 733302]]	F1 score: 0.9880369288357049 Confusion Matrix: [[297 6706] [74401 3918596]]
Pearson's, RuS	F1 score: 0.9320736586002475 Confusion Matrix: [[77 5] [8 101]]	F1 score: 0.5653548155660262 Confusion Matrix: [[4094 2909] [2414489 1578508]]
ANOVA + Pearson's, imbalanced	F1 score: 0.99986419880937 Confusion Matrix: [[59 35] [74 733537]]	F1 score: 0.9966814537565056 Confusion Matrix: [[678 6325] [7469 3985528]]
ANOVA + Pearson's, SMOTE	F1 score: 0.9930853727174025 Confusion Matrix: [[731737 670] [9475 725336]]	F1 score: 0.91442841804345 Confusion Matrix: [[1623 5380] [614138 3378859]]
ANOVA + Pearson's, RoS	F1 score: 0.9999461566047853 Confusion Matrix: [[733335 0] [79 733804]]	F1 score: 0.9889148098413013 Confusion Matrix: [[712 6291] [68068 3924929]]
ANOVA + Pearson's, RuS	F1 score: 0.9528795811518325 Confusion Matrix: [[91 1] [8 91]]	F1 score: 0.632300123431718 Confusion Matrix: [[4540 2463] [2141150 1851847]]

Figure 4.18: Random Forest results, first trained + tested D1 and then tested on D2

Random Forest had a notably higher training time than any of the other classifiers used - it took more than 800 seconds to train the model on the imbalanced D1, whereas all of the other models had a training time below 10 seconds, as visualized in table 4.20. The training time increased even further with the oversampled D1 dataset, making Random Forest an impractical choice compared to the other models. As seen in table 4.18, the TN scores were generally not better than they were for the decision tree test cases, but the TP scores were generally higher for Random Forest. The TP results of Random Forest were, however, more comparable to the TP results of XGBoost. Since the training time of Random Forest was unfavourable compared to the other models, but the performance evaluations for Random Forest were similar to some of the other test cases, it will be excluded from further testing.

In contrast, most of the Naive Bayes test cases, for both the Pearson's, ANOVA and the combined set, with both the SMOTE D1, the RoS D1, and the RuS D1 resulted in the highest TN score for any of the test cases in this section, with at least 6607 TN classifications

out of the total number of 7003 in the normal class - however, with subpar TP scores for these test cases, where the FN score was more than 3,000,000 for all test cases. As seen in table 4.19, all of the Naive Bayes test cases done with the imbalanced D1 resulted in lower TN classifications, with much higher scores for the TP.

	Naïve Bayes	
	Training and testing on the original 5% dataset	Testing the trained model on the new sample of the dataset
ANOVA, imbalanced	F1 score: 0.9994013596223166 Confusion Matrix: [[3 98] [580 733024]]	F1 score: 0.8009440195820478 Confusion Matrix: [[2075 4928] [1314677 2678320]]
ANOVA, SMOTE	F1 score: 0.7967137140281116 Confusion Matrix: [[598087 136005] [162162 570964]]	F1 score: 0.3908323330764422 Confusion Matrix: [[6745 258] [3021030 971967]]
ANOVA, RoS	F1 score: 0.7986133863806049 Confusion Matrix: [[601982 132110] [163231 569895]]	F1 score: 0.390915747116539 Confusion Matrix: [[6761 242] [3020776 972221]]
ANOVA, RuS	F1 score: 0.8325526461217826 Confusion Matrix: [[77 14] [18 82]] (base) kamil	F1 score: 0.3905153558176672 Confusion Matrix: [[6741 262] [3022009 970988]]
Pearson's, imbalanced	F1 score: 0.9997935184796224 Confusion Matrix: [[0 101] [0 733604]]	F1 score: 0.9973746419527619 Confusion Matrix: [[0 7003] [0 3992997]]
Pearson's, SMOTE	F1 score: 0.8039978983437742 Confusion Matrix: [[610916 123176] [164172 568954]]	F1 score: 0.3695732813178614 Confusion Matrix: [[6607 396] [3085878 907119]]
Pearson's, RoS	F1 score: 0.802712004126589 Confusion Matrix: [[609662 124430] [164810 568316]]	F1 score: 0.34817550036058725 Confusion Matrix: [[6642 361] [3149501 843496]]
Pearson's, RuS	F1 score: 0.832525114564723 Confusion Matrix: [[76 15] [17 83]]	F1 score: 0.34502361983409735 Confusion Matrix: [[6646 357] [3158733 834264]]
ANOVA + Pearson's, imbalanced	F1 score: 0.9992895844641267 Confusion Matrix: [[4 97] [745 732859]]	F1 score: 0.788438045399995 Confusion Matrix: [[2337 4666] [1383963 2609034]]
ANOVA + Pearson's, SMOTE	F1 score: 0.7960553040409589 Confusion Matrix: [[598822 135270] [163844 569282]]	F1 score: 0.3831478479809097 Confusion Matrix: [[6756 247] [3044682 948315]]
ANOVA + Pearson's, RoS	F1 score: 0.7963051529949552 Confusion Matrix: [[600524 133568] [165154 567972]]	F1 score: 0.3636108749859892 Confusion Matrix: [[6816 187] [3103819 889178]]
ANOVA + Pearson's, RuS	F1 score: 0.8273104635185056 Confusion Matrix: [[76 15] [18 82]]	F1 score: 0.36320438426221324 Confusion Matrix: [[6820 183] [3105035 887962]]

Figure 4.19: Naive Bayes results, first trained + tested D1 and then tested on D2

Model	**Training time**
Random forest	811.3402659893036s
Decision tree	7.125439882278442s
Naïve Bayes	0.3220810890197754s
XGBoost	6.133977890014648s

Figure 4.20: Training time in seconds for the four differen classifiers

The most notable results from the D2 test cases in this section, in regard to favourable confusion matrix scores, were the TN classifications from Naive Bayes when trained on the

oversampled and undersampled D1, as well as the TP classifications from XGBoost when trained on the imbalanced D1. Random Forest also showed notably good TP classification scores when trained with the imbalanced D1; however, the training time was more than 100 times higher than for XGBoost. In addition to this, the D2 TN performance evaluations for Random Forest were comparable to Decision Tree. This leads us to discard Random Forest due to its resource-intensive training time, with not much better results than Decision Tree and XGBoost.

The results from the correlation-based test cases in this section have shown us that generally, selecting a feature set based on correlation does improve the performance. For instance, referring back to **??**, the D2 F1-scores were not only lower than in 4.17, but the TN and TP scores were also more balanced when applying feature selection based on correlation. XGBoost in **??** had high TN scores, but for all test cases except one, the FN scores were higher than the TP scores throughout the test cases. In comparison, Table 4.17 shows that the correlation-based feature set has prevented some of the previous overfitting to the minority class. However, this possibly introduces some overfitting to the majority class instead, since the TN scores in Table 4.17 are generally lower than the FP scores.

Similar patterns can be seen for the correlation-based Decision Tree test cases in Table 4.16, where the D2 F1-scores are higher than they were in the initial Decision Tree test cases, as shown in Table 4.7. Again, the exclusion of features with redundancy based on correlation has prevented some of the previously apparent minority class overfitting and instead introduced some overfitting to the majority class. However, for Naive Bayes, the overfitting is still evident in the minority class, but again with much better F1-scores than in the initial test cases with only entropy and the full feature set.

We would like to further improve the performance of the models and therefore explore the potential of combining the strengths of the remaining classifiers. When a dataset is as imbalanced as the ones we are dealing with in this project, there is of course a certain amount of false classifications to be expected, especially for the minority class. However, since some of the test cases showed a relatively high TN classification score, the next step is to explore how we can train a model to recognise the patterns for both normal traffic and attack traffic. So far, the models have seemingly overfitted to either one of the classes, which is visible when introducing the trained models to new data in the form of D2. This leads us to speculate that combining Naive Bayes, which tends to overfit to the minority class, with XGBoost, which seems to overfit to the majority class, might provide some contrast to each other when combined. As mentioned previously, we have decided to discard Random Forest for further testing due to excessive training time, similar to why KNN and SVM were initially discarded. However, we would also like to do further testing on the combination of Decision Tree and XGBoost. Even though XGBoost is an ensemble method consisting of multiple trees, the XGBoost as implemented for our test cases focuses on optimising logistic regression, whereas Decision Tree simply makes splits based on feature values. This allows Decision Tree to capture essential patterns quickly but makes it

sensitive to overfitting, whereas XGBoost focuses on minimising errors through boosting, thereby capturing some of the more complex interactions between features that Decision Tree might miss.

Voting classifier

In this section, we will explore the combination of some classifiers from the previous test cases. Naive Bayes generally gave good score ratios for TN/FP, while decision tree and XGBoost generally gave a better ratio of scores for TP/FN. Therefore, we decided to implement an additional ensemble method combining some of these classifiers to improve their robustness.

As mentioned previously, random forest and XGBoost are also ensemble methods. They are defined as such because they base their prediction on an ensemble of decision trees. However, the definition of an ensemble method is simply a learning algorithm that trains multiple "base models" for the classification task at hand, and these base models do not have to be decision trees. By combining multiple base learners, the idea is to produce a better-performing model. As mentioned previously, random forest and XGBoost use bagging and boosting for the ensemble of trees, respectively. Both bagging and boosting can employ something called "voting" for aggregating the ensemble of predictions. When using voting for combining the predictions, it means that the final decision is made based on a vote that each model makes. For bagging, the voting is called majority voting because the final decision will be based on what class received the majority of the votes[50]. This type of voting is also employed by the voting classifier along with soft voting. Soft voting is when each model provides an estimate for the probability of each class being the correct prediction. The final decision is then made based on the average of the probabilities provided by each of the models in the ensemble.

The voting classifier is essentially another ensemble method that, as mentioned above, trains multiple models on the same data and makes the final predictions based on their votes. The decision to explore the voting classifier as a method for botnet detection was made since we wanted to combine the strengths of Naive Bayes and XGBoost to leverage their performance. In addition to this, as seen in the results from the previous test cases, the models made different errors when trying to classify the data. The goal was to reduce these errors and thereby the impact of a single model's mistakes by averaging the probabilities from each classifier while still capturing the different aspects and diverse perspectives of the data[64].

Voting classifier parameters	
estimators=[('dt', dt_clf), ('xg', xg_clf)] estimators=[('nb', nb_clf), ('xg', xg_clf)]	Indicates what estimators are combined
voting='soft'	Denotes that a soft voting mechanism is used for the final prediction
weights=None	Weights the class probabilities before averaging the sums of the predicted probalities
verbose=True	Prints the time elapse when the model is being fitted

Figure 4.21: The parameters specified for the voting classifier[3]

Similar to some of the other classifiers used in this project, the voting classifier is also implemented through the scikit-learn library in Python. Table 4.21 shows the parameters that were specified for the voting classifier test cases in this project. The *estimators* parameter specifies what classifiers should be trained on the data. This is specified to be either Naive Bayes and XGBoost or decision tree and XGBoost, each of which is specified with the same parameters as shown in Table 4.1. Then the type of *voting* mechanism is specified to be soft, which, as previously explained, means that it will base the final estimation on an average of the probability estimates provided by each model rather than just the prediction with the highest number of votes. The *weights* parameter specifies that each model should be weighted the same amount when making the final prediction. The last parameter, *verbose*, simply ensures that the time lapse for training the model is printed[3].

A soft voting classifier was combined XGBoost and Naive Bayes, with the aim of combining the general ability of classifying TN from Naive Bayes with an optimized ensemble method like XGBoost. Since random forest showed to be more computationally expensive but did not provide significantly better results than decision tree or XGBoost, the other voting classifier combined decision tree and XGBoost. This aimed to combine the more straightforward decision-making of decision tree with the iterative focus on minimizing errors of XGBoost.

The Naive Bayes classifier will assume that the features used for prediction of a given target variable are conditionally independent of each other. To give an example, this means that the model assumes that the features *min* and *dur* are considered separately when predicting if the traffic is botnet or normal - which is a naive assumption, since the shortest duration of the aggregated records, *min*, may very well be directly dependent on the duration of the specific traffic instance, *dur*, since there is a possibility that *min* is derived from *dur*. With this in mind, if we refer back to the Naive Bayes results in figure 4.19, it may suggest that when we oversample or undersample our dataset, the dependencies between the features in the normal traffic become less prominent when we remodel the minority class, allowing for a high TN score with Naive Bayes since it already explicitly assumes that the features are independent and therefore will not base its classification on that.

The very high TP rate when training on the imbalanced dataset suggests that the pat-

terns of the majority class (attack traffic) are strong enough that the assumption of feature independence in Naive Bayes does not significantly impact its classification of attack traffic. However, this same assumption causes Naive Bayes to struggle with classifying the minority class (normal traffic) correctly. This is because Naive Bayes overlooks the dependencies between features that might be crucial for recognizing patterns in the minority class.

In contrast, models like XGBoost, decision tree, and random forest can capture and utilize these dependencies between features when making splits in the feature space. This capability allows them to better distinguish between normal and attack traffic, even in the presence of class imbalance.

