FPGA-BASED NETWORK TRAFFIC CLASSIFICATION

USING MACHINE LEARNING

by

Mohammed Elnawawy

A Thesis presented to the Faculty of the
American University of Sharjah
College of Engineering
In Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in
Computer Engineering

Sharjah, United Arab Emirates

November 2019

Approval Signatures

We, the undersigned, approve the Master's Thesis of Mohammed Elnawawy

Thesis Title: FPGA-Based Network Traffic Classification Using Machine Learning.

Signature	**Date of Signature** (dd/mm/yyyy)
Dr. Tamer Shanableh Professor, Department of Computer Science and Engineering Thesis Advisor	
Dr. Assim Sagahyroon Professor, Department of Computer Science and Engineering Thesis Co-Advisor	
Dr. Raafat Aburukba Assistant Professor, Department of Computer Science and Engineering Thesis Committee Member	
Dr. Usman Tariq Assistant Professor, Department of Electrical Engineering Thesis Committee Member	
Dr. Fadi Aloul Head, Department of Computer Science and Engineering	
Dr. Lotfi Romdhane Associate Dean for Graduate Affairs and Research College of Engineering	
Dr. Naif Darwish Acting Dean, College of Engineering	
Dr. Mohamed El-Tarhuni Vice Provost for Graduate Studies	

Acknowledgement

I would like to thank my parents, my sister and my family for their consistent love and support. Thank you for always believing in me and encouraging me to become a better person over the years. You are truly the most precious treasure.

I would also like to thank my advisors Dr. Tamer Shanableh and Dr. Assim Sagahyroon for trusting my work, respecting my opinions during discussions, providing knowledge, support, and motivation throughout the different phases of this scientific work. It was a great pleasure working with you on this research.

Moreover, I would like to thank Mr. Hammam Orabi for his continuous help in my research whenever I sought his guidance. Your help means a lot to me.

Finally, I would like to thank the American University of Sharjah and the Department of Computer Science and Engineering for providing me with all the required resources and equipment to conduct my research and projects.

Dedication

To my mother…

Abstract

Traffic classification is the process of associating network traffic with the application or group of applications that generated it. It is an essential part of network management at datacentres and network operators due to its importance in traffic shaping, bandwidth allocation, and cybersecurity. Several techniques were investigated by researchers to classify traffic accurately with methods based on machine learning achieving encouraging results. In this work, we conduct several experiments using naïve Bayes, support vector machine, k-nearest neighbour, and random forest trees on two traffic datasets which are both publicly available. While the first dataset was collected in an uncontrolled environment that resembles real network behavior, the second was captured using a highly controlled environment. In the experiments conducted in this work, we look at the classifiers' performance and their effect on the classification accuracy and F-score. We also assess the suitability of extracted features using feature selection techniques. Moreover, we determine the optimal percentage of packets within a flow that need to be considered while extracting flow-level features. It is observed that when a larger number of packets is considered, the classification performance improves, but the required processing delay increases. Thus, we argue that 60% of packets in a flow would be a good compromise that ensures high performance in the least possible time. Several graphs are generated during each experiment to investigate the effect of varying each parameter on the classification performance. The results of our experiments indicate that random forest outperforms all other algorithms achieving a maximum accuracy of 98.5% and an F-score of 0.932. Finally, since software-based classifiers are usually slow and hence incapable of coping with the increasing amount of traffic within congested networks, we implement a highly pipelined random forest classifier on a Field-Programmable Gate Array (FPGA). The implementation makes use of the parallel architecture of the FPGA in accelerating such a time-consuming task. The implemented design is capable of achieving an average throughput of 163.24 Gbps which is more than twice the maximum throughput compared to reported work. This enables datacentres to achieve efficient online traffic classification given the dynamic nature of modern networks.

Keywords: ***Traffic classification, machine learning, random forest, feature extraction, FPGA.***

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ASIC	Application Specific Integrated Circuit
CBFS	Consistency-Based Feature Selection
CLB	Configurable Logic Block
CPU	Central Procession Unit
DM	Data Mining
DPI	Deep Packet Inspection
FPGA	Field-Programmable Gate Array
GT	Ground Truth
HDL	Hardware Description Language
IANA	Internet Assigned Numbers Authority
IDS	Intrusion Detection System
IGFS	Information Gain Feature Selection
ISP	Internet Service Provider
KNN	k-Nearest Neighbor
LUT	Look-up Tables
MAWI	Measurement and Analysis on the WIDE Internet
MCPS	Millions of Classifications per Second
ML	Machine Learning
NB	Naïve Bayes
OSI	Open System Interconnection
QoS	Quality of Service
RAM	Random-Access Memory
RBF	Radial Basis Kernel
RF	Random Forest
ROM	Read-Only Memory
SLA	Service Level Agreement
SPSM	Static Power Save Mode
SVM	Support Vector Machine
SWR	Stepwise Regression
UNB	University of New Brunswick
WNIC	Wireless Network Interface Card

Chapter 1. Introduction

Internet has been one of the most important inventions of the twentieth century. It went through major developments and evolvements since its inception. It was initiated as an attempt to devise a means of communication between computer systems across the globe that is not restricted by any borders. According to Statista, the number of active internet users has grown tremendously from 1.5 billion to around 3.9 billion in just ten years from 2008 to 2018 [1]. This means that more than half the population on the planet are now active users of the internet. To cope with the increasing number of internet users, companies are constantly working on enhancing their internet speeds. According to Cable, a company specialized in broadband analysis, some countries are now enjoying fast internet speeds up to 60 Mbps on average, which is an enormous amount of traffic [2]. This shows that internet has matured to be the fastest and easiest way of communication between mankind in our modern world. Nevertheless, internet is not just a method of communication, but it also involves other services like file and resource sharing, browsing the World Wide Web, and many more. The prosperity of the internet and its growing speeds has allowed more traffic to flow in and out of the average computing device. As a result, the openness to such a new paradigm has allowed us to explore new innovations that were not previously possible. However, it also opens doors for potential threats and malicious attacks that did not exist before the birth of the internet to infect everyday users of the network due to the numerous cyber threats and attacks that are conducted daily on the web. Therefore, researchers have realized the need for proposing several traffic classification techniques that help manage and control the flow of network traffic in order to alleviate the risks involved with the potential threats.

1.1. Traffic Classification

Traffic classification is the association of network traffic with the application or category of applications that generated them (for example, Skype, HTTP, SMTP, video streaming, and so on). Traffic classification is significantly important in our highly dynamic digital world for several reasons [3]. These reasons include ensuring the Quality of Service (QoS) and Service Level Agreement (SLA). It is also essential for troubleshooting abnormal network behaviour during unexpected downtimes whereby network administrators could potentially use it to pinpoint points of failure within the

network. Moreover, it is also used for traffic shaping and bandwidth allocation which regulate the flow of network packets to ensure the compliance with a specific traffic profile. In simple terms, service providers and datacentres implement traffic shaping to limit the traffic rate to a specific level in case it exceeds a pre-specified threshold. Moreover, one of the main motives behind the popularity of traffic classification and perhaps the most important one is cyber-security. The ease of access to internet resources has enabled hackers to craft new mechanisms by which they can exploit the numerous vulnerabilities that help them break into any system that is connected to the internet. Therefore, there is absolutely no system that is 100% safe once it gets connected to the internet. As a result, traffic classification is very important for security purposes since it helps recognize malicious classes of traffic that include viruses, trojans, spyware and many others. Once a specific flow of traffic has been labelled as malicious, an intrusion detection system (IDS) can then block out the malicious classes before they reach the user.

Traffic classification techniques are divided into four main mechanisms that are extensively used to categorize incoming traffic. These mechanisms are port-based, deep packet inspection (DPI) based, heuristic-based, and Machine Learning (ML) based techniques [4]. Port-based classification techniques are largely reliant on the port numbers of the transport layer of the open system interconnection (OSI). This mechanism was very beneficial and accurate since network applications used to have fixed port numbers as assigned to them by the Internet Assigned Numbers Authority (IANA). It was then easy to classify the traffic types based on their port numbers. Take browsers for example, since they usually use port 80 for HTTP connections, therefore, upon coming across a packet with port 80 as its source or destination ports we can immediately classify it as a browser packet. However, as communication protocols evolve, applications started varying their port numbers dynamically in order to obfuscate any means of classifying their traffic. This renders classifiers incompetent to accurately categorize the network's traffic. To overcome this pressing challenge, researchers started investigating various alternatives to port-based classification until deep packet inspection has emerged as a potentially powerful option.

DPI focuses on invasively checking the payload of the traffic looking for known signatures that relate to specific applications in order to categorize it. DPI proved to be one of the most accurate mechanisms of traffic classification since it looks right into

the contents of the packets, however, it is also one of the most time and resource consuming techniques. This is because pattern matching on application signatures requires lots of computing power besides the time required to compare a signature to a database of pre-saved signatures for classification purposes. In addition, some people are worried about the privacy of their communicated data since they do not wish to give permission to authorities in order to monitor their messages. Consequently, applications started overcoming this classification mechanism through encrypting the payloads of their packets to protect their contents. Accordingly, encryption renders DPI completely impractical since we can no longer comprehend the payload's meaning. In order to overcome DPI's hunger for resources, researchers have looked into the use of heuristic techniques for classification which tend to consume lesser resources, produce the output in shorter time at the expense of sacrificing the quality of the classification results. As a result, according to several research works, heuristics resulted in very low accuracies compared to other techniques. Finally, the research community has spotted the power of machine learning (ML) and data mining (DM) techniques in extracting essential information from traffic packets that can help a computer learn and categorize several applications with extraordinary accuracies.

1.2. Machine Learning

Machine learning is the study of algorithms, statistical models and methodologies which are applied to datasets in order to investigate interesting relationships among them that help computers take actions based on experience without the need to explicitly program them to do so. Machine learning is divided into two types of learning, supervised and unsupervised. Supervised learning involves algorithms that are informed about the expected output of some training instances in advance. It will then try to find relationships between inputs and their expected outputs such that a generalized model can be built that could potentially be applied to test samples that are not known to the trained model. On the other hand, unsupervised learning does not know about the expected output of the instances available for training, therefore, it tries to group instances based on some similarity measures among themselves. Supervised learning consists of classification, regression, and association, whereas unsupervised learning consists of clustering. Classification is the process of assigning instances into discrete classes or categories (for example, low, medium, and high). Regression is similar to classification since it attempts to predict a specific output value for an

instance, however, it is still different from classification since the output belongs to a continuous scale and not a discrete set of classes (for example, predicting a student's CGPA). Clustering is a set of algorithms that tries to group instances together based on some similarity measures discovering useful relationships between the different instances within the same cluster (for example, citizens belonging to the same socio-economic levels). Association is the process of uncovering interesting knowledge from an existing dataset that was not previously known (for example, if the weather was windy then the temperature is not high).

This work focuses on applying several supervised machine learning algorithms including naïve Bayes (NB), linear support vector machine (SVM), 2nd order SVM, k-nearest neighbour (KNN), and random forest (RF) to two different datasets with five distinct classes each in order to classify their traffic traces using MATLAB and Python. The performance of each algorithm is then assessed using its classification accuracy and average F-score. Several histograms were plotted in order to perform some pre-processing and analysis which helped us uncover interesting information about the datasets in hand. Two algorithms are used for feature selection to find out the most influential features that affect the classification performance, stepwise regression and random forest feature selection. Several experiments are then conducted using the different feature combinations resulting from the feature selection, as well as, the full set of extracted features. Other experiments were also performed in order to investigate the effect of varying training set sizes (10% to 90% of the datasets) in addition to the effect of using varying number of packets (10% to 100% of the packets in each flow) in a traffic flow in order to extract flow-level features. The ultimate goal is to study the effect of varying these parameters on the five ML algorithms mentioned above and then selecting the best classifier that could potentially classify network traffic with the highest accuracy and F-Score. However, several studies have shown that implementing traffic classifiers in software is causing a bottleneck in congested networks due to the enormous amount of traffic flowing through an average network nowadays. As a result, in order to mitigate the bottleneck caused by software, this research work attempts to implement the best classifier on a field-programmable gate array (FPGA) to exploit its parallel computing capabilities such that we can support high throughputs to enable real-time classification of traffic traces in congested networks.

1.2.1. Naïve Bayes. Naïve Bayes is a very simple probabilistic classification technique that was introduced in the early 1960s that focuses mainly on applying Bayes theorem. This classifier makes two assumptions, firstly, it assumes that all attributes are equally important and hence uses all attributes without inherent feature selection. The second assumption of naïve Bayes is that all attributes are statistically independent which is what makes this classifier naïve in the first place. This is because this assumption is almost never actually true since some attributes will always be related to one another. However, naïve Bayes tends to work really well in different applications, so we decided to use it as a start to raise our baseline classifier from the random classification which leads to an accuracy of around 20% (because five classes are used) to a higher accuracy. According to the Bayes rule the probability of event H given evidence E is given by Equation (1):

$$P(H|E) = \frac{P(E|H) \times P(H)}{P(E)} \quad (1)$$

where $P(H)$ is a priori probability of H (probability of event before evidence is seen), and $P(H|E)$ is a posteriori probability of H (probability of event after evidence is seen). When adopting this theorem into the naïve Bayes classifier, event is our traffic class and evidence is the value of each attribute used to build the classifier. Since we have many features to consider while building this classifier the general form of Equation (1) can be extended as shown in Equation (2):

$$P(H|E) = \frac{P(E_1|H) \times P(E_2|H) \times \ldots \times P(E_n|H) \times P(H)}{P(E)} \quad (2)$$

where E_1, E_2, …, E_n represent the different attribute-value pairs.

The way this works is by computing the probability of each class of the five classes occurring given the attribute-value pairs. Eventually, the instance is then classified as the class with the highest conditional probability of all classes. This explains why this classifier typically works even though the independence assumption is strongly violated. It is due to the fact that classification using naïve Bayes does not require precise probability predictions but rather it assigns the instance to the class with the highest prediction.

The way this classifier handles data is through performing a frequency count on the categorical attributes, whereas numeric attributes are usually handled by trying to fit a probability distribution like Gaussian distribution to the numeric values.

Nevertheless, Gaussian is not the only distribution used by naïve Bayes, in fact it uses kernel density estimation which is a weighting function used to estimate random variables' density functions. Kernel density estimation usually tends to be less overfitted than other distributions while usually yielding more accurate results. Therefore, in our work we will be using kernel density estimation with all the numeric attributes. One of the major advantages of naïve Bayes is its ability to handle missing values as it simply does not consider them when computing the required probabilities. However, naïve Bayes suffers from a huge drawback when having several redundant attributes since they worsen the independence assumption even further.

1.2.2. Support vector machine. Support vector machine is another popular ML algorithm that, in its very basic configuration, tries to find a separating hyperplane between the instances of different classes such that any new instance would be classified according to where they fall around the plane. Therefore, unlike naïve Bayes, SVM is not a probabilistic classification technique as it does not compute any probabilities but rather assigns an instance to a class based on the separating hyperplane. In its default settings, SVM usually works as a binary classifier that handles only two classes. It also works as a linear classifier that attempts to find a linear hyperplane that separates the two classes. SVM finds the optimal separating hyperplane through maximizing the distance between the separating hyperplane and the critical points at the decision boundary which are known as support vectors. Therefore, SVM's task is to try to maximize the distance between the dotted line and the two solid lines in Figure 1.1. This results in a quadratic programming problem. However, SVM gained popularity since it is presumed to be one of the best text classifiers. This classifier works by computing the resulting outcome from running the test instance through the hyperplane equation. In case of a two-dimensional plane, the outcome is computed using a straight line and in case the value fell above the line then the instance is labelled as the class that resides above the line and vice versa.

There are two main problems with SVM in its basic setup, the first being its inability to separate non-linearly separable datasets. In this case, SVM will fail to find a plane that separates the two classes. Therefore, one of the configurations of SVM is to use a kernel-based classifier that tries to map the dataset into a higher-dimensional space in an attempt to find a hyperplane that could potentially separate the two classes in that space. Figure 1.2 depicts the issue of non-linear datasets. This issue is very

important to us because the nature of network traffic is usually non-linear, and hence we are required to use the more sophisticated kernel-based classifier in order to help us find the separating hyperplane. Therefore, in our experiments we try both the linear and the 2nd order polynomial kernel to assess the performance of non-linear SVM classifiers on the traffic dataset. The second problem with SVM is its inability to handle more than two classes since it is a binary classifier by definition. There are two main solutions to this challenge. One can use a one vs-one scheme which tries to build a classifier for each pair of output classes in the dataset. This usually leads to creating $\frac{n(n-1)}{2}$ classifiers where n is the number of classes in the dataset. Therefore, if we consider a dataset that has five classes each, we will end up with 10 one-vs-one classifiers. For example, Skype-browser classifier, Skype-Mail classifier, and so on. Testing is then performed through running the test instance through all 10 classifiers and then assigning the instance to the class that yields the highest count. The advantage of this approach is that it usually yields better accuracies since it depends on a majority vote mechanism. However, the drawback of the one-vs-one approach is the fact that it is very time and resource consuming to build such a high number of classifiers, in addition to the need to run the test instance through so many classifiers before classifying it. The second approach is the one-vs-all scheme which tries to build one classifier per output class whereby the target class is treated as the positive class and all other classes are treated as the negative class. This results in n classifiers where n is the total number of classes in the dataset. For example, Skype-others, Spotify-others, and so on. Therefore, each dataset in hand would require building only 5 classifiers which is half the number required for the one-vs-one approach. We can already tell the major advantage of one-vs-all compared to one-vs-one since we would need lesser number of binary classifiers and hence lesser computation resources and lesser time. However, this approach usually tends to result in less accurate classifications when compared to the one-vs-one approach. In this work, we decided to use the kernel-based one-vs-one approach to build the best classifiers in terms of classification accuracy since our network traffic dataset is considered a multi-class non-linearly separable dataset.

1.2.3. K-nearest neighbour. KNN is perhaps one of the simplest yet most effective ML algorithms when it comes to classification problems. It is considered as a non-parametric classification method that is not based on the parametric method of probability distributions. It is also known as instance-based learning or lazy learning

because the generated model from the training data is not a mathematical formula, neither a tree-based structure but it is actually the training instances themselves stored in the memory of the computer system. After that, when a test instance is to be classified, the classifier simply looks at the closest k neighbours and conducts a majority vote of the output class. The majority class is then assigned to the test instance as the classification output. In its very basic setup KNN computes the distance of the test instances from all the training instances. The distance between two instances can be defined using several distance measures like the Euclidean or the Manhattan distance depending on the problem in hand, however, Euclidean remains the most popular distance measure for KNN. Euclidean distance is very intuitive for numerical attributes since it is defined as shown in Equation (3):

$$Euclidean\ Distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (3)$$

However, for categorical attributes distance can be computed as 0 if the two instances have the same attribute value, and 1 if the two instances have different attribute values. The main configuration of KNN is to choose a value for K. The smaller the K, the more sensitive the classification is to noisy data. The higher the K the more smoothing happens, and we might end up underfitting the data instead of overfitting. In this work, we used the default configuration for K in MATLAB which is 1.

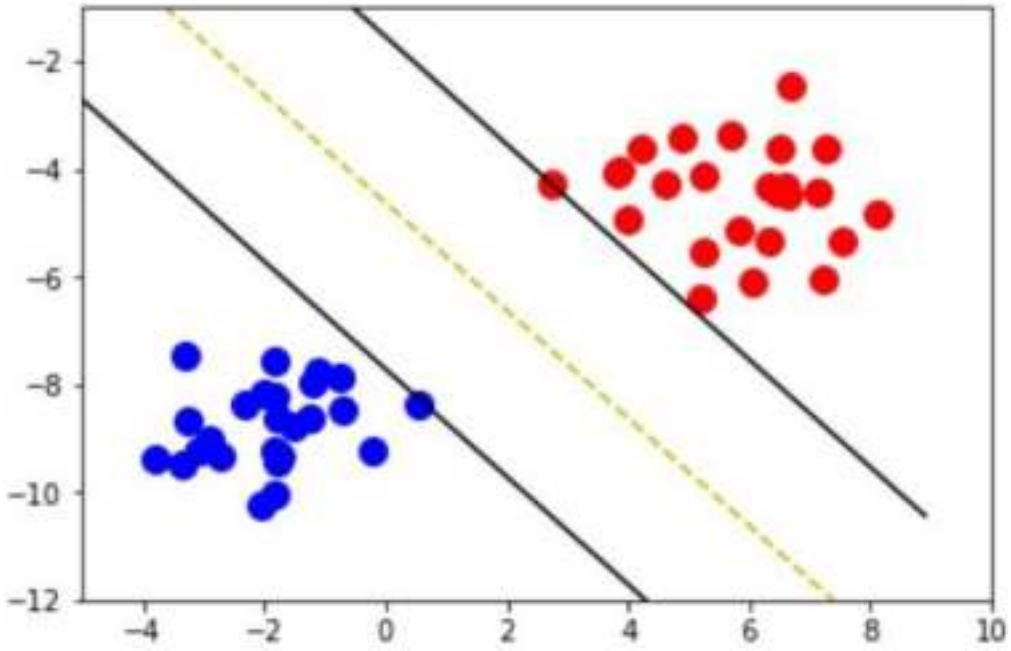


Figure 1.1: Linear SVM

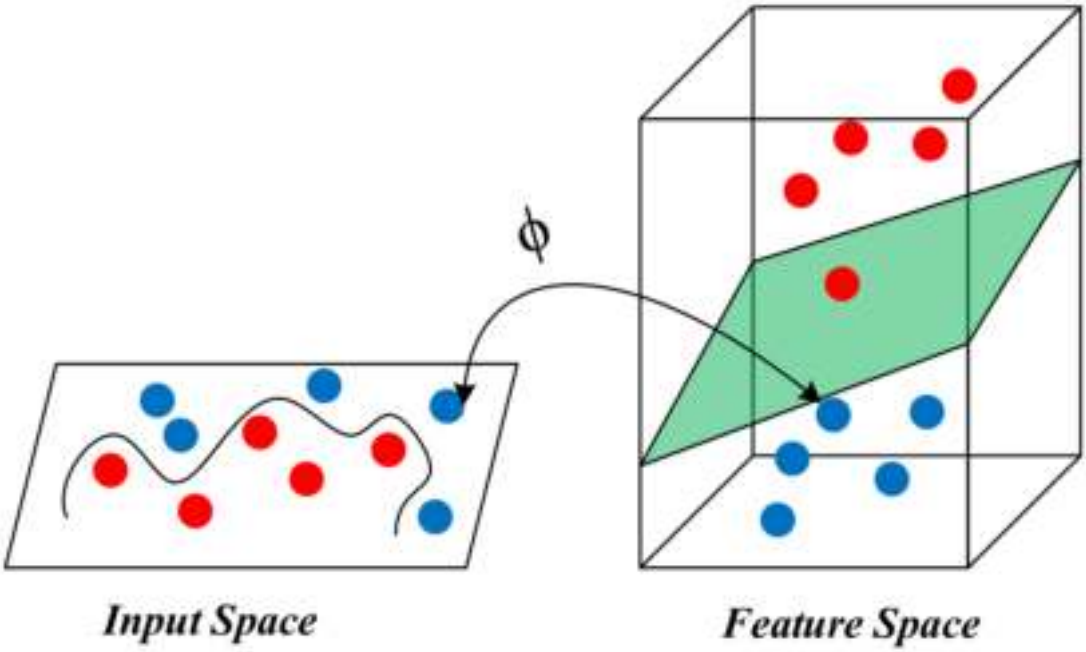


Figure 1.2: Non-linear SVM

KNN is a very powerful algorithm since it is very simple to implement and use. In addition, it is somehow easily comprehendible and logical to classify instances according to their closest lookalike. Therefore, it is easy to justify a decision to the customer in case this was used in a bank loan application, for example. Another major strength is the fact that it is usually very good at handling noisy datasets since it can take into account more than one nearest neighbour. On the contrary, it usually requires lots of memory to store all the training instances such that they can be used when classifying new instances. This is not the case with other classifiers that usually find a model that generalizes the behaviour of all training instances without consuming as much memory. Besides, it might also be computationally intensive since the classifier needs to compare a new instance to all the training instances in order to decide on the nearest neighbours. Therefore, the complexity could be in the order of O(n) where n is the number of training instances. Nevertheless, new algorithms have been devised to reduce the complexity to the order of $O(\log_2(n))$. These enhancements include the use of kD-trees and ball trees that break down the sample space into different non-overlapping regions that include the different training instances. Therefore, we can investigate only the regions close to the test instance instead of the entire sample space. This reduces the complexity significantly and hence simplifies the complexity of KNN classification.

1.2.4. Random forest. Random forest is an ensemble learning method that works by constructing multiple decision trees instead of just one tree in an attempt to enhance the classification performance. Random forests can be used for both classification problems and regression problems, therefore, if it was a classification problem, they output the resulting class based on a majority vote that picks the class that occurs the most from all decision trees. Another implementation of a random forest uses probabilities to choose the class with the highest probability across all trees in the forest. On the other hand, if we use it for regression then the output is simply the average of all the values generated by all the regression trees. Decision trees are very popular in the field of machine learning because they perform really well on non-linearly separable datasets. Keeping in mind that our traffic datasets are usually non-linear, this might indicate that decision trees are the best option to go about this problem. Besides, they are usually resilient to feature transformations and scaling, are really good at ignoring irrelevant attributes, and usually result in models that are easy to comprehend and inspect. Therefore, decision trees are usually preferred in applications like loan banking as one can easily explain their decision to a customer through the generated model. However, decision trees suffer from a great deficiency since they tend to overfit if grown too deep, which means that they could suffer from a very high variance in their predictions. This is because they can learn very rare and irregular behaviours in the datasets and hence might not generalize well on other datasets if they were not pruned properly. Nevertheless, random forests are always a great solution to the overfitting problem of decision trees. The mechanics of random forests allow the majority vote, probability calculation, or the average computation which are usually good ways of cancelling the effect of overfitting.

The main driver behind fighting overfitting is the bagging or bootstrapping method. Bootstrapping involves randomly sampling with replacement from the original training dataset in order to create several “bags”. Bagging is essential because it ensures that the variance is reduced and hence overfitting is resolved to some extent. After that, a decision tree can be fit to each bag and the output of the decision trees can be combined thereafter. In case we had to fit several decision trees to the original dataset we might end up with highly correlated trees since the dominating attributes would be the same in almost all trees. An even worse situation could happen if we end up with exactly the same tree multiple times. Therefore, bootstrapping is the mechanism by

which we create several datasets of the same size as the original dataset while decorrelating them. This will ultimately result in decorrelated decision trees and hence a more effective forest. Building decision trees usually depends on a purity measure of attributes at each split whereby we choose the purest attribute and split the training instances according to it. One of the most popular purity measures is the entropy which indicates the number of bits required to represent specific information. However, this is the only place where random forest differs from normal decision trees. Random forests usually choose a random subset of features at each split in the learning process in what is known as feature bagging. This step is also essential to avoid the correlation of the different trees since a purity measure has a very high chance of picking the same attributes at corresponding splits of different trees. The most important parameter in random forests is probably the number of trees used to build the forest. The following are the steps taken by a random forest classifier in building a generalized model that resembles any dataset:

Given the number of trees (N) and the number of training instances (m)

for n = 1:N

1. Sample, with replacement, m training instances from the training set
2. Train a decision tree f_n
    a. At every split pick a random subset of the features (default is the square root of the number of variables for classification)
    b. choose one or more features to split on based on some purity measure like entropy or Gini index that give the locally optimal condition for splitting

The previous algorithm will result in a random forest that looks like Figure 1.3.

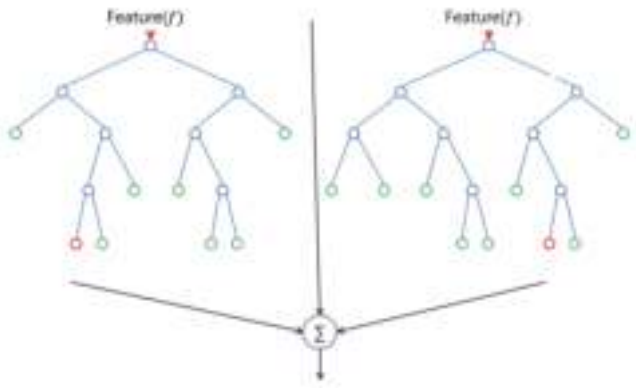


Figure 1.3: Random Forest

Finally, if we were to use a simple majority vote, when a test instance is run through the forest, each of the decision trees will result in an output class. The random forest classifier will then choose the most occurring class and declare the test instance as one of this class's instances. On the other hand, if we were to use a probability-based random forest algorithm, each tree in the forest will eventually output the probability of a test instance belonging to each class. For example, a tree's output would indicate that the test instance belongs to class Skype with a probability of 20%, belongs to class Browser with a probability of 30%, and so on. After that, the probabilities resulting from all trees will be aggregated and the class with the highest probability will be assigned to the test instance. In most cases, probability-based random forests perform slightly better than the simple majority vote ones. Therefore, in our work we examine both algorithms using MATLAB's majority-based library and Python's sklearn library which makes use of a probability-based random forest algorithm.

1.3. Field-Programmable Gate Array

Field-programmable gate arrays are semiconductor chips that consist of configurable logic blocks (CLBs) that are wired together using programmable interconnects [5]. The overall structure of an FPGA is shown in Figure 1.4. This enables CLBs to be reprogrammed in the field to perform different functionalities even after the chip has been already manufactured. This is the major advantage of FPGAs over application specific integrated circuits (ASICs), since ASICs are burnt on silicon to perform a single operation. Central processing units (CPUs) are considered one form of ASICs since their architecture restricts them to perform specific operations as entailed by the instruction set supported by their design. Therefore, digital circuit designers exploit the re-programmability feature of FPGAs to implement and extensively test their designs before manufacturing the actual ASICs. This helps cut down the huge cost of manufacturing the chip and then testing it before introducing it to the market, which in turn lowers the development cost and shortens the time required to market an FPGA-based product. Moreover, FPGAs offer a great feature compared to any CPU or ASIC in general, which is their ability to execute instructions in parallel powered by the independence of all of the CLBs. This means that each CLB can operate on a specific task regardless of the activity of other CLBs. This enables true parallel execution as opposed to the sequential execution of instructions on a normal CPU. Consequently, FPGAs gained popularity in emerging fields that include machine

learning, cloud computing, aerospace and defence and many more. Table 1.1 summarizes the main differences between FPGAs and normal CPUs.

Table 1.1: FPGA VS. CPU Comparison

Criteria	FPGA	CPU
Instruction Set	Customizable	Fixed
Architecture Complexity	Low	High
Programming Language	Verilog, VHDL, System Verilog C	C, C++, Python, Java
Power Consumption	Relatively Lower	Relatively Higher
Use	Reconfigurable	General-Purpose
Marketing Time	Short	Long
Execution	Parallel	Sequential
Price	Expensive	Relatively Cheaper

CLBs consist of basic hardware components that include memory, look-up tables (LUT), logic and multiplier units. CLBs can be interfaced to external sensors and actuators using the wide range of programmable Input/output (I/O) ports. LUTs are arrays that replace complex and time-consuming operations by a quicker array indexing operation. They usually store the expected outcomes of a specific operation that correspond to particular inputs. In addition, each CLB has a devoted register file which is a storage component that performs a similar operation to Random-Access Memory (RAM). This enhances the speed of the FPGA even further. Besides, each CLB contains flip flops that help in implementing sequential digital designs. Moreover, CLBs usually contain multiplexers to route different inputs to the CLB's output. Figure 1.5 depicts a simple representation of the internal architecture of a CLB.

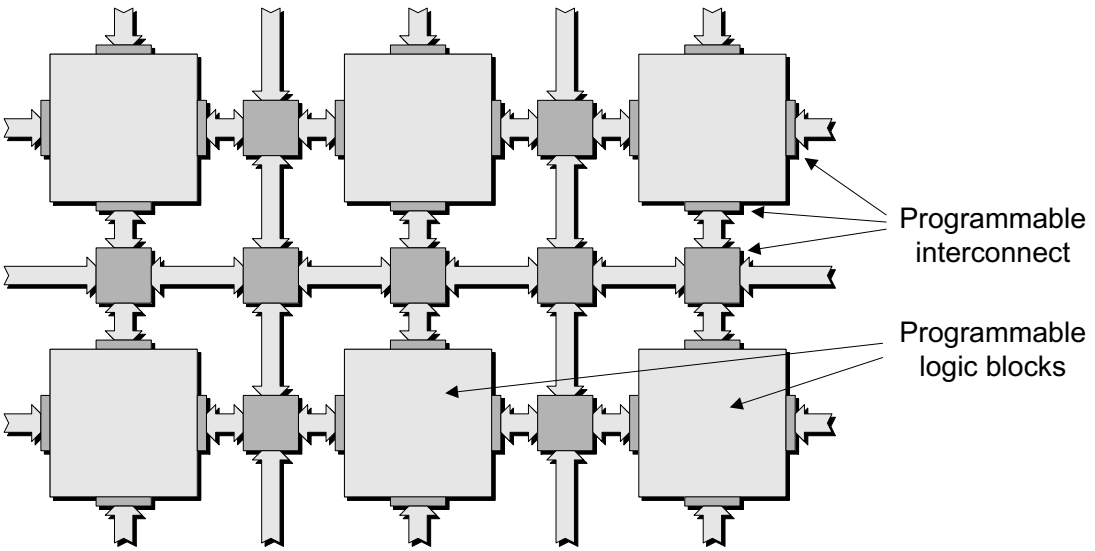


Figure 1.4: FPGA Architecture [6]

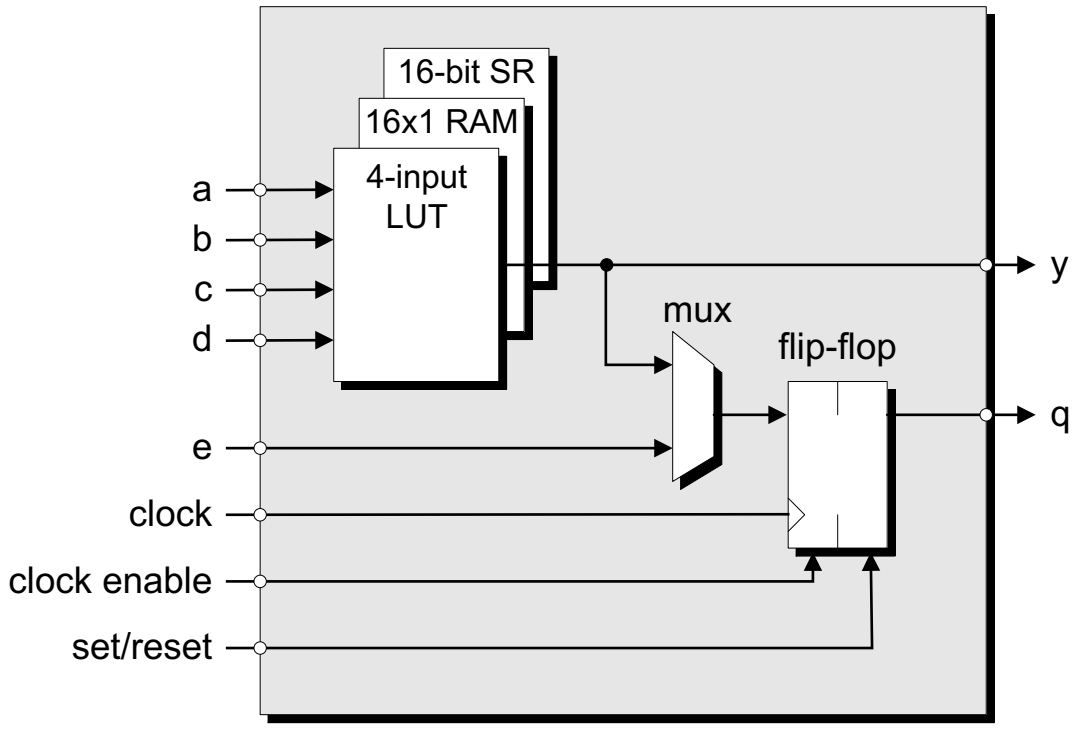


Figure 1.5: Configurable Logic block Architecture [6]

Chapter 2. Background and Literature Review

Traffic classification has recently drawn people's attention due to its importance in the networking field. Researchers have realized the need to devise new traffic classification techniques that could cope with the increasing speeds of modern computer networks. Traffic classification can be categorized into four different classes; namely port-based, DPI-based, heuristics-based, and ML-based. In this chapter, we review different mechanisms of traffic classification with emphasis on machine learning.

2.1. Port-Based Classification

Three major improvements in the networking field have affected the accuracy of port-based classification technique [7]. The creation of brand-new applications that have no IANA registered port numbers has forced these new applications to use ports that were registered to other applications. In addition, application developers started deliberately using port numbers that are known to be registered for other applications to disguise their traffic and bypass specific traffic filters. Moreover, the IPv4 address depletion problem causes many servers to be using the same public IP address while offering their services through different port numbers. All of these factors have rendered port-based classification incapable of correctly classifying network traffic. According to [8], port-based techniques are capable of identifying only 30-70% of the total network traffic.

2.2. DPI-Based Classification

It was 1998 when engineers discovered the power of DPI-based traffic classification [7]. DPI uses pattern matching to look for known application signatures in the payload of the network packets. Upon identifying a known signature, the packet is then classified as the traffic of the known application. Many attempts have been devised to implement DPI tools that can look into matching packets with their known signatures. One of the most popular tools of DPI-based classification is the L7-filter which is a Linux-based classifier that identifies packets based on their application layer data [9]. The DPI-based technique tends to offer a very high accuracy in classifying network packets regardless of their disguised port numbers, nevertheless, this method suffers from critical drawbacks that weaken the possibility of making use of such technique in real life systems. One of the major technical deficiencies of DPI is its

hunger for a very high memory consumption and processing time. Huge amounts of memory are required in order to store the different signatures of the applications under supervision, while a strong processing power is required because of the computationally intensive process of comparing a packet's signature to a huge database of known signatures.

Researchers realized DPI's need for a huge processing power and memory requirements. As a result, they started their search for optimized DPI-based algorithms which would reduce its resource consumption. In [10], the authors worked on designing a fast and memory-efficient system that leverages the capabilities of a multi-core architecture. They devised a new compression algorithm known as CSCA to perform regular pattern matching that reduces 95% of the memory consumption. Moreover, by introducing some optimization algorithms they were able to accelerate the matching speed such that the total throughput achieved was in the order of Gbps in 4-cores Servers. Besides solving the processing and memory requirements issue, this paper neglects another major drawback of DPI which resides in its offensive violation to privacy terms and conditions. By definition, DPI invasively looks into the content of the network packets in order to find known signatures. This puts the privacy of the communicating parties at risk because their information become vulnerable to potential threats and eavesdropping by the monitoring applications. To counter this issue, various network applications started encrypting the payloads of their packets in an attempt to deny DPI its advantage of getting to know the content of the transferred message [11]. A similar issue is using a proprietary protocol to transfer packets of a specific application. These protocols are usually not known to most filters and firewalls which also means that they will not be recognized and hence the packets will not be classified correctly. Therefore, with the wide use of encryption and proprietary software and protocols in modern networks, DPI does not stand a chance of being used in real life applications anymore.

2.3. Heuristic-Based Classification

DPI-based techniques usually tend to result in very high classification accuracies provided the traffic traces are not encrypted and use standard protocols. However, they are very demanding in terms of resource utilization and time required to classify the traffic traces. As a result, researchers turned their attention towards heuristic techniques that can perform a similar functionality to that of DPI. Heuristic techniques

are experience-based techniques that are used to cut down the time required to perform a computationally intensive task at the expense of sacrificing the quality of the result. They do not always guarantee a correct solution, but they tend to provide a good approximation of the actual solution. Besides solving the time and resource requirements issue of DPI, heuristics also resolve its privacy problem. This is because heuristic techniques are usually more concerned with the nature of the traffic flow rather than invasively looking at the payload of the actual traffic traces.

Several research works have investigated the importance and the power of heuristic techniques. One of the most interesting writings on the use of heuristics in traffic classification proposes a novel multilevel mechanism that looks at traffic flows at different levels of rising details [12]. The three levels used in their work are the social, the functional, and the application levels. At the social level, the authors capture the number of hosts with which the host under supervision communicates. This is called as the popularity of that host. At the functional level, they capture the role of a host within a network including whether that host is offering a particular service or consuming services offered by other hosts. For example, if a host uses a specific port number for most of its operations then it is most likely to be a provider host. At the application level, they look at the exchange of information at the transport layer in an attempt to identify the source application.

This approach comprises numerous advantages since it does not look at the packet payloads, as well as, having no knowledge of information apart from what is available to all flow collectors. However, the accuracy of heuristic techniques in this domain is usually not so promising since heuristics are prone to many errors in classification. This is shown by the accuracy of the proposed model in [12] which is about 95% accuracy on only 80 to 90% of the traces. This accuracy might be sufficient in some applications, however with the increasing number of threats on the internet nowadays, we would like to push the limits of traffic classification even further. This is because network administrators would not tolerate such a high level of inaccuracy which might compromise the security of their systems.

Therefore, the first three methods of traffic classification proved incapable of correctly classifying different types of traffic for reasons that were explained earlier. As a result, researchers have directed their attention towards devising new machine learning techniques or enhancing already existing ones in an attempt to enrich the

classification process. In Section 2.4, we discuss existing ML techniques that were proposed in the literature.

2.4. Machine Learning-Based Classification

The numerous challenges that come with the previously mentioned traffic classification techniques have encouraged researchers to look into an alternative that could potentially overcome those difficulties. People realized the power of machine learning in correctly classifying traffic traces since it relies mainly on flow statistics that are independent of port numbers or payload. Therefore, machine learning offered a greater flexibility in terms of the different features required to classify network packets. As a result, a classifier does not need to know the content of a packet in order to be able to classify it, but rather, the classifier will now be capable of classifying traffic traces by simply observing the behaviour of the network generated by the applications under supervision. This flexibility is also augmented with the speed required to construct an online classifier which can classify packets on the fly unlike the DPI-based technique that consumes a great amount of time trying to find a matching signature for the application. Moreover, machine learning techniques tend to offer a greater deal of quality results when compared to heuristic techniques. This is why the literature is rich with all sorts of machine learning algorithms that were adopted in the traffic classification domain. In this section we investigate the different machine learning approaches taken by researchers to implement a traffic classifier.

Machine learning algorithms are key players in classifying encrypted traffic by making use of their important statistical information. The statistical information consists mainly of flow-level features like packet inter-arrival times, average size, maximum size, and many more [13]. A flow is defined as a series of packets that share the same source and destination IP addresses, source and destination port numbers, and protocol. In addition, features can also be extracted at the packet-level whereby we obtain information like protocol, source and destination port numbers [14].

A published study examines the very common issue of huge power consumption caused by wireless Network Interface Cards (WNICs) in order to facilitate wireless communication between a mobile device and an access point [15]. Existing solutions include the Static Power Save Mode (SPSM) implemented by the 802.11 standard. SPSM allows WNICs to sleep in order to save energy and wake up periodically at every other beacon frame to receive any buffered packets. However, SPSM suffers from a

huge overhead due to sending a PS-poll frame from the mobile device to the access point requesting for its buffered frames. In addition, the huge delay between every other beacon frame causes SPSM to have a long waiting time in order to receive its buffered frames. In order to overcome the issues of SPSM, researchers proposed Adaptive SPSM which allows the WNIC to switch from sleep to awake mode based on a specific threshold of network activity regardless of the network type. This means that a low priority flow could wake up the WNIC which again does not optimize the power consumption. The authors suggest a context-aware listen interval algorithm that makes use of machine learning algorithms in order to classify traffic into four different priority levels. Based on the priority level the system will then decide whether to switch on or off the WNIC in order to minimize the power consumption while keeping the waiting time to a minimum in order not to suffer from long delays before receiving the buffered packers. For example, if the traffic was of the highest priority then the WNIC will always remain in the awake mode, whereas if the traffic was of the least priority the WNIC will have a very long sleep time before it periodically wakes up to check for the buffered packets. The sleep time will then vary according to the priority of the traffic flow.

In their work, they used the traffic generated by nine different applications which were then grouped into the four priority levels that form a dataset of 1350 instances [15]. VoIP applications constitute applications of the highest level, while offline gaming applications that only fetch advertisements when connected to the internet are considered of least priority. They also used some applications that represent a buffering stream as the second highest level of priority. Finally, a varied level of priority consists of applications that generate traffic levels which vary between the buffering stream and the low priority. The varying priority consists mainly of mailing applications or any sort of social media traffic that could potentially be generated at a very high rate at times of breaking news or at a low rate at steady times. In their work, the authors used two algorithms for feature selection; consistency-based feature selection (CBFS) and information gain feature selection (IGFS) this resulted in three different sets of features including the original set of features. The authors experimented with eight different machine learning algorithms including Naïve Bayes, C4.5, Random Forest, KNN and others. They assess the performance of the different algorithms mainly using accuracy and F-score. In general, KNN and Random Forest secured the

highest classification accuracies of 99.62 and 99.48 respectively. Their F-scores were also the highest compared to others with KNN scoring 0.996 and Random Forest scoring 0.995. On the other hand, Naïve Bayes was one of the worst classifiers scoring an accuracy of 93.25 and an F-score of 0.933.

Other research studies have looked into classifying traffic traces into more fine-grained categories. This means that the classification would assign packets to the applications that generated them as in [16]. In this work, the authors used two datasets; one of them is publicly available by the University of New Brunswick (UNB) which consists of 14 different classes including Skype, Filezilla, Facebook, Torrent, Twitter and many more. In their work the authors were able to extract an initial set of 111 features. After that, they used two feature selection algorithms from the WEKA tool to find out the reduced set of features that simplifies the complexity of the learnt model without compromising the quality and accuracy of classification. The CfsSubsetEval feature selection method resulted in a set of only three features, while the ChiSquaredAttributeEval method resulted in a set of twelve features. WEKA was then used to test four different algorithms, namely J48, Random Forest, KNN, and Bayes Net. The results of their experiments show that KNN (k=1) and Random Forest have the best performances with accuracies of 93.94% and 93.74% respectively.

Another study proposed an online Support Vector Machine (SVM) traffic classifier that was implemented using the CoMo project infrastructure which is an open source software for passive tracking of network traffic [17]. The authors chose SVM as a classification algorithm since it proved to be one of the best classification techniques, yet one of the most computationally expensive algorithms. They select eight different classes to work with which are web browsing, peer-to-peer, DNS, email, network operation, encrypted traffic, chat, and attacks. Each class is modelled as a test function that is used to determine the probability of an instance belonging to this class. CoMo consists mainly of five modules that include capture, export, store, query and supervise incoming traffic traces. Therefore, the paper attempts to build a feature extraction module on top of the CoMo infrastructure, as shown in Figure 2.1, that helps extract the necessary features required by the SVM classifiers.

One of the main objectives of their work is to optimize the code used for the SVM classification to reduce the time it takes to classify traffic [17]. As a result, they try to parallelize the operation of their SVM algorithm by off-loading the assessment

of each test function to a different thread. In testing the implemented system, they used a dual Xeon PC augmented with 24 cores running at a clock speed of 2.6 GHz and supported with 48 GB of RAM. They attempt to capture a 60-second trace of traffic and measure the time it takes to classify its packets into their respective application types. The results obtained from their experiments prove that the classification of the 60-second trace is completed within 21.5 seconds on average. In addition, results show that the system can work at speeds reaching up to 600 Mbps.

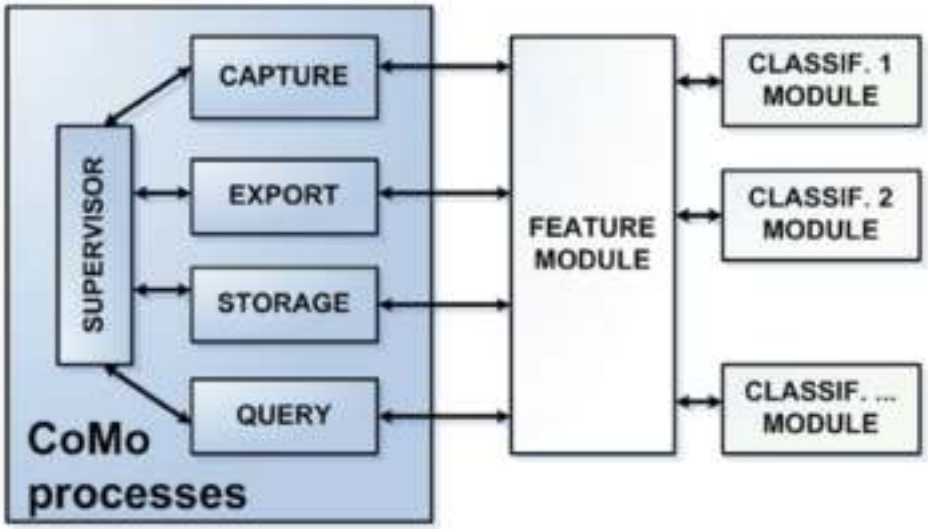


Figure 2.1: SVM Classifier Using CoMo Infrastructure [17]

This piece of work offers numerous advantages to the networking community [17]. One of the main strengths of this paper is their novel approach to classification whereby they demonstrate a hyperthreading mechanism that speeds up the execution of a computationally intensive ML algorithm like SVM. Besides, it also lays down the foundations of a software-defined classification approach that could support the increasing growth in network link rates. On the contrary, a great drawback of software systems is their deficiency when it comes to handling extremely high throughputs due to their limitation in terms of supported network speeds. This becomes a serious problem at datacentres and Internet Service Providers (ISPs) operating at 10s and 100s of Gbps. This limitation paves the way for the introduction of hardware accelerators like Field Programmable Gate Array (FPGA) that can be used to perform the computationally intensive ML operations at speeds of 100s of Gbps. The enormous speeds supported by FPGA accelerators allow the research in the traffic classification

field to prosper due to the huge capabilities offered. Therefore, more complicated ML algorithms can now be run directly on hardware eliminating the software bottleneck. Section 2.5 discusses innovations within this field that makes use of the power of FPGAs in traffic classification applications.

2.5. Hardware-Based Traffic Classifiers

FPGA accelerators are becoming very useful in speeding up computationally intensive operations which drew a lot of attention to its usefulness to overcome the software bottleneck of traffic classifiers. As a result, many researchers in this field started implementing their DM algorithms on FPGAs with the aim of improving the classification throughput. In this section, we look at some FPGA traffic classifiers, examine their strengths and weaknesses, and compare their performances.

2.5.1. C4.5 implementation. The first challenge in building a classifier is usually selecting the most appropriate features that will result in the highest classification accuracy. The system proposed in [18] suggests using eight candidate features, similar to the ones mentioned earlier in Section 2.4, that describe traffic traces. Next, they consider different combinations formed by these features such that they end up having six different feature sets. After that, they evaluate the performance of each feature set using the C4.5 decision tree algorithm which is shown to result in the best performance in their literature review. In order to evaluate those feature sets, the authors use the WEKA tool along with the very famous 10-fold cross-validation technique for offline training. Finally, the feature set that results in the highest overall accuracy (around 97%) was used in the final implementation of the classifier.

According to the methods reviewed by the authors in their literature review, a pre-processing discretization step helps improve classification accuracy significantly since it helps reduce the noise of the dataset [18]. For example, packet size could range from 0 bytes to +infinity bytes in theory and hence if packet size was discretized using fixed intervals of packet sizes this could help build a better classifier. However, the paper suggests that such a discretization step would consume a very high number of comparisons and hence they propose an Optimized Decision Tree (ODT) algorithm that integrates the discretization phase within the classification phase such that each non-leaf node in the binary tree will now compare the feature in hand to both the upper and lower boundaries of the discretization interval instead of actually discretizing then

comparing to only the discretized values. This cuts down the discretization overhead while still filtering the unwanted noise.

The ODT algorithm performs well as long as the tree is balanced, however, the complexity of imbalanced trees grows linearly with the number of tree levels [18]. This encouraged the authors to come up with a parallel data structure that makes use of balanced binary trees. Another technique suggested by the paper is the Divide and Conquer (DQ) algorithm which looks at creating multiple trees called as range trees that test for only one feature each. Each leaf node in a range tree stores a Bit Vector (BV) where each bit in that BV corresponds to an application type. Therefore, the output of each range tree will be a BV that consists of 0s and 1s to indicate whether an input matches the corresponding application type. Of course, the BV might show 1s for more than one application type and hence the BV is only considered as a partial result for this feature. The final classification result will then be obtained through performing a bitwise AND operation on all the resulting BVs of all range trees. The speedup with DQ is obtained through the fact that each binary range tree can be executed in parallel.

In order to optimize the previously mentioned algorithms for FPGA implementation, the authors decide to use pipelined architectures for both ODT and DQ whereby at each clock cycle one input is consumed and one output is generated [18]. ODT is implemented on an FPGA by mapping each tree level to a pipeline stage as shown in Figure 2.2 (a). Each tree node is stored in the Distributed RAM of the FPGA such that at each pipeline stage a Processing Element (PE) retrieves the tree node from the RAM and performs the necessary comparisons and then determines the node to be visited next. The input instance is fed to the next stage along with the address of the next node. On the other hand, each range tree is run simultaneously in parallel using the DQ algorithm, merging the resulting BVs eventually using a bitwise AND. Each level in a range tree is mapped to a different pipeline stage as shown in Figure 2.2 (b).

The results of the two implementations prove that DQ uses fewer FPGA resources when compared to ODT [18]. More importantly, DQ accomplishes a higher throughput when compared to ODT. This paper suggests a novel idea in mapping an ML algorithm onto a pure hardware implementation, especially with the DQ algorithm that parallelizes all operations by nature in addition to the use of a pipelined architecture that boosts the throughput. This technique offers a great advantage to datacentres and ISPs that handle enormous amounts of traffic every second since with such a high

FPGA throughput ISPs can quickly classify each traffic trace and take corrective actions when needed. Perhaps one of the drawbacks of such a paper is that it does not discuss the remedies if the bitwise AND results in a BV of only 0s which means failing to classify a traffic trace. Moreover, the paper starts by pointing out the need for an FPGA implementation which lies in the very low throughput of software implementations, however, the authors fail to compare their achieved throughput to that of software. Hence, towards the end of the paper the reader feels that a very crucial piece of information was left out.

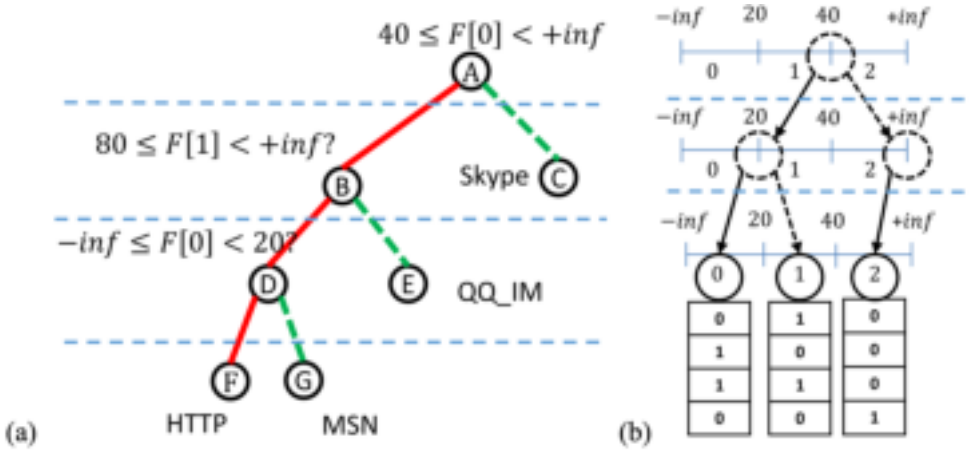


Figure 2.2: C4.5 Classifier on FPGA. (a) ODT Algorithm. (b) DQ Algorithm. [18]

2.5.2. SVM implementation. The study in [19] suggests a different approach to traffic classification which uses SVM on an FPGA accelerator. In contrast to the previous paper, the authors demonstrate both a software and a hardware implementation of their algorithm using NetFPGA 10 G FPGAs that incorporate four network interfaces at 10 Gbps each. Initially, the authors train their system offline using the LIBSVM library. The ground truth, which are the actual class labels of data instances, of the obtained traffic traces is usually determined using tools like L7-filter, GT tool, and inspection of system logs. There are two main stages in the classification process. Firstly, flow reconstruction which looks at the individual packets and tries to extract the flow-level features, and secondly, using the support vectors to classify those features. They used a computer that has two 6-core Xeon X5650 processors running at 2.66 GHz and 12 GB RAM. The results of their software SVM implementation shows that they were able to achieve a maximum throughput of 8.1 Gbps which can still cause a bottleneck at heavily occupied networks.

Figure 2.3 shows the hardware architecture used to implement the SVM algorithm on the FPGA. It consists mainly of the flow builder that performs flow reconstruction and then passes the flow-level features to the many computation units that perform classification. These computation units are duplicated such that they can exploit the highly parallel nature of the FPGA in order to make the classification process faster. Moreover, as an additional way of improving the throughput, the operations of the computation units are pipelined such that at each pipeline stage a new support vector is processed by each operation at every clock cycle. The result of each operation is fed into the next operation of the computation unit.

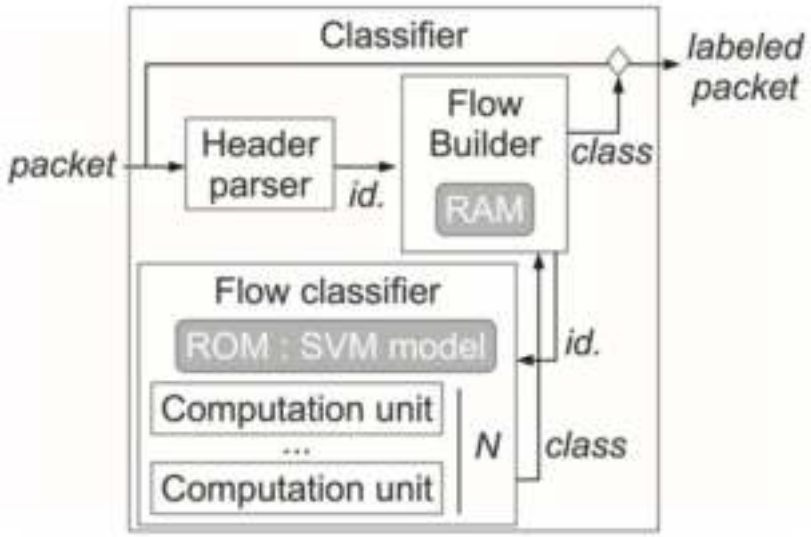


Figure 2.3: SVM Classifier on FPGA [19]

The collected traffic traces do not look linearly separable and hence the implementation initially uses Radial Basis Functions (RBF) kernel for the SVM classification. The results obtained with such a kernel show an accuracy of 95.8% on average. However, the authors suggest a different kernel that can be easily computed using the CORDIC algorithm as an alternative to RBF since it is more suitable for hardware as it does not include the computationally expensive exponent calculation. Upon implementing such a kernel, the accuracy of classification increased to almost 96.9%.

One of the weaknesses of their work is the fact that the generated SVM model was stored inside a Read-Only Memory (ROM) instead of the faster RAM. This entails incurring a very long time in the process of reconfiguring the FPGA in case the SVM

model needed to be retrained when new application types started showing up in the network.

2.5.3. Other implementations. C4.5 and SVM are not the only algorithms used for traffic classification. In fact, one of the very interesting papers to look at is [20] since it uses another well-known decision tree-based algorithm which is simple Classification and Regression Trees (CART). This paper follows a very similar approach to that used by the C4.5 implementation discussed earlier whereby they construct multiple range trees for each feature and then combine the resulting BVs towards the end. This FPGA implementation was able to achieve an accuracy of 96.8%. However, the most interesting finding of this paper is the fact that they tried most of the well-known ML algorithms on their dataset and summarized their results in Table 2.1. The Kappa measure usually indicates how well an algorithm performs compared to a random classifier, while the F-score incorporates both precision and recall in a single measure.

Table 2.1: Algorithms' Performance [20]

Algorithm	Accuracy (%)	Kappa	F-Score
C4.5	97.2	0.968	0.972
Simple CART	96.8	0.964	0.968
Simple CART (No Discretization)	93.9	0.931	0.939
KNN	96.1	0.955	0.961
Naïve Bayes	94.7	0.939	0.947

By carefully inspecting these results we can tell that decision tree-based algorithms usually perform better than others. This could be explained by the fact that the dataset in hand requires a non-linear classifier to discriminate between its classes and hence a decision tree is more effective in non-linearly separating the distinct classes. We can also notice that C4.5 usually performs better than Simple CART, which might be the result of having unstable trees generated by CART as opposed to the slightly more stable C4.5 trees. A very interesting observation is the fact that discretization helps significantly improve classification performance as shown by the discrepancy between the discretized CART and the non-discretized CART. This supports the idea mentioned earlier that discretization reduces noise in the dataset and hence results in a better classifier. KNN algorithm tends to show decent performance as a traffic classifier, however, it still falls behind decision trees. As the dimensionality of traffic instances increases, the distance between data points becomes less representative. Therefore, having many flow-level features might actually backfire

when KNN is used which is known as the curse of dimensionality. KNN is more susceptible to the curse of dimensionality than decision trees and hence this explains why decision trees are superior to KNN. Finally, naïve Bayes is proved to have the worst results in this domain. This could happen due to two reasons, firstly, the attribute independence assumption of naïve Bayes could be extremely violated and secondly, if the only way that separates applications is the way in which traffic features are correlated (for example, minimum, maximum and variance of packet sizes) then naïve Bayes will not result in an accurate classifier. Nevertheless, one of the main deficiencies of all the reviewed papers is overlooking the need to mention their sample size or the confidence level of their accuracies. As a result, this makes their results unreliable and their reporting process incomplete since we cannot really tell how good their classifiers are when it comes to a real dataset with an enormous number of packets.