	XGBoost + Naïve Bayes ensemble	
	Training and testing on D1	Testing the trained model on D2
ANOVA, imbalanced	F1 score: 0.9998067321938837 Confusion Matrix: [[4 97] [0 733604]]	F1 score: 0.898522048120804 Confusion Matrix: [[537 6466] [720079 3272918]]
ANOVA, SMOTE	F1 score: 0.9782509017451732 Confusion Matrix: [[726774 7318] [24588 708538]]	F1 score: 0.6595053977032136 Confusion Matrix: [[6772 231] [2023289 1969708]]
ANOVA, RoS	F1 score: 0.9775321217467199 Confusion Matrix: [[724686 9406] [23556 709570]]	F1 score: 0.7057146648909637 Confusion Matrix: [[4338 2665] [1808475 2184522]]
ANOVA, RuS	F1 score: 0.93717277486911 Confusion Matrix: [[85 6] [6 94]]	F1 score: 0.591403444848012 Confusion Matrix: [[4534 2469] [2311336 1681661]]
Pearson's, imbalanced	F1 score: 0.9997968990258452 Confusion Matrix: [[1 100] [0 733604]]	F1 score: 0.9970052259244319 Confusion Matrix: [[110 6893] [3358 3989639]]
Pearson's, SMOTE	F1 score: 0.9839501278713721 Confusion Matrix: [[729880 4212] [19334 713792]]	F1 score: 0.827539507552016 Confusion Matrix: [[1619 5384] [1162457 2830540]]
Pearson's, RoS	F1 score: 0.9954137675995497 Confusion Matrix: [[731012 3080] [3649 729477]]	F1 score: 0.8189604798369503 Confusion Matrix: [[1519 5484] [1212137 2780860]]
Pearson's, RuS	F1 score: 0.9318697411822746 Confusion Matrix: [[83 8] [5 95]]	F1 score: 0.5997568325014322 Confusion Matrix: [[4034 2969] [2277166 1715831]]
ANOVA + Pearson's, imbalanced	F1 score: 0.9998463992301869 Confusion Matrix: [[18 83] [1 733603]]	F1 score: 0.8453634995063772 Confusion Matrix: [[1157 5846] [1056374 2936623]]
ANOVA + Pearson's, SMOTE	F1 score: 0.98884859531051 Confusion Matrix: [[729898 4194] [12167 720959]]	F1 score: 0.7472074325133027 Confusion Matrix: [[3875 3128] [1602937 2390060]]
ANOVA + Pearson's, RoS	F1 score: 0.9941992265211234 Confusion Matrix: [[729487 4605] [3906 729220]]	F1 score: 0.7509017575249488 Confusion Matrix: [[3903 3100] [1584014 2408983]]
ANOVA + Pearson's, RuS	F1 score: 0.9476439790575916 Confusion Matrix: [[86 5] [5 95]]	F1 score: 0.6034729072460713 Confusion Matrix: [[4543 2460] [2262153 1730844]]

Figure 4.22: The performance evaluations of the voting classifier with XGBoost and Naive Bayes

As seen in the table 4.22, the highest F1-score on D2 were 0.997 when training the voting classifier with the Pearson's feature set on the imbalanced D1 set. Even though this test case managed to get a very high TP score, the TN score was very low - which would

be effective for detecting botnet traffic, however the system would not be very usable if it also filtered out all the normal traffic. Therefore, this does not seem to be the optimal solution. None of the test cases managed to combine the high TN score from Naive Bayes with the more preferable TP score from XGBoost, and generally the oversampled and undersampled test cases had a lower F1-score with the voting classifier than they did when just using the XGBoost on its own.

The combination of decision tree and XGBoost gave the most optimal results for testing on D2 for any of the test cases in this project - when using the combined feature set of ANOVA and Pearson's, the results showed both high F1-scores as well as better TP and TN ratios than for any previous test cases.

	XGBoost + Decision Tree ensemble	
	Training and testing on D1	Testing the trained model on D2
ANOVA, imbalanced	F1 score: 0.999874509079053 Confusion Matrix: [[62 39] [57 733547]]	F1 score: 0.970511436703453 Confusion Matrix: [[1601 5402] [210987 3782010]]
ANOVA, SMOTE	F1 score: 0.9861594943134079 Confusion Matrix: [[734048 44] [20259 712867]]	F1 score: 0.8799084344531729 Confusion Matrix: [[1756 5247] [842300 3150697]]
ANOVA, RoS	F1 score: 0.9863095024251376 Confusion Matrix: [[734092 0] [20083 713043]]	F1 score: 0.9493023400038316 Confusion Matrix: [[1898 5105] [368773 3624224]]
ANOVA, RuS	F1 score: 0.9790656393205903 Confusion Matrix: [[90 1] [3 97]]	F1 score: 0.705130029403435 Confusion Matrix: [[3000 4003] [1810530 2182467]]
Pearson's, imbalanced	F1 score: 0.9998048786863643 Confusion Matrix: [[61 40] [129 733475]]	F1 score: 0.9910618455361909 Confusion Matrix: [[42 6961] [50247 3942750]]
Pearson's, SMOTE	F1 score: 0.9928501295531504 Confusion Matrix: [[733246 846] [9644 723482]]	F1 score: 0.8392814977558899 Confusion Matrix: [[1321 5682] [1092946 2900051]]
Pearson's, RoS	F1 score: 0.9999488828499103 Confusion Matrix: [[734092 0] [75 733051]]	F1 score: 0.9776926685844715 Confusion Matrix: [[964 6039] [155498 3837499]]
Pearson's, RuS	F1 score: 0.9529054253415202 Confusion Matrix: [[89 2] [7 93]]	F1 score: 0.7908443642728387 Confusion Matrix: [[2447 4556] [1370851 2622146]]
ANOVA + Pearson's, imbalanced	F1 score: 0.9998537205426938 Confusion Matrix: [[72 29] [93 733511]]	F1 score: 0.9778545948794322 Confusion Matrix: [[1820 5183] [155214 3837783]]
ANOVA + Pearson's, SMOTE	F1 score: 0.9929748688487143 Confusion Matrix: [[733267 825] [9482 723644]]	F1 score: 0.958693650822364 Confusion Matrix: [[5688 1315] [303635 3689362]]
ANOVA + Pearson's, RoS	F1 score: 0.999952972222063 Confusion Matrix: [[734092 0] [69 733057]]	F1 score: 0.9837972892971938 Confusion Matrix: [[1572 5431] [109054 3883943]]
ANOVA + Pearson's, RuS	F1 score: 0.9633588363526786 Confusion Matrix: [[88 3] [4 96]]	F1 score: 0.6218306249429757 Confusion Matrix: [[4243 2760] [2185546 1807451]]

Figure 4.23: The performance evaluations of the voting classifier with XGBoost and decision tree

As seen in table 4.23, the F1-score was generally higher than for the previous voting classifier test cases - however this was still mainly due to the TP classifications being very high compared to the TN. When training the classifier with the combined ANOVA + Pear-

son's feature set on D1 oversampled with SMOTE, it did manage to classify 5688 normal traffic instances correctly in the D2. However, this test case classified 303635 attack traffic instances incorrectly, which would most likely not be a successful result for a botnet detection system.

This test case did not show overfitting and had a F1-score of 0.958 where 81.2% of the normal traffic was classified correctly and 92.4% of the attack traffic was classified correctly. Even though these numbers are not necessarily optimal for a botnet detection system, they do not show an evident bias towards either of the target variable classes. A few of the other test cases did have higher F1-score, however they all showed signs of overfitting.

It was difficult to conclude the specific metrics for an optimal botnet detection system through these results, as all the other the performance evaluations of the voting classifiers did not show a clear preference for any of the feature sets or the oversampled, undersampled or imbalanced dataset. For instance, in table 4.22, the highest TN scores were provided when either undersampling or oversampling the dataset - except when using the Pearson's feature set, of which only the undersampled test case showed high TN scores. In addition to this, as seen in table 4.23, for the ANOVA + Pearson's test cases, the imbalanced set, RuS and RoS all showed signs of either underfitting or overfitting.

We found that generally for many of the test cases for all of the classifiers, a big part of the FN results were caused by specific feature values being very imbalanced between the classes. When oversampling D1, this imbalance of feature values was amplified depending on the method of oversampling - specifically RoS caused a lot of feature value imbalance when using it to balance out the minority class. This is most likely because RoS duplicates existing instances directly. A specific example is shown in table 4.24, where the voting classifier is trained on an oversampled D1, resulting in 3496 FN classifications, but when all records that have the value of 0 for the feature *stddev* are removed, the performance scores are perfect for D1. Referring to table A in appendix A, we can see that when we oversample the dataset with RoS, the normal traffic will have 4.85 as many records where *stddev* is 0 compared to the attack traffic.

Soft voting classifier (XGBoost + Naïve Bayes), ANOVA + Pearson's + RandomOverSampler		
	Training + testing on D1	Testing on D2
Keeping records where stddev is 0	F1 score: 0.9941992265211234 Confusion Matrix: [[729487 4605] [3906 729220]]	F1 score: 0.7509017575249488 Confusion Matrix: [[3903 3100] [1584014 2408983]]
Removing records where stddev is 0	F1 score: 0.9978757530358378 Confusion Matrix: [[141059 1596] [2 611233]]	F1 score: 0.9904582874760801 Confusion Matrix: [[639 122] [19887 1112055]]

Soft voting classifier (XGBoost + Naïve Bayes), ANOVA + Pearson's + SMOTE		
	Training + testing on D1	Testing on D2
Keeping records where stddev is 0	F1 score: 0.98884859531051 Confusion Matrix: [[729898 4194] [12167 720959]]	F1 score: 0.7472074325133027 Confusion Matrix: [[3875 3128] [1602937 2390060]]
Removing records where stddev is 0	F1 score: 0.9985943260633422 Confusion Matrix: [[147073 1045] [21 611214]]	F1 score: 0.9932481611200319 Confusion Matrix: [[597 164] [13648 1118294]]

Figure 4.24: Table showing the how the feature value imbalance effects the performance with both SMOTE and RoS

When the records containing the feature values that were very imbalanced were removed from the test set without removing them from the training set, we got closer to ideal scores on the subset of features that is the combination of the chosen features based on the ANOVA score and Pearson's coefficient. This led us to the hypothesis that the majority of the FN and FP classifications across the test cases were caused by some specific feature values being significantly imbalanced between normal and attack traffic. Therefore, oversampling and undersampling may not be optimal solutions for botnet detection.

As mentioned above, our findings show that the best method for classifying botnet traffic based on an imbalanced dataset is by training a soft voting classifier with both the XGBoost ensemble method and Naive Bayes, using entropy-based feature selection that considers both a high correlation between the features and the target variable, as well as low correlation between the features. We also found that this does not fully address the most substantial feature value imbalances between the attack and normal traffic. Therefore, an optimal solution would not be to emphasize this through oversampling or undersampling the imbalanced dataset.

Imbalanced feature values

In our analysis of the dataset, we observed several features exhibiting significant imbalances in their values. These imbalances could potentially introduce bias in our test cases, thereby impacting the reliability and prediction capabilities of our models. To further analyze this imbalance and the impact it has on the predictions, we calculated which features had the most feature value imbalance. The calculations were made by evaluating the variety of the feature values. If any feature has a disproportionately high occurrence of a specific value for a specific class, the imbalance may cause a large number of records in the opposite class to be misclassified, as the model is trained to associate that feature value predominantly with the "wrong" class.

In order to take a better look at the specific feature values causing this imbalance and thereby further address it, we have calculated an imbalance ratio based on how many times a feature value is represented in one class compared to the other class. These are shown in Appendix A. We have taken the full feature set and created four test cases where we applied both undersampling with RuS, oversampling with RoS and SMOTE, and keeping the

original records in the dataset as is. Then, for each of the most represented feature values in each of the test cases, we calculated the imbalance from the normal class to the attack class and the attack class to the normal class—that is, for the feature values that appear in both classes. The feature value imbalance is first calculated for D1, since this is the dataset that the model is trained on and therefore, this is the dataset where a large imbalance of the feature values between classes can influence the patterns learned by the model and subsequently impact the model's ability to predict which class any new data should belong to.

In the tables shown in Appendix A, we included every feature value imbalance exceeding 2 for the 10 most represented feature values for each feature, since this will indicate that a feature value is at least twice as represented for one of the classes (normal or attack) than for the other. The normal to attack traffic ratio for a given feature value represents the proportion of normal traffic instances compared to attack traffic instances. It indicates how much more prevalent some feature values are in the normal traffic relative to attack traffic, and vice versa for attack to normal traffic ratio, where the table shows what feature values are much more prevalent in the attack traffic compared to the normal traffic.

The feature value imbalance was especially present in the attack to normal traffic ratio in D2, as seen in the rightmost column in Table A.3—in this table, the dataset is neither undersampled nor oversampled and is therefore imbalanced between the benign and attack records, which is why many of the feature values are represented more in the attack traffic than in the normal traffic. Therefore, the feature value imbalance is to be expected.

However, the feature value imbalance for D1 is more interesting, primarily due to the oversampling of the dataset. In Table A, we can see that when we oversample D1, many feature values will be more than twice as represented for the normal class than for the attack class, which will most likely introduce a bias towards the normal class. This is further emphasised in Table A, where we can see that the feature values that in the imbalanced D1 are more than twice as represented for the attack traffic are eliminated when oversampling the minority class. This is, of course, to be expected to a certain degree; however, this will dilute some of the defining features for the patterns in the majority class.

When looking at the normal to attack traffic ratio of the RandomOverSampler test case, it is clear that many feature values have increased representation in the normal traffic. The purpose of applying oversampling and undersampling to a dataset is to balance the dataset—this means that even though the RandomOverSampler has balanced the records to an even number, the feature values inside the records are not necessarily as balanced and proportionally distributed in the minority class that has been oversampled as they are in the untouched majority class. The variety of the feature values will depend on the method used for oversampling. As mentioned previously, RoS just makes additional copies of the records that already exist in the dataset, while SMOTE makes new records

that consist of different feature values comprised of the 5 nearest neighbour records of a sample from the minority class. This may explain why the imbalanced feature values are more prominent in the RoS oversampled normal traffic than the attack traffic, since the records are just replicated directly, causing the existing feature values to be amplified according to the ratio that the minority class needs to be amplified in order to reach balance between the classes, instead of being created as new but plausible representations of the minority class.