Chapter 3. Problem Statement

After studying the reported results in the published literature that attempted to tackle the problem of traffic classification, we have identified a few shortcomings in their approaches. First of all, the reviewed work did not consider a systematic approach towards choosing the optimal number of packets to be considered within a flow in order to extract flow-level features. Most papers simply speak of the first *n* packets within a flow without defining a methodology to quantify *n*. The problem is that the more the considered packets, the better is the classifier's performance but also the more the time delay incurred when performing the classification. This entails the need to find out the most optimal number of packets that guarantees a reasonable classification performance while minimizing the incurred delay. In addition, the reviewed work did not consider some flow-level features when extracting the required features for classification. As a result, we propose new flow-level features that were not discussed earlier in the literature and we intend to study their effect on the classification performance. Moreover, another pressing issue is the fact that most of the literature in this field attempts to build classifiers that make use of the port numbers. At first sight, this might sound reasonable since it leads to a very high accuracy in most of the existing literature, nevertheless, these results are not too accurate. Researchers agree that port numbers are becoming an obsolete way of classifying traffic since applications try to dynamically disguise their used port numbers in order to obfuscate any means of traffic classification. Therefore, in the worst-case scenario, applications will be able to randomly choose port numbers in order to completely mislead the traffic classifiers. In such situations, port numbers become useless in the classification process as they might not carry any information related to the target application. Therefore, our aim is to build an efficient classifier that does not depend on port numbers in an attempt to make it ready for future enhancements in network applications.

Traffic classification is usually a time and resource consuming operation due to the need to extract several flow-level features that aid in correctly categorizing traffic traces. Therefore, the implementation and performance of a software-based traffic classifier has not proven to cope well with the enormous amount of traffic flowing in and out of several networks nowadays. As mentioned in the literature, software classifiers tend to form bottlenecks at congested networks, which hastened the need for hardware-accelerated traffic classifiers. Hence, another interesting shortfall in the

literature is the fact that almost all papers did not study and analyse hardware-implemented random forest network traffic classifiers. Even though the majority of papers agree that decision trees are the best algorithms used in the traffic classification domain, no one has ever looked into the implementation of a random forest traffic classifier on FPGAs. One of the main features of the random forest algorithm is its ability to generalize well and avoid overfitting to the training set when compared to single decision trees like C4.5. Therefore, in this research, we intend to study the properties and performance of random forests on some traffic traces compared to other popular machine learning algorithms like naïve Bayes, SVM, and KNN. In addition, we assess the overfitting properties of random forests through conducting several cross-validation experiments. If proven that random forest outperforms the other algorithms, we would then focus on fine-tuning the parameters of the generated random forest, like the number of trees, the number of features considered at each split, and the minimum leaf size, and study their effect on the classification performance.

In order to tackle the problem of bottlenecks introduced by software-based network traffic classifiers, we assess the performance of the hardware implementation of random forests on an FPGA in comparison to their software counterpart. Consequently, we compare the performance of the software model to that of the hardware implementation in terms of accuracy, F-score, and throughput to find out the obtained speedup using the hardware accelerator over the software implementation. This is carried out in order to find out the effect of a hardware-accelerated random forest network traffic classifier. In doing so, we intend to design a fast, efficient, and accurate random forest traffic classifier on an FPGA. As a result, we shall look into design techniques that make use of the parallel execution capabilities of FPGAs using pipelined architectures. This in turn would be a very important addition to existing traffic classification technologies and cybersecurity measures at datacentres and internet service providers.

Chapter 4. Datasets

In order to perform the experiments of this research work, we needed traffic traces to be able to build classifiers that can uncover relationships between the different instances. We had two options to collect traffic traces. The first option is to capture traffic traces using a special setup that will listen to a specific network channel to collect packets going through a particular access point. To do so, we might need to use packet capture software like Wireshark [21] or tcpdump [22]. This option would have been an easy one since we can build our own dataset at our convenience without the need to look up another dataset that was used in the literature. However, this option does not allow us to compare the obtained results to those of existing implementations of traffic classifiers since it will be the first work on such a dataset. Therefore, we chose another option which is to find out existing datasets that have been extensively used in the literature in order to be able to assess the performance of our classifiers against other researchers' work. In this chapter, we discuss three different datasets that were previously used in the literature and we discuss the feasibility of using each of them in the work presented here.

4.1. The MAWI Dataset

The Measurement and Analysis on the WIDE Internet (MAWI) group is a research group that has conducted several internet traffic experiments including analysis, evaluation and verification of several captured traffic traces since the inception of the WIDE Project [23]. The WIDE project focuses on evaluating the conditions of several real-world networks. They simply assess whether a network behaves in a similar manner to its design specifications, learn from abnormal network behaviour, and act accordingly. To support the continuity of research in the traffic analysis field the MAWI group has created and maintained a traffic data repository which contains several traffic traces that are collected with specific goals in mind. Some traffic traces resulted from a weekly capture experiment from the main IX link of WIDE to DIX-IE. MAWI call these traffic traces as samplepoint-G. Other traffic traces were captured on a daily basis since the 1st of June 2006 until today at the transit link of WIDE to the upstream link of the ISP. These traffic traces are called as samplepoint-F. To protect the privacy of the traffic owners, MAWI has made sure that all of their datasets are anonymized through scrambling the IP addresses in the packet headers.

At first sight, MAWI datasets look like a very precious treasure for any traffic classification project. Therefore, we had to dive deep into one of their capture files in order to assess whether they could be of any help to this research. We looked at the traffic traces in the PCAP file captured on the 31st of December 2016. The capturing process ran for almost 15 minutes and resulted in a PCAP file of almost 4 GB of data and around 65 million packets. The major strength of this dataset is the fact that MAWI has provided detailed statistics about each of the protocols included in the capture file. Therefore, according to MAWI, around 98.17% of the packets belong to the IPv4 protocol while 1.83% belong to the newly introduced IPv6. They also demonstrate similar statistics about other protocols like ICMP and IPSec. Moreover, they look deep into the transport layer protocols like TCP and UDP showing that their packets represent 53.81% and 6.92% of the total number of packets in the capture file respectively. Despite the abundance of information about the traffic traces collected by MAWI, they still lack one crucial piece of information that renders MAWI completely useless for our research. This work focuses on classifying traffic as per the application that generated such traffic. The problem with the MAWI dataset is the absence of any form of ground truth that could tell us the application that generated individual packets. For example, both Chrome and Skype use TCP, however, this dataset will only tell us that a packet uses the TCP protocol. Therefore, we still cannot differentiate between specific applications that use the same protocol. Hence, this dataset was finally discarded from our research work due to the absence of its ground truth.

4.2. The UNIBS Dataset

The telecommunication networks group at the University of Brescia in Italy is a research group that specializes in the field of computer and telecommunication networks [24]. The group is involved with several national and international research projects which resulted in several publications [25, 26]. Their most recent project revolves around a platform that combines software and hardware to accelerate the process of developing and testing wireless solutions. This project is sponsored by the EU Horizon 2020 program. They have also produced lots of useful software tools that are used in the networking field including Ground Truth (GT) which is a very efficient tool that captures and analyses traffic traces with their corresponding ground truth information [27]. To facilitate the continuity of the research in the networking field, this group has captured several traffic traces along with their associated ground truth

information collected using the GT tool. The traffic traces were captured at the edge router of the university's campus network on three consecutive days starting 30th of September 2009 until 2nd of October 2009. They used twenty workstations running the GT tool to collect the traces. As a result, the UNIBS dataset was captured in a non-controlled environment whereby all types of traffic were allowed to flow into the system with almost no restrictions, and then GT was used to obtain the ground truth of the captured packets. Therefore, the UNIBS dataset is more representative of real network traffic. Similar to MAWI, this group has also made sure that all of the captured packets are anonymized, as well as, payload-stripped in order to ensure the privacy of their owners. The anonymization process was conducted using a software known as Crypto-Pan. This dataset overcomes MAWI's greatest drawback since it includes its ground truth information which enables us to determine the application that generated almost every packet. The ground truth file was in the form of a text file that determines the application of each packet using the combination of source IP, destination IP, source port, destination port, and protocol.

The UNIBS dataset resulted in traffic traces of around 27 GB of data which resemble about 79000 traffic flows. After anonymizing the dataset and removing any payload related information, the traffic traces reached 2.7 GB. This dataset consists of both TCP and UDP traffic only with the majority of traffic belonging to TCP. Some of the provided statistics regarding this dataset show that almost 61.2% of the flows belong to browser applications which consists of both HTTP and HTTPS traffic, 5.7% are mail traffic (POP#, IMAP4, SMTP), 27.7% are torrent traffic, and 5.2% belong to Skype. These tend to be the most dominating applications in the UNIBS dataset and hence they will be our first choice of applications later on when we start the practical work. Upon inspecting the ground truth file of the UNIBS dataset, we found out a minor issue within the way the output classes are represented. We found lots of entries with different variations of the same application, for example, Skype appeared in the ground truth file Skype, Skype.exe, skype, skypePM.exe. This was also repeated with other classes like the browser class having variations like Safari, Safari Webpage, firefox-bin, firefox.exe. Therefore, before using the ground truth file we had to clean it up in such a way that these variations are replaced with a single class name for consistency purposes. Eventually, the resulting ground truth consisted of five main classes that were used in our work, namely, Browser, Skype, Mail, BitTorrent, and RSS feed. There were

also other classes that rarely appeared in the ground truth file and hence they were completely discarded when we filtered the UNIBS dataset. So far, UNIBS appears to be a very good start to our research work, and indeed it was. However, the main issue with the UNIBS dataset is the fact that it dates back to 2009, almost 10 years ago. Therefore, in the last 10 years different applications might have changed their protocols or their communication mechanisms. As a result, we do use UNIBS in our research, but we also tried looking for more recent datasets in order to cope with the recent advancements in the networking field.

4.3. The UNB Dataset

The Canadian institute for cybersecurity (CIC) is a broad multidisciplinary research and development institute that focuses on multiple research topics including social sciences, computer science, engineering, and many others [28]. The institute is based at the University of New Brunswick (UNB) in Canada where they harness all of their effort, energy and manpower to come up with new innovative research that could potentially benefit our dynamic world. Their aim is to become one of the pioneers in teaching and research methodologies in Canada by the year 2021 in the field of cybersecurity. As a result, they try to offer a competitive and well-organized environment for the best researchers to come together and devise new research methodologies in the field. Therefore, in order to aid the research in the cybersecurity field, the CIC has worked on collecting several network traffic traces such that they can be inspected and analysed for potential cybersecurity threats. Some traffic traces were collected to study the effect of intrusion detection and prevention systems. Others were obtained to investigate the impact of denial of service attacks and other malware on Android devices. However, the dataset of the most importance to us in this research is the VPN-nonVPN traffic dataset since it contains numerous interesting network applications like Skype, YouTube, Spotify, Torrent, and many others which serve the purpose of our research. The ground truth of the UNB dataset is also provided which makes this dataset very useful to our work. However, in order to record the ground truth, the collectors of this dataset used a much smarter way compared to the GT tool of the UNIBS dataset. While capturing the anticipated traffic all unnecessary applications and services were shut down in order to ensure that the generated traffic belongs only to the target application. Only then the capturing process was started. Therefore, the UNB dataset consists of several PCAP files where each file belongs to a

single application. The collection of the UNB traces was rather conducted in a very controlled environment that ensured that all services except the target service were shut down before the collection process started. Even though this might not be as representative of the real-life networks as the UNIBS dataset, it is still worth experimenting on such a dataset as it is more recent compared to UNIBS.

In capturing the traffic traces, the researchers used packet analysers like Wireshark and tcmpdump which resulted in a total traffic of 28 GB of data. Unlike the UNIBS dataset, the UNB dataset was not payload-stripped which means that the application layer data was present in all the capture files. Some statistics from this dataset show that VoIP applications including Skype represent the majority of around 20% of the captured packets. It was also interesting to find out that very popular applications like YouTube and Spotify appear in the UNB dataset since we believe that these modern-day services would add a great value to our research work as they help us cope with the advancing nature of internet traffic nowadays. Therefore, when picking traffic classes from this dataset we ensured that we choose the most trending and popular applications in order to add value to the traffic classification research community while maintaining some of the classes used from the UNIBS dataset for performance comparability purposes. We have selected five different classes from the UNB dataset, namely, Skype, Netflix, Torrent, YouTube, and Spotify.

In 2014, Microsoft has announced the use of a proprietary telephony protocol to replace the old protocol used in Skype. This means that datasets before 2014 including UNIBS would include skype traffic that uses the old protocol, which might render modern day traffic classifiers incapable of correctly categorizing Skype traffic. This issue might also be true for other applications. Therefore, the major advantage of the UNB dataset compared to UNIBS is the fact that this dataset was captured in 2016 which makes it quite recent compared to the 2009 UNIBS dataset. As a result, UNB represents a major player in our traffic classification research. However, one of the greatest drawbacks of UNB dataset is the presence of some impurities in the dataset. Even though the owners mention that they turn off all other services before capturing the target service, we were still able to find out lots of packets that did not belong to Skype, for example, in the Skype PCAP file. In order to verify this, we had to contact the owners of the UNB dataset, who said that they tried their best to remove all impurities, but they do not guarantee that it is 100% pure.

Chapter 5. Methodology

As mentioned earlier, this work focuses on traffic classification using machine learning which involves supervised learning methods including naïve Bayes, SVM, KNN, and random forest. In this chapter, we look into some of the pre-processing steps performed before running the datasets through the mentioned classifiers. In addition, we describe the detailed approach taken towards extracting useful features from the UNIBS and the UNB datasets. Moreover, we discuss the feature reduction algorithms used in order to choose the features that best represent the datasets in hand. We also provide a detailed flowchart that describes the entire process starting from the pre-processing, going through the training phase and finally terminating with the testing and evaluation phase. After that, we describe the mechanics of the four ML algorithms employed in this work.

5.1. Pre-processing Step

In this work, the MATLAB environment and the Python language are used throughout the different phases to implement the ML algorithms mentioned in Section 1.2. The UNIBS and UNB datasets come in the form of different PCAP files. Therefore, we needed a way of converting the PCAP files into a readable format to MATLAB and Python. This is because PCAPs were not directly supported by MATLAB and Python at the time this research was conducted. After that several pre-processing steps were used in order to filter the noisy instances, inspect the distribution of the resulting instances, and so on. Next, we describe a detailed step-by-step guide of the implemented procedure:

1. The PCAP file is launched using the Wireshark software to inspect the several frames contained within the file. Figure 5.1 shows the Skype file from the UNB dataset. As mentioned earlier, the UNIBS PCAP contained traffic capture of three consecutive days, resulting in a total of over 20 million packets. This was more than the required number of packets to perform our machine learning analysis. Therefore, we have decided to use a subset of around 100000 packets. There are two ways to filter such a subset, either by randomly sampling 100000 packets from the whole file or by selecting consecutive 100000 packets from one of the three files. While the first option would result in a more representative sample of the original traffic traces, it suffers from a very dangerous threat which is the fact that by randomly sampling the packets we might

end up losing the flow-level information which requires the packets to be consecutive in nature to avoid any interruption to the flow of traffic. Therefore, we decided to go with the second option and choose 100000 consecutive packets from the capture file.

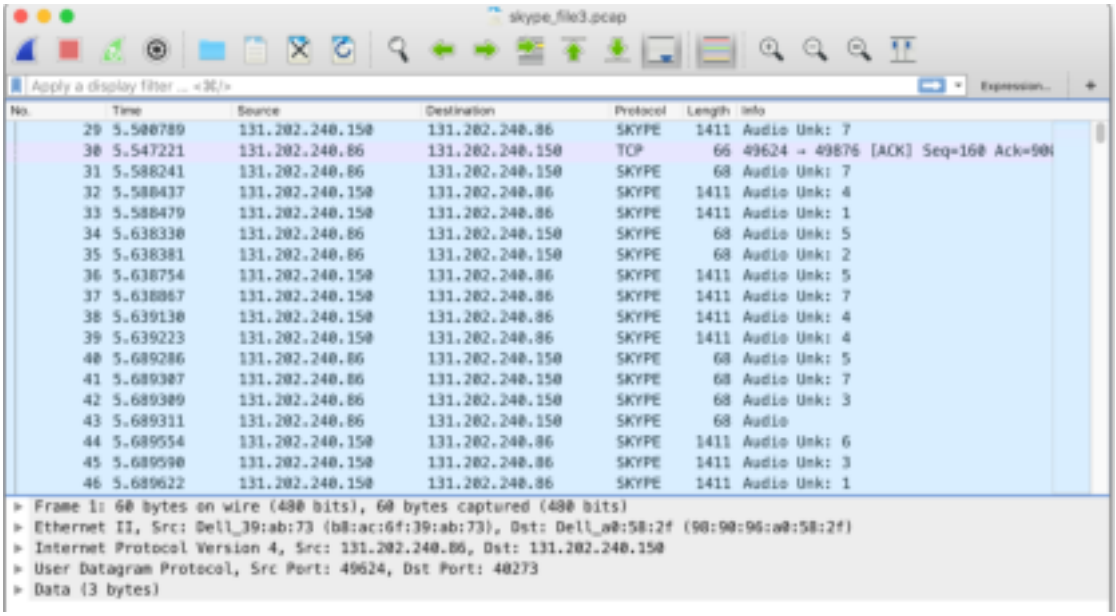


Figure 5.1: Skype PCAP File

Similarly, this was also done with the UNB datasets, except that UNB has different PCAP files for different classes of traffic. Therefore, in order to create a balanced dataset. The five files skype_file3, netflix3, spotify4, Torrent01, and youtube5 were chosen from the set of files of the UNB dataset and the first 20000 consecutive packets were chosen from each file resulting in a total of 100000 packets as well.

2. Wireshark was used to export the first 100000 packets in the UNIBS trace of the 30th of September 2009 into JSON format which is supported by MATLAB. Also, 20000 packets from each of the UNB dataset files were exported into JSON resulting in the UNB dataset that consists of a total of 100000 packets. Figure 5.2 and Figure 5.3 show the steps to export the packets into JSON.

3. JSON files are read into MATLAB where each packet is represented as a struct containing its header information.

4. UNIBS packets are labelled by looping through its ground truth file and locating source and destination IP addresses, source and destination port numbers, and protocol. If a match is found, then the packet is labelled using the class associated to the combination of the five attributes mentioned above.

On the other hand, UNB was very easy to label since packets come in different files already, therefore, upon reading each of the five files into MATLAB the label was associated to the read packet according to the file name.

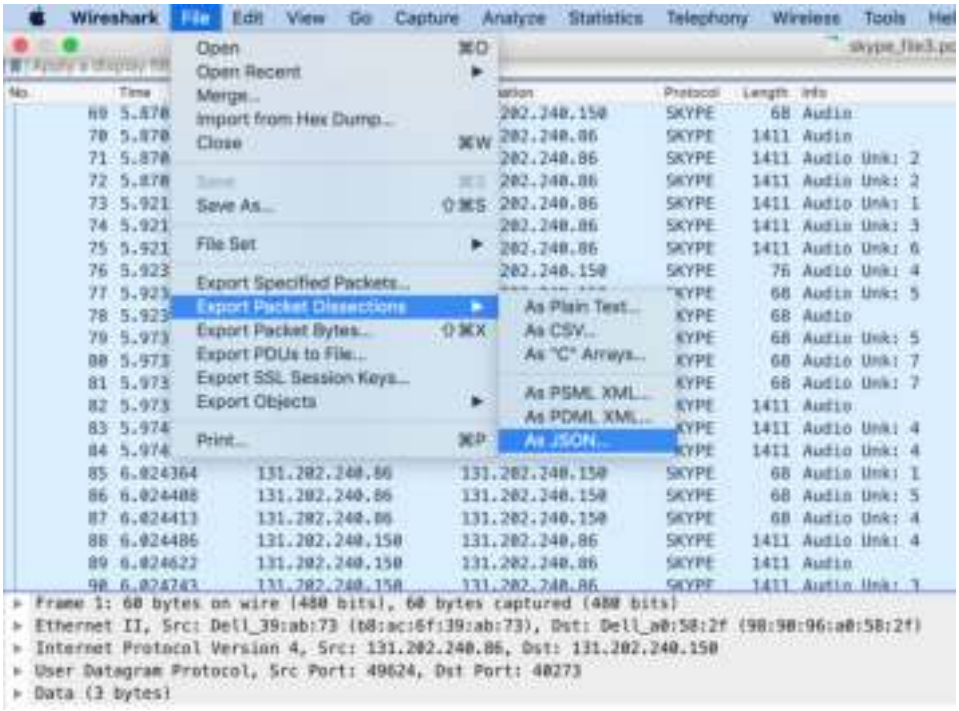


Figure 5.2: Export PCAP as JSON

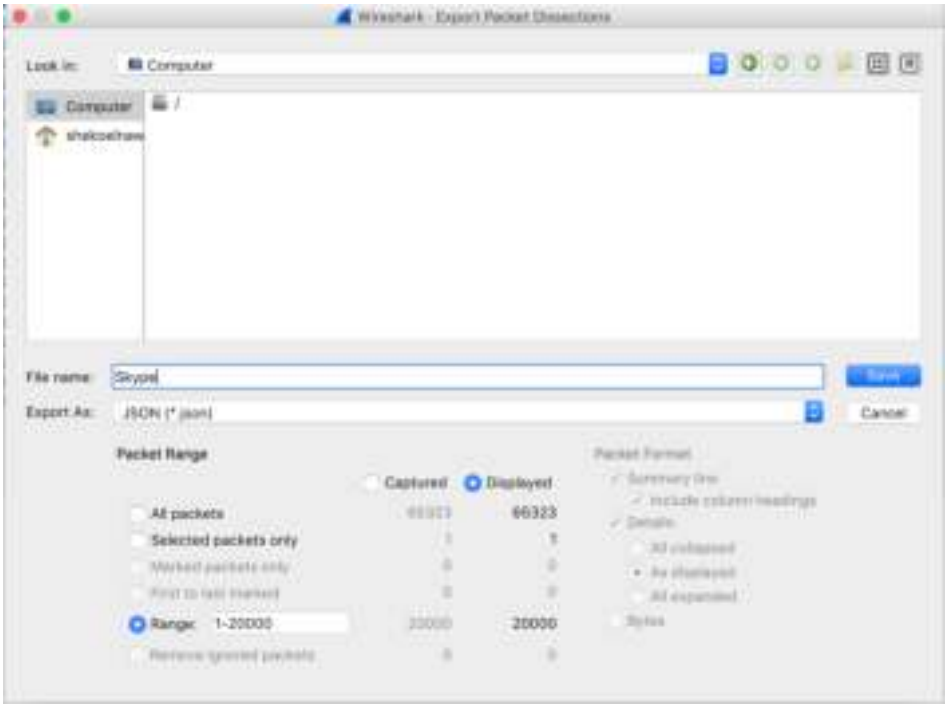


Figure 5.3: Export Properties

5. Packets are then labelled according to their flow number based on the definition of a flow mentioned in Section 2.4. Therefore, after this step each packet will have its own flow number in the struct.
6. Packets are filtered to remove any non-labelled packets or any packet with no transport layer header like ARP, ICMP, and IGMP packets since it will not be possible to extract useful features for these protocols.

This step resulted in the following packet count in the UNIBS dataset:

Skype:	20776
Browser:	22249
BitTorrent:	946
Mail:	1291
RSS:	279

The packet count of the UNB dataset looks as follows:

Skype:	19951
Netflix:	20000
Torrent:	19986
Spotify:	19885
YouTube:	19996

As we can see, the UNIBS dataset is unbalanced, whereas the UNB dataset is almost equally balanced. We took this as an opportunity to investigate the performance of the different traffic classifiers on different datasets that are both balanced and unbalanced. This will allow us to assess the impact of unbalanced training data on the accuracy of such classifiers.
7. A set of 26 packet-level and flow-level features was then extracted from both the datasets. A summary of all the extracted features is shown in Table 5.1. Appendix A contains a glossary of feature definitions. Note that, the flow-level features that use entropy were not previously discussed in the literature, therefore, one of the contributions of this work is proposing the use of entropy-based flow-level features.

Table 5.1: Complete List of Extracted Features

Packet-Level Features	**Flow-Level Features**		
Source Port	Minimum Frame Length	Maximum Frame Length	Mean Frame Length
	Median Frame Length	Variance Frame Length	Entropy Frame Length
Destination Port	Flow Size Frame Length	Minimum Capture Length	Maximum Capture Length
	Mean Capture Length	Median Capture Length	Variance Capture Length
Protocol	Entropy Capture Length	Flow Size Capture Length	Minimum Interarrival Time
	Maximum Interarrival Time	Mean Interarrival Time	Median Interarrival Time
Time to Live	Variance Interarrival Time	Entropy Interarrival Time	Number of Packets in Flow
	Flow Duration		

5.2. Feature Histograms

In order to inspect and investigate the UNIBS and UNB datasets at a much lower and fine-grained level we had to plot histograms for each of the 26 extracted features. To study the impact of each feature on the output class, we split the histogram of each feature into five histograms representing the values of the extracted feature belonging to every class. In this section, and for brevity, we comment on the most interesting histograms and then we simply include the remaining histograms in Appendix B. The next few plots represent the histograms of all features for every output class. As we can see in Figure 5.4 source port could be a very effective attribute in separating Skype traffic from the rest of the classes since Skype tends to have only one source port value. On the other hand, RSS tends to have a very small range of source ports while browser tends to be spread over a wide range of source ports.

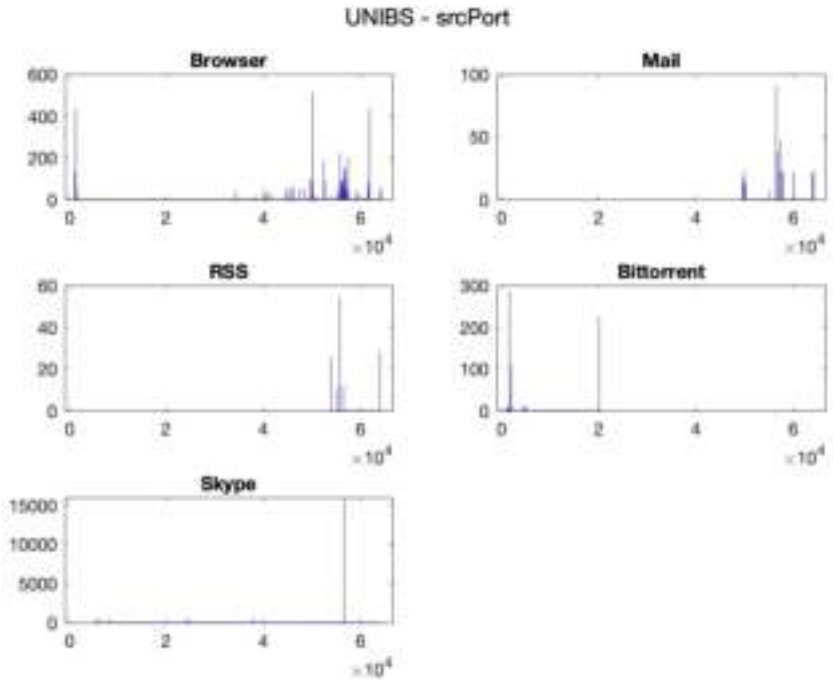


Figure 5.4: UNIBS Source Port Histogram

Figure 5.5 shows the destination port histograms for the UNIBS dataset. If we inspect the destination port histograms shown in Figure 5.5, we would find out that destination port could be a very effective feature in pointing out BitTorrent traffic since this is the only application that tends to use higher port numbers unlike the other four applications. It is also worth mentioning that the combination of source port and destination port could be a very good indicator of Browser traffic, since Browser is the only application that tends to have a fixed destination port number (port 80) while having a spread distribution of source port numbers. The case with mail is similar since they have a spread source port distribution, but their destination port is concentrated towards the lower end of the port numbers. Therefore, our initial observation from these histograms is that if we use a combination of source, as well as, destination ports we could very easily classify almost all classes in the UNIBS dataset. This observation will either be confirmed or denied later on using the results of running the UNIBS dataset through the five classifiers mentioned earlier.

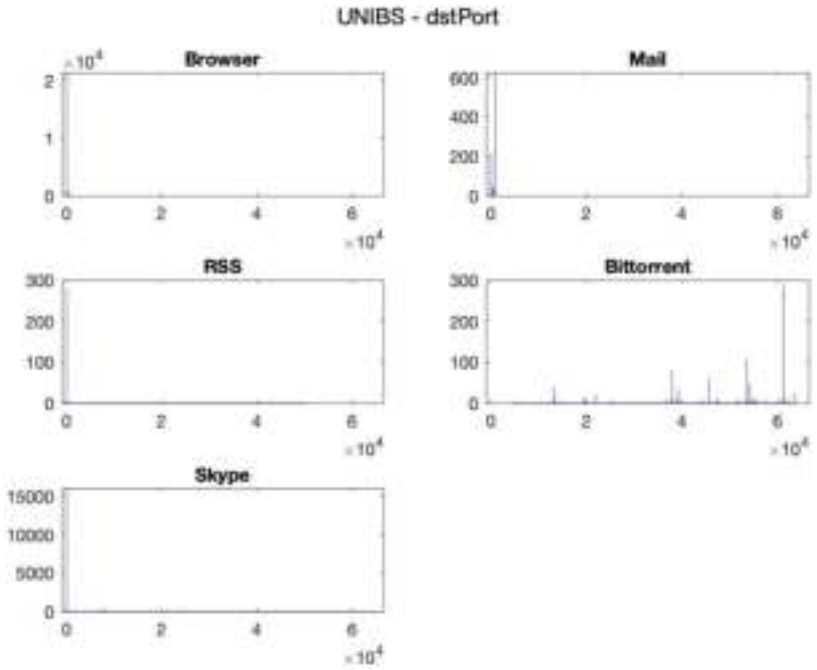


Figure 5.5: UNIBS Destination Port Histogram

Figure 5.6 shows the histograms for maximum capture length, which also shows a good potential to differentiate some traffic classes from one another. As we can see the variance of the maximum capture length for Skype tends to be very large, whereas

that of mail is significantly smaller, therefore, this feature might be a very interesting feature for the classification algorithms.

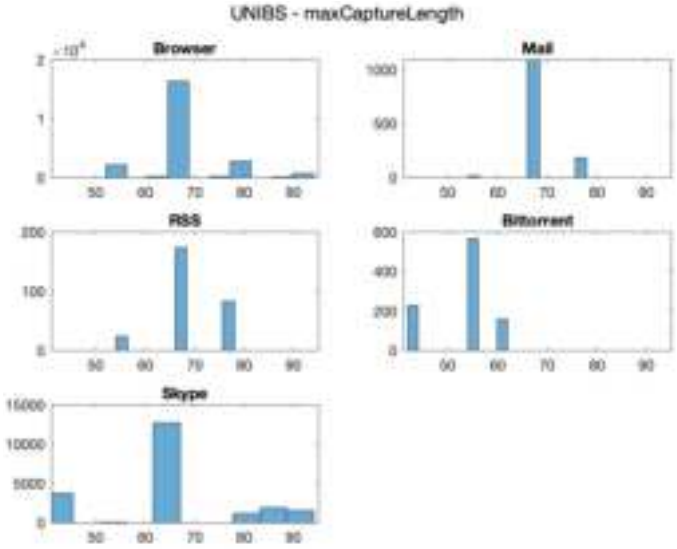


Figure 5.6: UNIBS Maximum Capture Length Histogram

On the other hand, the histograms of other features prove that they might not be very useful in differentiating the different classes due to the overlap between their values. Figure 5.7 shows the histogram of entropy capture length. As we can see from the plots, all the histograms tend to have almost exactly the same distribution. Therefore, the first intuition says that entropy capture length might be discarded when performing feature selection later on.

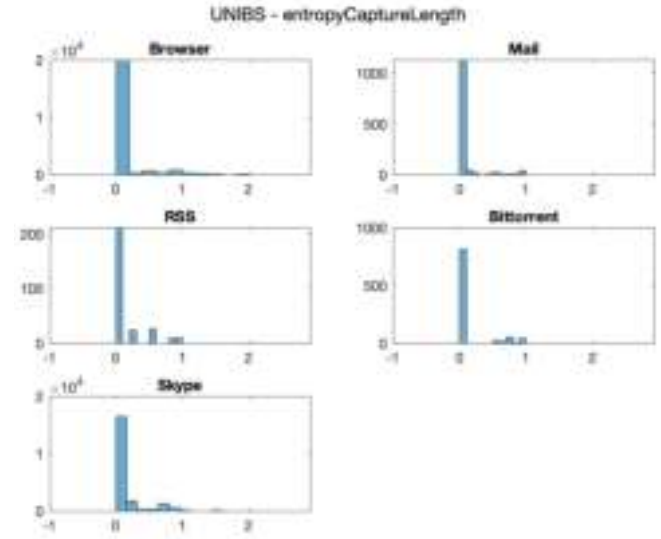


Figure 5.7: UNIBS Entropy Capture Length Histogram

The same argument also applies to the median of the inter-arrival time shown in Figure 5.8.

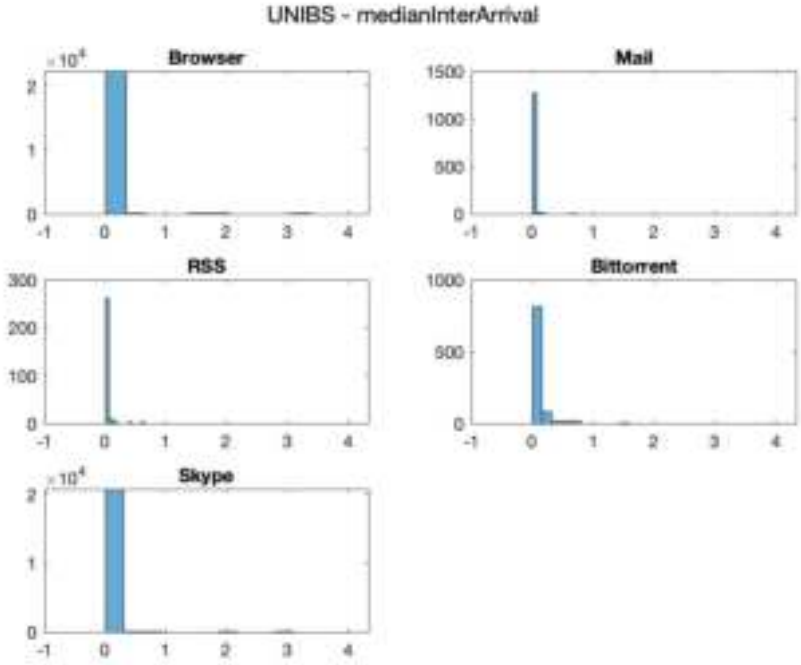


Figure 5.8: UNIBS Median Inter Arrival Time Histogram

When it comes to the UNB dataset the source and destination ports become even more decisive as shown in Figure 5.9 and Figure 5.10, respectively. We can see that the five classes have distinct source and destination ports, and hence by simply looking at the port numbers we would expect the classifiers to obtain really high accuracies and F-scores. This might indicate that using port numbers alone can be enough to obtain the correct traffic class. However, this could raise a potential problem since as mentioned earlier in the literature review, modern applications tend to dynamically change their port numbers in order to obfuscate any means of traffic classification. Therefore, if we run into a more sophisticated situation where applications change their port numbers dynamically, our classifiers might fail to identify them. This problem will be investigated later in this work.

If we inspect the entropy inter-arrival time of the UNB dataset shown in Figure 5.11 we can clearly tell that it could be very useful in recognizing the Skype traffic as its entropy inter-arrival tends to have a distribution that looks very different from the other classes. On the contrary, median frame length might be used to differentiate

between Torrent and Skype on one hand, and the other three classes on the other hand. Unfortunately, it cannot really distinguish by itself between individual classes due to the very similar distributions shown in Figure 5.12.

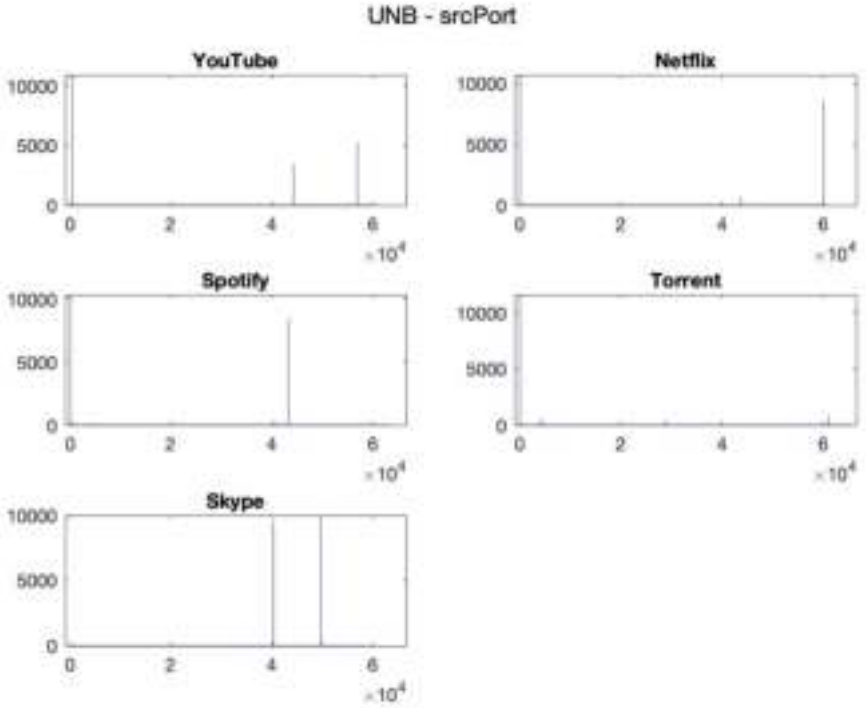


Figure 5.9: UNB Source Port Histogram

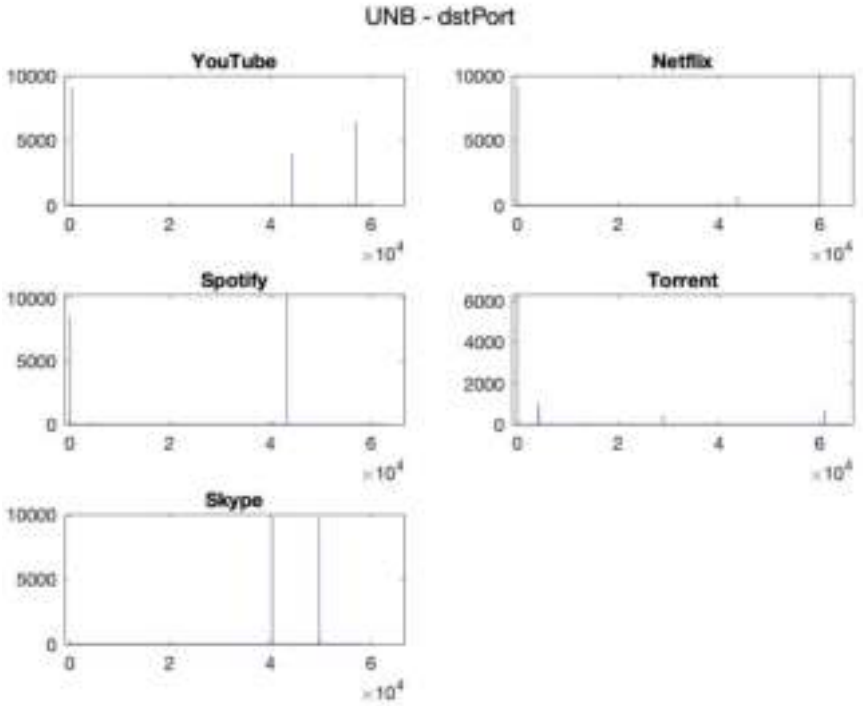


Figure 5.10: UNB Destination Port Histogram

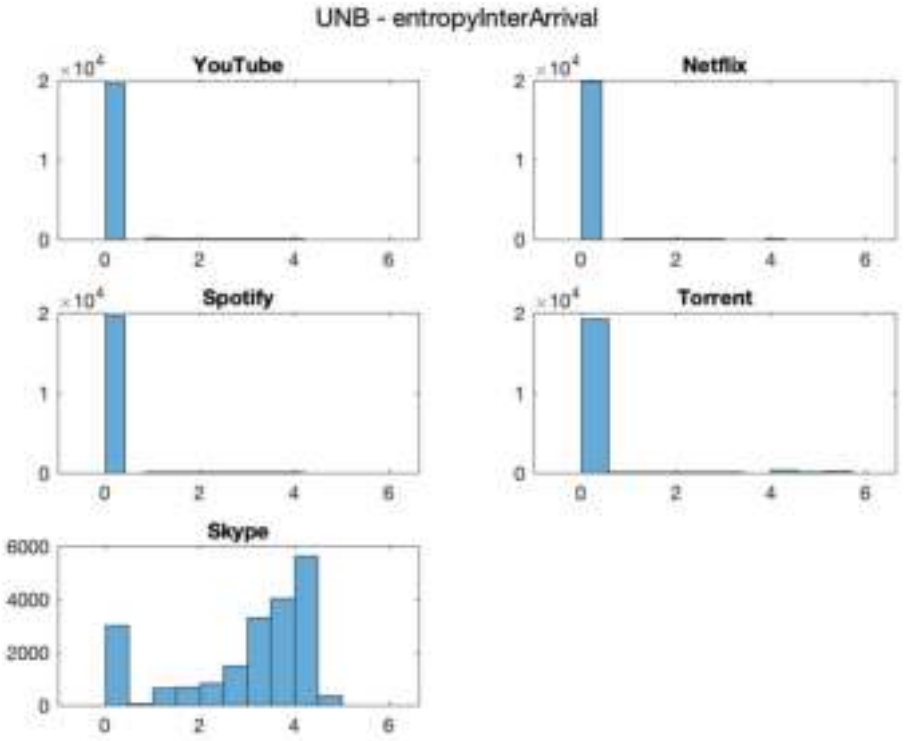


Figure 5.11: UNB Entropy Inter Arrival Time Histogram

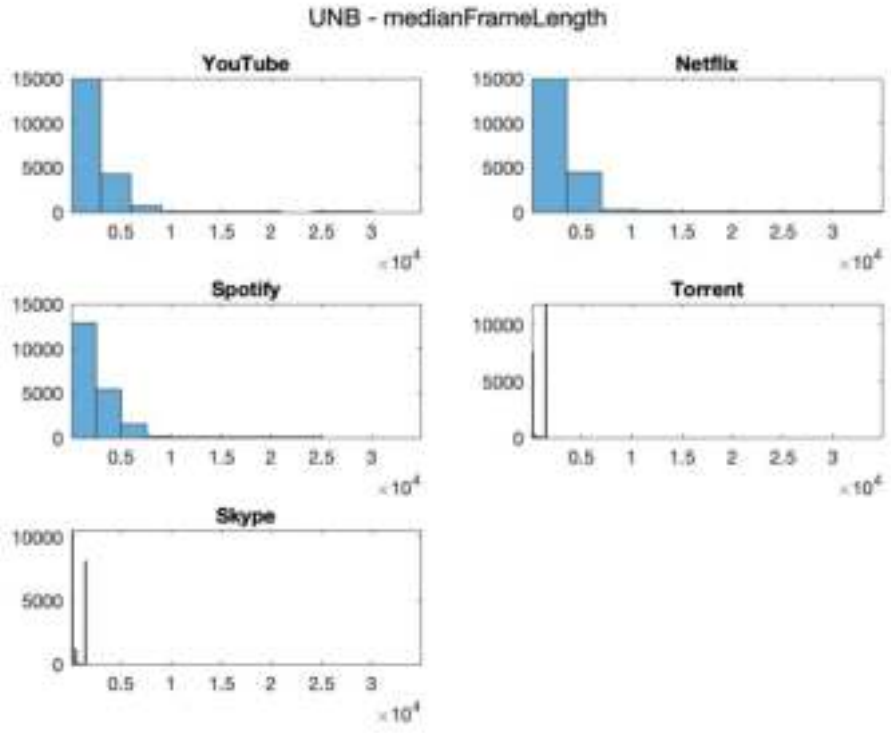


Figure 5.12: UNB Median Frame Length Histogram

5.3. Feature Selection

As mentioned in Section 5.1, we were able to extract 26 features from the two datasets. These features were chosen based on an exhaustive list of features gathered from all the previous research works discussed within the literature review, besides additional features that we thought might enhance the classification performance. This means that choosing these 26 features might not be based on a scientific approach or might not take into account the knowledge of the networking domain. This might cause some issues since we might have some redundant attributes that could potentially misguide the classification algorithms used, especially the naïve Bayes algorithm that is quite sensitive to redundancies. In addition to redundant attributes, we might also have completely irrelevant attributes that might not just degrade the performance of our classifiers in terms of classification accuracy or F-score, but it also slows down the process of training and testing our classifiers. Therefore, feature selection is very important since it helps streamline the interpretability of the model, avoids overfitting by eliminating irrelevant attributes through reducing the variance of the model, and reduces the time and resources required to build the model.

In this section we discuss two feature selection algorithms, namely stepwise regression and random forest feature selection, that were used in our research to select the most optimal features that describe the two datasets in hand. The feature selection algorithms were integrated into the five traffic classifiers that were built in this research. The way we do this is by applying feature selection to the training phase only. Therefore, before training any of the classifiers we run one of the two feature selection algorithms to obtain the indices of the most influential attributes to the classification process. The training phase will then build the classifiers based on the selected features only. Next, the indices of the selected features are then be passed on to the testing phase such that only those features will be used to classify any new instance.

5.3.1. Stepwise regression (SWR). Stepwise regression is a popular regressor feature selection mechanism by which a regression model is iteratively built by adding or deleting features at every step [29]. Assume that we have n potential features (in our case $n = 26$) and a single class attribute which is the traffic class. Regression models usually have an additional intercept term β_0 and hence the number of features is $n+1$ (in our case 27). At the beginning, stepwise regression attempts to build a regression model that consists of only one feature which is the feature with the highest correlation

to the output class. By definition, this feature will result in the largest partial F-statistic. After that, stepwise regression investigates the other n-1 features to find out the one that generates the highest partial F-statistic. The second feature is then added to the model given that its partial F-statistic is larger than the value of the F-random variable for adding a feature to the model. This F-random variable is known as f_{in}. The partial F-statistic for the second feature is calculated using Equation (4):

$$f_2 = \frac{SS_R(\beta_2|\beta_1, \beta_0)}{MS_E(x_2, x_1)} \tag{4}$$

where MS_E represents the mean square error for the model generated using the two features x_2 and x_1, and $SS_R(\beta_2|\beta_1, \beta_0)$ represents the regression sum of square due to β_2 given $\beta_1,\ \beta_0$.

Once we have added the second feature to the regression model, it is now time to decide whether the first feature needs to be eliminated. We do so by calculating the F-statistic represented by Equation (5):

$$f_1 = \frac{SS_R(\beta_2|\beta_1, \beta_0)}{MS_E(x_2, x_2)} \tag{5}$$

In case f_1 is smaller than the F-random variable to remove variables from the regression model, we call it f_{out}. In MATLAB, the minimum p value for a term to be removed is called 'premove'. The value of premove in our experiments was set to 0.025.

Stepwise regression will keep on performing this systematic method to investigate the remaining potential candidate features, with the stopping condition being the inability to add or eliminate any more features. Even though stepwise regression tends to perform feature selection in a very scientific and systematic manner which reduces the complexity significantly compared to performing a brute force feature selection, it also suffers from some drawbacks. Stepwise regression is not always guaranteed to select the most influential features that best describe the output class since it selects the attributes based on sample estimates of the actual model weights. Therefore, this could potentially lead to a small room for error when selecting the important attributes. In addition, this is also a computationally intensive algorithm since the complexity of stepwise regression is in the order of $O(n^2)$. Nevertheless, the fact that we only use feature selection during training does not affect the real-time classification of traffic. Table 5.2 and Table 5.3 show the selected features using stepwise regression from the UNIBS and UNB datasets, respectively.

Table 5.2: UNIBS Features Selected by Stepwise Regression

Packet-Level Features	**Flow-Level Features**		
Source Port	Minimum Frame Length	Maximum Frame Length	Mean Frame Length
	Median Frame Length	Variance Frame Length	Entropy Frame Length
Destination Port	Minimum Capture Length	Maximum Capture Length	Mean Capture Length
	Median Capture Length	Variance Capture Length	Entropy Capture Length
Protocol	Minimum Interarrival Time	Maximum Interarrival Time	Variance Interarrival Time
Time to Live	Entropy Interarrival Time		

Table 5.3: UNB Features Selected by Stepwise Regression

Packet-Level Features	**Flow-Level Features**		
Source Port	Minimum Frame Length	Maximum Frame Length	Mean Frame Length
Destination Port	Median Frame Length	Variance Frame Length	Entropy Frame Length
Protocol	Minimum Interarrival Time	Entropy Interarrival Time	
Time to Live			

5.3.2. Random forest. As discussed earlier, random forest is an ensemble method for classification that combines several trees together in order to reduce overfitting and enhance the classification accuracy. By definition, random forest selects a random subset of features at every step and assesses the purity of the instances according to the chosen subset of features. A very popular purity measure that is widely used with decision trees is entropy. Therefore, at every node in each tree, random forest picks the feature or set of features that best split the training instances according to the purity measure. The structure of random forest defines, by default, the most important features in the dataset, since we can always find the most important attribute at the very first node of a decision tree followed by the second most important feature and so on. Therefore, the random forest ranks the different features by importance on its own.

Random forests offer two simple methods for feature selection, namely, mean decrease impurity and mean decrease accuracy. When performing feature selection using the mean decrease impurity method, the algorithm usually computes to what extent each feature reduces the weighted impurity in every single tree in the forest. The overall importance of that feature can then be computed as the average decrease in the weighted impurity resulting from this feature across all the trees within the forest. After that, features are then ranked based on this measure. On the other hand, mean decrease accuracy focuses on measuring the direct effect of each feature on the overall accuracy of the model. This is done through permuting the values of each feature and computing how much the permutation reduces the accuracy of the random forest model. Therefore, the result could be thought of as permuting important features would highly reduce the

accuracy of the model, whereas permuting insignificant features would have almost no impact on its accuracy.

Table 5.4 and Table 5.5 show the selected features using random forest from the UNIBS and UNB datasets, respectively.

Table 5.4: UNIBS Features Selected by Random Forest

Packet-Level Features	**Flow-Level Features**		
Source Port	Maximum Frame Length	Mean Frame Length	Median Frame Length
	Variance Frame Length	Entropy Frame Length	Minimum Capture Length
Destination Port	Maximum Capture Length	Mean Capture Length	Median Capture Length
	Flow Size Capture Length	Minimum Interarrival Time	Maximum Interarrival Time
Protocol	Mean Interarrival Time	Median Interarrival Time	Variance Interarrival Time
Time to Live	Entropy Interarrival Time		

Table 5.5: UNB Features Selected by Random Forest

Packet-Level Features	**Flow-Level Features**		
Source Port	Minimum Frame Length	Maximum Frame Length	Mean Frame Length
Destination Port	Median Frame Length	Variance Frame Length	Entropy Frame Length
	Flow Size Frame Length	Minimum Capture Length	Maximum Capture Length
Protocol	Mean Capture Length	Median Capture Length	Variance Capture Length
Time to Live	Flow Size Capture Length	Median Interarrival Time	Entropy Interarrival Time

5.4. Discretization

Discretization is the process of converting a continuous variable into a discrete counterpart such that they can be more suitable for numerical analysis and digital representations. Discretization is very important in improving the performance of classification algorithms since it usually yields better results and sometimes speeds up the execution of the machine learning algorithm used [18]. The network features extracted from the UNIBS and the UNB dataset are all numeric except for the port numbers and the protocol. Hence, discretization could clearly help in improving the classification performance if the numeric attributes of the UNIBS and the UNB dataset were discretized before being presented to the classification algorithms. Discretization could be divided into two main categories; unsupervised and supervised discretization. In this section, we discuss the two types of discretization, highlight the differences between the two and eventually point out the more suitable technique for our traffic classification problem.

Unsupervised discretization is the process of quantizing the feature values in the absence of any knowledge about the classes of the different instances in the training set [30]. Therefore, it is very helpful in clustering problems where the class attribute is

unknown in the first place. Unsupervised discretization can be divided into equal-width and equal-frequency discretization. Equal-width discretization focuses on splitting the range of attribute values into a number of equal intervals regardless of the distribution of the attributes' values. Unfortunately, this discretization technique causes uneven distribution of the training instances among the different intervals which could eventually burden the ability of this feature to contribute to a more representative model. Equal-frequency discretization divides the training instances into a number of predefined bins based on the distribution of the feature values. Therefore, equal-frequency discretization results in bins containing the same number of instances. This technique is also known as histogram equalization since it will result in a uniform feature histogram [30].

Supervised discretization is the process of quantizing the feature values while respecting the knowledge of class labels of all instances in the training set [30]. As a result, it is mainly used with supervised learning techniques like classification and regression. Traffic classification is a supervised learning process since we know in advance the class label of each training instance, hence we are going to use supervised discretization on our two datasets. As mentioned in [31], the most commonly used supervised discretization mechanism is the entropy-based discretization, where we sort the instances by the feature's value and look for potential splitting points such that the subintervals are as pure as possible. Consequently, we place our splitting boundaries at the points where the information or entropy required to represent the individual class values is the least. To further illustrate the process of entropy-based discretization, consider the binary classification example given in Figure 5.13 where the numeric values represent the packet sizes and the labels are either "Skype" (S) or "Non-Skype" (N).

64	65	68	69	70	71	72	75	80	81	83	85
S	N	S	S	S	N	N	S	N	S	S	N
						S	S				

Figure 5.13: Binary Classification Example

We would then use Equation (6) to calculate the entropy or the information required to represent class values as follows:

$$Entropy = -\sum_{i} p_i \log_2 p_i \tag{6}$$

where p_i is the probability of class i to occur in the particular interval.

The entropy is calculated for each of the 11 possible candidates for boundary placement in the previous example, and then the boundary that results in the least entropy is then picked for discretization. To demonstrate the working of such an algorithm, let us compute the entropy of "packet size < 71.5" which splits the feature values into two regions where the first runs from 64 to 71 resulting in four Skype packets and two Non-Skype packets, while the second region runs from 72 to 85 resulting in five Skype packets and three Non-Skype packets. The Entropy is calculated using Equation (7) and it is equal to 0.939 bits. This process is repeated recursively until some stopping criterion is met. One of the popular stopping criterions is called the MDL principle [31].

$$Entropy\ ([4,2],[5,3]) = \frac{6}{14}\ Entropy([4,2]) + \frac{8}{14}\ Entropy([5,3]) \tag{7}$$

The supervised discretization described in [32] is very similar to that of [31] with the exception of using a Gini index-based measure instead of the entropy measure used by Fayyad and Irani. Therefore, in this work we will try the two algorithms described in [31, 32] since they are readily available in the WEKA tool. We conduct a cross-validation experiment whereby we train five different classifiers namely naïve Bayes, KNN, linear SVM, 2nd order SVM, and random forest on the two datasets, UNIBS and UNB. Meanwhile, we shall be recording the training time, testing time, classification accuracy and F-score on the original dataset, as well as, the discretized datasets using both discretization algorithms mentioned above. We then conduct a comparison between the results of the discretized and the non-discretized datasets based on the criteria mentioned earlier (training time, testing time, classification accuracy and F-score). By doing so, we study the effect of discretization on the classification performance, and hence, we decide whether or not it is necessary to discretize our datasets before applying any machine learning algorithm.

5.5. Conducted Experiments

Now that feature selection has been performed and we have identified the most important features, we start training and testing our traffic classifiers. We have conducted three different sets of experiments, whereby each one tries to look at a

specific parameter and tries to investigate the impact of varying such a parameter. We wanted to address the issue of overfitting models, which means that the generated model would not generalize well to new instances that were never seen by the classifier. In addition, through our literature review we have noticed that the research community has not investigated the impact of varying the training set size on the classification performance. Furthermore, a very important aspect that was missing from the literature is the most optimal number of packets used to extract flow-level features. We perform those experiments on seven different subsets of features to investigate the effect of using each combination of features on the performance of the classifiers. The seven feature subsets are:

1. All Features – This includes all the 26 extracted features
2. All Features without port numbers – This is very similar to combination 1 but after removing source and destination ports
3. Port numbers only – This includes only source and destination ports to check whether ports are truly the most decisive features behind classification
4. Stepwise regression (SWR) features – This includes the features selected by the SWR algorithm
5. Stepwise regression (SWR) features without port numbers – This is very similar to combination 4 but after removing source and destination ports
6. Random forest (RF) features – This includes the features selected by the RF algorithm
7. Random forest (RF) features without port numbers – This is very similar to combination 6 but after removing source and destination ports

The reason why we try the same combination without the port numbers is because of our initial claim that ports are the most decisive features, and hence we wanted to investigate the performance of our classifiers in the absence of port numbers. The port-less experiments are very important since in case port numbers were dynamically changed by the different applications, we assume that port numbers, in the worst-case scenario, will not influence the classification process. Therefore, we aim to build classifiers that could counteract the port obfuscation process and still be able to classify traffic without the need to have fixed port numbers per application. In this section we look at the different experiments performed within the course of this research.

5.5.1. Cross-validation. Overfitting is a very serious issue that affects the performance of classification models because building a model with 100% accuracy on the training set that cannot generalize well to new unseen instances could be disastrous if implemented in real life. Therefore, cross-validation is one of the best ways to assess overfitting. Cross-validation is the process of dividing the complete dataset into k different folds with approximately equal number of instances that do not overlap. After that, we use k-1 folds as training data and the remaining fold as a test data to test the generated model. The process is repeated until all k folds have been used as testing data exactly once. This results in k different models with k different classification accuracies and F-scores. The overall classification accuracy and F-score are then found out to be the average of the k accuracies and F-scores. By doing this, we try to eliminate the luck factor that might be caused by simply taking a fixed training set and test set without repetition. A better measure of overfitting would be repeating cross-validation several times which is usually referred to as repeated cross-validation or Monte Carlo. In this work, we used 10-fold cross-validation. Figure 5.14 shows a flow chart of the cross-validation experiment.

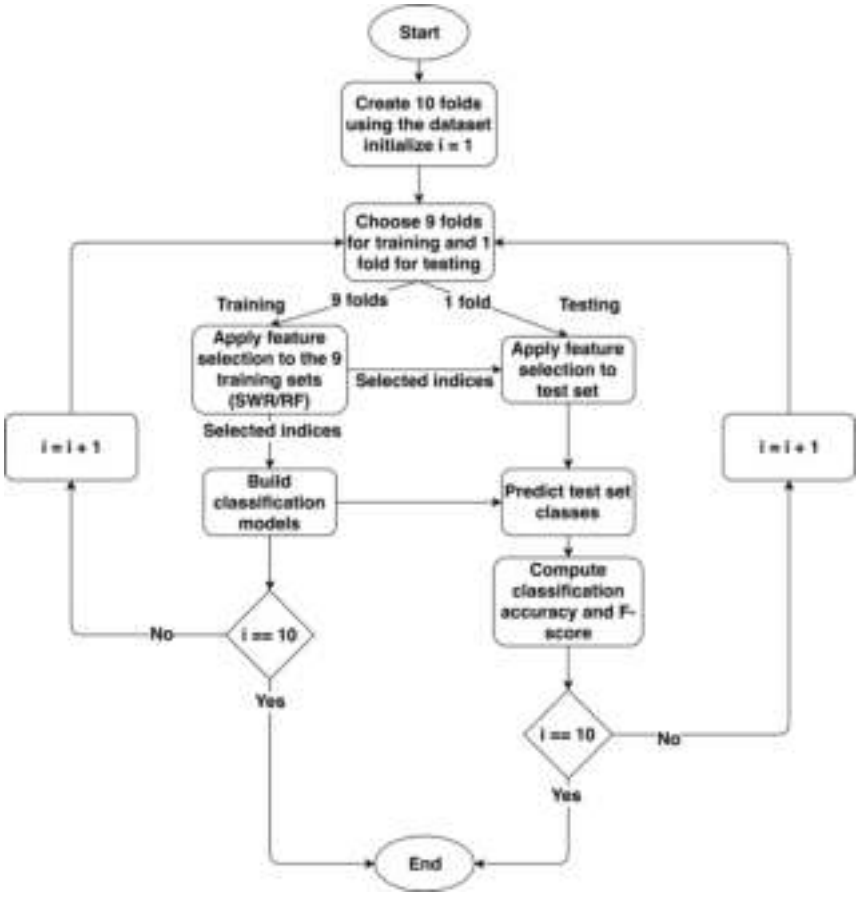


Figure 5.14: Flowchart of the Cross-Validation Experiment

5.5.2. Various packet percentage within a flow. As mentioned earlier, a flow is a series of packets sharing the same source and destination IP addresses, source and destination port numbers and protocol. Therefore, if we were to implement a real-time traffic classifier, we would need to wait for a number of packets within a flow to arrive in order to extract their flow-level features. One can immediately spot a trade-off in this scenario, since considering 100% of the packets within a flow would logically yield the best performance, nevertheless, it would also mean slowing down the classification process as we would need to wait for a longer period of time before starting to classify the flow. Therefore, in this experiment we vary the percentage of packets used to extract flow-level features in every flow such that the packet percentage varies from 10% to 100% of the packets in a flow using the 10-fold cross-validation method. After that, we generate two plots for each subset of the seven feature subsets mentioned earlier. The two plots represent the classification accuracy against the packet percentage and the F-score against the packet percentage. Moreover, we also plot the average wait time to receive the required packets against the packet percentage. This was computed as the arrival time of the last packet to be considered within the flow minus the arrival time of the first packet within the flow. We only consider arrival times as this is the major contribution of delay within real-time systems since classification time is usually negligible compared to the packet arrival time. Of course, we would expect the classification accuracy and F-score to go up as we increase the percentage of used packets. However, our intention from this experiment was to find out the most optimal packet percentage required to give us the best performance given the wait time. Figure 5.15 shows a flow chart of the various packet percentage within a flow experiment.

5.5.3. Various training set sizes. In order to inspect the impact of different training sizes on the performance of the different classifiers, we will also vary the training set sizes from 10% to 90% of the whole dataset while fixing the packet percentage at 100%. We will then compute the classification accuracies and F-scores for each of the seven feature subsets mentioned earlier. We will then plot both the classification accuracy and F-score against the training set size. In doing this, we use the holdout method that simply allocates n% of the instances to the training set and the remaining (100-n) % instances are held out as the test set. Figure 5.16 shows a flow chart of this experiment.