For the SMOTE test case, the feature value representation ratio is more even; however, the features that have multiple feature values with high ratios are different from the normal to attack traffic and the attack to normal traffic. In the normal traffic (the minority class that has been oversampled), the features that have more than four feature values with a higher ratio of representation are *sport*, *dport*, *state*, *stddev*, *max*, and *dpkts*. For the attack traffic, the features that have more than four feature values with a higher ratio of representation are *daddr*, *pkts*, *bytes*, *spkts*, *sbytes*, and *dbytes*.

The RandomUnderSampler shows the least amount of highly represented feature values when measuring across both normal and attack traffic. Only two features, *pkts* and *spkts*, show more than four feature values with a ratio of more than 2, specifically for the attack traffic.

As mentioned previously, table A.3 shows the imbalance in D2. The feature value ratios were only calculated for the original imbalanced version of this dataset, as D2 was never over- or undersampled for previous experiments, as it is created to be an imbalanced dataset for validation purposes and is never used for training. This dataset showed to be the most imbalanced dataset when compared to the previous test cases, since only one feature value of *dport* for normal traffic had a relatively high representation compared to the attack traffic, whereas all the features had at least one value that was represented with a ratio of more than 2 for the attack traffic. To further explore the imbalance ratio shown through the most represented feature values in both normal and attack traffic and how this may affect the results from testing and training the different models, we would like to experiment further with adjusting the specific feature values.

To further visualise how the imbalance is represented in the test cases, the largest feature value count for each feature in the ANOVA + Pearson's set is shown in Table 4.25 when using both SMOTE, RoS, and the imbalanced D1 with the voting classifier with decision tree and XGBoost. If we refer back to Table 4.23, we can see that the number of misclassified attack traffic instances is very similar to the number of features where the value of *stddev* is 0. In relation to this, the RoS and imbalanced feature values are very similarly represented — which is logical, since the additional samples of the minority class that are created when using RoS are direct copies of already existing records.

	Largest feature value count when using RoS			
	Training + testing on D1		Testing on D2	
	Feature value	Count	Feature value	Count
stddev	0	65	0	163660
sum	0	50	0	346
dur	0	50	0	346
min	0	50	0	601
sport	48374	18	103212	17997
max	0	50	0	346
rate	0	50	0	346

	Largest feature value count when using SMOTE			
	Training + testing on D1		Testing on D2	
	Feature value	Count	Feature value	Count
stddev	0	9471	0	309861
sum	0	9431	0	93554
dur	0	9431	0	43820
min	0	9431	0	93557
sport	26501	23	5567	40913
max	0	9431	0	93554
rate	0	9431	0	43280

	Largest feature value count on imbalanced dataset			
	Training + testing on D1		Testing on D2	
	Feature value	Count	Feature value	Count
stddev	0	84	0	133177
sum	0	63	0	547
dur	0	63	0	547
min	0	63	0	780
sport	48374	11	103212	19421
max	0	63	0	547
rate	0	63	0	547

Figure 4.25: The values that are most represented in each feature, all test cases uses the voting classifier with decision tree and XGBoost

As seen for the voting classifier test cases in table 4.17, the best results specifically for TN classifications in D2 were achieved using SMOTE, where it correctly classified 5688 out of 7003 instances. This is most likely due to SMOTE creating more "randomised" new instances of the minority class, which adds variety to the trained model and makes it easier for the model to recognise the new instances of normal traffic present in D2 but not in D1. In contrast, for RoS, the oversampling merely multiplies the already existing samples of normal traffic, of which there are already few in D1. As a result, when the trained model is tested on D2, it does not recognise all the patterns of normal traffic when presented with an "organically" larger minority class.

However, the majority class will experience a loss in the TP predictions from this, since its own amplification of its defining feature values lies in that it is the majority class and will therefore have a larger index of feature values to be defined on. By oversampling the minority class, we will inevitably reduce some of that amplification since we redistribute the feature values.

pkSeqID	flgs	proto	saddr	sport	daddr	dport	pkts	bytes	state	dur	mean	stddev	sum	min	max	spkts	dpkts	sbytes	dbytes	rate	srate	drate	attack	category	subcategory
2	e	arp	192.168.100.7	-1	192.168.100.14	-1	2	120	CON	0.000131	0.000131	0	0.000131	0.000131	0.000131	1	1	60	60	7633.5883	0	0	1	DoS	HTTP
263646	e s	tcp	192.168.100.147	47157	192.168.100.7	80	5	770	REQ	32.005363	0	0	0	0	0	5	0	770	0	0.124979	0.124979	0	1	DoS	TCP
263647	e s	tcp	192.168.100.147	47098	192.168.100.7	80	5	770	REQ	31.96143	0	0	0	0	0	5	0	770	0	0.125151	0.125151	0	1	DoS	TCP
525797	e s	tcp	192.168.100.147	9762	192.168.100.7	80	5	770	REQ	59.491379	0	0	0	0	0	5	0	770	0	0.067237	0.067237	0	1	DoS	TCP
525798	e s	tcp	192.168.100.147	9763	192.168.100.7	80	5	770	REQ	59.491379	0	0	0	0	0	5	0	770	0	0.067237	0.067237	0	1	DoS	TCP
3152934	e	arp	192.168.100.149	-1	192.168.100.3	-1	2	120	CON	0.002297	0.002297	0	0.002297	0.002297	0.002297	1	1	60	60	435.35046	0	0	1	DDoS	UDP
3152935	e	arp	192.168.100.148	-1	192.168.100.3	-1	2	120	CON	0.000562	0.000562	0	0.000562	0.000562	0.000562	1	1	60	60	1779.3594	0	0	1	DDoS	UDP
3414894	e	arp	192.168.100.147	-1	192.168.100.3	-1	2	120	CON	0.00155	0.00155	0	0.00155	0.00155	0.00155	1	1	60	60	645.16125	0	0	1	DDoS	UDP
3513966	e	arp	192.168.100.149	-1	192.168.100.3	-1	1	60	INT	0	0	0	0	0	0	1	0	60	0	0	0	0	1	DDoS	UDP
3576887	e	udp	192.168.100.149	51838	27.124.125.250	123	2	180	CON	0.048565	0.048565	0	0.048565	0.048565	0.048565	1	1	90	90	20.59096	0	0	0	Normal	Normal
3577249	e	udp	192.168.100.3	53586	202.12.27.33	53	2	1251	CON	0.15618	0.15618	0	0.15618	0.15618	0.15618	1	1	70	1181	6.402869	0	0	0	Normal	Normal
3577250	e	udp	192.168.100.3	18535	202.12.27.33	53	2	1209	CON	0.162404	0.162404	0	0.162404	0.162404	0.162404	1	1	70	1139	6.157484	0	0	0	Normal	Normal
3577251	e	udp	192.168.100.150	39053	192.168.217.2	53	2	172	INT	2.501064	0	0	0	0	0	2	0	172	0	0.39983	0.39983	0	0	Normal	Normal
3577252	e	udp	192.168.100.150	60153	8.8.8.8	53	2	172	CON	0.007359	0.007359	0	0.007359	0.007359	0.007359	1	1	86	86	135.88803	0	0	0	Normal	Normal
3619115	e	tcp	192.168.100.147	2185	192.168.100.3	1	2	120	RST	0.000079	0.000079	0	0.000079	0.000079	0.000079	1	1	60	60	12658.228	0	0	1	Reconnais	Service_Scan
3619116	e	tcp	192.168.100.150	2567	192.168.100.3	1	2	120	RST	0.000068	0.000068	0	0.000068	0.000068	0.000068	1	1	60	60	14705.881	0	0	1	Reconnais	Service_Scan
3619117	e	tcp	192.168.100.148	1510	192.168.100.3	1	2	120	RST	0.000068	0.000068	0	0.000068	0.000068	0.000068	1	1	60	60	14705.881	0	0	1	Reconnais	Service_Scan
3619118	e	tcp	192.168.100.147	2186	192.168.100.3	1	2	120	RST	0.000096	0.000096	0	0.000096	0.000096	0.000096	1	1	60	60	10416.666	0	0	1	Reconnais	Service_Scan
3623806	e	tcp	192.168.100.147	4182	192.168.100.3	1	2	120	RST	0.000059	0.000059	0	0.000059	0.000059	0.000059	1	1	60	60	16949.152	0	0	1	Reconnais	Service_Scan
3623807	e	tcp	192.168.100.150	4564	192.168.100.3	1	2	120	RST	0.000058	0.000058	0	0.000058	0.000058	0.000058	1	1	60	60	17241.378	0	0	1	Reconnais	Service_Scan
3623808	e	tcp	192.168.100.148	3507	192.168.100.3	1	2	120	RST	0.000074	0.000074	0	0.000074	0.000074	0.000074	1	1	60	60	13513.512	0	0	1	Reconnais	Service_Scan
3623809	e	tcp	192.168.100.147	4183	192.168.100.3	1	2	120	RST	0.000047	0.000047	0	0.000047	0.000047	0.000047	1	1	60	60	21276.595	0	0	1	Reconnais	Service_Scan
3659550	eU	udp	192.168.100.149	43758	192.168.100.5	18234	1	60	INT	0.000214	0.000214	0	0.000214	0.000214	0.000214	1	0	60	0	0	0	0	1	Reconnais	Service_Scan
3659551	e	icmp	192.168.100.5	303	192.168.100.14	3A47	1	70	URP	0	0	0	0	0	0	1	0	70	0	0	0	0	1	Reconnais	Service_Scan
3659552	e	udp	192.168.100.147	35226	192.168.100.5	20411	1	60	INT	0	0	0	0	0	0	1	0	60	0	0	0	0	1	Reconnais	Service_Scan
3668450	e	tcp	192.168.100.3	54766	192.168.100.14	4433	2	134	RST	0.00011	0.00011	0	0.00011	0.00011	0.00011	1	1	74	60	9090.9091	0	0	1	Theft	Keylogging
3668451	e	tcp	192.168.100.3	42954	192.168.100.15	4433	2	134	RST	0.000112	0.000112	0	0.000112	0.000112	0.000112	1	1	74	60	8928.5712	0	0	1	Theft	Keylogging

Figure 4.26: Some of the BoT-IoT traffic instances where the standard deviation is 0 [1]

Table 4.26 shows a sampled selection of the traffic instances in D1 where *stddev* is 0. We can see that they vary widely across source IP addresses, protocols, durations, bytes, type of attack, etc. This indicates that even though *stddev* is a feature highly correlated with the target variable according to the ANOVA score, it is not specifically representative of one type of traffic. This makes sense, since the standard deviation is not based on the records as singular instances but shows how the records generally deviate from the standard duration within an aggregated group. This would make *stddev* an obvious choice as a sorting criterion for filtering out which records are grouped together. To do this, we filtered D1 first based on the ascending order of *stddev* and then based on the descending order of *mean*.

With the help of the generated features that were initially discarded due to not being included in D2, we can assume that the groups are aggregated based on the source address, the destination address, and the state of the transaction, since those features also correlate with the aggregated groups of the records. The connection between the generated features based on the aggregation of records and *saddr* and *daddr* features can be seen in Table A.4 in Appendix A.

The reason for the generated features being aggregated based on these three features is most likely that it makes it easier to identify sources that are frequently initiating specific requests or connections to certain destinations, which can often be indicative of specific attacks.

Building on the knowledge that the models are showing difficulties in predicting the traffic instances where the standard deviation is 0, we ran some test cases without including

stddev in the selected feature set, as seen in Figure 4.26. However, this did not show any significant enhancement in the performance evaluation; specifically, the TN scores were lower in D2 compared to the previous test cases, and the confusion matrix showed signs of overfitting to the majority class. This is most likely because it removes a feature from the minority class that is highly correlated with the target variable and therefore informative for classification, making it even harder to recognise any patterns than it already is due to the class imbalance.

Removing stddev from the selected feature set		
	Training + testing on D1	Testing on D2
SMOTE	F1 score: 0.9929844077391796 Confusion Matrix: [[733302 790] [9503 723623]]	F1 score: 0.9364892156205792 Confusion Matrix: [[1330 5673] [460352 3532645]]
RoS	F1 score: 0.9999550169081294 Confusion Matrix: [[734092 0] [66 733060]]	F1 score: 0.9841813676361947 Confusion Matrix: [[916 6087] [105282 3887715]]

Figure 4.27: Running the decision tree + XGBoost model where standard deviation is removed from the feature set

A solution to the feature value imbalance may be to handle some feature values individually. An approach for this could be to set a threshold for how much a given feature value can be represented in one type of traffic compared to the other. For example, if the normal traffic has 4.85 times as many records where the *stddev* feature has the value of 0 compared to attack traffic, there would likely be a partiality towards the normal traffic when the model is trying to classify a record with *stddev* as 0. Consequently, a significant portion of predictions could also be misclassified if there is still a substantial amount of attack traffic with this specific feature value. According to the threshold, a mask could be added to some of the imbalanced feature values to dilute the imbalance, similar to handling missing values in a dataset. Another solution may be to handle the imbalance of the dataset by creating more authentic instances of the minority class, which would not just amplify the feature values that are already represented in the minority class, but also possibly create more diversity for diluting this imbalance.

4.6 Wrapper methods

As explained in the preceding sections, the analysis of feature contribution in dataset training can take various forms, offering different types of insight into the model's performance and feature patterns. One such approach is the utilisation of wrapper methods, which involves finding a selected set of features by actually training the model and evaluating the resulting performance at runtime.

There are multiple ways of finding a feature set through evaluating the performance of the models; we have chosen to explore forward selection and backward selection, as implemented by the Sequential Feature Selector in Python. The best model performance in the previous section resulted from training the model with the combined set of ANOVA and Pearson's features, which consisted of seven independent variables. Therefore, the wrapper methods in this section will also choose the seven best features. Additionally, since the voting classifier gave the best result, it will also be used in this section.