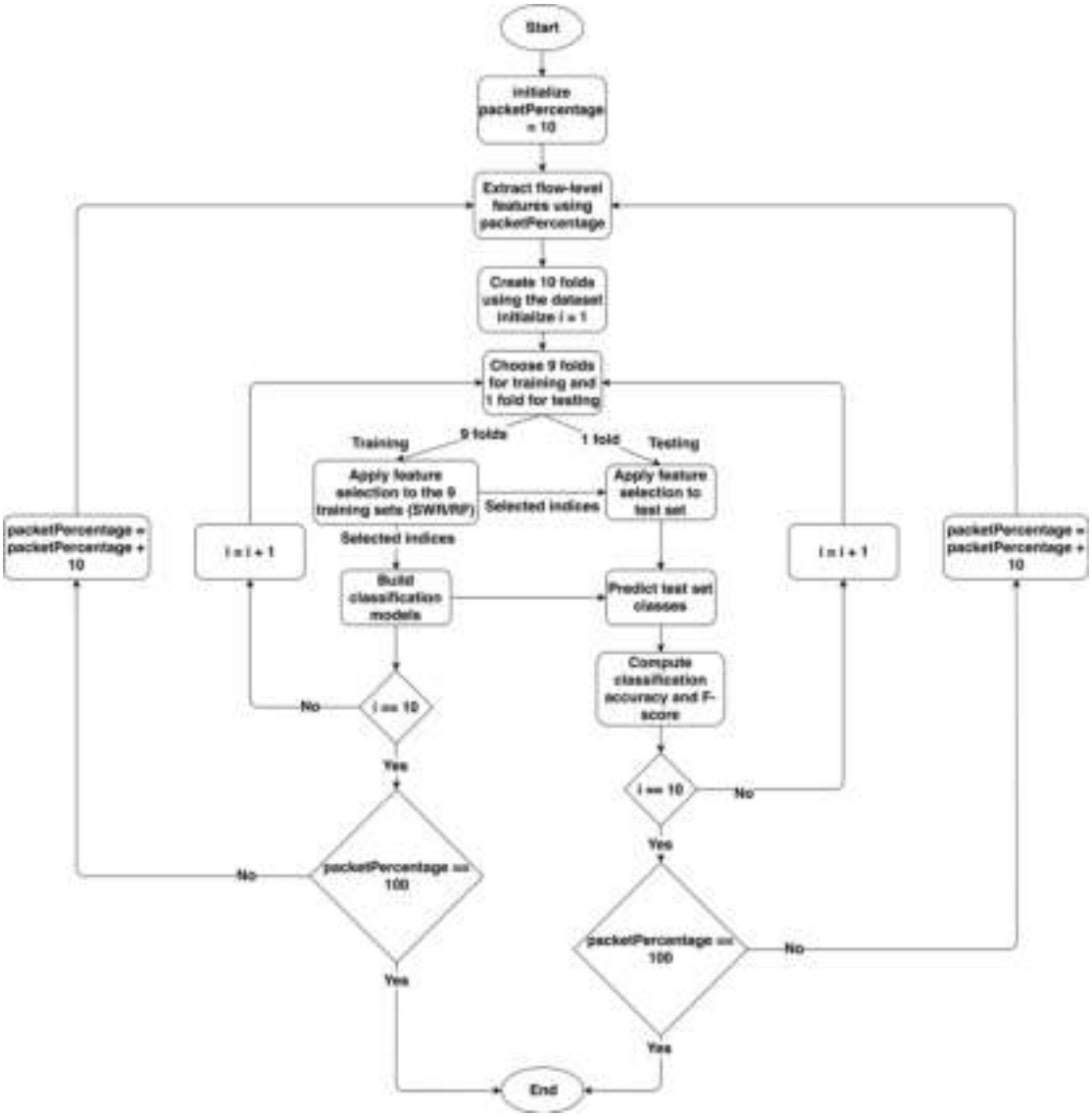


Figure 5.15: Flowchart of the Various Packet Percentage within a Flow Experiment

5.5.4. Random forest parameter tuning. Before running our experiments, we wanted to fine-tune the key parameters of the random forest algorithm in order to be able to build the model that yields the best classification. The three main parameters of the random forest algorithm are number of trees in the forest, number of features to select at random for each decision split, and the minimum number of training instances reaching the leaf nodes of the tree.

In order to find out the optimal number of trees within the random forest that would lead to the best classification performance, we study what is known as the out-of-bag error against the number of trees. As mentioned earlier, the random forest algorithm randomly samples m instances with replacement from the training set in

order to build each tree. This sampling technique is known as bootstrapping where a particular instance has a probability of $1 - \frac{1}{m}$ of not being picked in the training set. Therefore, the probability of an instance being placed in the test set is $\left(1 - \frac{1}{m}\right)^m \approx e^{-1} \approx 0.368$. Therefore, the test set would consist of approximately 36.8% of the instances for each tree within the forest. These 36.8% of instances are what is known as out-of-bag instances for each tree. Therefore, in order to investigate the effect of increasing the number of trees within the forest, we plot the average out-of-bag error against the number of trees within the forest. We do this by varying the number of trees from 1 to 500. Figure 5.17 and Figure 5.18 show the average out-of-bag error against the number of trees for the UNIBS and the UNB datasets, respectively.

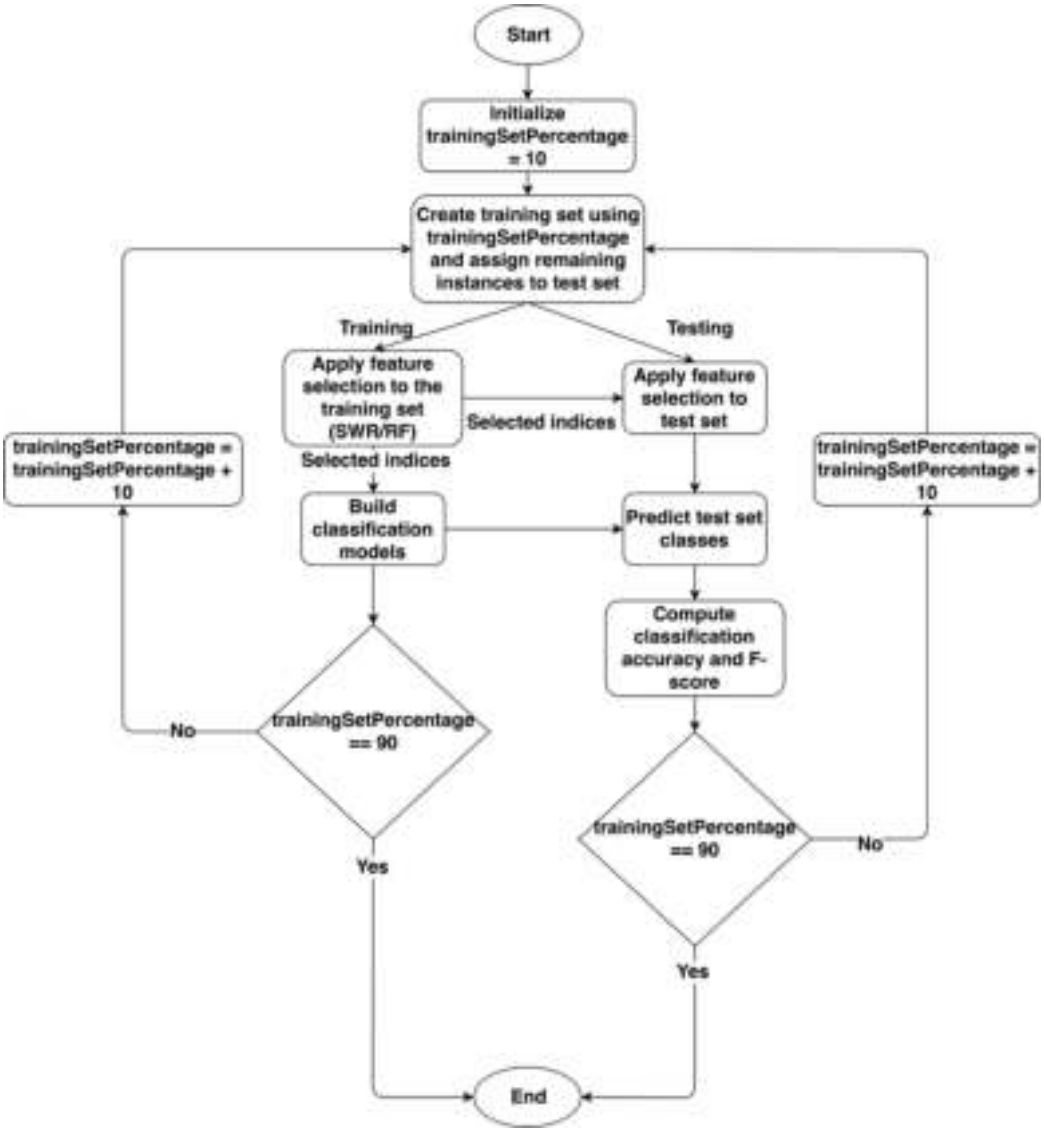


Figure 5.16: Flowchart of the Various Training Set Sizes Experiment

By inspecting the average out-of-bag error for both datasets, we can tell that the results are self-explanatory to a great extent. As the number of trees within the forest increases, we would expect the classification accuracy to increase and the error to reduce. On the other hand, once the number of trees exceeds a specific threshold, the gain in classification performance becomes insignificant. Keeping in mind that the more the number of trees in the forest, the more the time it takes to build the forest and potentially the more the time it takes to classify a test instance as it has to be routed down a greater number of trees. Even though the different trees can operate in parallel during the testing phase, unbalanced trees within the forest might lead to a slightly longer testing time as we would have to wait for all trees to finish before performing a majority vote or a probability-based decision. Therefore, it is important to pick the optimal number of trees in the forest such that nearly the best classification performance is obtained while maintaining a reasonably small number of trees. With such graphs it is apparent that the optimal number of trees usually resides at the elbow of the graph which translates to approximately 50 trees using both datasets. Therefore, in all upcoming experiments we use 50 trees in all of our random forest models.

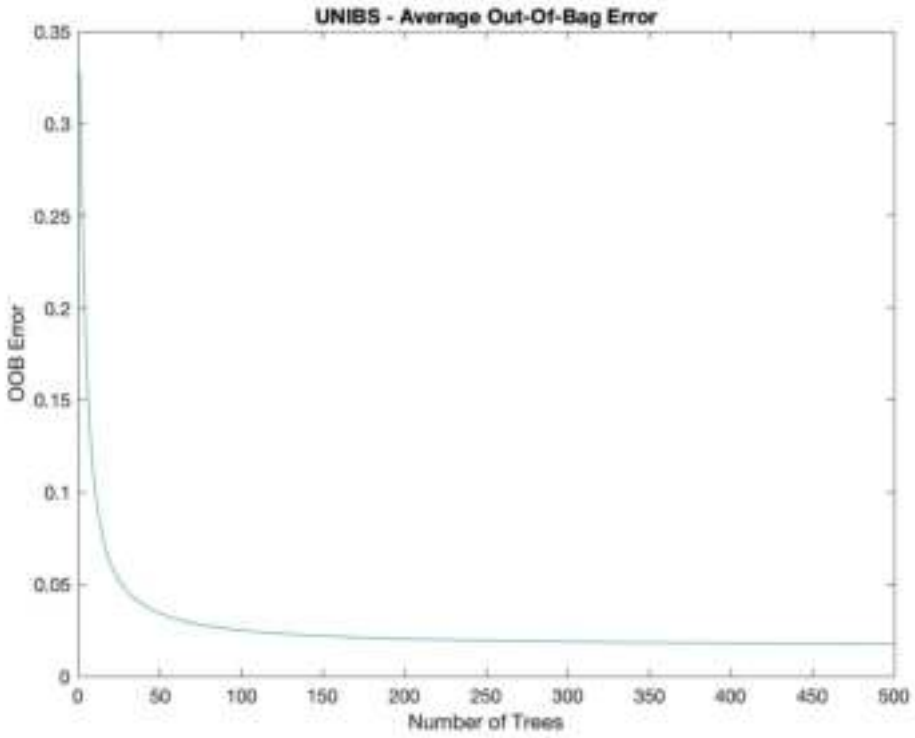


Figure 5.17: UNIBS – Average Out-Of-Bag Error Against Number of Trees

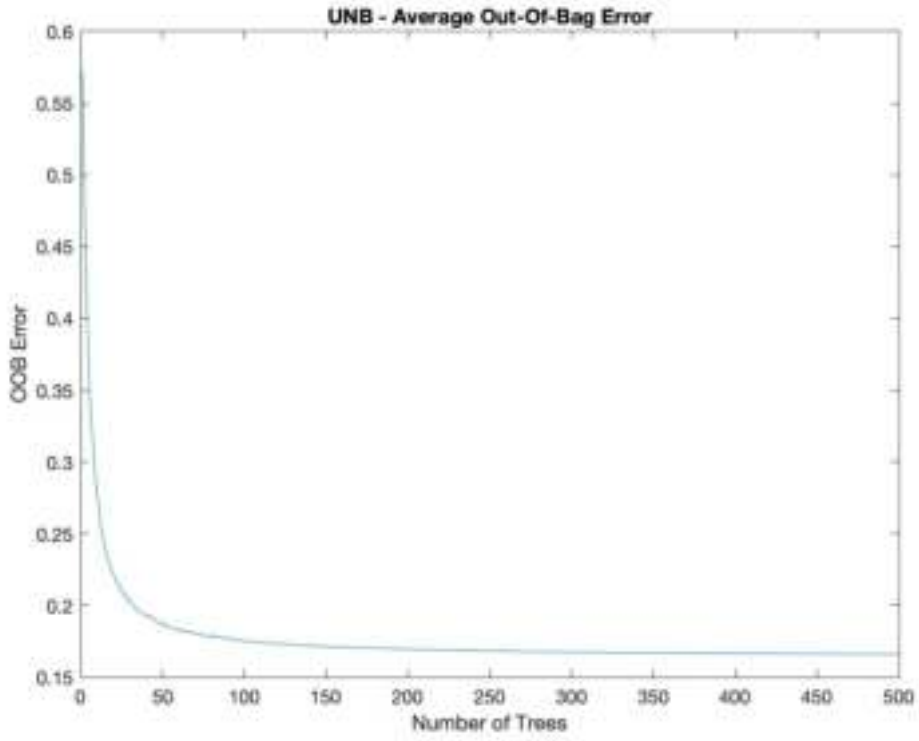


Figure 5.18: UNB – Average Out-Of-Bag Error Against Number of Trees

The next parameter to be tuned in our work is the number of features to select at random for each decision split. As mentioned earlier, while detailing the steps of training a random forest, the algorithm usually picks a random subset of features at every decision node to split the training set on. The default number of features chosen randomly is usually the square root of the total number of features in the dataset for classification problems. In our case, we were able to extract 26 features from the datasets under investigation. Hence, the default number of randomly picked features at each split is $\sqrt{26} \approx 5$ features. This is what we would like to verify through plotting the out-of-bag error against the number of considered features at each split. We vary the number of features from 1 all the way to the full feature vector (26). Figure 5.19 and Figure 5.20 show the average out-of-bag error against the number of features for the UNIBS and the UNB datasets, respectively.

Similar to the "number of trees" parameter, we expect that considering a greater number of features at each decision split would reduce the out-of-bag error. This is evident from the shape of Figure 5.19 and Figure 5.20. The same analogy applies here, the more the number of considered features the more time it takes to build the model.

However, the elbow of the graphs will usually give us the best number of features beyond which increasing the number of features leads to no significant improvement in the out-of-bag classification error. By inspecting the two figures we can conclude that the elbows of the two graphs lie around 5 features. This result proves that the default value of $\sqrt{26}$ yields the best out-of-bag performance while keeping the number of features to a minimum. Therefore, the default number of features will be used in building all of our models in this work.

Next, we attempt to tune the minimum leaf size parameter of the random forest algorithm. Minimum leaf size is simply the minimum number of training instances that reach a leaf node within the tree. If more instances are allowed to reach the leaf nodes, then we make sure we are not overfitting the model since noisy instances will not have a great impact on the structure of the trees as noisy instances can result in further decision splits that are not necessary. On the contrary, the drawback is that not much details are extracted from the large number of instances as we might learn more differences among the leaf node instances if we allowed the algorithm to keep on splitting. Therefore, we are faced by a tradeoff scenario where we would like to find out the minimum number of instances reaching the leaf nodes without overfitting the model to the training set. Keeping in mind that the default minimum leaf size is usually 1, we decided to plot the out-of-bag error against the minimum leaf size. We vary the minimum leaf size from 1 all the way to 1000 instances. Figure 5.21 and Figure 5.22 show the average out-of-bag error against the minimum leaf size for the UNIBS and the UNB datasets, respectively. Figure 5.21 and Figure 5.22 clearly reflect what we anticipated earlier. The lesser the minimum leaf size, the lower is the out-of-bag-error. As we increase the minimum leaf size, we incur more classification errors. Therefore, we decided to stick to the default value of the minimum leaf size (1) as it yields the least out-of-bag error. As for the overfitting problem, the cross-validation technique mentioned earlier shall give a better idea about whether the generated random forest overfits to the training set when a minimum leaf size of 1 is used.

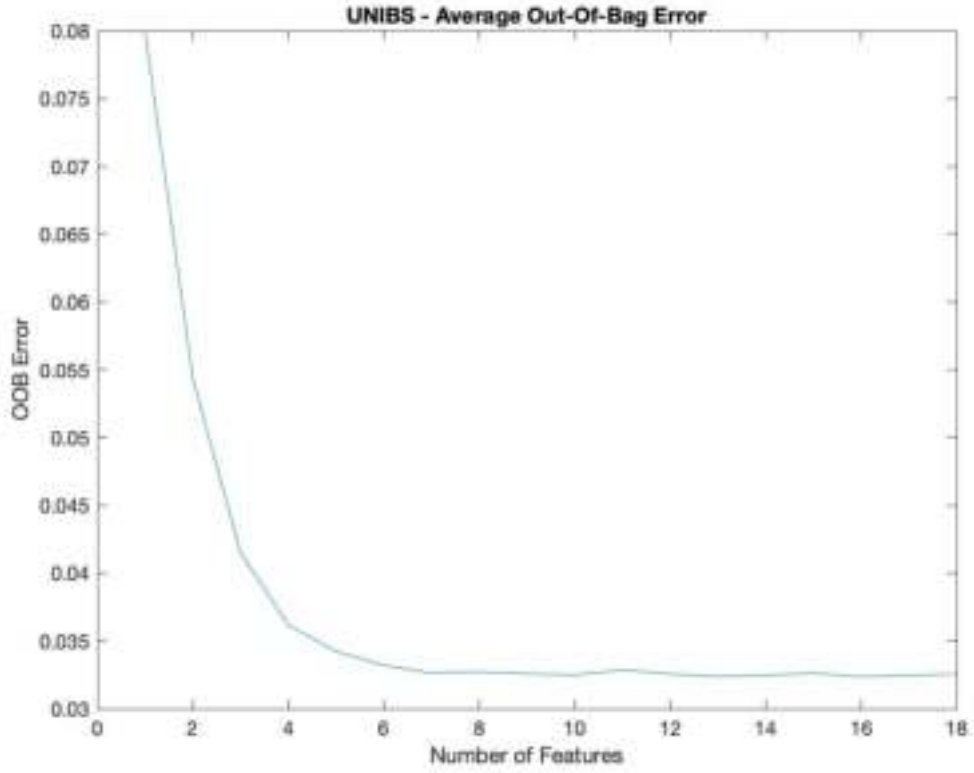


Figure 5.19: UNIBS – Average Out-Of-Bag Error Against Number of Features

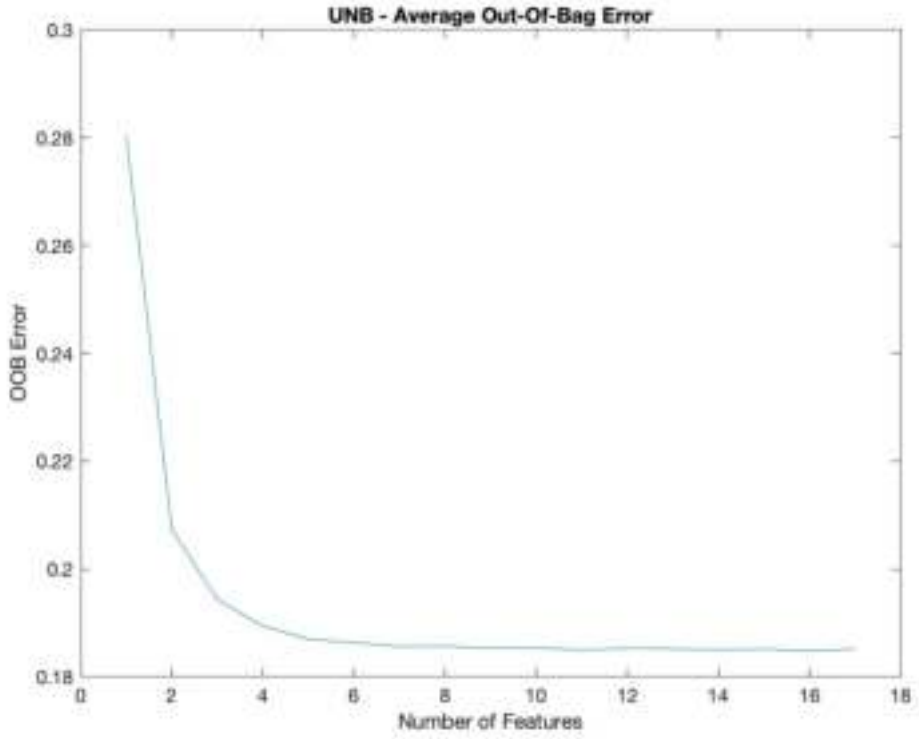


Figure 5.20: UNB – Average Out-Of-Bag Error Against Number of Features

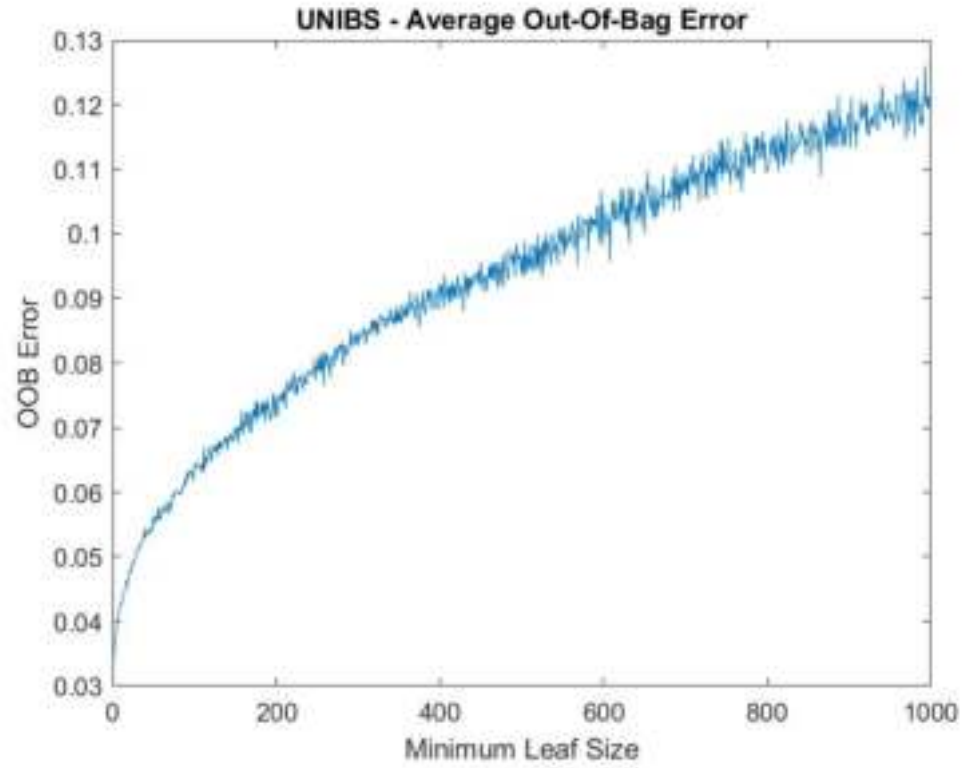


Figure 5.21: UNIBS – Average Out-Of-Bag Error Against Minimum Leaf Size

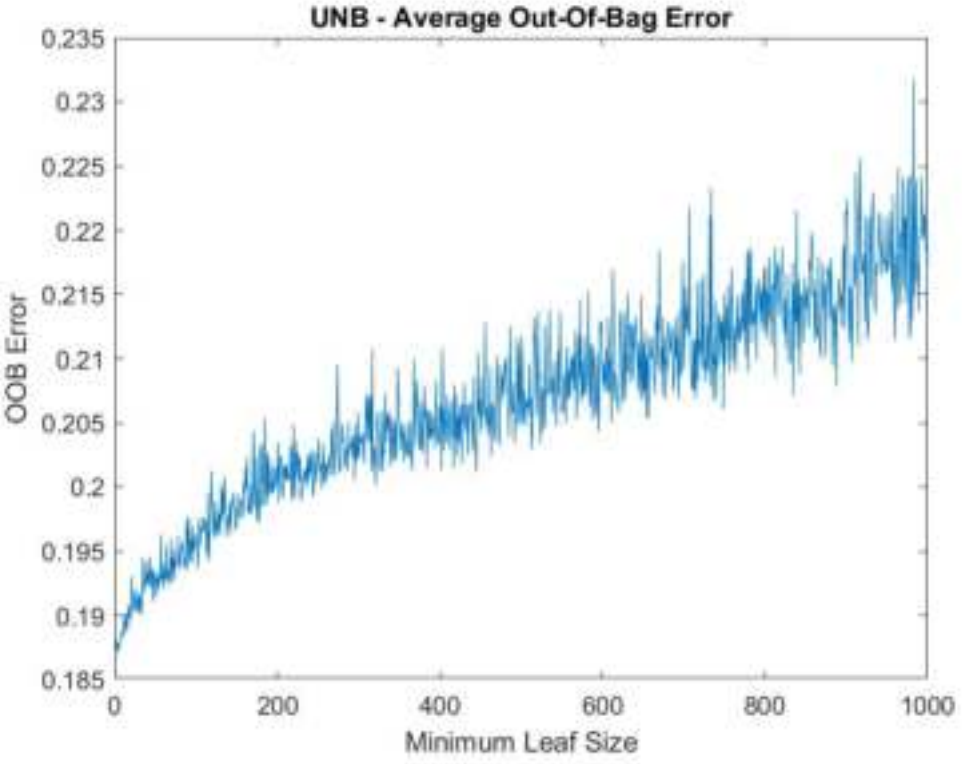


Figure 5.22: UNB – Average Out-Of-Bag Error Against Minimum Leaf Size

Chapter 6. Random Forest Hardware Design

In the experimental results demonstrated in Chapter 7, we show that the random forest algorithm tends to outperform all other algorithms in terms of classification accuracy and F-score. Therefore, we choose to proceed with designing a hardware accelerator that is based on a random forest classifier in order to speed up the classification process through the use of hardware. This is true since hardware usually executes an algorithm much faster compared to the same algorithm in software due to the fact that dedicated hardware components perform the repetitive classification steps without incurring the cost of any overhead processes introduced by software. In doing so, we try to exploit the highly parallel architecture of FPGAs to accelerate the design even further.

Authors in [33] suggest that there are two possible architectures for a random forest implementation on hardware. The first implementation is a memory-centric implementation whereby the comparison attributes and values of all nodes of the forest are stored in memory. Since a data instance will only traverse one node at each level, therefore, only the node information of that one node that needs to be traversed will be loaded from the tree level memory into the tree level comparator. Hence, only one comparator is used at each tree level. This enables a very quick context switching from one random forest model to another through simply loading new node information into the tree level memory. In addition, it also helps reduce the FPGA resource consumption in terms of the number of required comparators since only one comparator is used per tree level. On the other hand, due to the heavy reliance of a memory-centric architecture on the memory of an FPGA, this architecture requires a significant amount of on-chip memory. The second architecture is a comparator-centric architecture which is very similar in design to that of the memory-centric architecture except that it uses one comparator per node in the tree. This results in a very high consumption of FPGA comparators, while requiring no memory elements to store the node information. This is because each comparator will now hold the static value of the comparison attribute at that node. This means that context switching now takes a significant amount of time when we want to load a new random forest model onto the FPGA due to the need to change all static values of all comparators within the forest.

Keeping in mind that we have two datasets, this means that we will require the context switching feature of the memory-centric architecture. Otherwise, using the

comparator-centric architecture will mean that we will have to design two models to embed the static comparison values into their comparators. This is an infeasible approach that will consume a considerably high amount of time. Therefore, in this work, and with noticeable modifications to the implementation suggested in [33], our implementation will follow the memory-centric approach. In this chapter, we discuss the hardware design of our random forest classifier in detail.

6.1. Data Memory

In order to perform traffic classification on digital hardware, the features extracted from each network packet must first be encoded in a way that can be understood by the random forest accelerator. Recall that we were able to extract 26 features earlier from the UNIBS and the UNB datasets. Therefore, we opted for encoding the 26 features using binary numbers where each feature is encoded as a 58-bit fixed-point number. Fixed-point was chosen instead of floating-point since it usually results in a much simpler hardware design which tends to be faster than a floating-point architecture. After inspecting the maximum and minimum values of the features in the two datasets, we realized that the integer part of the 58-bit fixed-point number shall not exceed 30 bits. Therefore, each of the 26 features is encoded such that 30 bits resemble the integer part and 28 bits are used to describe the fractional part of the number. This results in an encoding scheme that requires 1508 bits to describe the features of one network packet. Consequently, the packets are stored in the data memory according to the format shown in Figure 6.1. It shows that features were arranged one after the other starting from feature 0 all the way to feature 25. The numbers shown above each feature are the start and end bits of each feature. For example, the MSB of feature 0 is at bit 1507 and the LSB is at bit 1450.

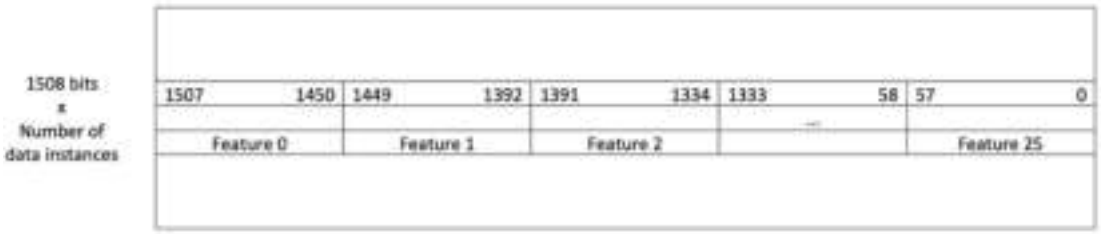


Figure 6.1: Data Memory Arrangement

6.2. Random Forest Overview

As mentioned earlier, FPGAs are semiconductor devices that can be used to

design digital circuits while decomposing the design into smaller modules that can run in parallel. This is perhaps one of the major advantages of FPGAs since it enables a hardware designer to breakdown their complex designs into smaller pieces that can execute instructions simultaneously unlike the sequential execution of normal CPUs. Therefore, our main objective while designing the random forest classifier in hardware is to identify independent components that can work simultaneously without affecting the operation of one another. The most obvious independent components are the individual trees within the forest since a test instance is simply passed down each tree regardless of the output of the other trees. The structure of the independent trees is shown in Figure 6.2 which offers an overview of our hardware-based random forest design.