Forward feature selection (FFS) works by starting with an empty set of features. For each iteration, it will run a classifier and add the feature to the feature set that maximises the performance of the model according to the specified metric. For our test cases, we chose the voting classifier with decision tree and XGBoost as the model and F1-score as the performance metric. This process continues until a specific number of features in the set is reached or until the F1-score does not improve. In contrast to FFS, backward feature selection (BFS) starts with the full set of features. For each iteration, a feature will be removed if its removal results in the highest possible F1-score for the model.[56]

For this project, the Sequential Feature Selector from the scikit-learn library in Python was used to implement both the FFS and BFS with the parameters shown in Table 4.28.[3] The voting classifier with XGBoost and decision tree was defined as the classifier to be optimised since this model generally achieved the most desirable results with the filter methods. Additionally, 5-fold cross-validation was used to ensure that the entire dataset was utilised for training and testing and that the features chosen as the best would represent a more reliable estimate of the model's performance.

Sequential Feature Selector parameters	
estimator=voting_clf	Denotes the type of classifier used, which in this case is the voting classifier with XGBoost + decision tree
n_features_to_select=7	The amount of features that should be selected
direction='backward', direction='forward'	Defines whether the feature selection is based on forward selection or backward selection
scoring='f1_weighted'	What performance metric the feature selector should optimize through the chosen features
cv=5	5-fold cross-validation

Figure 4.28: The parameters that were chosen for the Sequential Feature Selector to implement the wrapper methods [3]

These two processes were performed on the imbalanced D1 dataset with the full feature set to either validate or provide a contradiction to the previously selected features. Through the feature selection process with FFS, we identified seven features that should impact the F1-score of the voting classifier. The seven features chosen through FFS were *saddr*, *daddr*, *stddev*, *spkts*, *dpkts*, *srate*, and *drate*. The feature set defined through BFS included *flgs_number*, *saddr*, *daddr*, *pkts*, *min*, *sbytes*, and *srate*, as shown in Figure 4.29. The

features that overlap between the sets are *saddr*, *daddr*, and *srate*—these are all features that would generally be considered very informative when trying to classify traffic as normal or attack.

Normal traffic would originate from a wide range of arbitrary IPaddresses and generally show predictable patterns with legitimate destination addresses. Attack traffic may communicate with a C&C server that originates from or is directed to specific IP addresses associated with these servers—bots within a botnet might also be deployed within the same or similar IP ranges. Attack traffic may also be indicated through the source rate through burst patterns or low and slow attacks, whereas normal traffic would show more variability that responds to organic user patterns.

	Wrapper method results	
	Selected features	Time
Forward selection	'saddr', 'daddr', 'stddev', 'spkts', 'dpkts', 'srate', 'drate'	4251.116105079651s
Backward selection	'flgs_number', 'saddr', 'daddr', 'pkts', 'min', 'sbytes', 'srate'	13096.721642017365s

Figure 4.29: The selected features by each wrapper method and their training time

The feature sets that were found by using FFS and BFS were used for training the voting classifier combining decision tree and XGBoost, since this is the classifier that got the most favorable results with the filter methods. The undersampled dataset have not been included, since it primarily has introduced redundant results with low F1-scores in previous test cases. In the table below, it can be seen that both the BFS and FFS showed near-perfect results for D1. The second column shows an extremely low results for all test cases, meaning that the feature set as selected by wrapper methods do not showcase superiority to the correlation-based feature sets. The second column shows that the TN predictions were much higher than the TP, since none of the attack traffic instances were classified correctly. This is most likely due to the model overfitting when trying to find the patterns that will give the highest performance, rather than actually recognizing the meaningful patterns in the data. When looking at the results in table 4.30, it is evident that the trained models are biased towards the minority class - the model might overcompensate by focusing too much on the minority class, therefore placing emphasis on the minority class to correct for the imbalance. When oversampling, the patterns in the minority class are not "organic", they are artificially amplified, which can make the model too sensitive for these technically non-existing patterns that do not occur when the minority class is larger due to consisting of more real traffic instances, which is the case for D2.

	Wrapper validation on decision tree + XGBoost voting classifier	
	Training and testing on D1	Testing the trained model on D2
FFS feature set, imbalanced	F1 score: 1.0 Confusion Matrix: [[101 0] [0 733604]]	F1 score: 6.065452798032215e-06 Confusion Matrix: [[6941 62] [3992997 0]]
FFS feature set, SMOTE	F1 score: 0.9999870503223464 Confusion Matrix: [[734091 1] [18 733108]]	F1 score: 6.069814517091071e-06 Confusion Matrix: [[6946 57] [3992997 0]]
FFS feature set, RoS	F1 score: 1.0 Confusion Matrix: [[734092 0] [0 733126]]	F1 score: 6.065452798032215e-06 Confusion Matrix: [[6941 62] [3992997 0]]
BFS feature set, imbalanced	F1 score: 1.0 Confusion Matrix: [[101 0] [0 733604]]	F1 score: 6.058474024896348e-06 Confusion Matrix: [[6933 70] [3992997 0]]
BFS feature set, SMOTE	F1 score: 0.9999972737520891 Confusion Matrix: [[734092 0] [4 733122]]	F1 score: 6.0890059516322565e-06 Confusion Matrix: [[6968 35] [3992997 0]]
BFS feature set, RoS	F1 score: 0.9999993184380233 Confusion Matrix: [[734092 0] [1 733125]]	F1 score: 6.058474024896348e-06 Confusion Matrix: [[6933 70] [3992997 0]]

Figure 4.30: Performance of the feature sets as chosen by forward and backward feature selection

Even though it did manage to correctly predict the minority class, the model did not manage to predict the attack traffic, making it useless for the purpose of botnet detection. In addition to this, BFS was the most demanding feature selection method in terms of computational power - selecting a feature set using this method took 13096.721 seconds, while the training time of FFS was 4251.116 seconds. This, in comparison with the time it took to calculate the scores used for the entropy and correlation-based feature selection as shown in 4.15, also emphasizes that the wrapper methods are not feasible for reducing the feature set when compared to the filter methods.

Chapter 5

Interpretation of Results

In this chapter, we will present an overview of the results from the previous chapter as well as an analysis of the insights it provides in relation to the detection of botnet traffic. The purpose of this is to understand how well the models performed and conclude whether machine learning is a comprehensive tool for identifying attack traffic in IoT networks. We will also identify which features, variables, and models provided the most notable performance and highlight any limitations we found. Specifically, we will review the models, feature importance, and feature selection methods by examining the results and patterns from the last chapter. These insights will also be used to set a scope for possible future directions.

5.1 Insights from the Machine Learning Models

For our experiments, four different models were chosen: Gaussian Naive Bayes, decision tree, random forest, and logistic regression with XGBoost. Initially, a few other models were considered, namely SVM and KNN. However, SVM and KNN were discarded in the initial testing stages due to being too computationally costly.

Naive Bayes

As mentioned previously, we chose Gaussian Naive Bayes since it assumes a normal distribution of the feature values, whereas Bernoulli Naive Bayes assumes that all the feature values are binary. This is not the case for most of the features relating to network traffic, such as protocol, destination/source IP, and bytes—these are continuous values rather than binary, which is why Gaussian Naive Bayes was deemed more appropriate. The Naive Bayes test cases received high F1-scores when using the imbalanced dataset; however, this was due to the almost complete set of attack traffic instances being classified correctly, while none of the minority class normal traffic was classified correctly. This would not be feasible in an intrusion detection system since even though almost all attack traffic would

be detected and blocked, it would also block all normal traffic. When oversampling or undersampling the dataset, Naive Bayes achieved the best True Positive rate for the correct classification of normal traffic when cross-validating on the D2 dataset, with around 6000 correct classifications and 700 misclassifications. For the D1 dataset, it repeatedly misclassified around 10000 normal instances. The steady number of misclassifications suggests that it is most likely due to the imbalance of specific feature values when the dataset is either oversampled or undersampled.

XGBoost, Decision Tree, and rRandom Forest

XGBoost, decision tree, and random forest showed comparable results. When cross-validating with the D2 dataset, the confusion matrices were similar for most of the test cases, showing signs of overfitting to either the majority or the minority class. For most of the test cases where D1 was undersampled, the performance models on D2 showed relatively high FN scores and either signs of underfitting or overfitting to the minority class. This indicates that undersampling the majority class does not effectively capture the underlying patterns. When keeping D1 imbalanced, the performance on D2 showed definite signs of overfitting to the majority class for most test cases; however, this was less prevalent when oversampling with SMOTE and using the ANOVA feature set.

The similarities in the results are most likely because XGBoost and random forest are ensemble methods that are trained on multiple decision trees. However, random forest had a longer training time, leading us to discard it for the voting classifier.

Voting Classifier

The idea of introducing the voting classifier originated from our own ideas of creating a hybrid model that would combine the TN classifications of Naive Bayes with the TP of XGBoost. The voting classifier was chosen since it would train multiple models on the same dataset and then make a final prediction based on a voting system, which averages the probabilities provided by each model. The voting classifier was also trained on a combination of decision tree and XGBoost to give further context to the performance of the voting classifier as a whole.

Essentially, the goal of combining the models was to balance out their biases and variances, thus creating a more robust model that would not be as sensitive to the characteristics of a single classifier. This approach showed only slight improvement in some test cases, but it did result in the most favourable scores for detecting attack traffic correctly when validating on the D2 dataset. The most optimal performance came from training the voting classifier with decision tree and XGBoost on the D1 dataset that was oversampled with SMOTE, using the combined feature set of Pearson's and ANOVA.

This raises the question of why this combination resulted in the most favourable results. When the decision tree and XGBoost were trained on the same test case separately, they achieved very similar results. Since both models separately showed a clear bias towards

the majority class, it is possible that the addition of a soft voting classifier, which combines them to make an average of their predictions, helped dilute this bias. The combination of their outputs could capture a wider range of patterns. This effect was particularly notable when combined with SMOTE, which introduces new minority class samples that might be considered more original by the model due to them not being direct copies. When using RoS, the new samples are just duplicates, which likely does not capture any complexities of the minority class distribution. This lack of diversity might cause overfitting in the test case with RoS instead of SMOTE, as seen in 4.23.

5.2 Insights from the Feature Selection

For the feature selection, we initially focused on filter methods. Filter methods are based on statistics rather than the actual performance of the model at runtime. This approach allows for a thorough examination of feature characteristics while maintaining neutrality towards specific classifiers.

Balancing Uniqueness and Redundancy

Initially, the sequential purpose features were removed from the dataset since they would introduce a new feature value for each traffic instance without actually denoting anything other than the sequence of the traffic instances for administrative purposes. These features revealed more about the context of the traffic rather than the content. This would introduce a lot of uniqueness without revealing anything descriptive about the packet headers of the traffic instances.

In addition to this, we also wanted to discard the features with too much redundancy that would also not be descriptive of the individual packets of the traffic instances, but on the other end of the spectrum, in that these features would reveal information about the content of the traffic, but would not have enough differing values within the features to show any informative variance.

The Shannon entropy measures the average information content of the features, thereby quantifying which features carry more or less information. This was calculated for the features in order to identify which features did not have enough diverse feature values and which features did have a diverse amount of values and therefore more information content. This allowed us to remove any features where the distribution of values was too uniform and therefore would have a pattern that is too vague for predictive modelling. There is no correct way to set a threshold for this, but when we calculated the threshold values, a notable gap appeared between the lowest threshold values between 0-5 and the higher threshold values from 8 and up. This indicated a natural distinction for the diverse distribution of feature values, which is why we chose to discard the lower threshold values below 5.

Additionally, the generated features that were only present in D1 were also discarded.

This was done for the purpose of validating with D2—D2 did not contain these features, so if we had trained the model with them, it would not be possible to test the saved model on D2 without generating them. The choice not to generate them was made both because it would be extremely computationally expensive and because the set of features already in D2 covers what would typically be expected to be contained in a network traffic dataset; therefore, we did not find it necessary.

When looking at the selected features based on the entropy method described above, the chosen feature values are obvious choices for basing the prediction on diverse value patterns.

- *sport:* The source port depends on the type of communication and the device it originates from, allowing for a wide range of values.
- *dur:* The duration of the traffic instances can vary significantly in a network with multiple components.
- *mean, stddev, sum, min, max:* All based on the durations of a given group of aggregated records, where the aggregation is based on the records having the same source IP, destination IP and transaction state.
- *rate:* Different devices and systems will generate traffic at different rates—a thermostat may send and receive data more frequently than motion-activated lights, for example.
- *srate:* This is specifically for the packets transmitted by the source and also varies depending on the source.

As seen in the table, the reduction of the feature set based on entropy did introduce lower performance when training on D1, but also introduced much higher performance when testing on D2 for Naive Bayes and XGBoost. For decision tree and random forest, the F1-scores were lower with the entropy-selected feature set, but the confusion matrices were generally more balanced, showing less bias towards either class.

Pearson's and ANOVA

We found the Shannon entropy to be appropriate for this feature selection since the measurement of the diversity of a feature is based on the single feature, and the entropy will be the same independently of what the diversity is of the other features. This allows us to gain a clearer understanding of its standalone importance in the dataset.

We also wanted to measure the correlation between features, but we wanted this to be a separate step in the process because we wanted a modular and easy-to-interpret approach to feature selection, allowing us to address different aspects of feature relevance systematically. Pearson's and ANOVA were regarded as appropriate for measuring the correlation of the variables in the dataset due to being widely used in the state of the art.