In Figure 6.2, we can see that individual trees are being instantiated starting from Tree 1 all the way to Tree *n*, where *n* is the number of trees in the forest. It is clear that the network packets consisting of 1508 bits are routed down all trees at the same time. Before doing so, the test packet is registered at an input register which introduces a one-cycle delay. This input register is very important since it helps us have more control over the flow of signals in the random forest as it keeps the whole design well-behaved using a master clock. Once the test instance reaches the trees it goes into the different levels of the trees. Each tree level will simply pass on the packet to the next level within the tree for more checks, along with the address of the next node in the tree. The execution of each level within the tree is also one more aspect that requires attention. Each tree level examines only one packet at a time, therefore, instead of treating the entire tree as one bulky component, we can simply make use of the fact that a tree level checks one packet at a time and hence we can insert pipeline stages between the different tree levels. This is yet another aspect where we exploit the parallel execution capabilities of FPGAs, since now tree levels can operate simultaneously and independently with respect to all other levels within the tree. The pipelined architecture of the trees within the forest imply the need for another design constraint. For the design to be well-synchronized, all trees must have the same number of levels such that a test instance can spend the exact number of clock cycles in the pipeline and hence the outputs of all trees can be ready at the same instant in time. That is why Figure 6.2 shows all trees consisting of $m+1$ levels where level m is the last level in all trees. The output of level *m* is either the class label in case of a majority-based random forest or

class probabilities in case of a probability-based random forest. After that, the outputs of all trees are fed into a module known as “Class Tally”. In simple words, the Class Tally module will simply aggregate the results of all trees and will then pass the results to the “Voter” module. The Voter module will eventually choose the most occurring class (majority-based) or the class with the highest probability (probability-based) to be the class label of the data instance. The Class Tally and the Voter modules are discussed further in later sections. Lastly, to further add on to our highly pipelined architecture, an output register is used to simply register the output class such that it can be displayed to the user in a timely manner. Notice that the input and output pipeline registers are triggered on the positive edge of the clock.

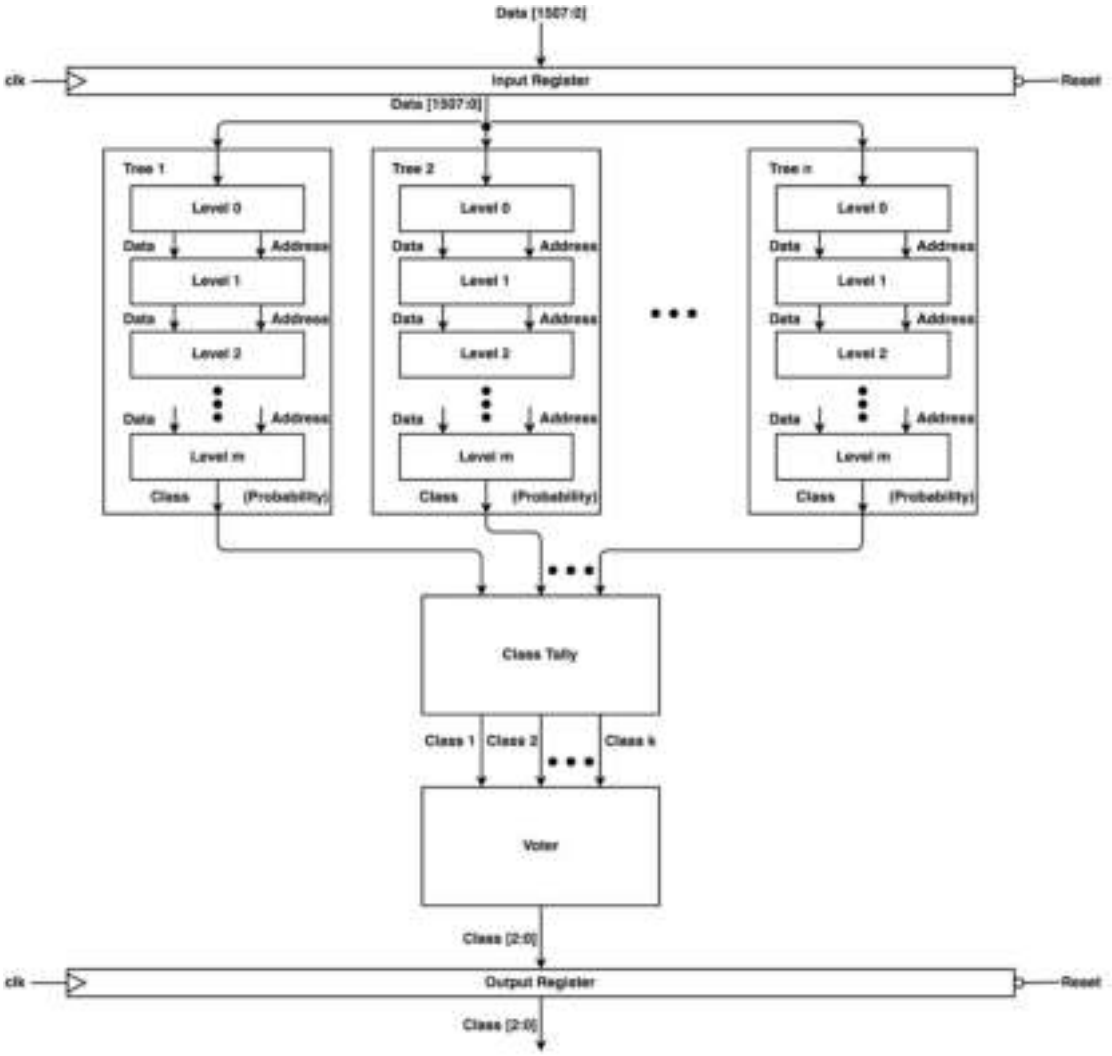


Figure 6.2: Hardware-Based Random Forest Design Overview

6.3. Tree Level

To understand the hardware design of a tree level, consider the graphical representation of a decision tree shown in Figure 6.3.

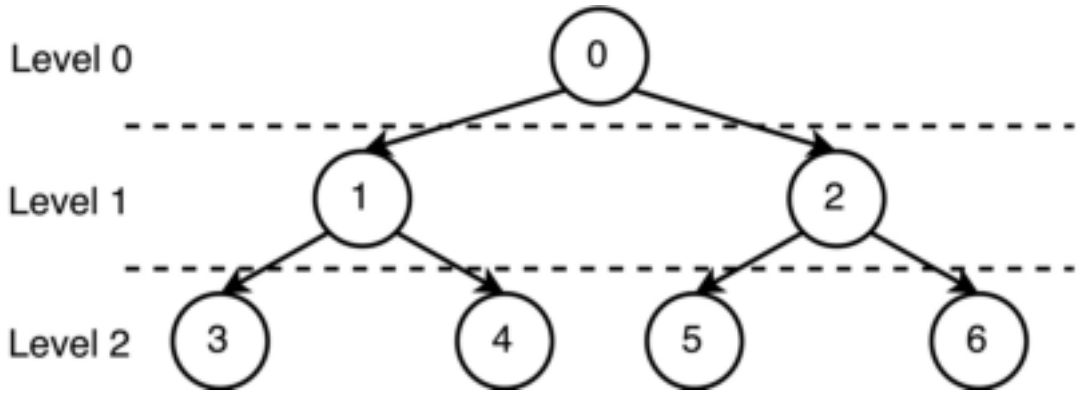


Figure 6.3: Decision Tree Structure

There are many design aspects we can pick out from Figure 6.3. First of all, a tree level m consists of 2^m nodes. Let's call the root node (0^{th} node of level 0) node 0. If we consider a binary decision tree where each node has only two children, the storage address in memory of the first child and the second child can be calculated using Equation (8) and Equation (9), respectively:

$$Left\ Child\ Address = 2 * Parent\ Address + 1 \quad (8)$$

$$Right\ Child\ Address = 2 * Parent\ Address + 2 \quad (9)$$

This means that in each tree level we would have to store the information of all the nodes within that level. Nevertheless, in this case we will be faced by a slight design obstacle when we store the node information in memory. Let us take level 2 as an example to illustrate the design problem. If we were to store the information of nodes 3, 4, 5, and 6 in memory, they would be stored as the 0^{th}, 1^{st}, 2^{nd}, and 3^{rd} nodes, respectively, within level 2. Therefore, Equations (8) and (9) will no longer provide us with the right node addresses. We noticed that the published literature that implemented a hardware-based random forest did not tackle such a problem. In the worst-case scenario the design would fail as Equations (8) and (9) will not work properly, but in the best-case scenario this might result in a waste of memory resources as we would need to duplicate the entire tree contents at each level in the tree. As a result, one of the contributions of this work is the introduction of the concept of effective address within our design, whereby the effective address of a node within the tree level can be

calculated using Equation (10). Effective address helps eliminate the previously mentioned problem as we would no longer need to duplicate tree contents at each level in the tree.

$$Effective\ Address = Original\ Address - 2^m + 1 \quad (10)$$

If we were to work out the effective address of nodes 3, 4, 5, and 5 using m = 2 since m stands for the level number, they will turn out to be 0, 1, 2, and 3 respectively. This design necessitates the use of complete trees in order for the previous equations to work properly. As a result, if we were to zoom into one of the levels of a tree, we would see the hardware structure shown in Figure 6.4. First of all, we can see the effective address being computed using a subtractor and an adder. The effective address is then used to index the tree memory which holds the information about all nodes within the particular tree level (more on this in Section 6.4). Notice that the tree memory is triggered on the negative edge of the clock since all pipeline registers are triggered on the positive edge of the clock. This allows signals to stabilize before we can request for node information from memory. The tree memory will fetch the feature index which is simply the index of the feature being checked within this node. As mentioned earlier, 26 features have been extracted, so feature index is a number between 0 and 25. The feature index is then used as selection lines to a multiplexer that will enable the value of only the feature under inspection to pass through for comparison.

In addition to the feature index, the tree memory will also provide us with the feature threshold which is the value against which the feature is compared. For example, if the node was checking if "packet size > 5" then the feature threshold will be 5. Next, both the selected feature and its threshold are passed to a comparator that checks whether the selected feature is greater than the feature threshold. The result of the comparison is then used in the calculation of the next address. The way we calculate the next address is by first shifting the parent address one bit to the left, which is mainly the part where we multiply the parent address by 2 in Equations (8) and (9). After that, depending on the result of the comparison we either add a 1 or a 2 to the shifted address. This results in calculating the child address. Lastly, in order to design a highly pipelined architecture, a state register is used at the end of each tree level in order to store the outcomes of that level. In the tree level register, we simply register the network packet and the child address such that they can be passed on to the next tree level. In the last level of the tree, we also register either the class number (majority-based) or the class

probabilities (probability-based) which are also fetched from the tree level memory. In the intermediate tree levels, we can simply leave the class signal unconnected, whereas in the last tree level we can leave next address and data unconnected resulting in a more general design for tree levels.

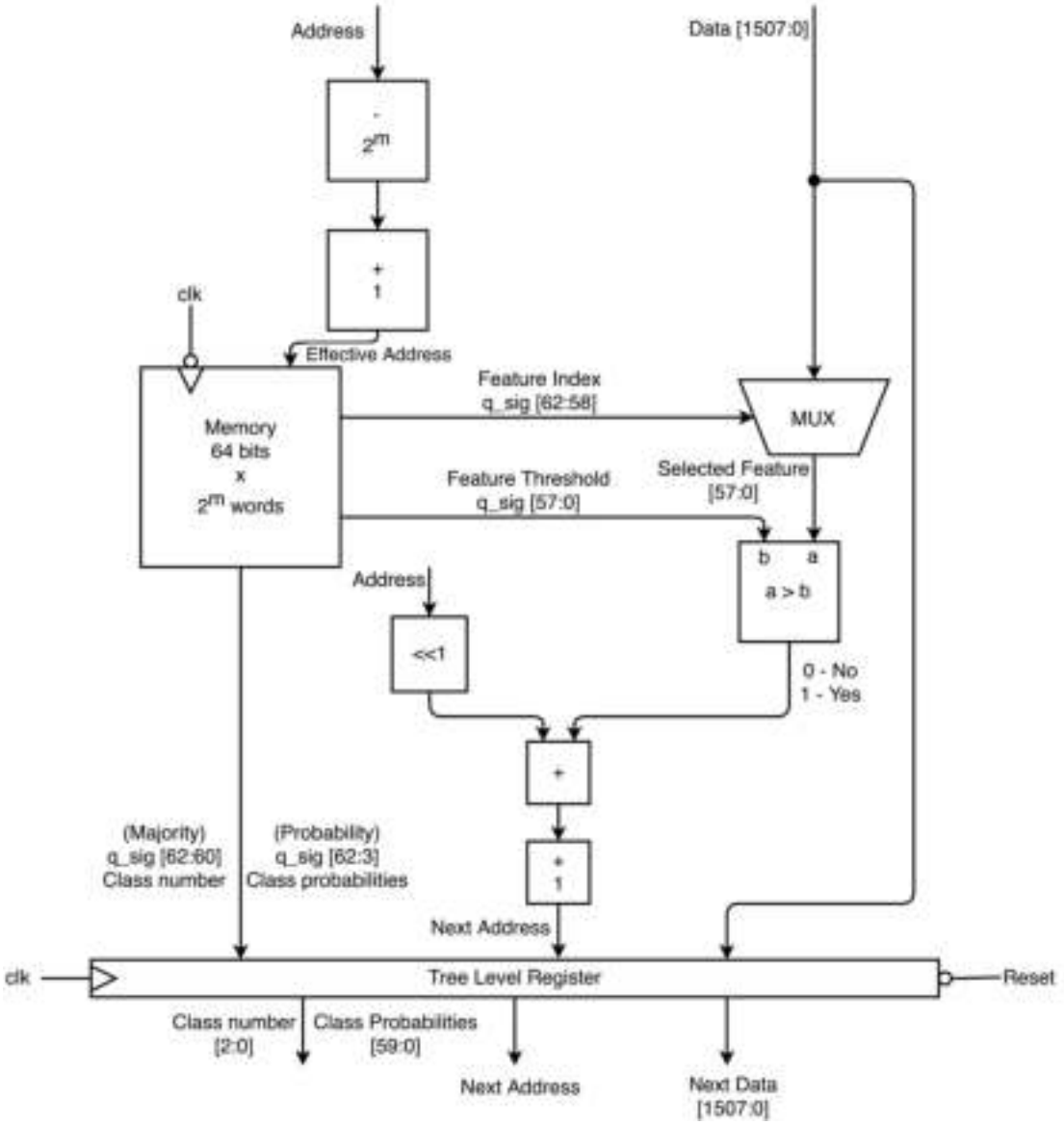


Figure 6.4: Tree Level Architecture

6.4. Tree Memory

Figure 6.5 gives a better insight into the design of the tree memory and shows how the different fields of a node in the tree are stored in the memory. The tree memory is a 64-bit memory where the MSB is a flag that indicates whether a node is a leaf node.

If a node was not a leaf node, it means that bit 63 is a 0 indicating that what follows is the feature index of that particular node. Twenty-six features would require 5 bits to index each of them and hence the feature index is usually stored at bits 62-58. The 58-bit feature threshold comes immediately after the feature index taking up the space from bit 57 to bit 0. The structure of a non-leaf node is fixed for both the majority-based and the probability-based algorithms.

The situation is slightly different when it comes to leaf nodes. This is because in leaf nodes we usually want to store either the class label (majority-based) or class probabilities (probability-based). In both cases, and since this is a leaf node, the 63rd bit is always set to 1. In case of a majority-based random forest algorithm, the class label is stored from bits 62 to 60. Only three bits are used for the class label since we are dealing with only five classes in our problem. The rest of the bits in a majority-based model are don't cares (bits 61-0). On the other hand, if we were dealing with a probability-based model, we simply store the probability of each class as a 12-bit fixed-point number, where 1 bit is used for the integer part and 11 bits are used for the fractional part. Finally, the size of the tree memory at level m is usually 64 bits (per node) * 2^m words since level m usually has 2^m nodes.

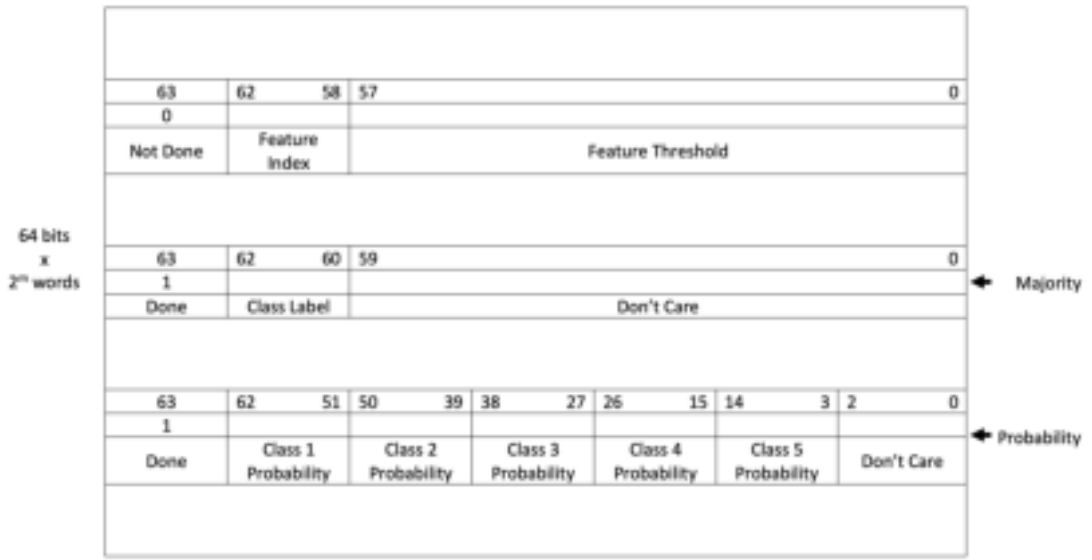


Figure 6.5: Tree Memory Design

To exploit the parallel capabilities of the FPGAs, we use the on-chip memory to act as the tree memory and store all the node information. This is due to the ability of an FPGA to restructure its on-chip memory on-demand such that each tree level can

have simultaneous access to its tree level memory without creating a memory access bottleneck at the other levels of the same tree or even other trees in the forest. By doing so, all levels in all trees can fetch their node information at the same time.

6.5. Class Tally (Majority-Based)

In a majority-based model, after each tree casts its vote what we need to do is to find out how many trees voted for each class. This is exactly the task of the Class Tally module in the majority-based design. We simply have a counter module for each of the five classes, where the input to each module is the votes by all the decision trees in the forest. The output of each counter is the number of times class *k* was selected by the decision trees of the random forest, where in our problem $k = 5$ since we have five classes. Figure 6.6 shows the general design of the Class Tally module for a majority-based algorithm.

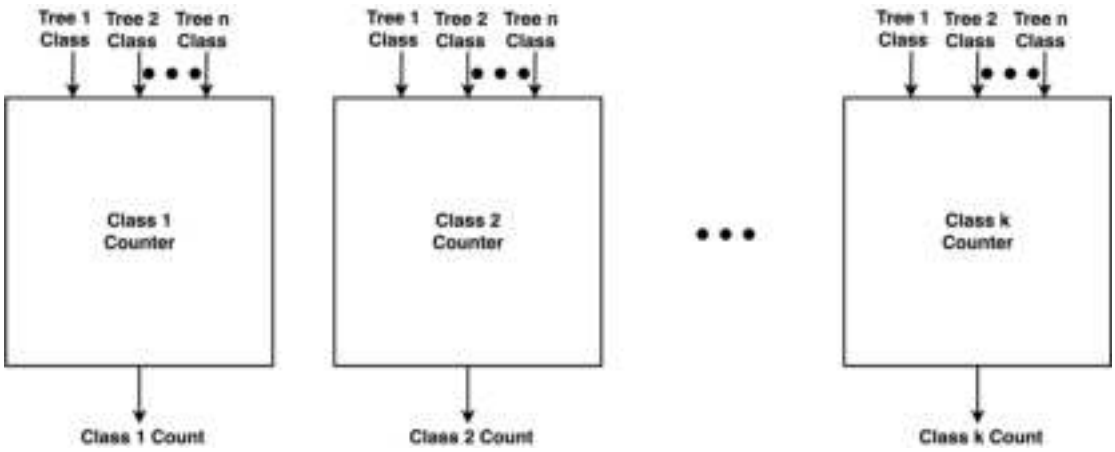


Figure 6.6: Class Tally Module Design (Majority-Based)

6.6. Class *k* Counter (Majority-Based)

If we zoom into one of the class counters in the Class Tally Module, we would find a number of comparators that is equal to the number of trees in the random forest, *n*. Each comparator will simply check whether the output class of each tree is equal to *k*, the class number the module is concerned with. If they are equal, a 1 is output otherwise a 0 is output. Eventually, an adder adds up the number of ones resulting from the different comparators. By doing so, we calculate the number of times a class *k* was chosen by the decision trees in the random forest. Figure 6.7 shows the hardware design of the Class *k* Counter module.

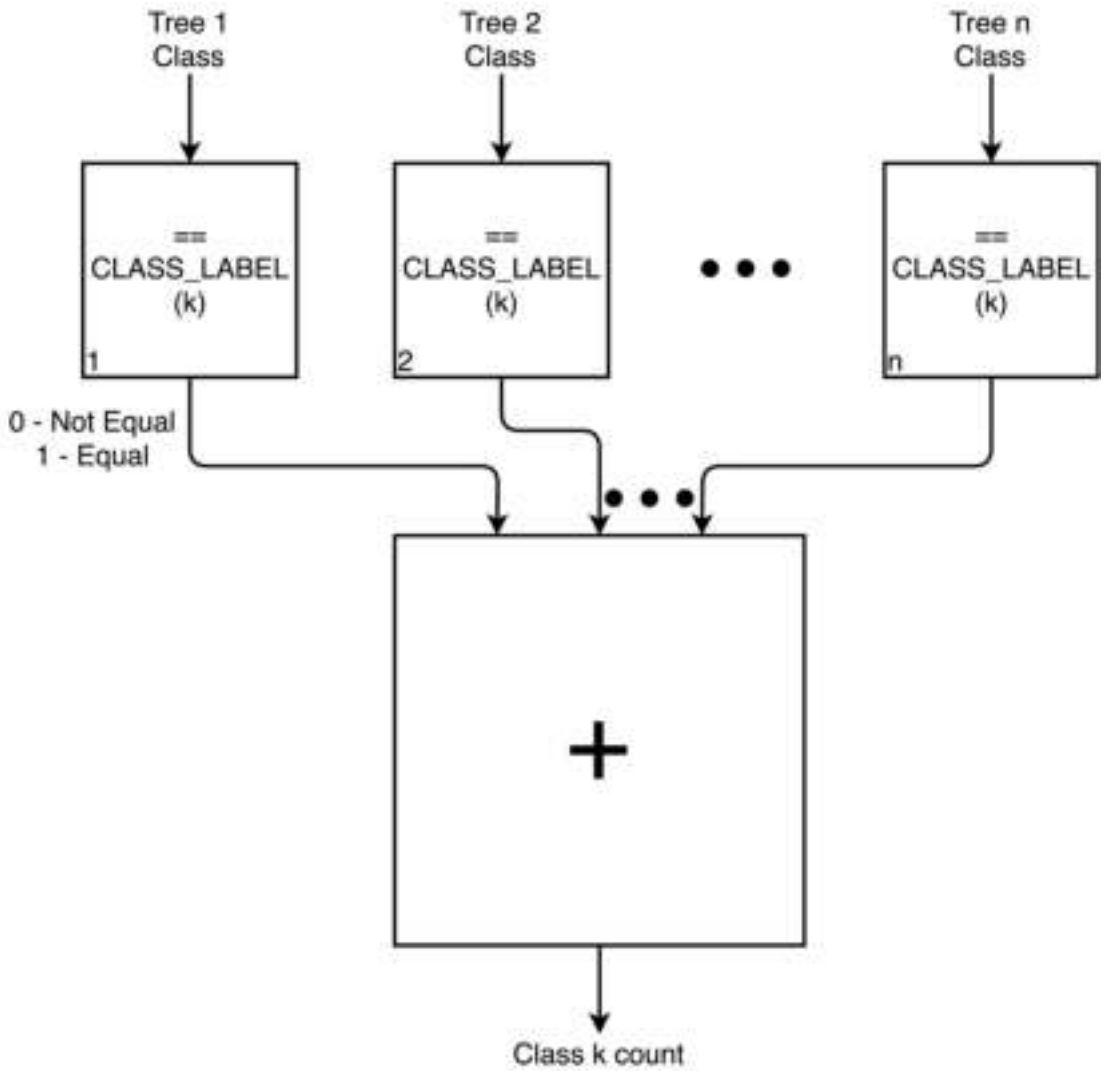


Figure 6.7: Class k Counter Architecture

6.7. Voter (Majority-Based)

Upon receiving the total count of each of the *k* classes, we are now only left with selecting the most occurring class. The way we do this is by simply comparing the *k* counts and choosing the class with the highest count. Figure 6.8 shows the design of the Voter module in a majority-based algorithm. We compare the values of class 1 count and class 2 count using a comparator. The result of the comparison is used as a selection line to a multiplexer. If the comparison turns out to be in favour of class 1, we would simply route class 1 count through the first multiplexer, and vice versa. In order to keep track of which class count was routed through the multiplexer, we concatenate the CLASS_LABEL, which is a number from 1 to *k* that represents the class, and the class count corresponding to it. After that, the routed class count will go into the next

comparator to be compared against class 3 count. This process repeats until the last multiplexer routes the CLASS_LABEL with the highest count declaring it as the majority class. Notice that in case of a tie between two class counts, this architecture gives more priority to class 1, then class 2, followed by class 3, 4, and lastly class 5 with the least priority.

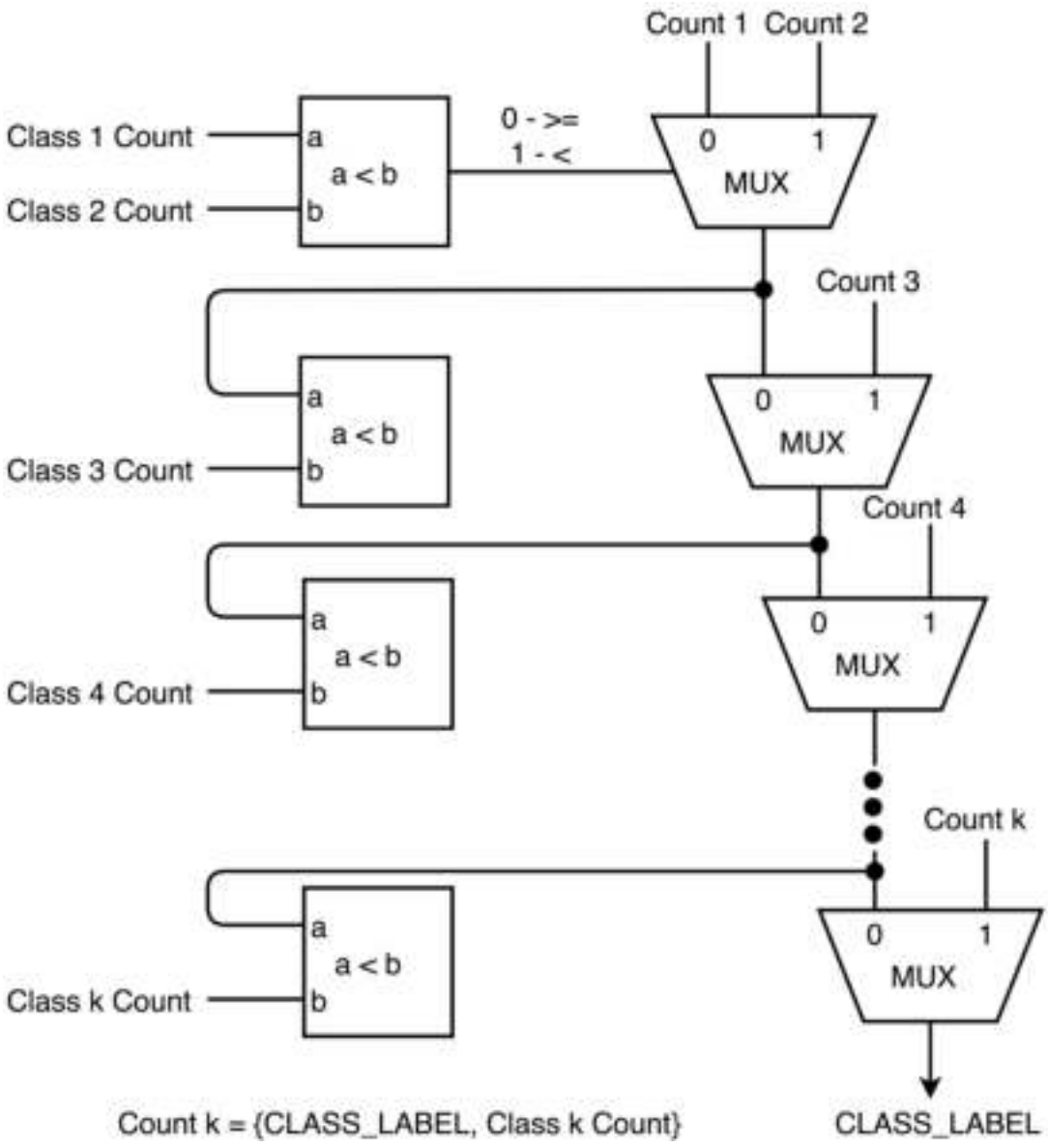


Figure 6.8: Voter Module Architecture (Majority-Based)

6.8. Class Tally (Probability-Based)

The Class Tally module of a probability-based algorithm, unlike that of the majority-based one, is more concerned with finding the average probability of each class resulting from the probabilities obtained from each decision tree. To do so, we

need to simply add up the corresponding probabilities of each class and then divide by the number of trees in the forest. However, division is a very expensive hardware operation that requires lots of hardware and time to execute. Therefore, we simplify our design by avoiding the division operation since it is not required anyway as calculating the sum of all probabilities would suffice as we would eventually pick the highest sum. Hence, we can see in Figure 6.9 that we have an adder for each class, resulting in five adders in total, and each adder is concerned with class probabilities that correspond to one class. For example, the first adder adds up class probabilities from bit 59 to bit 48 which correspond to the probability of class 1of all decision trees. The result of the Class Tally module is the sum of all probabilities for each class.

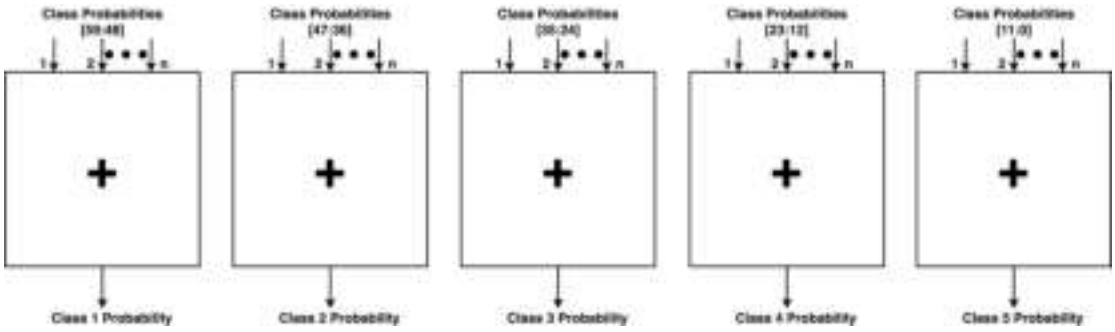


Figure 6.9: Class Tally Module Design (Probability-Based)

6.9. Voter (Probability-Based)

The Voter module of the probability-based algorithm is very similar in concept to that of the majority-based algorithm. This is because we would like to select the class with the highest sum of probabilities as opposed to the most occurring class. The only difference in the Voter module of the probability-based model is the fact that we pass class probabilities instead of class counts to the comparators and the multiplexers. Consequently, the Count k inputs to the multiplexers represent the concatenation of CLASS_LABEL and the class k probability instead of class k count. Figure 6.10 shows the architecture of the Voter module of a probability-based model.