Pearson's measured the correlation between each pair of features of the independent variables. When measuring the correlation between these, we preferably want features that do not correlate too much, since this would indicate redundant information, similar to the reason for setting an entropy threshold. However, this type of correlation calculation allows for a lot of overlapping results between each pair of features. In order to assemble comparable correlation scores, the average correlation for each feature was calculated.

The features with the highest average feature correlation that were discarded were *max*, *mean*, *rate*, and *srate*.

For ANOVA, the correlation between each of the independent variables and the target variable was measured, in order to measure information gain from a feature through how much it affects the target variable, which we want to predict. For measuring this correlation, we aim for the highest measured scores.

The features that were discarded due to having too low a correlation with the target variable were *sport*, *min*, *srate*, and *mean*.

The models were also trained with the shared set of Pearson's and ANOVA. This was done to test if it would give the best results if both types of correlation calculations were accounted for when training the model; however, this would also introduce the possibility of "diluting" the correlation since some of the features included in the ANOVA subset for having a strong correlation with the target variable were not included in the chosen Pearson's subset for having a low correlation with the other independent variables. For example, *sport* was included in the Pearson's feature set but scored the lowest on the ANOVA scores. The features that are included in both ANOVA and Pearson's are only *sum*, *stddev*, and *dur*, but using only these for training the model resulted in only 15.91% of the normal attack traffic being classified correctly.

As stated previously, the best overall results in this project came from training the voting classifier with the combined set of features as defined by the ANOVA and Pearson's correlation scores. This resulted in 92.4% correctly predicted attack traffic classifications.

Wrapper

As mentioned previously, the results from filter methods are based on statistics rather than the actual run-time of training the model, which is where the wrapper methods become interesting. In order to give some more context to the field of feature selection, we decided to also find selected feature sets based on FFS and BFS, where the models are iteratively tested with new features that are only added to the final feature set if they improve the performance of the model. The initial results on D1 were near perfect, but for D2, the voting classifier failed to classify any attack traffic correctly when using the feature sets as defined by the wrapper methods.

The chosen features were features that would generally be considered informative when distingushing attack traffic from normal traffic, such as *saddr*, *daddr* and *srate*, however they were initially discarded from the filter methods. The *saddr* and *daddr* did not

have a high enough entropy score, which would mean that they introduce too much redundancy and the feature values are simply not informative enough. For *srate*, it was both too highly correlated with other features and not enough with the target variable.

This means that, according to our findings, the most useful features for detecting botnet traffic are *dur*, *stddev*, *sum*, *max*, *min*, *sport* and *rate*. The success of the filter methods was most likely due to them not relying of the performance of the model, rather than the statistical properties of the feature values and relevance both in relation to the other features and the target variable. This would create a more generalized and robust feature set with clear criteria that is not biased towards a specific type of model, performance metric or dataset imbalance.

sum, *dur*, *min*, *stddev* and *max* all relate to the duration of the traffic instances. This is relevant for detecting botnet traffic, since very short or long durations can indicate specific types of attacks. For example, a short duration with a high packet count might suggest a burst attack, whereas a long duration with a consistent packet rate might indicate a persistent intrusion or data exfiltration attempt.

sport is the source port - in IoT networks, certain devices typically communicate over specific ports. Unusual source ports could indicate compromised devices or unauthorized access attempts. For example, if a device that normally uses port 80 (HTTP) suddenly starts using a different port, it may suggest that the device has been compromised and is being used for malicious purposes, such as scanning the network for vulnerabilities or exfiltration of data.

rate describes the packets per second in the transaction, where a high rate can be indicative of high-volume attacks like (D)DoS, in which a large number of packets are sent to overwhelm the target. Alternatively, an unusually low rate might signify stealthy or low-and-slow attacks designed to evade detection.

5.3 Limitations of the Study

One of the initial limitations of this project established early in the process is the range of machine learning models used for the experiments. In order to train and test the KNN and SVM with a large dataset with many features, a significant amount of computing resources is necessary. An initial test run for each of these models was conducted to set a benchmark for their computational costs within our experimental setup, which showed that it would take many hours to train the models. This limited the scope of the project, especially since KNN was one of the models that generally performed the best according to the literature review. Both KNN and SVM are able to capture more complex and non-linear relationships in the data, which would be useful for recognising complex patterns in the network traffic.

Another factor that possibly limited the scope of the study was the feature set of the BoT-IoT dataset. In D1, a range of additional features were generated specifically for the purpose of multiclass classification of the specific types of attack traffic. These were not generated for the full dataset and therefore were not included in the trained models for this project. Many of the features that exceeded the entropy threshold, as presented in table 4.5, were from this generated set of features, highlighting them as possibly very relevant for training the models. A solution could be to generate them ourselves, but that would be a complex and cumbersome operation as the complete dataset is very large and all of the traffic instances would have to be taken into account when calculating them since they describe aggregated feature values. For instance, to calculate the total number of bytes per source IP, one would have to go through the whole dataset with more than 71.5 million records to find all instances with the given source IP. Nevertheless, the feature engineering of this project is fully based on the BoT-IoT dataset and therefore there is no guarantee that they would translate to a dynamic real-time traffic analysis environment and still be optimal for the performance of the model. Continuous updates based on both the threat landscape and analysis of the network traffic would be the best option for ensuring a relevant feature set for a real-life IoT network.

For this project, botnet detection is limited as a theoretical concept based on statistical measures and the runtime of the algorithms. This is a limitation in that it does not reflect a true IoT network scenario, even though certain steps were taken in the process to replicate this scenario. For instance, the classifiers employed for this project would have to be scalable as well as not require too many computational resources if they were to be implemented in an IoT network managing real-time data, which is also why KNN and SVM were excluded. However, for botnet detection in a real IoT network, additional security measures would have to be considered, such as any adversarial attacks designed to deceive machine learning detection techniques. The threat landscape is constantly evolving, so the botnet detection technique would not only need a robust detection mechanism to be successful but also need to be adaptable to changes and new evasion techniques. Since our solution is trained on historical data in a simulated setup, a future concern would be how to handle zero-day attacks that do not conform to the patterns recognised by the model. Furthermore, the evaluation of our models was limited to offline data analysis, which might not fully capture the more dynamic and unpredictable nature of live network traffic.

5.4 Future Directions and Recommendations

This section will explore directions for future research and further exploration of this project.

Deeper Classification Analysis

Using machine learning models for classification can feel like a black box approach, especially when dealing with a large dataset where it is impossible to check all misclassified instances to analyse and understand where the model became confused.

Doing a deeper analysis of both the specific instances that were misclassified, as well as the ones that were correctly classified, may lead to a clearer understanding of why this is happening and exactly what feature values are causing the models to overfit. This would require large-scale data analysis, since both datasets consist of close to 4 million traffic instances. The analysis of the feature value imbalance provided an insight into the patterns that confused the model in our test cases; however, in order to understand the full context and further refine the models, we would have to look more closely into the characteristics of the predictions.

Additional Datasets for Fine-Tuning

In this project, the evaluation of the models is based on testing the models on two datasets—each dataset imbalanced with different ratios, but with normal traffic as the minority class in both. For additional training and testing that would further confirm how robust the models are, it would be useful to test the trained classifier on a dataset that is not imbalanced and one that has normal traffic as the majority class, where the balance of the dataset is organic and not due to oversampling or undersampling.

Considerations of Computational Resources

The machine learning models and feature selection methods that were not chosen for our experiments, even though they were widely used in relevant literature — SVM and KNN — were primarily discarded due to being too resource-intensive. The evaluation of the resource consumption of the models adds further nuances to the performances of the models. For example, decision tree and random forest showed similar results for some test cases based on the performance evaluation scores, but the training and testing process for random forest consumed around 811 seconds, whereas the decision tree required around 7 seconds. This was underlined as a reason for random forest not being part of an optimal solution for a detection system; however, not much emphasis was placed on this aspect of evaluation. Highlighting the classification time and memory as crucial performance metrics for each test case could have distinguished one model as more favourable than the others and offered more nuance, rather than making them look as similar as they do with the evaluation metrics that we did choose to use.

This would have provided a broader understanding of the relative strengths and weaknesses of the models, which is especially useful to take into account when dealing with limited computing power, which is often the case with IoT devices and complex dataset, which is often the case with large amounts of network data.

Considerations for a Realistic IoT Botnet Detection System

If the findings of this project were to be implemented into a real-life botnet detection system for IoT devices, a hybrid approach may be used for the placement. By making part of the detection system edge- or cloud-based as well, the system would be able to handle large amounts of traffic, and most of the attack traffic would be detected early before entering the internal network. As we can see in the results for the voting classifier, when using the SMOTE oversampled dataset with the combined feature set as selected by Pearson's and ANOVA, aiming to predict attack traffic would result in both the majority of the normal traffic and attack traffic being classified correctly. This means the majority of the normal traffic would be accepted, while the majority of the botnet traffic would be deterred before reaching the internal network. According to the performance evaluation of this test case, a relatively large amount of more than 300,000 attack traffic instances were misclassified as normal traffic. If this misclassified botnet traffic were to enter the internal network or any IoT devices, a smaller and more lightweight system might not be sufficient to validate the traffic again. Therefore, the use of edge- or cloud-based resources for additional validation of any FN data might be an optimal solution in order to not saturate the possibly limited resources of the IoT network.

A relatively large amount of normal traffic is also misclassified as being attack traffic—more than 18% for the D2 test case. This would effectively mean that almost a fifth of the normal traffic would get discarded, which is another issue that should be handled in order to effectualise a botnet detection system and make it as user-friendly as possible.

When making an IDS scalable for IoT devices, neural networks may be an evident choice since neural networks can eliminate the need for manual feature engineering in favour of automatically learning any relevant features from the raw IoT data.

Incorporation of Neural Networks

Neural networks offer an approach for enhancing the scalability and effectiveness of botnet detection systems in IoT environments. Unlike traditional machine learning models that rely on manual feature engineering, as seen in this project, neural networks can automatically learn and extract relevant features from raw IoT data. This capability makes them particularly suitable for complex and dynamic environments where feature relevance may change over time. Future research could explore the implementation of various deep learning techniques such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs). CNNs can be particularly effective for analysing spatial data and patterns in network traffic, while RNNs are well-suited for sequential data and can help in understanding temporal dependencies in network activity. Leveraging these advanced neural network architectures can significantly improve the accuracy and robustness of detection systems in detecting sophisticated and evolving cyber threats in IoT networks[11].

Integration with IoT Device Firmware

According to the OWASP IoT top 10, many of the vulnerabilities in IoT devices are related to the device firmware[65]. Integrating botnet detection mechanisms directly into the firmware of IoT devices can provide an additional layer of security and enhance the overall defence strategy. With advancements in processing power and optimisation techniques, it is now feasible to embed lightweight machine learning models within IoT devices. This integration allows for real-time detection and mitigation of threats at the device level, reducing the reliance on centralised network-level defences and potentially lowering latency in threat response. Such an approach can decentralise the detection process, making the overall system more resilient to attacks that target network infrastructure. Furthermore, embedding detection capabilities in the firmware ensures that even if network connectivity is disrupted, the devices themselves can continue to monitor and respond to threats autonomously. Future research could focus on optimising these models for low power consumption and ensuring they can operate effectively within the limited computational resources available on most IoT devices.

Exploration of Adversarial Machine Learning

Adversarial machine learning represents a critical area of research for enhancing the security and robustness of botnet detection systems. Attackers often manipulate input data to evade detection systems, creating adversarial examples that can deceive even well-trained models[66]. Future work could focus on developing adversarial training techniques where models are exposed to such adversarial examples during the training process, thereby improving their resilience against manipulation. This involves generating adversarial samples that mimic potential attack patterns and incorporating them into the training dataset. Additionally, research could explore the use of defensive mechanisms such as adversarial detection, where models are trained to identify and filter out adversarial inputs. Understanding and mitigating the effects of adversarial attacks can lead to the development of more robust and reliable botnet detection systems capable of withstanding sophisticated evasion tactics employed by attackers.

Chapter 6

Conclusion

In this section, we will conclude on the problem formulation from section **??**. We will compare the three initial questions to our final results and their interpretation. The goal of this project was to research the field of botnets and how they can be efficiently detected with machine learning. To achieve this, we analysed the state-of-the-art works to examine the multitude of tools and directions to explore. We chose to work with the BoT-IoT dataset since it was widely used in the reviewed literature, along with four machine learning models—Naive Bayes, decision tree, random forest, and XGBoost. We wanted to focus on supervised learning algorithms with relatively low time complexity, and they were all extensively used as binary classifiers in the literary works. In addition, XGBoost was combined with decision tree and Naive Bayes in a soft voting classifier.

Initially, we trained and tested the models on a 5% sample of the BoT-IoT dataset (D1) as generated by the creators. However, this still left us with doubts as to how effective the models would be for detecting botnets in a real-life scenario, where the model would not only be used to classify the 20% test set of the dataset that it was trained on. Therefore, we decided to take a part of the full BoT-IoT dataset and create a new sample (D2) for further testing purposes to investigate how the training of the model can be optimised and fine-tuned to not only perform well on one set of data. Since both datasets were imbalanced, with attack traffic being the majority class, the F1-score as well as the confusion matrix were used for measuring performance. The additional model validation on D2 provided some substance to the performances of the models, as good results on D1 did not necessarily translate to good results on D2 due to overfitting. This gave us extra context for how to handle the dataset imbalance and what features to add or remove in order to achieve both sufficient and robust performance.

How can we ensure proper pattern recognition through training of machine learning models when handling imbalanced datasets, which are prevalent in botnet detection scenarios where normal traffic significantly outnumbers malicious traffic?

We found that the incorporation of an additional dataset gave us more context for optimisation of the model. D2 had a larger minority class than D1, requiring the model to consider a more widespread pattern for the minority class.