6.10. Hardware Platform

Now that we have detailed the hardware design of a random forest classifier, it is time to start implementing it on a real hardware platform. In this work, we have used the DE2-115 development board manufactured by Terasic Inc [34]. The DE2-115 board features a Cyclone IV E FPGA chip designed and manufactured by Altera (now Intel).

It is a well-known educational board used for testing and prototyping academic and research projects. It is considered one of the most appealing FPGA boards due to its low cost, low power and an enormous supply of logic, memory and DSP capabilities. The FPGA chip on the DE2-115 board offers 114,480 configurable logic blocks (CLBs), up to 3.9 Mbits of on-chip RAM, 266 multipliers and 529 GPIO pins [34]. Besides, the DE2-115 board offers a wide range of on-board memory including 2 MB SRAM, 128 MB SDRAM, 8 MB flash memory, and 32 kb EEPROM. Moreover, the DE2-115 board features several I/O devices including a 16x2 LCD display, 26 LEDs, 4 pushbuttons and 18 switches. Furthermore, the DE2-115 allows connections to numerous external I/O devices, like keyboard, mouse, VGA monitor, camera, microphone, speaker, Ethernet, RS-232 communication port, Secure Digital card and infrared. Figure 6.11 shows the DE2-115 board with all of its peripherals.

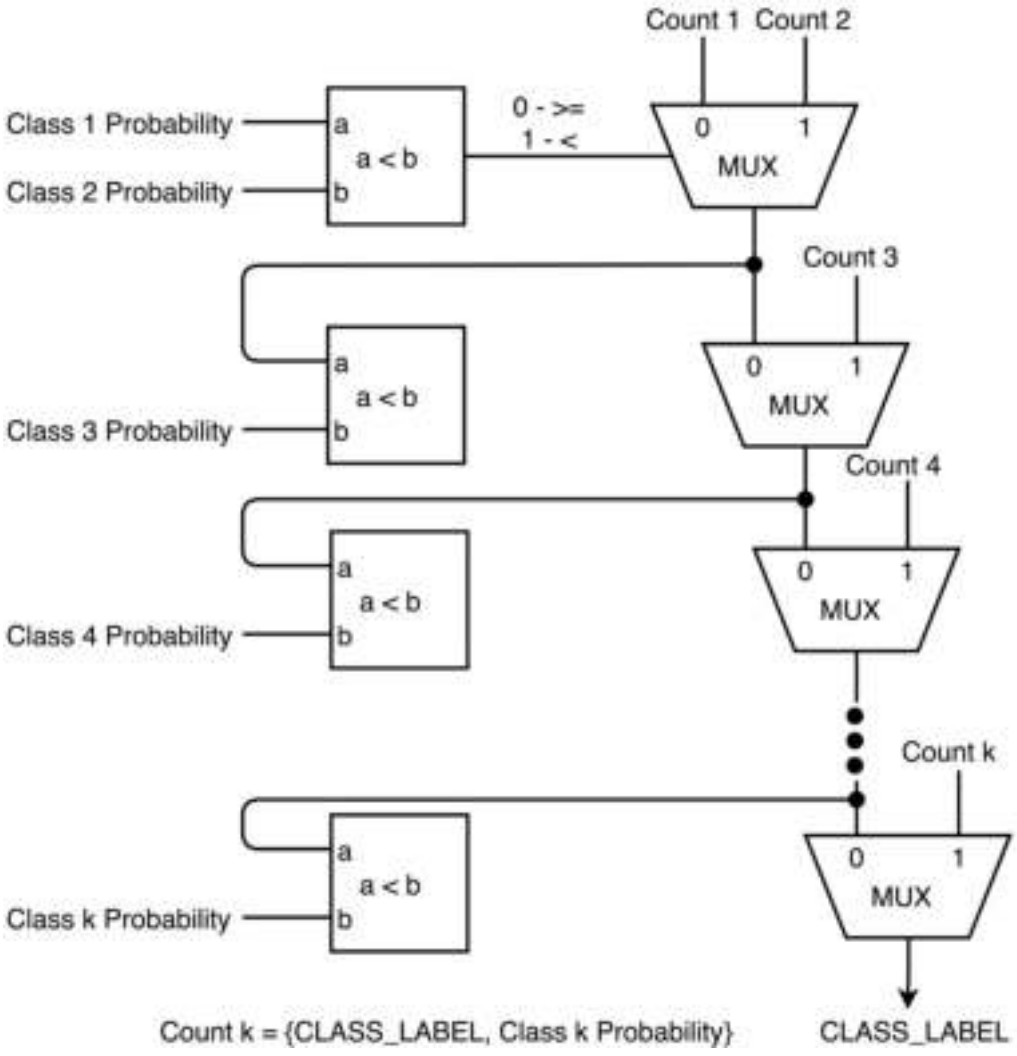


Figure 6.10: Voter Module Architecture (Probability-Based)

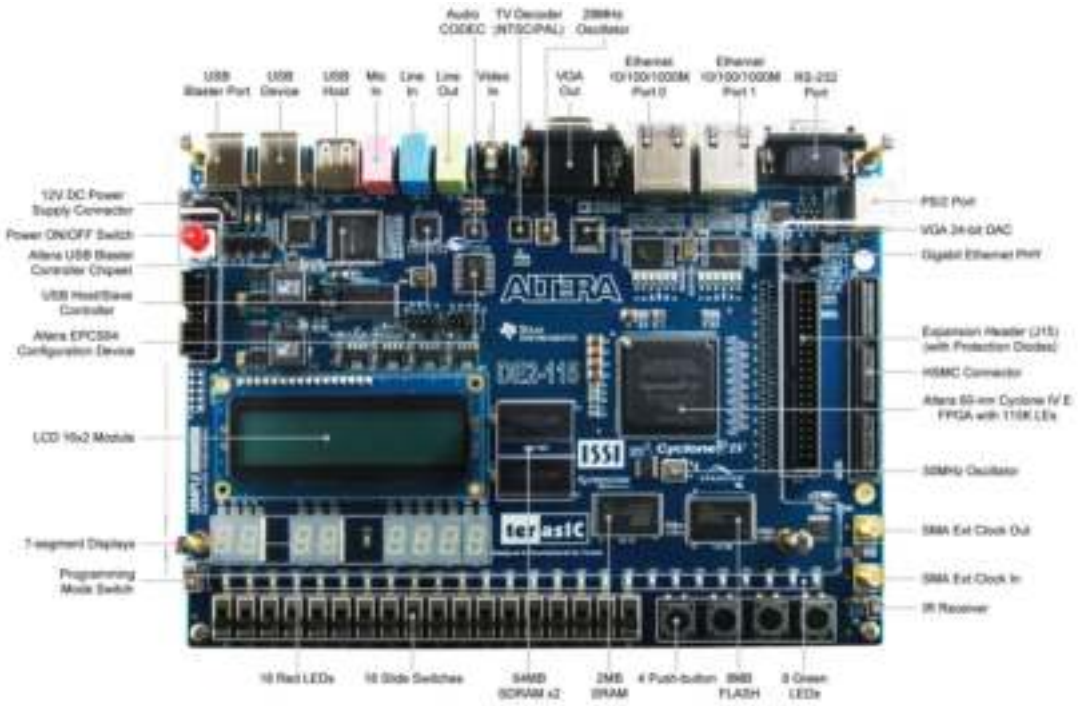


Figure 6.11: DE2-115 Development Board [34]

Chapter 7. Experimental Results

In this chapter, we discuss the results obtained using the various experiments described in Chapter 5. Before discussing the obtained results, we introduce the performance metrics used to assess the performance of the several classifiers built during the experiments. Therefore, in Section 7.1 we discuss our performance measures.

7.1. Performance Measures

In order to define the performance measures used, consider the simple confusion matrix and example shown in Table 7.1 and Table 7.2, respectively. We use Table 7.1 and Table 7.2 as basis for our discussion.

Table 7.1: Confusion Matrix Template

Actual\Predicted	Yes	No
Yes	True Positive (TP)	False Negative (FN)
No	False Positive (FP)	True Negative (TN)

Table 7.2: Classification Example

Actual\Predicted	Yes	No
Yes	20000	0
No	1000	1

One can notice that the classifier of Table 7.2 is biased towards class 'Yes' since it classifies almost all data instances to that class. To assess the performance of such a classifier, we discuss important performance measures like accuracy, precision, recall, and F-score.

7.1.1. Accuracy. The classification accuracy is defined as the percentage of instances classified as their true class labels. It is also known as the recognition rate since it resembles the percentage of test set instances that are correctly classified. Therefore, we can compute the accuracy of a classifier using Equation (11):

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} * 100\% \quad (11)$$

Therefore, using the values of Table 7.2 the accuracy can be calculated as:

$$Accuracy = \frac{20000 + 1}{20000 + 1 + 1000 + 0} * 100\% = 95.2383\%$$

where 20000 is the true positive, 1 is the true negative, 1000 is the false positive, and 0 is the true negative.

Hence, by simply looking at the classification accuracy one might think that this is a very good classifier. Indeed, it looks good at first sight, however, such a high classification accuracy might be misleading in several situations. By closely inspecting the confusion matrix, we realize that this classifier almost classifies every instance as 'Yes' regardless of whether it actually is a 'Yes' or a 'No'. Therefore, it is somehow biased towards class 'Yes'. This problem arises significantly when the dataset in hand is unbalanced since we have 20000 instances that are actually 'Yes' and only 1001 that are 'No'. Of course, this type of classifier is not a really good classifier since it blindly classifies instances into the majority class. This might create some issues in specific applications including our traffic classification application. Consider the case where 'Yes' means safe traffic and 'No' means malicious traffic. If we were using such a classifier that blindly assigns all traffic traces to the majority class which is safe, then we might incur a huge cost of allowing malicious traffic that is very dangerous to enter our system. Although classification accuracy tends to give us a good feel for the classifier's performance, unfortunately, it does not capture this problem. Therefore, we needed better performance measures that capture the goodness of classification. Nevertheless, we would still use accuracy as it serves as a good initial indicator in assessing the classifiers.

7.1.2. Precision. Precision is defined as the percentage of instances that were classified as X and are actually X. Therefore, precision is sometimes referred to as the exactness of the classifier. Precision is usually computed using Equation (12):

$$Precision = \frac{TP}{TP + FP} \tag{12}$$

Therefore, using the values of Table 7.2 the precision of class 'Yes' can be calculated as:

$$Precision = \frac{20000}{20000 + 1000} = 0.9524$$

Whereas, the precision of class 'No' can be calculated as:

$$Precision = \frac{1}{1 + 0} = 1$$

where 20000 is the true positive, 1 is the true negative, 1000 is the false positive, and 0 is the true negative.

Upon inspecting the precision for both classes, we can tell that the classifier is good at assigning instances to the actual classes that they come from. So, this

performance measure on its own does not solve the issue of accuracy mentioned earlier, therefore, we look into using a complementary performance measure that, along with precision, tackles the accuracy's issue.

7.1.3. Recall. Recall is defined as the percentage of instances that are actually X and were labelled as X by the classifier. It is also known as the completeness or the sensitivity of the classifier. Recall is computed using Equation (13):

$$Recall = \frac{TP}{TP + FN} \tag{13}$$

Therefore, using the values of Table 7.2 the recall of class 'Yes' can be calculated as:

$$Recall = \frac{20000}{20000 + 0} = 1$$

Whereas, the recall of class 'No' can be calculated as:

$$Recall = \frac{1}{1 + 1000} = 0.000999$$

where 20000 is the true positive, 1 is the true negative, 1000 is the false positive, and 0 is the true negative.

When we consider the recall values for both classes, it becomes apparent that the classifier is performing well with the 'Yes' class, however, it fails noticeably with the 'No' class. Therefore, this measure tackles the issue of unbalanced data as mentioned earlier. Now that we know about precision and recall pointing out specific problems within the performance of the classifier, we can combine them in one single measure that captures the properties of both precision and recall.

7.1.4. F-score. F-score is the harmonic mean of precision and recall; therefore, it incorporates both the precision and recall in a single value. Hence, F-score is maximum at the value of 1 and minimum at the value of 0. F-score is also known as F-measure or F1 score and can be computed using Equation (14):

$$FScore = \frac{2 * Precision * Recall}{Precision + Recall} \tag{14}$$

Therefore, using the values of Table 7.2 the F-score of class 'Yes' can be calculated as:

$$FScore = \frac{2 * 0.9524 * 1}{0.9524 + 1} = 0.9756$$

Whereas, the F-score of class 'No' can be calculated as:

$$FScore = \frac{2 * 1 * 0.000999}{1 + 0.000999} = 0.0020$$

Finally, the average F-score of this classifier is computed as:

$$Average\ FScore = \frac{0.9756 + 0.0020}{2} = 0.4888$$

This value is a perfect depiction of the problem shown earlier when using the accuracy. Therefore, F-score is considered more reliable in capturing the performance of classification algorithms, and hence, F-score will be our basis for assessing the performance of our classifiers. The only issue with F-score is that it can only be calculated for binary classification problems that include only two classes, which is not the case with our traffic classification problem that has five classes in each dataset. Nevertheless, there is a very simple turnaround to overcome this limitation. The idea is to transform an $n*n$ confusion matrix where $n > 2$ into several 2*2 confusion matrices in a similar fashion to the one-vs-one or the one-vs-all approach discussed earlier in Section 1.2 when detailing the SVM algorithm. In our case we transform the 5*5 confusion matrix into five 2*2 confusion matrices representing the true positive, true negative, false positive, and false negative of each class using the one-vs-all approach. The F-score is then calculated for each class and the average F-score is then computed as the overall F-score of this classifier.

7.2. Software-Based Classifier Performance

In this section, we look into the results obtained from running the different machine learning experiments, discussed in detail in Chapter 5, on the UNIBS and UNB datasets. We highlight the main findings of these experiments and try to analyse the reasons that led to those findings.

7.2.1. Discretization. In this experiment, we ran the original datasets, as well as, the discretized versions of the two datasets based on the two discretization algorithms described in [31, 32] through five different classifiers, namely naïve Bayes, linear SVM, 2nd order polynomial SVM, KNN, and random forest. In doing so, we used a 10-fold cross-validation technique. For each classifier, we recorded four main parameters, namely, training time, testing time, classification accuracy and F-score. The objective of this experiment is to study the effect of discretization on the UNIBS and the UNB datasets and discover whether discretization improves the classification performance.

Figure 7.1 and Figure 7.2 show the time spent while training the five different classifiers on the UNIBS and the UNB datasets, respectively. It is evident from the figures that SVM, whether linear or polynomial, takes a considerably huge amount of

time to be trained on a network traffic dataset. This could be explained by the difficulty in obtaining a separator that distinguishes the different classes due to the highly non-linear nature of the traffic datasets. On the contrary, naïve Bayes, KNN, and random forest tend to show a reasonable amount of training time with random forest being slightly higher than the other two. Another interesting observation from the two graphs is the fact that discretization reduces the training time of the SVM classifiers using the UNIBS dataset while it increases the training time using the UNB dataset. This is justified by the artificial nature of the UNB dataset which in turn makes it more difficult for the SVM classifiers to find the separating plane unlike the real-life UNIBS dataset. We can also notice that discretization usually increases the time it takes to build a random forest classifier using both datasets. This could be a drawback of discretization while using the random forest algorithm, since the time to obtain the model increases with the usage of discretization. The cases of naïve Bayes and KNN look trivial as the change in training time looks slightly insignificant when comparing the training time before and after discretization.

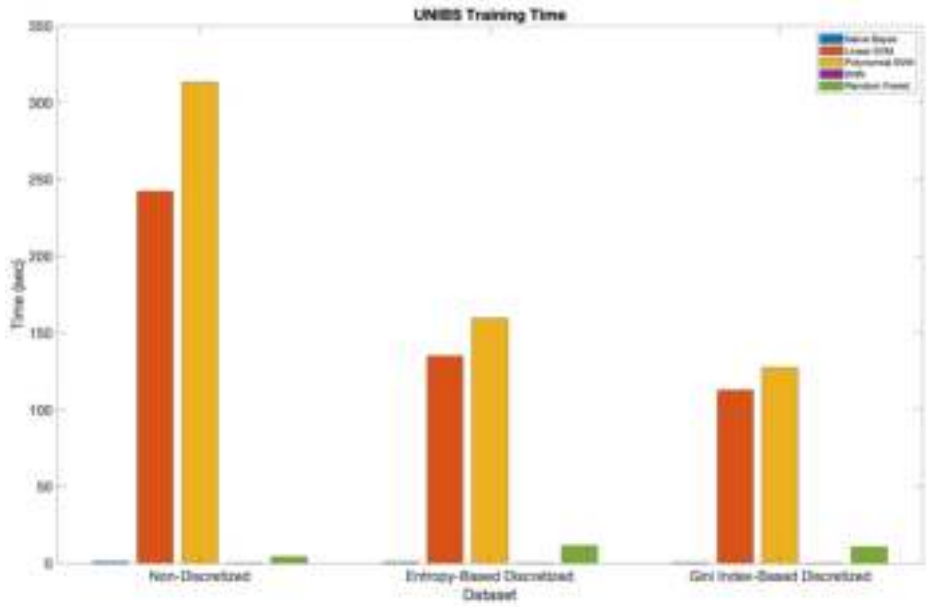


Figure 7.1: UNIBS Training Time

To summarize the change in training time between the non-discretized dataset and the discretized datasets using the two discretization algorithms, we calculate the percentage change using Equation (15):

$$Percentage\ Change = \frac{New\ value - Old\ value}{Old\ value} * 100\% \quad (15)$$

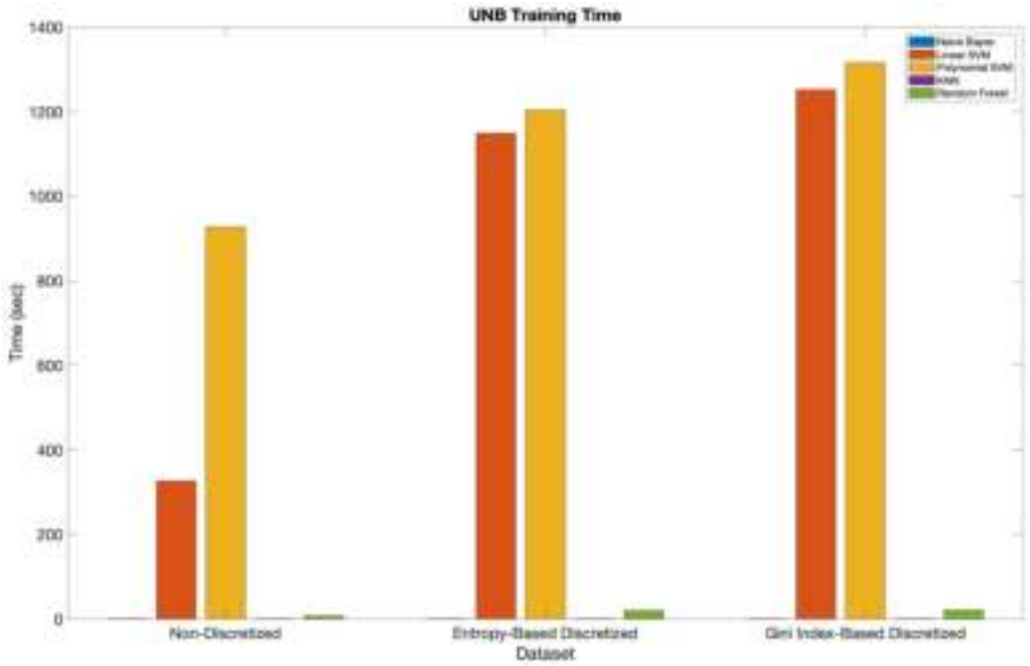


Figure 7.2: UNB Training Time

where new value represents the training time of the discretized dataset, while old value represents the training time of the non-discretized dataset. Table 7.3 and Table 7.4 summarize the percentage change of training time between the non-discretized and the discretized datasets using UNIBS and UNB datasets, respectively. A positive percentage change indicates an increase in training time after using the respective discretization algorithm compared to the non-discretized dataset, and vice-versa.

Table 7.3: Percentage Change in Training Time Using UNIBS Dataset

Algorithm	Entropy-Based Discretization	Gini Index-Based Discretization
Naïve Bayes	-45.2 %	-54.4 %
Linear SVM	-44.3 %	-53.5 %
2nd Order SVM	-49.1 %	-59.3 %
KNN	0 %	0 %
Random Forest	175.9 %	153.5 %

Table 7.4: Percentage Change in Training Time Using UNB Dataset

Algorithm	Entropy-Based Discretization	Gini Index-Based Discretization
Naïve Bayes	-59.1 %	-58.3 %
Linear SVM	252.0 %	283.9 %
2nd Order SVM	29.7 %	41.9 %
KNN	43. 6 %	25.6 %
Random Forest	175.1 %	191.7 %

Keeping in mind the ultimate objective of this work, which is to use an FPGA in order to achieve online traffic classification, usually training is done offline with the

use of a hardware accelerator. Therefore, despite the fact that percentage changes shown earlier look disappointing at first sight, we can tolerate an increase in training time as long as offline training is adopted. Therefore, in the case of an online traffic classifier we would be more concerned with the testing time rather than the training time. As a result, we decided to record the testing time using the same setup as before, while recording the difference in testing time between the non-discretized and the discretized datasets. Figure 7.3 and Figure 7.4 show the time spent on testing using the five different classifiers on the UNIBS and the UNB datasets, respectively.

The testing time plots demonstrate the difficulty in finding a simple naïve Bayes model using the non-discretized datasets. However, upon discretizing the UNIBS and the UNB datasets, the naïve Bayes model was able to classify the data instances in a much shorter time. Unlike naïve Bayes, the polynomial SVM classifier shows a significant increase in testing time when operating on the discretized datasets. This behaviour is also observed when using the random forest algorithm to classify the test instances even though the increase in testing time is not as huge as the polynomial SVM. This could be a serious drawback that stands in the way of using discretization on the network traffic datasets, since an increase in testing time defeats the purpose of having an online traffic classifier. Especially, if the increase in testing time mainly affects polynomial SVM and random forest which, as will be seen later, are the classifiers with the best performance in classifying network packets.

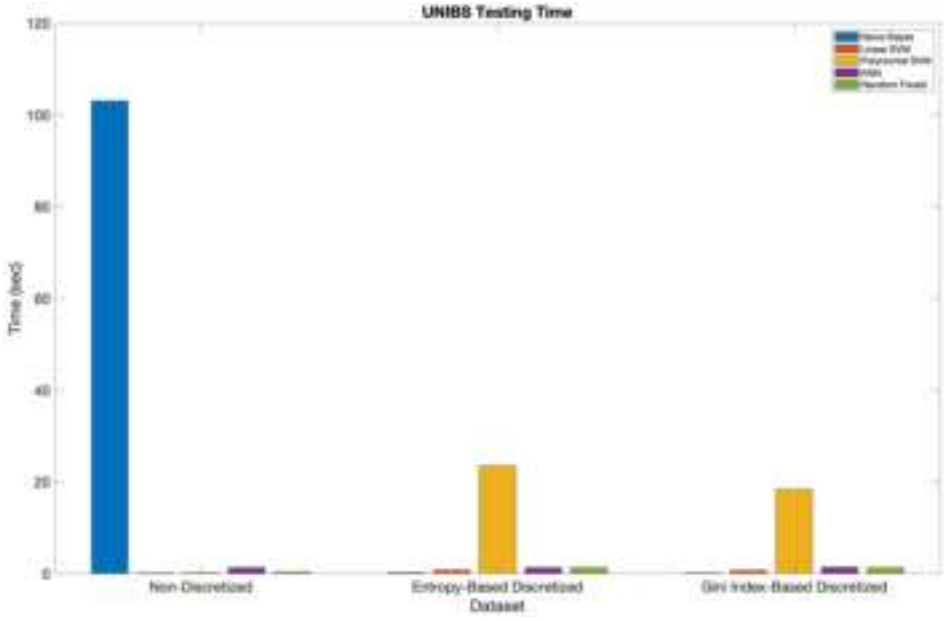


Figure 7.3: UNIBS Testing time

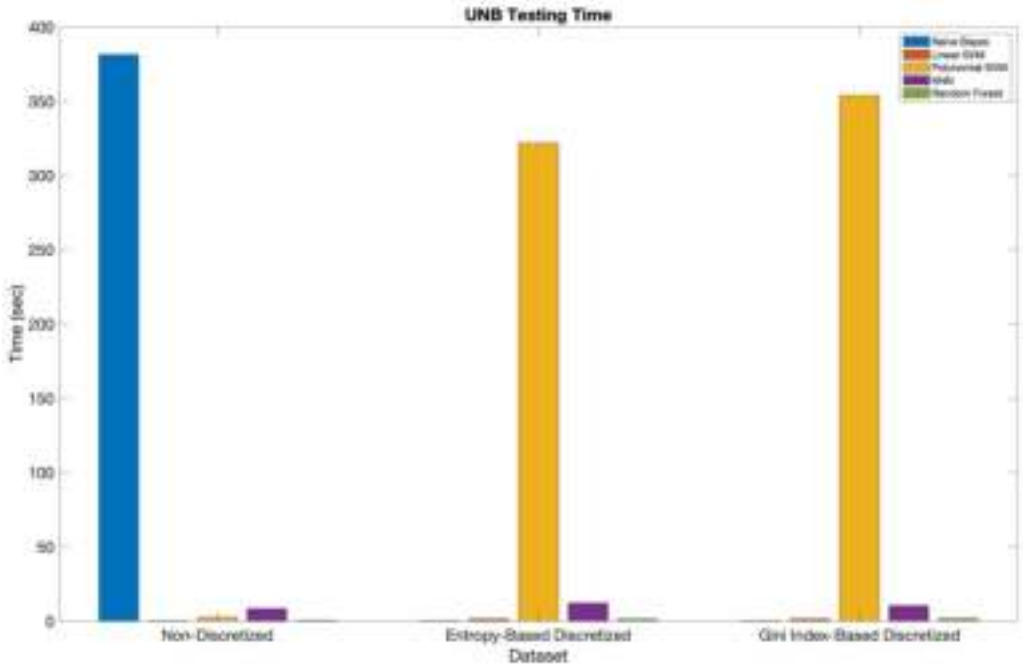


Figure 7.4: UNB Testing time

Table 7.5 and Table 7.6 summarize the percentage change of testing time between the non-discretized and the discretized datasets using UNIBS and UNB datasets, respectively.

Table 7.5: Percentage Change in Testing Time Using UNIBS Dataset

Algorithm	Entropy-Based Discretization	Gini Index-Based Discretization
Naïve Bayes	-99.8 %	-99.8 %
Linear SVM	660.6 %	568.1 %
2nd Order SVM	13308.0 %	10407.4 %
KNN	-2.5 %	6.8 %
Random Forest	218.9 %	209.7 %

Table 7.6: Percentage Change in Testing Time Using UNB Dataset

Algorithm	Entropy-Based Discretization	Gini Index-Based Discretization
Naïve Bayes	-100.0 %	-100.0 %
Linear SVM	827.0 %	914.2 %
2nd Order SVM	13672.5 %	15049.7 %
KNN	41.9 %	23.6 %
Random Forest	183.9 %	189.6 %

Now that we had a look at the time it takes to build and test the five machine learning algorithms using the UNIBS and UNB datasets, we shift our attention to the performance of those classifiers in terms of classification accuracy and F-score. This is very important in order to obtain a preliminary look on how well each of the five classifiers generalize to the datasets under investigation. Therefore, we use the 10-fold

cross-validation technique in order to have a rough idea on whether the five classifiers overfit to the datasets before diving into deeper analysis of the performance of each algorithm.

Figure 7.5 and Figure 7.6 show the classification accuracy of the five different classifiers on the UNIBS and the UNB datasets, respectively. As we can see in the accuracy results, random forest tends to outperform all other algorithms on the non-discretized datasets. The use of discretization only helps other algorithms like SVM, and naïve Bayes get closer to random forest but never surpass it. On the other hand, random forest seems unaffected by the use of discretization. This could be explained by the fact that the random forest algorithm itself discretizes the dataset on the go while building the trees that constitute the forest. Therefore, external discretization does not seem to help much in the classification accuracy of random forest. Another interesting observation is the fact that discretization does not seem to affect the KNN algorithm at all. We give credit here to the ability of KNN to handle numeric attributes really well while calculating the distance between two instances. Hence, KNN does not seem to require the help of discretization to improve its performance.

Table 7.7 and Table 7.8 summarize the percentage change of classification accuracy between the non-discretized and the discretized datasets using UNIBS and UNB datasets, respectively.

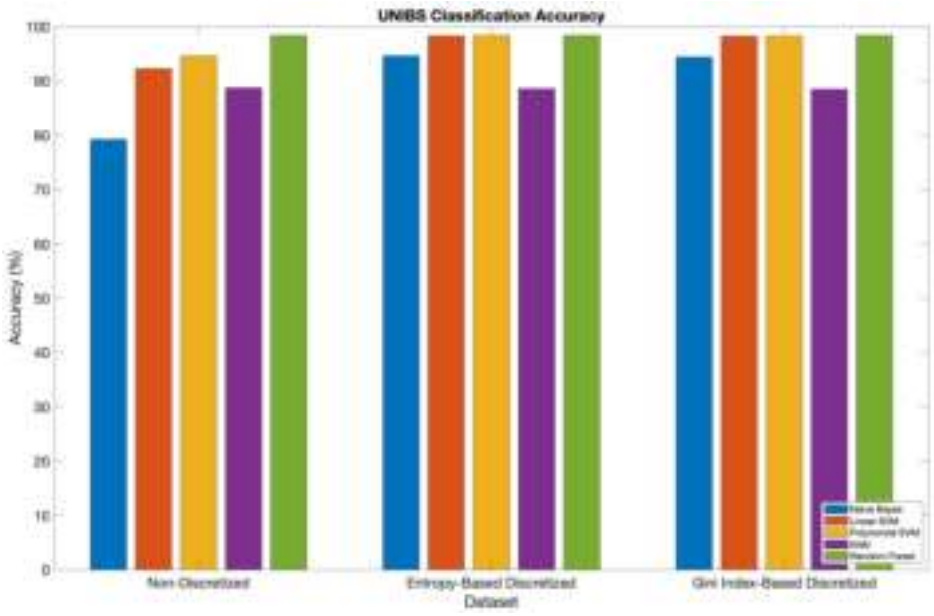


Figure 7.5: UNIBS Classification Accuracy with Discretization

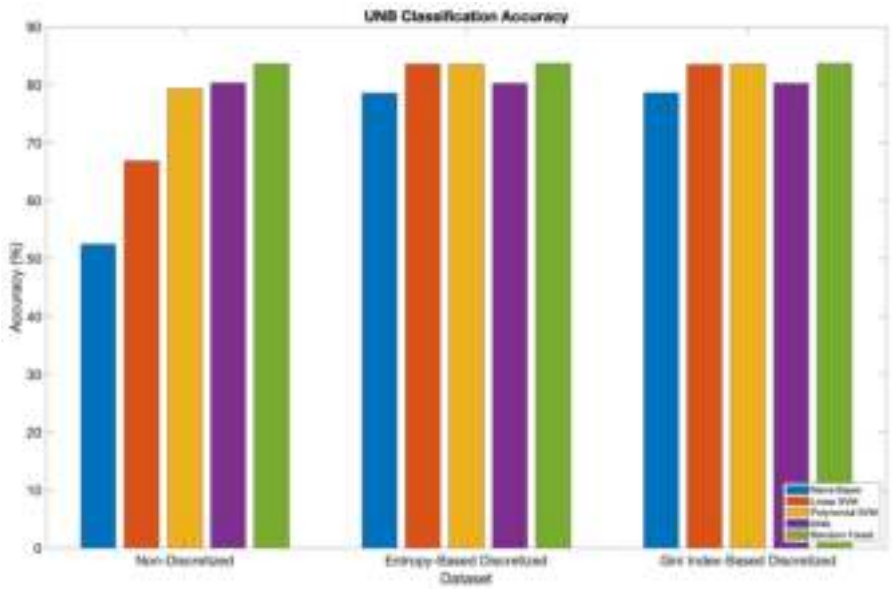


Figure 7.6: UNB Classification Accuracy with Discretization

Table 7.7: Percentage Change in Classification Accuracy Using UNIBS Dataset

Algorithm	Entropy-Based Discretization	Gini Index-Based Discretization
Naïve Bayes	19.5 %	19.2 %
Linear SVM	6.6 %	6.5 %
2nd Order SVM	4.0 %	4.0 %
KNN	-0.1 %	-0.2 %
Random Forest	0.06 %	0.01 %

Table 7.8: Percentage Change in Classification Accuracy Using UNB Dataset

Algorithm	Entropy-Based Discretization	Gini Index-Based Discretization
Naïve Bayes	49.8 %	49.8 %
Linear SVM	25.0 %	24.9 %
2nd Order SVM	5.4 %	5.3 %
KNN	-0.1 %	-0.1 %
Random Forest	0.08 %	0.07 %

We also inspect the performance of the five classifiers on the two datasets using another important performance metric which is the F-score. Therefore, while conducting the pervious experiment we also record the F-scores of all models while recording the accuracy. Figure 7.7 and Figure 7.8 show the F-score of the five different classifiers on the UNIBS and the UNB datasets, respectively. The F-score bar plots further solidify the conclusions drawn from the accuracy bar plots. We notice that random forest outperforms all other algorithms on the non-discretized dataset. We also notice that discretization does not help in improving the performance of the random forest algorithm due to its built-in discretization. Nevertheless, discretization helps

SVM and naïve Bayes get closer to the random forest but never exceed its F-score. Finally, KNN proves once more its resilience to discretization.