When studying the state of the art within the field of botnet detection, the performance evaluations left us with an inquiry as to what would happen if new data were introduced, in addition to the original dataset that the models were trained on. How is the issue of overfitting fully addressed if we have not observed a trained model in multiple contexts? This question led us to adding more testing data for further fine-tuning of the parameters.

Many of the initial test cases showed perfect results when training and testing on D1. However, when testing the trained model on D2, it resulted in a much lower performance evaluation, indicating overfitting to either the minority or majority class. This would not be feasible for a realistic botnet detection system, causing us to doubt the performance results of previous studies. We concluded that most of the test cases were prone to overfitting when looking at the TN and TP scores, but that the F1-score generally improved when adding entropy-based and correlation-based feature selection. The best test case that did not show obvious signs of overfitting, along with a high F1-score of 0.958, was the combination of XGBoost and decision tree in a voting classifier with the combined feature set of ANOVA and Pearson's, where SMOTE was used to oversample the minority class. This is most likely due to the combination of ANOVA and Pearson's that considers both the correlation between the features and the correlation between each feature and the target variable, as well as SMOTE, which oversamples the minority class by not just copying traffic instances from the minority class but using five samples of the minority class to create a "new" synthetic and randomised sample, introducing more variety rather than just exactly copying the already existing patterns.

In comparison, most of the test cases would have a relatively high TP score at the cost of a low TN score, and vice versa. Naive Bayes generally had the lowest performance evaluation and did not show much enhancement from being combined with XGBoost. Random forest was very resource intensive, and the performance evaluation was similar to XGBoost and decision tree, making it a redundant choice when factoring in the training time. Most of the other test cases with XGBoost and decision tree with the voting classifier did not improve much, as the performance evaluations were very similar to the performance from the test cases with only XGBoost.

Returning to the most optimal performance evaluation, which was the voting classifier with XGBoost and decision tree when training the model on the dataset oversampled with SMOTE and the combined feature set of ANOVA and Pearson's, we received an F1-score of more than 0.958 and a TN score of 5,688 and a TP of 3,689,362 in D2. This effectively means

that 81.2% of the normal traffic is classified correctly, while 92.4% of the attack traffic is classified correctly. For comparison, the same test case but with RoS instead of SMOTE resulted in 97.27% of the attack traffic being classified correctly, while only 28.94% of the normal traffic was classified correctly. This is an evident sign of overfitting, meaning that the trained model would most likely never conform correctly to newly introduced data. Although the RoS test case correctly classified a larger percentage of attack traffic, it has shown not to be an optimal solution for capturing the dataset's underlying patterns. The 303,635 misclassified instances of the attack traffic for the SMOTE test case would likely not be appropriate for a critical network; however, it does not exhibit signs of overfitting, providing a more solid foundation for potential further enhancements in classification.

What are the most effective strategies to optimise machine learning models for botnet traffic detection in IoT networks?

The selection of the most influential features proved valuable when training the models for the detection of botnet traffic. First, the complete set of BoT-IoT features was reduced based on the feature entropy. This seemed to eliminate overfitting for some of the test cases. Then, the feature set was chosen by calculating correlations based on ANOVA and Pearson's, and using the combined feature set. ANOVA measures the correlations between the features and the target variable, for which we want a high score to denote that the features are highly correlated with the class that we wish to predict. Pearson's measures the correlation between the features, for which we want a lower coefficient since a feature correlating too much with the other features denotes redundancy.

In addition, feature sets were chosen with FFS and BFS, which gave subpar results. The reason for this is most likely that when using wrapper methods, the iterations are not focused on a statistical baseline similarly to ANOVA and Pearson's, but instead are based on just optimising the given model with the given dataset at runtime. The resulting feature set will therefore be less transferable to another context, which in this case is a new dataset.

For the most optimal result, the feature set was the combined feature set as defined by both ANOVA and Pearson's. Additionally, SMOTE was used to handle the imbalanced dataset for the only test case that did not show evident signs of overfitting. This did increase the feature value imbalance, resulting in a still rather large number of misclassifications.

This leads us to the conclusion that filter methods perform better when validating on new data due to better generalisation and focusing on the statistical relationships both between the features and the target variable. This allows for a versatile and robust method that is not tied to a specific model's performance at runtime. Furthermore, using SMOTE to oversample the minority class and thus provide extra substance for recognising the pat-

terns also resulted in the most favourable solution for a botnet detection system.

Which network traffic features are most influential in identifying potential botnet activities in IoT networks, and how can these features be systematically evaluated and selected?

As mentioned above, the best performance came from the combined set of ANOVA and Pearson's. One might think that the intersection of ANOVA and Pearson's might give the best results since the features that are not included in both individual sets might skew the results as they did not get a high enough score for both. However, this was not the case, as using only the intersection of the ANOVA and Pearson's feature sets resulted in a very low FN score on both D1 and D2, suggesting that a feature set only consisting of three features is too small for the model to recognise the patterns of the minority class.

An initial removal of features was done since some of them were deemed irrelevant for the purpose of detecting botnet traffic. These were all "sequential" features, meaning that they represented an attribute about the order of traffic instance for administrative purposes. Two of these were unique packet IDs, *pkSeqID* and *seq*, while two of them represented the start time and end time of the instance in Unix time, which essentially is also a type of sequence number. For this dataset, it was used for calculating some of the other features, such as the three features that are the intersection between ANOVA and Pearson's feature set: *sum*, *stddev*, and *dur*. These all represent generated data based on the duration of aggregated traffic instances—allowing us to use these for detecting suspicious traffic based on grouped traffic instances. For instance, this would allow for easier detection of (D)DoS attacks.

Elaborating on the specific features, it is interesting that the *dur* feature is calculated to have a low correlation with the other features when quite a few of them are generated based on *dur* feature values. In addition, it is highly correlated with the target variable. This is most likely because the duration of a traffic instance will often vary widely and is often highly correlated with the attack type, giving indicators to whether the transaction is an attack and what kind of attack it could be. *sum* represents the sum of the durations of the aggregated records — a large duration sum may suggest data exfiltration where a persistent connection is needed, while a very short duration may represent brief scanning. *stddev* will represent the duration variability of the aggregated records, where a high standard deviation could suggest a burst attack. This feature proved to be an essential consideration for the FN classifications — many of the misclassified attack instances had 0 as a value for *stddev*. Since this feature denotes how "spread out" the durations are within the aggregated groups, the reason for this causing many FN classifications is probably because the *dur* feature is very vital in distinguishing attack traffic from normal traffic, so when there is no deviation of the durations because they are all the same within a group,

it is much harder to classify them.

These are the three features that both correlate highly with the target variable, which is a binary classification of normal or attack traffic, as well as having low correlation with the other features.

The additional two features that were added to the feature set through the ANOVA score were *max* and *rate*, meaning that they have high correlation with the target variable but are also above the threshold for correlation with the other features. *max* represents the upper duration bound of the aggregated records — if this is short, it can mean that the transaction state of the aggregated traffic instances does not need a longer maintained connection to be fulfilled, ruling out data exfiltration, for instance. *rate* is the packet rate per second in the given transaction — if this is high, it could denote a UDP attack, while a slow rate might indicate a HTTP request attack.

sport and *min* are the remaining features added to the final set through only having a low Pearson's correlation coefficient, meaning that they do not have a high correlation with the other features but are also below the ANOVA correlation threshold. *min* represents the lower duration bound of the aggregated records, which could for instance denote the difference between a burst attack or an HTTP POST request attack. Even though it might indicate a specific type of attack traffic, the lower bound of the durations can be influenced by outliers or edge cases that are not representative of the accumulated instances. If this feature captures some isolated instances of very low traffic durations, it might lead to low correlation with the other features. *sport* is the source port number — this might have low correlation with both the other features and the target variable since it is often randomised or dynamically allocated.

From this, we can conclude that a majority of the relevant features for the detection of IoT botnet traffic are based on the duration of the traffic instances. In addition to the correlation-based feature selection, wrapper methods were employed in order to test whether these two methods would complement or overthrow the statistical-based results. The wrapper methods utilised were forward selection and backward selection, and even though they managed to get a near-perfect performance evaluation on D1, the performance evaluation when testing on D2 was much lower.

This thesis aimed to explore how to efficiently detect botnet traffic in an imbalanced dataset by employing machine learning techniques. Throughout the course of the project, we explored numerous machine learning models and accompanying feature selection techniques. The analysis and experiments demonstrated that through the use of correlation-based filter methods, a soft voting classifier combining decision tree and XGBoost can correctly classify more than 92% of the botnet traffic in an IoT network, which was val-

idated by introducing the trained model to an additional dataset. The exposure of the trained model to a secondary dataset with new and re-balanced data points provided a novel approach to fine-tuning the classifier, giving further insight into the optimisation of the machine learning algorithm.

Bibliography

[1] Nickolaos Koroniotis, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset. *Future Generation Computer Systems 100*, 100:779–796, 2019.

[2] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[3] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. *Scikit-learn: Machine Learning in Python*, volume 12, pages 2825–2830. 2011.

[4] Guido Van Rossum. *The Python Library Reference, release 3.8.2*. Python Software Foundation, 2020.

[5] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[6] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.

[7] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[8] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[9] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[10] Matija Stevanovic and Jens Myrup Pedersen. Machine learning for identifying botnet network traffic. April 2013.

[11] Ying Xing, Hui Shum, Hao Zhao, Dannong Li, and Li Guo. Survey on botnet detection techniques: Classification, methods, and evaluation. *Mathematical Problems in Engineering*, pages 1–24, 2021.

[12] Saman Taghavi Zargar, James Joshi Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE Communications Surveys Tutorials*, 15(4):2046–2069, 2013.

[13] David Dagon, Guofei Gu, Cristopher P. Lee, and Wenke Lee. A taxonomy of botnet structures. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 325–339. IEEE, 2007.

[14] David Dittrich and Sven Dietrich. P2p as botnet command and control: A deeper insight. pages 41–48, 2008.

[15] Daniel Plohmann, Khaled Yakdan, Elmar Gerhards-Padilla, Michael Klatt, and Johannes Bader. A comprehensive measurement study of domain generating malware. In *This paper is included in the Proceedings of the 25th USENIX Security Symposium*, pages 263–278, 2016.

[16] Jing Qiu, Zhihong Tian, Chunlai Du, Qiang Zuo, Shen Su, and Binxing Fang. A survey on access control in the age of internet of things. *IEEE Internet Things Journal*, 7(6):4682–4696, 2020.

[17] Industrial internet of things market size, share trends analysis report by component, by end use, by region and segment forecasts, 2022 - 2030: Industrial internet of things market size, share trends analysis report by component (solution, services, platform), by end use (manufacturing, logistics transport), by region, and segment forecasts, 2022 - 2030, Sep 13 2022. Copyright - GlobeNewswire, Inc; Last updated - 2023-12-03.

[18] Muhammad Shafiq, Zhihong Tian, Ali Kashif Bashir, Xiaojiang Du, and Mohsen Guizani. Corrauc: A malicious bot-iot traffic detection method in iot network using machine-learning techniques. *IEEE Internet of Things Journal*, 8(5):3242–3254, 2021.

[19] Vitaly Morgunov and Yaroslav Shmelev. Iot threats in 2023. 2023.

[20] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. Security for the internet of things: A survey of existing protocols and open research issues. *IEEE Communications Surveys Tutorials*, 17(3):1294–1312, 2015.

[21] Omaima Bamasaq, Aiiad Albeshri, Li Cheng, Daniyal Algazzawi, and Majda Wazzan. Internet of things botnet detection approaches: Analysis and recommendations for future research. *Applied Sciences*, 11(12):5713, 2021.

[22] Ashley Woodiss-Field, Michael N Johnstone, and Paul Haskell-Dowland. Examination of traditional botnet detection on iot-based bots. *Sensors*, 24(3):1027, 2024.

[23] Sérgio S.C. Silva, Rodrigo M.P. Silva, Raquel C.G Pinto, and Ronaldo M. Salles. Botnets: A survey. *Computer Networks*, 57(2):378–403, 2012.

[24] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[25] Mads Stege, Peter Issam El-Habr, Jesper Bang, Nicola Dragoni, and Simon Nam Thanh. Survey on botnets: Incentives, evolution, detection and current trends. *Future Internet*, 13(8):198, 2021.

[26] Antonio de Paula Silva, Marcelo Rogerio Silva, Antonio Francisco Pinto, Renato Munhoz Salles, and Murilo Martinello. Botnets: A survey. *Computer Networks*, 57(2):378–403, 2013.

[27] Michael Bailey, Evan Cooke, Farnam Jahanian, Yunjing Xu, and Manish Karir. A survey of botnet technology and defenses. *Cybersecurity Applications Technology Conference For Homeland Security, CATCH*, pages 299 – 304, 2009.

[28] Ahsan Nazir, Jingsha He, Nafei Zhu, Ahsan Wajahat, Xiangjun Ma, Faheem Ullah, Sirajuddin Qureshi, and Muhammad Salman Pathan. Advancing iot security: A systematic review of machine learning approaches for the detection of iot botnets. *Journal of King Saud University - Computer and Information Sciences*, 23(10):32, 2023.

[29] Alaa Alhowaide, Izzat Alsmadi, and Jian Tang. An ensemble feature selection method for iot ids. *2020 IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys)*, pages 41–48, 2020.

[30] Zainab Alothman, Mouhammed Alkasassbeh, and Sherenaz Al-Haj Baddar. An efficient approach to detect iot botnet attacks using machine learning. *Journal of High Speed Networks*, 26(3):241–254, 2020.

[31] Sherif Saad, Issa Traore, Ali Ghorbani, Bassam Sayed, David Zhao, Wei Lu, John Felix, and Payman Hakimian. Detecting p2p botnets through network behavior analysis and machine learning. *2011 Ninth Annual International Conference on Privacy, Security and Trust*, pages 1–7, 2011.

[32] Taghi M. Khoshgoftaar, Joffrey L. Leevy, John Hancock, and Jared M. Peterson. An easy-to-classify approach for the bot-iot dataset. *2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI)*, pages 172–179, 2021.

[33] Satish Pokhrel, Robert Abbas, and Bhulok Aryal. *IoT Security: Botnet detection in IoT using Machine learning*. 2021.

[34] Nookala Venu, Aarun Kumar, and Sanyasi Rao Allanki. Botnet attacks detection in internet of things using machine learning. *NeuroQuantology*, 20(4):743–754, 2022.

[35] Jiyeon Kim, Minsun Shim, Seungah Hong, Yulim Shin, and Eunjung Choi. Intelligent detection of iot botnets using machine learning and deep learning. pages 324–327, 2019.

[36] Mustafa Alshamkhany, Wisam Alshamkhany, Mohamed Mansour, Mueez Khan, Salam Dhou, and Fadi Aloul. Botnet attack detection using machine learning. *2020 14th International Conference on Innovations in Information Technology (IIT)*, pages 203–208, 2020.

[37] Alejandro Guerra-Manzanares, Hayretdin Bahsi, and Sven Nomm. Hybrid feature selection models for machine learning based botnet detection in iot networks. *2019 International Conference on Cyberworlds (CW)*, pages 1–22, 2020.

[38] Moemedi Lefoane, Ibrahim Ghafir, Sohag Kabir, and Irfan-Ullah Awan. Machine learning for botnet detection: An optimized feature selection approach. *ICFNDS 2021: The 5th International Conference on Future Networks Distributed Systems*, page 195–200, 2021.

[39] Rajesh Kalakoti, Sven Nomm, and Hayretdin Bahsi. In-depth feature selection for the statistical machine learning-based botnet detection in iot networks. *IEEE Access*, 10(1):94518 – 94535, 2022.

[40] Mohammed Al-Sarem, Faisal Saeed, Eman H. Alkhammash, and Norah Saleh Alghamdi. An aggregated mutual information based feature selection with machine learning methods for enhancing iot botnet attack detection. pages 1–20, 2021.

[41] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer, 2013.

[42] Sunil Ray. Naive Bayes Classifier Explained: Applications and Practice Problems of Naive Bayes Classifier. `https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/`, 05 2024.

[43] Irina Rish. An empirical study of the naive bayes classifier. *IJCAI 2001 workshop on empirical methods in artificial intelligence*, 3(22):41–46, 2001.

[44] Trevor LaViale. Deep dive on knn: Understanding and implementing the k-nearest neighbors algorithm. `https://arize.com/blog-course/knn-algorithm-k-nearest-neighbor/`, 03 2023.

[45] Anshul Saini. Guide on support vector machine (svm) algorithm. https://www.analyticsvidhya.com/blog/2021/10/support-vector-machinessvm-a-complete-guide-for-beginners/, 01 2024.

[46] Anshul Saini. What is Decision Tree? [A Step-by-Step Guide]. https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/, 05 2024.

[47] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[48] Sruthi E R. Understand Random Forest Algorithms With Examples (Updated 2024). https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/, 05 2024.

[49] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[50] Introduction to xgboost algorithm in machine learning. https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/, 04 2024.

[51] Deval Shah. Logistic Regression: Definition, Use Cases, Implementation. https://www.v7labs.com/blog/logistic-regression#h1, 03 2023.

[52] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11:1601–1604, 2010.

[53] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[54] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[55] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[56] B. Venkatesh and J. Anuradha. *A Review of Feature Selection and Its Methods*, volume 19, pages 3–26. 02 2019.

[57] Sampath Kumar Gajawada. ANOVA for Feature Selection in Machine Learning']. https://towardsdatascience.com/anova-for-feature-selection-in-machine-learning-d9305e228476, 10 2019.

[58] Jason Brownlee. Racluster examples - argus- auditing network activity. https://openargus.org/oldsite/racluster.examples.shtml, 03 2012.

[59] Aayush Bajaj. Performance Metrics in Machine Learning [Complete Guide]. https://neptune.ai/blog/performance-metrics-in-machine-learning-complete-guide, 12 2023.

[60] Claude E Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[61] Jason Brownlee. The Complete Guide on Overfitting and Underfitting in Machine Learning. https://www.simplilearn.com/tutorials/machine-learning-tutorial/overfitting-and-underfitting, 03 2024.

[62] Jason Brownlee. SMOTE for Imbalanced Classification with Python. https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/, 03 2021.

[63] Jason Brownlee. Random oversampling and undersampling for imbalanced classification. https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/, 01 2021.

[64] MARCIN RUTECKI. Voting Classifier for Better Results. https://www.kaggle.com/code/marcinrutecki/voting-classifier-for-better-results, 02 2023.

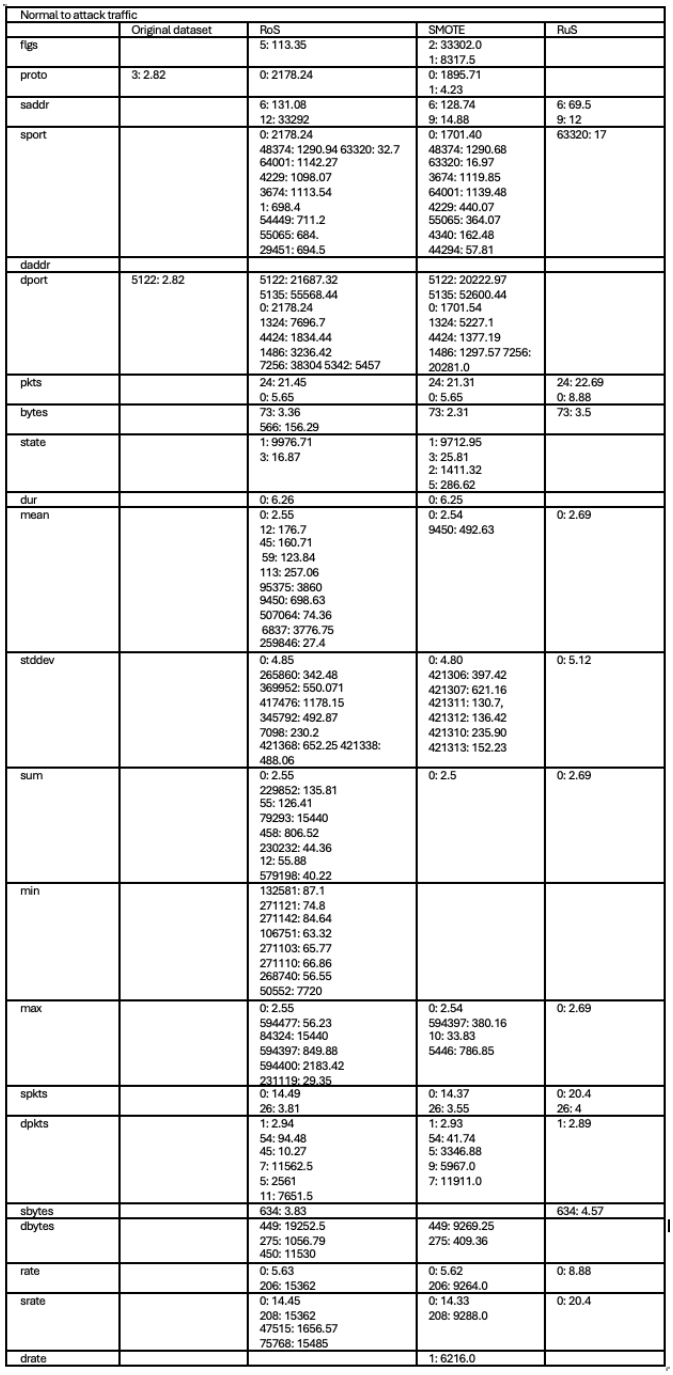
[65] The OWASP IoT top 10 vulnerabilities and how to mitigate them. https://www.sisainfosec.com/blogs/the-owasp-iot-top-10-vulnerabilities-and-how-to-mitigate-them/, 05 2023.

[66] Apostol Vassilev, Alina Oprea, Alie Fordyce, and Hyrum Anderson. Adversarial machine learning: A taxonomy and terminology of attacks and mitigations. *NIST Trustworthy and Responsible AI*, pages 1–99, 1 2024.

Appendix A

Appendix

b



Normal to attack traffic				
	Original dataset	RoS	SMOTE	RuS
flgs		5: 113.35	2: 33302.0 1: 8317.5	
proto	3: 2.82	0: 2178.24	0: 1895.71 1: 4.23	
saddr		6: 131.08 12: 33292	6: 128.74 9: 14.88	6: 69.5 9: 12
sport		0: 2178.24 48374: 1290.94 63320: 32.7 64001: 1142.27 4229: 1098.07 3674: 1113.54 1: 698.4 54449: 711.2 55065: 684. 29451: 694.5	0: 1701.40 48374: 1290.68 63320: 16.97 3674: 1119.85 64001: 1139.48 4229: 440.07 55065: 364.07 4340: 162.48 44294: 57.81	63320: 17
daddr				
dport	5122: 2.82	5122: 21687.32 5135: 55568.44 0: 2178.24 1324: 7696.7 4424: 1834.44 1486: 3236.42 7256: 38304 5342: 5457	5122: 20222.97 5135: 52600.44 0: 1701.54 1324: 5227.1 4424: 1377.19 1486: 1297.57 7256: 20281.0	
pkts		24: 21.45 0: 5.65	24: 21.31 0: 5.65	24: 22.69 0: 8.88
bytes		73: 3.36 566: 156.29	73: 2.31	73: 3.5
state		1: 9976.71 3: 16.87	1: 9712.95 3: 25.81 2: 1411.32 5: 286.62	
dur		0: 6.26	0: 6.25	
mean		0: 2.55 12: 176.7 45: 160.71 59: 123.84 113: 257.06 95375: 3860 9450: 698.63 507064: 74.36 6837: 3776.75 259846: 27.4	0: 2.54 9450: 492.63	0: 2.69
stddev		0: 4.85 265860: 342.48 369952: 550.071 417476: 1178.15 345792: 492.87 7098: 230.2 421368: 652.25 421338: 488.06	0: 4.80 421306: 397.42 421307: 621.16 421311: 130.7, 421312: 136.42 421310: 235.90 421313: 152.23	0: 5.12
sum		0: 2.55 229852: 135.81 55: 126.41 79293: 15440 458: 806.52 230232: 44.36 12: 55.88 579198: 40.22	0: 2.5	0: 2.69
min		132581: 87.1 271121: 74.8 271142: 84.64 106751: 63.32 271103: 65.77 271110: 66.86 268740: 56.55 50552: 7720		
max		0: 2.55 594477: 56.23 84324: 15440 594397: 849.88 594400: 2183.42 231119: 29.35	0: 2.54 594397: 380.16 10: 33.83 5446: 786.85	0: 2.69
spkts		0: 14.49 26: 3.81	0: 14.37 26: 3.55	0: 20.4 26: 4
dpkts		1: 2.94 54: 94.48 45: 10.27 7: 11562.5 5: 2561 11: 7651.5	1: 2.93 54: 41.74 5: 3346.88 9: 5967.0 7: 11911.0	1: 2.89
sbytes		634: 3.83		634: 4.57
dbytes		449: 19252.5 275: 1056.79 450: 11530	449: 9269.25 275: 409.36	
rate		0: 5.63 206: 15362	0: 5.62 206: 9264.0	0: 8.88
srate		0: 14.45 208: 15362 47515: 1656.57 75768: 15485	0: 14.33 208: 9288.0	0: 20.4
drate			1: 6216.0	

Figure A.1: Normal traffic feature value imbalance in D1

Attack to normal traffic				
	Original dataset	RoS	SMOTE	RuS
flgs	0: 4878.67 5: 67.5		5: 67.5	
proto	4: 5322.83 3: 35377.97 0: 3.53	3: 4.59	3: 3.59	3: 4.5
saddr	1: 19810.79 2: 16197.07 4: 14836.7 3: 18543.7 6: 58.57 9: 425.5 11: 28.72 12: 2.3	2: 2.10 1: 2.57 3: 2.4	1: 2.21 4: 4.03	1: 2.70 2: 2.07 3: 2.27
sport	63320: 235		5: 271.21	
daddr	16: 98987.87 18: 28229.87 21: 103713.75 13: 3859 10: 3657 12: 3110 11: 2968.0	16: 12.88 18: 3.68 21: 13.43	16: 13.69 18: 4.76 21: 17.51 20: 23.32	16: 13.29 18: 3.62 21: 13.5
dport	6650: 155371.82 3057: 2305.5 0: 3.53	6650: 20.23	6650: 29.16	6650: 20.21
pkts	88: 50394.6 110: 107099 62: 17778.35 44: 206198 1: 13186.63 117: 126888	88: 6.54 110: 13.94 62: 2.31 44: 27.12	99: 228.02 88: 16.2 71: 91.39 110: 594.16 62: 3.97 44: 15.22 4: 121.8 1: 3.8	88: 6.2 110: 13.5 62: 2.17 1: 2.36 44: 31.
bytes	1361: 74571.6 1172: 34618.42 1432: 25899.6	1361: 9.78 1172: 4.49 1432: 3.38	1287: 195.31 1361: 148.01 1545: 196.14 1172: 15.18 1444: 92.89 1469: 87.06 1597: 30.14 1602: 29.28	1361: 9.2 1172: 4.57 1432: 5.
state	4: 12795.2 8: 387583 3: 455.21	8: 50.5	7: 195.22 8: 387583	8: 52.5
dur	0: 1227.67			
mean	0: 3004.25 259846: 284			
stddev	0: 1585.69			
sum	0: 3004.22			
min	0: 14838.75 184179: 285			
max	0: 3004.22			
spkts	72: 46080.66 80: 211025 62: 69791.83 26: 2014.83 0: 530.56 46: 16900.77 14: 45831	72: 5.98 62: 9.08 80: 27.37 46: 2.19 14: 5.98	72: 13.4 83: 881.46 62: 27.66 80: 102.83 85: 969.56 3: 117.46 46: 5.71 14: 8.59	72: 5.58 80: 29.5 62: 9 46: 2.77 14: 5.33
dpkts	0: 15684.44 1: 2612.2 17: 14745.4 31: 4875.5 45: 749.8 54: 81.36	0: 2.04	0: 2.19 17: 3.46	0: 2 17: 2.2
sbytes			510: 422.75 804: 220.9 548: 352.08 644: 282.52 466: 113.81 686: 191.64 945: 33.11	
dbytes	0: 15684.44 375: 29508.95 25: 55291 102: 4755.37 194: 502.33	0: 2.04 375: 3.83 25: 7.21	0: 2.21 375: 4.19 25: 14.48 102: 2.14	0: 2 375: 3.9 25: 8.25
rate	0: 1365.17			
srate	0: 532.07			
drate	0: 8389.81			

Figure A.2: Attack traffic feature value imbalance in D1

d

New dataset, imbalanced		
	Normal to attack traffic	Attack to normal traffic
flgs		0: 296.48 9: 35423.66 12: 3762.22 6: 213.7 5: 53.25
proto		5: 9913.08 6: 7.34 1: 3523.77
saddr		5: 1772.91 6: 1722.2 7: 1640 8: 1610.38 10: 19.56 13: 66.15 15: 28.65
sport		82962: 59205
daddr		34: 7949.8 37: 9139.78 40: 61249.85 39: 205172.5 30: 611.18 31: 456.8 28: 475.75 29: 498.52
dport	97109: 5.26	127909: 23797.13 127908: 913.14 53201: 4490
pkts		168: 342.236 340: 2737.14 424: 10470.14 277: 23672.53 0: 597.35 459: 42028 532: 24784 1: 2928 509: 1445.81
bytes		278: 7690.23 3560: 1777.61
state		10: 176481.15 5: 13.1 4: 425.5 12: 3355.66
dur		0: 526.92 411341: 4267
mean		0: 246.64 280371: 4461 322024: 4342 281412: 4320 317738: 4308
stddev		0: 458.35 207119: 211 207103: 270.66 207099: 161.25 207102: 623 207097: 83.71 207101: 112.4 207106: 174 207100: 244.5
sum		0: 246.63 376572: 4290 336225: 4213 384423: 4191
min		0: 481.22 230100: 4784 180117: 4321 176884: 2156 203923: 4279 205417: 2102 207760: 2094
max		0: 246.63 218471: 4347 288331: 4290 223728: 4234 305558: 2097
spkts		1: 565.5 326: 7920.84 139: 164.16 367: 26211.14 247: 9025.71 383: 35278 405: 12979.6 24: 927.75 439: 81.2 2: 132.57
dpkts		1: 686.24 0: 441.49 62: 2231.61 161: 7996.81 109: 1627.02 185: 580.42 200: 2894.5 227: 1829 212: 115.73 239: 416
sbytes		2390: 6218.23
dbytes		1240: 8685.86 0: 441.49 79: 3966.21 322: 579.64
rate		0: 606.83
srate		0: 566.68
drate		0: 548.52

Figure A.3: The feature value imbalance in D2

e

pkSeqID	flgs	proto	saddr	sport	daddr	dport	pkts	bytes	state	dur	mean	stddev	sum	min	max	spkts	dpkts	sbytes	dbytes	rate	srate	drate	attack	category	subcatego
284	e s	tcp	192.168.100.149	52820	192.168.100.3	80	11	2708	RST	4.91103	4.91103	0	4.91103	4.91103	4.91103	8	3	2008	700	2.036233	1.425363	0.407247	1	DoS	HTTP
285	e s	tcp	192.168.100.149	52822	192.168.100.3	80	11	3516	RST	4.910644	4.910644	0	4.910644	4.910644	4.910644	8	3	2816	700	2.036393	1.425475	0.407279	1	DoS	HTTP
286	e s	tcp	192.168.100.149	52824	192.168.100.3	80	9	1473	RST	4.91041	4.91041	0	4.91041	4.91041	4.91041	6	3	773	700	1.629192	1.018245	0.407298	1	DoS	HTTP
287	e	tcp	192.168.100.148	51972	192.168.100.7	80	8	2265	RST	4.350441	4.350441	0	4.350441	4.350441	4.350441	5	3	935	1330	1.609032	0.919447	0.459724	1	DoS	HTTP
288	e s	tcp	192.168.100.148	51974	192.168.100.7	80	9	2712	RST	4.349761	4.349761	0	4.349761	4.349761	4.349761	6	3	1382	1330	1.839182	1.149488	0.459795	1	DoS	HTTP
289	e s	tcp	192.168.100.148	51976	192.168.100.7	80	9	2558	RST	4.349176	4.349176	0	4.349176	4.349176	4.349176	6	3	1228	1330	1.839429	1.149643	0.459857	1	DoS	HTTP
290	e s	tcp	192.168.100.148	51978	192.168.100.7	80	9	2022	RST	4.348809	4.348809	0	4.348809	4.348809	4.348809	6	3	692	1330	1.839584	1.14974	0.459896	1	DoS	HTTP
840288	e	arp	192.168.100.5	-1	192.168.100.149	-1	8	480	CON	4.283724	4.283724	0	4.283724	4.283724	4.283724	4	4	240	240	1.634092	0.700325	0.700325	1	DoS	UDP
1109	e	tcp	192.168.100.148	52266	192.168.100.7	80	8	2157	RST	4.127463	4.127463	0	4.127463	4.127463	4.127463	5	3	827	1330	1.695957	0.969118	0.484559	1	DoS	HTTP
1110	e	tcp	192.168.100.148	52268	192.168.100.7	80	8	2199	RST	4.126905	4.126905	0	4.126905	4.126905	4.126905	5	3	869	1330	1.696186	0.969249	0.484625	1	DoS	HTTP
1111	e	tcp	192.168.100.148	52270	192.168.100.7	80	8	2034	RST	4.126637	4.126637	0	4.126637	4.126637	4.126637	5	3	704	1330	1.696297	0.969312	0.484656	1	DoS	HTTP
1112	e	tcp	192.168.100.148	52272	192.168.100.7	80	8	2192	RST	4.12623	4.12623	0	4.12623	4.12623	4.12623	5	3	862	1330	1.696464	0.969408	0.484704	1	DoS	HTTP
1113	e	tcp	192.168.100.148	52274	192.168.100.7	80	8	2181	RST	4.125817	4.125817	0	4.125817	4.125817	4.125817	5	3	851	1330	1.696634	0.969505	0.484753	1	DoS	HTTP
1114	e	tcp	192.168.100.148	52276	192.168.100.7	80	8	2074	RST	4.12555	4.12555	0	4.12555	4.12555	4.12555	5	3	744	1330	1.696744	0.969568	0.484784	1	DoS	HTTP
1115	e	tcp	192.168.100.148	52278	192.168.100.7	80	8	2019	RST	4.125318	4.125318	0	4.125318	4.125318	4.125318	5	3	689	1330	1.696839	0.969622	0.484811	1	DoS	HTTP
1116	e	tcp	192.168.100.148	52280	192.168.100.7	80	8	2092	RST	4.125089	4.125089	0	4.125089	4.125089	4.125089	5	3	762	1330	1.696933	0.969676	0.484838	1	DoS	HTTP
844000	e	arp	192.168.100.149	-1	192.168.100.5	-1	7	420	CON	4.01502	4.01502	0	4.01502	4.01502	4.01502	5	2	300	120	1.494389	0.996259	0.249065	1	DoS	UDP
167186	e	arp	192.168.100.147	-1	192.168.100.7	-1	5	300	CON	4.00007	4.00007	0	4.00007	4.00007	4.00007	4	1	240	60	0.999982	0.749987	0	1	DoS	TCP
835437	e	arp	192.168.100.147	-1	192.168.100.7	-1	5	300	CON	3.99043	3.99043	0	3.99043	3.99043	3.99043	4	1	240	60	1.002398	0.751799	0	1	DoS	UDP
291	e	tcp	192.168.100.149	52826	192.168.100.3	80	8	1398	RST	3.886361	3.886361	0	3.886361	3.886361	3.886361	5	3	698	700	1.801171	1.02924	0.51462	1	DoS	HTTP
292	e s	tcp	192.168.100.149	52828	192.168.100.3	80	11	2828	RST	3.885858	3.885858	0	3.885858	3.885858	3.885858	8	3	2128	700	2.573434	1.801404	0.514687	1	DoS	HTTP
294	e s	tcp	192.168.100.149	52832	192.168.100.3	80	9	1496	RST	3.885313	3.885313	0	3.885313	3.885313	3.885313	6	3	796	700	2.059036	1.286898	0.514759	1	DoS	HTTP
293	e s	tcp	192.168.100.149	52830	192.168.100.3	80	11	3176	RST	3.885272	3.885272	0	3.885272	3.885272	3.885272	8	3	2476	700	2.573822	1.801676	0.514764	1	DoS	HTTP
171688	e	arp	192.168.100.149	-1	192.168.100.5	-1	5	300	CON	3.870311	3.870311	0	3.870311	3.870311	3.870311	4	1	240	60	1.033509	0.775132	0	1	DoS	TCP
509668	e	arp	192.168.100.147	-1	192.168.100.7	-1	5	300	CON	3.717161	3.717161	0	3.717161	3.717161	3.717161	4	1	240	60	1.07609	0.807068	0	1	DoS	TCP

pkSeqID	TnBPSrcIP	TnBPDstIP	TnP_P SrcIP	TnP_P DstIP	TnP_Per Proto	TnP_Per_ Dport	AR_P_Proto_ P_SrcIP	AR_P_Prot o_P_DstIP	N_IN_Conn _P_DstIP	N_IN_Con n_P_SrcIP	AR_P_Proto_P _Sport	AR_P_Proto_P _Dport	Pkts_P_State_ P_Protocol_P_ DestIP	Pkts_P_State_ P_Protocol_P_ SrcIP	attack	category	subcate gory
284	123641	145654	630	754	825	825	1.24757	1.24099	90	76	2.23986	1.27818	754	628	1	DoS	HTTP
285	123641	145654	630	754	825	825	1.24757	1.24099	90	76	2.24003	1.27818	754	628	1	DoS	HTTP
286	123641	145654	630	754	825	825	1.24757	1.24099	90	76	1.83284	1.27818	754	628	1	DoS	HTTP
287	19405	19405	71	71	825	825	1.87472	1.87472	8	8	1.83889	1.27818	71	71	1	DoS	HTTP
288	19405	19405	71	71	825	825	1.87472	1.87472	8	8	2.06908	1.27818	71	71	1	DoS	HTTP
289	19405	19405	71	71	825	825	1.87472	1.87472	8	8	2.06936	1.27818	71	71	1	DoS	HTTP
290	19405	19405	71	71	825	825	1.87472	1.87472	8	8	2.06953	1.27818	71	71	1	DoS	HTTP
840288	480	480	8	8	8	8	1.86753	1.86753	1	1	1.86753	1.86753	8	8	1	DoS	UDP
1109	42871	45082	154	164	962	962	1.66224	1.42371	17	16	1.93824	0.620291	164	154	1	DoS	HTTP
1110	42871	45082	154	164	962	962	1.66224	1.42371	17	16	1.9385	0.620291	164	154	1	DoS	HTTP
1111	42871	45082	154	164	962	962	1.66224	1.42371	17	16	1.93862	0.620291	164	154	1	DoS	HTTP
1112	42871	45082	154	164	962	962	1.66224	1.42371	17	16	1.93882	0.620291	164	154	1	DoS	HTTP
1113	42871	45082	154	164	962	962	1.66224	1.42371	17	16	1.93901	0.620291	164	154	1	DoS	HTTP
1114	42871	45082	154	164	962	962	1.66224	1.42371	17	16	1.93914	0.620291	164	154	1	DoS	HTTP
1115	42871	45082	154	164	962	962	1.66224	1.42371	17	16	1.93924	0.620291	164	154	1	DoS	HTTP
1116	42871	45082	154	164	962	962	1.66224	1.42371	17	16	1.93935	0.620291	164	154	1	DoS	HTTP
844000	23460	23460	391	391	7	7	1.74345	1.74345	65	65	1.74345	1.74345	7	7	1	DoS	UDP
167186	61284	61284	401	401	5	5	1.24998	1.24998	100	100	1.24998	1.24998	5	5	1	DoS	TCP
835437	53760	53760	896	896	5	5	1.253	1.253	100	100	1.253	1.253	5	5	1	DoS	UDP
291	123641	145654	630	754	825	825	1.24757	1.24099	90	76	2.05848	1.27818	754	628	1	DoS	HTTP
292	123641	145654	630	754	825	825	1.24757	1.24099	90	76	2.83078	1.27818	754	628	1	DoS	HTTP
294	123641	145654	630	754	825	825	1.24757	1.24099	90	76	2.31642	1.27818	754	628	1	DoS	HTTP
293	123641	145654	630	754	825	825	1.24757	1.24099	90	76	2.8312	1.27818	754	628	1	DoS	HTTP
171688	62244	62244	417	417	5	5	1.29189	1.29189	100	100	1.29189	1.29189	5	5	1	DoS	TCP
509668	300	300	5	5	5	5	1.34511	1.34511	1	1	1.34511	1.34511	5	5	1	DoS	TCP

Figure A.4: Two tables showing a section of the D1 dataset, showing how some of the records may be aggregated based on source and destination address [1]