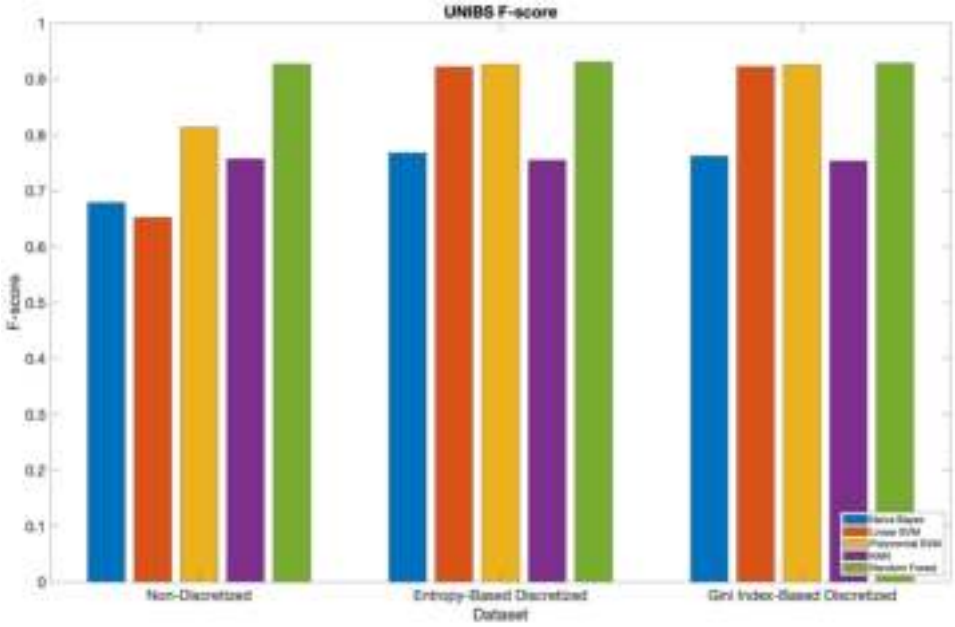


Figure 7.7: UNIBS F-score with Discretization

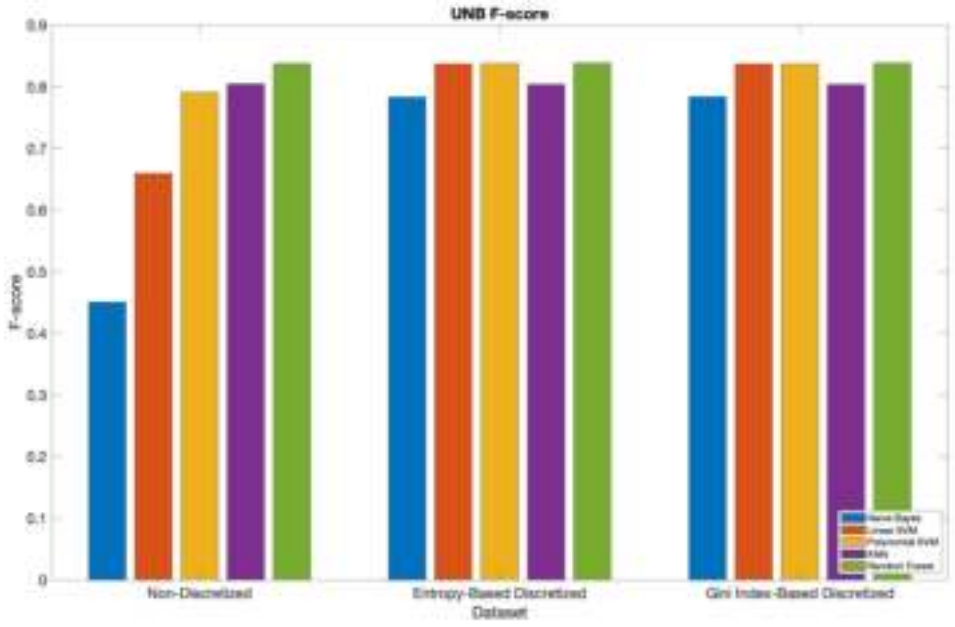


Figure 7.8: UNB F-score with Discretization

Table 7.9 and Table 7.10 summarize the percentage change of F-score between the non-discretized and the discretized datasets using UNIBS and UNB datasets, respectively.

Table 7.9: Percentage Change in F-score Using UNIBS Dataset

Algorithm	Entropy-Based Discretization	Gini Index-Based Discretization
Naïve Bayes	13.1 %	12.2 %
Linear SVM	41.3 %	41.4 %
2nd Order SVM	13.9 %	13.8 %
KNN	-0.3 %	-0.5 %
Random Forest	0.4 %	0.2 %

Table 7.10: Percentage Change in F-score Using UNB Dataset

Algorithm	Entropy-Based Discretization	Gini Index-Based Discretization
Naïve Bayes	73.8 %	73.8 %
Linear SVM	27.0 %	27.0 %
2nd Order SVM	5.9 %	5.9 %
KNN	-0.1 %	-0.1 %
Random Forest	0.08 %	0.07 %

Looking at the results obtained from the discretization experiment, we can conclude that random forest is usually not affected by discretization due to its built-in discretization mechanism that discretizes numeric attributes at each decision node. This is implicitly done by the decision node as it usually checks whether the attribute is above or below a specific threshold. We can also conclude that even though discretization helps SVM and naïve Bayes close in the performance gap with random forest, they never surpass the performance of the random forest. Keeping in mind that discretization usually increased the training and testing time of random forest significantly, whereby the increase in time incurred to test new instances is not justified by the very small increase in random forest performance, we can safely discard the idea of discretization and hence we can move on to perform the remaining experiments on the non-discretized dataset.

7.2.2. Cross-validation. In this experiment, we ran the two datasets through five different classifiers, namely naïve Bayes, linear SVM, 2nd order polynomial SVM, KNN, and random forest. We used the typical 10-fold cross-validation mechanism for testing in order to assess their overfitting performance. Below are the results obtained for each dataset.

7.2.2.1. UNIBS results. Figure 7.9 shows the classification accuracy obtained using the UNIBS dataset. First of all, by simply observing the all features performance we can notice that most classifiers perform nearly the same with their accuracies reaching around 99% except for the naïve Bayes classifier that falls behind at almost 90%. The discrepancy in the naïve Bayes classifier could be explained by its naïve assumption of feature independence. Therefore, it might be the case that features are

not independent and hence the classification accuracy deteriorates when using naïve Bayes. Another reason could be the fact that it treats all features equally and considers them having the same importance, hence feature selection is not performed by naïve Bayes and therefore, the usage of irrelevant features in classification might be another reason for its bad performance. Moreover, the fact that we do not use discretization in this experiment could be another reason for the poor performance of naïve Bayes. The second set of results that belongs to ports only might be a good justification for the all features case since using port numbers only has boosted the accuracy of naïve Bayes in a way that enables it to catch up with the other classifiers. This might also suggest that port numbers are the most dominant features that tremendously affect the classification performance. This confirms our initial concerns that were discussed earlier when plotting histograms of the different features. Also, it suggests that if applications were to dynamically change their port numbers in order to deceive traffic classifiers, we might fall into a serious problem of not having reliable port numbers that can facilitate the classification process.

As a result, we have also tried different sets of experiments that rely on the original features except that port numbers were removed before training the different classifiers. This was done on purpose to mimic the worst-case scenario where port numbers might be completely random and add no value to the classification process. The new set of results also shown in Figure 7.9 show that the accuracy drops when port numbers are removed which totally makes sense as we would expect classifiers to find more difficulties in classifying traffic traces when port numbers are missing or randomized. Nevertheless, we notice that random forest, which is by far the best performer on the UNIBS dataset, does not drop significantly as it saturates at almost 97% with a drop of only 2% in accuracy from the experiments that included port numbers. This is a very promising finding as we can be sure that even if port numbers were completely randomized, we can still be able to classify traffic with a very high accuracy. Another observation shows that feature selection using stepwise regression or random forest feature selection does not greatly impact the performance of the classifiers when compared to the all features results. This is true for both the no ports and the port results. This indicates that despite the fact that less features were used to classify traffic with stepwise regression or random forest feature selection, we were still able to obtain the same performance. Therefore, feature selection has enabled us to

reduce the dimensionality of our problem without sacrificing the classification performance. Another important observation is the performance difference between linear SVM and 2nd order polynomial SVM. Even though the performance is not so different when including the port numbers, the difference becomes apparent in the no ports case. Polynomial SVM tends to always perform better than linear SVM regardless of whether feature selection has been used. This also confirms our initial doubts that traffic traces are usually not linearly separable in nature. However, the performance was not so different as it was almost always about 2.5% different.

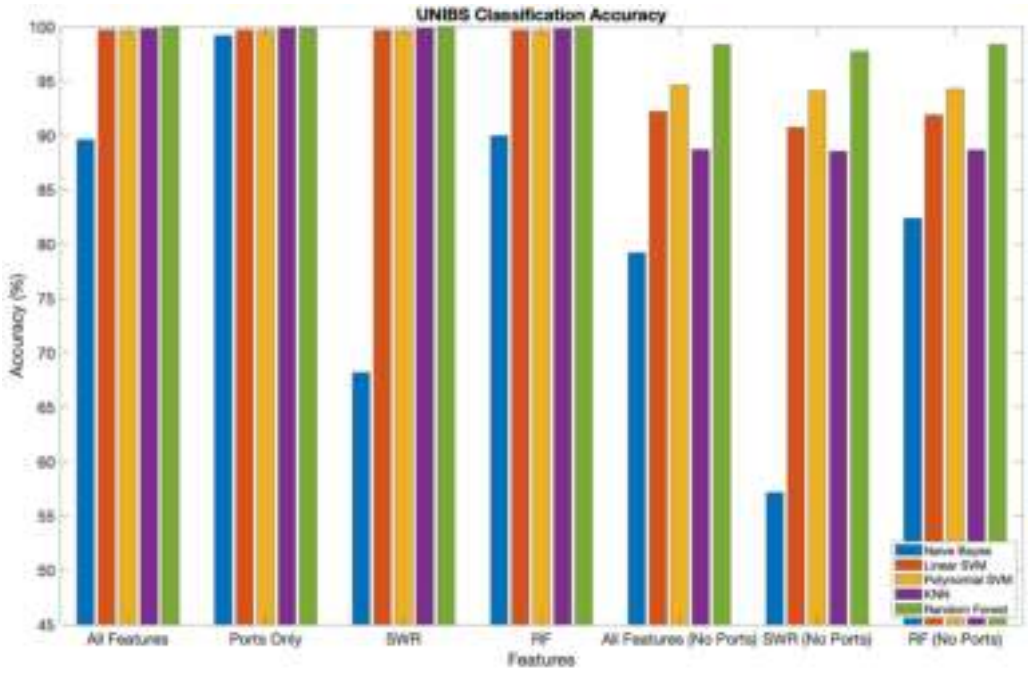


Figure 7.9: UNIBS Classification Accuracy

Figure 7.10 shows the F-scores of the different classifiers in the same experiments on the UNIBS dataset. We can see from the bar plot that they look very similar to the classification accuracies, which indicates that those classification accuracies were actually a good descriptor of the performance. However, The F-score results tend to highlight the difference in performance between random forest and all other classifiers. This is fairly obvious in the no ports cases where random forest is always above 0.9 whereas other classifiers are almost always below 0.8. We can conclude from this bar plot that random forest is a clear winner when it comes to classifying the UNIBS traffic traces. In addition, such a high F-score using the cross-validation method suggests that random forests are indeed not overfitting to the training

data as we explained earlier. Therefore, random forest becomes a very good option to avoid overfitting. Another observation that can be extracted from the F-score bar plot is that the features selected by the random forest feature selection algorithm yields a higher F-score than the all features (no ports) case when building a random forest classifier. This is one case where feature selection can in fact boost the classification performance after removing irrelevant features. Also, RF features tend to provide slightly better results when compared to the SWR features.

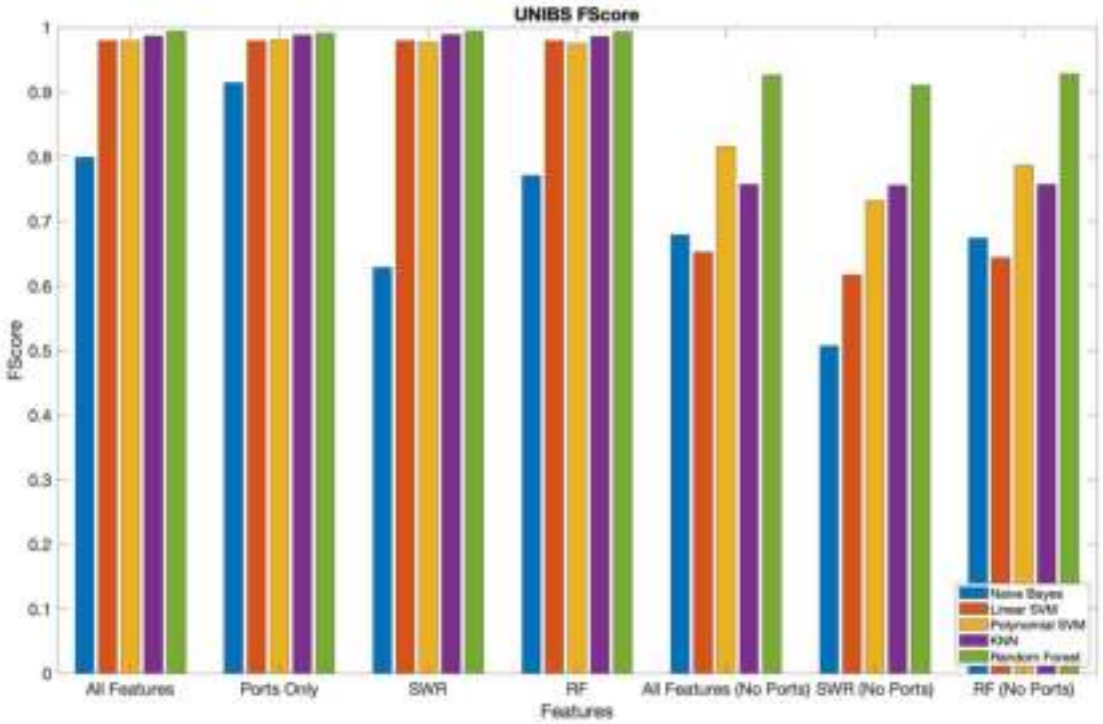


Figure 7.10: UNIBS F-score

If we were to compare our results to existing works that were done on the UNIBS dataset, we would notice that the maximum accuracy obtained by [35] and [36] is 90.6% and 91.28%, respectively, whereas the minimum accuracy that we obtained using random forest regardless of which feature set was used is about 98%. Similarly, the highest F-score recorded in [35] is around 0.964 when performing the testing on the training set itself, however, when they used a different test set their highest F-score was around 0.70. Similarly, the work published in [36] was able to achieve an average F-score of 0.9162. On the other hand, we were able to successfully achieve a minimum F-score of approximately 0.93.

If were to rate the performance of the five classifiers on the UNIBS dataset using cross-validation, naïve Bayes would come last due to its very poor performance, followed by the linear SVM classifier. After that comes the intense competition between KNN and polynomial SVM which almost always favours polynomial SVM in the no ports results except for the one case of F-score results of SWR (no ports). However, KNN F-scores also surpass polynomial SVM in the port results. Nevertheless, since no port results are of more importance to us in this research, we would also favour polynomial SVM over KNN. Finally, the best classifier by far goes to random forest which outshines all other classifiers.

7.2.2.2. UNB results. By inspecting the UNB dataset accuracy results shown in Figure 7.11, we can see a very similar behaviour to the results of the UNIBS dataset in case port numbers were included in the experiments. One important feature of the bar plot is the deterioration of naïve Bayes in all the port cases compared to the UNIBS dataset. This further illustrates the failure of naïve Bayes to classify traffic traces accurately. Therefore, from this plot we can conclude that naïve Bayes is definitely not a good traffic classifier. The ports only result also shows that all classifiers can classify traffic traces with accuracies reaching almost 100% using only port numbers. However, this is rather a drawback and not an advantage because dynamically changing port numbers might collapse such classifiers. We would also expect the no port results to drop compared to the port results as what happened with the UNIBS dataset. However, the accuracy drop in the UNB dataset is way larger than that of the UNIBS dataset as the accuracies barely reach around 84%. This significant drop could be explained by the fact that the UNB dataset is, as we mentioned earlier during our initial analysis of the datasets, not 100% pure. Therefore, the presence of impurities in the UNB dataset might have resulted in the significant accuracy drop when port numbers were removed. We also notice that the no port results are almost identical whether feature selection was used or not, which indicates the ability of feature selection to reduce the dimensionality of our problem without impacting the performance of the classifiers.

Figure 7.12 shows the F-scores obtained using the UNB datasets. The results are fairly good with random forest reaching almost 0.84 F-score across the no port experiments. Again, the UNB dataset raises the question of whether polynomial SVM is better than KNN as a traffic classifier. The UNB dataset disagrees with the UNIBS dataset, since KNN usually shows higher accuracies and F-scores compared to

polynomial SVM. Nevertheless, polynomial SVM still performs better than linear SVM which further solidifies our claim that network traffic is not linearly separable. We can then generalize this claim since it seems to be the case with two different datasets.

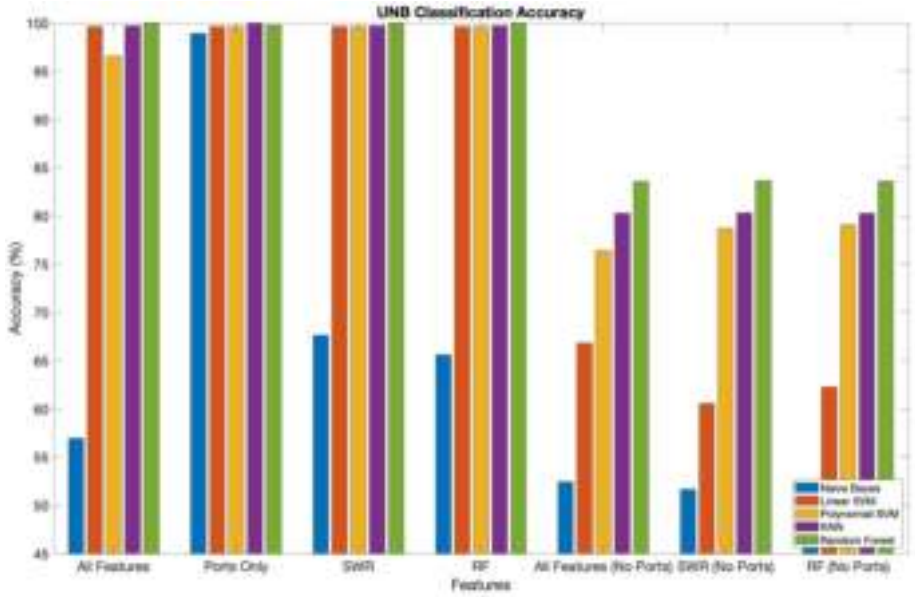


Figure 7.11: UNB Classification Accuracy

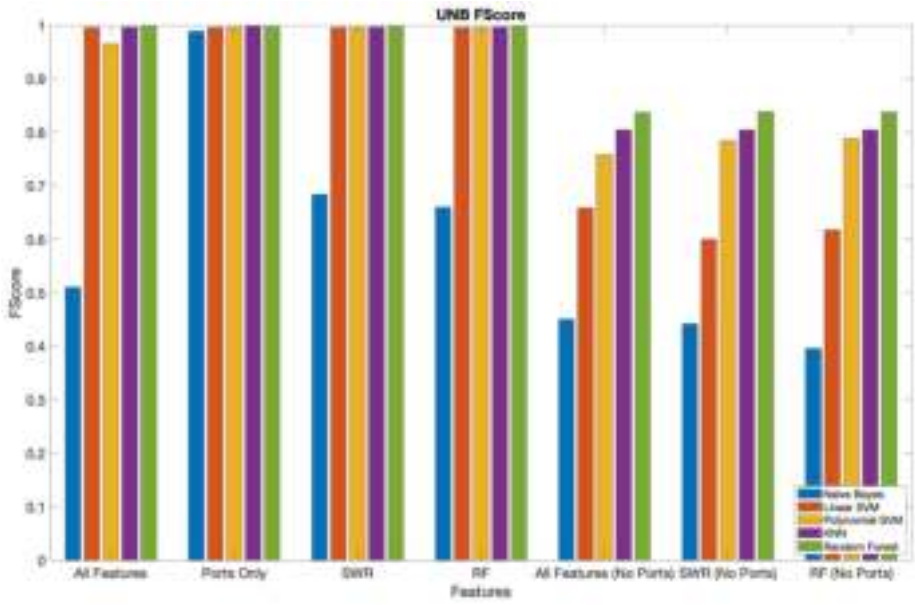


Figure 7.12: UNB F-score

If we were to rate the performance of the different classifiers on the UNB dataset, we would obviously rank them, in ascending order, as naïve Bayes, linear SVM, polynomial SVM, KNN, and random forest.

After performing the cross-validation experiment on two different datasets, we can conclude that the best classifier that can point out different traffic classes accurately while avoiding overfitting is the random forest algorithm. We can also conclude that stepwise regression and random forest feature selection tend to have very similar performance in dimensionality reduction while maintaining the classification accuracy and F-scores very close to the all features results. Moreover, we can also say that even if applications disguise port numbers completely to obfuscate the traffic classification process, we can still build classifiers that can classify traffic with very high accuracies and F-scores.

7.2.3. Various packet percentage within a flow. In the reviewed literature, the authors have always overlooked a very important parameter in the field of traffic classification which is the number of packets considered within a flow to extract flow-level features. It logically makes sense that the more the number of packets considered, the better is the classifier at classifying traffic. Nevertheless, this also means that we would have to wait for a longer period of time in order to receive the required packets before the classification process can be started, which in turn impacts the real-time classification requirement entailed by the quick and dynamic modern networks. Therefore, we needed to find the most optimal number of packets that would yield a reasonable performance. However, traffic flows can vary significantly in total number of packets. One might find a flow with only two packets, but we can also find flows with thousands of packets. Therefore, it is unfair to quantify them in terms of number of packets per flow, rather we use the percentage of packets in a flow. We conduct an experiment that plots the classification accuracy and F-score against the percentage of packets in a flow using four classifiers, naïve Bayes, linear SVM, KNN, and random forest. The reason why we do not use polynomial SVM in this experiment is because it requires a huge amount of time to build the classifier while not adding a great value to the accuracies since it generates results that are very close to KNN. After that we try to deduce some important features from the most important plots. To have a look at the remaining plots of this experiment refer to Appendix C.

7.2.3.1. UNIBS results. Figure 7.13 shows the classification accuracy of the four classifiers using the UNIBS dataset and all features including port numbers. As we can see from the plot naïve Bayes performs the worst starting at almost 50% accuracy when the packet percentage is only 10%, then it rises gradually in an abrupt manner. We can

also see that all the other classifiers have a consistent accuracy which is very close to 100% regardless of the packet percentage. This plot matches with the results of the previous experiment since it shows that if port numbers are present there is almost no need to use flow-level features. However, as we mentioned before this might not always be the case since port numbers might be dynamically changed by the different applications and therefore, we should treat them as if port numbers did not exist in the first place.

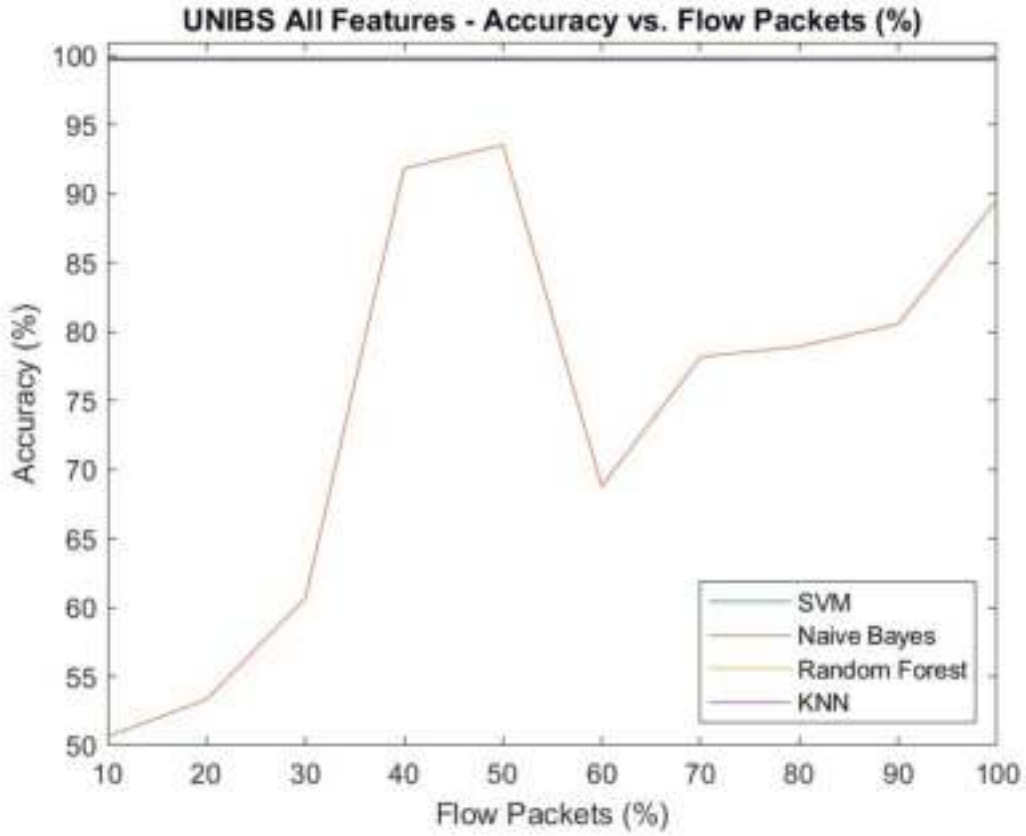


Figure 7.13: UNIBS All Features – Accuracy vs. Flow Packets (%)

Figure 7.14 shows F-score results for all features including port numbers. It can be seen that it has a very similar shape to that of the accuracy which confirms our previous speculations.

In order to investigate the effect of not having port numbers, we display the accuracy results of all features with no ports in Figure 7.15. In general, this figure shows clearly that as we increase the percentage of packets considered for flow-level feature extraction the accuracy of the classification increases. This is exactly what we expected to happen in the first place. We can also observe that naïve Bayes is very unpredictable

with its sharp spikes at times, however, SVM, KNN, and random forest show the same overall trend. This plot also shows that random forest is consistently favorable over both SVM and KNN, which also confirms the findings of the cross-validation experiment.

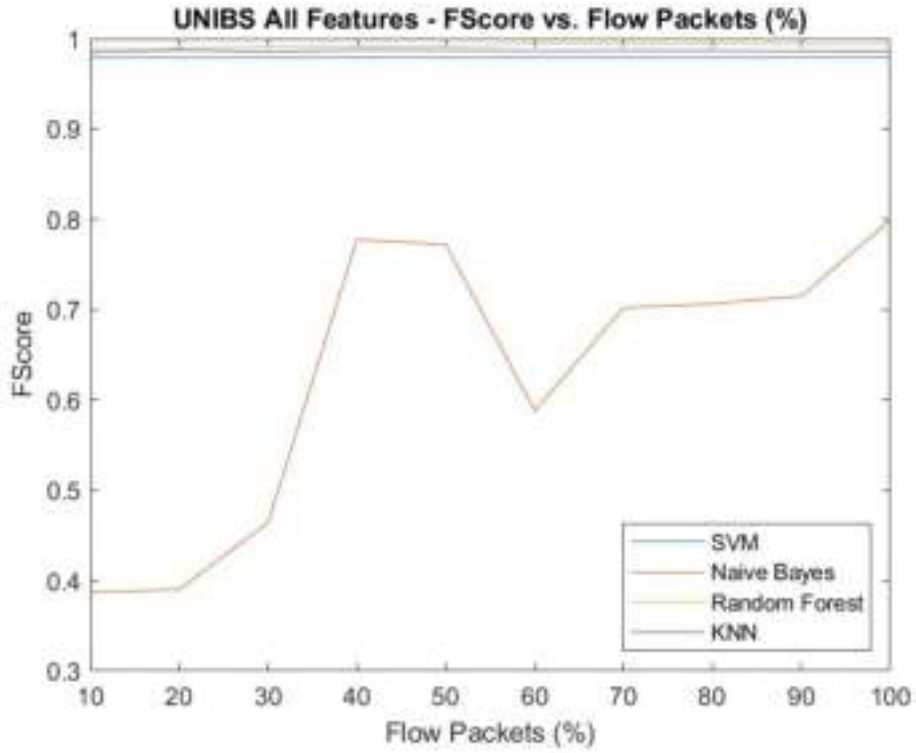


Figure 7.14: UNIBS All Features – F-score vs. Flow Packets (%)

If we look at the F-score of the all features no ports experiment shown in Figure 7.16 we would also observe a very similar outline to that of the accuracy. However, the F-scores tend to show the significant rise in the performance of random forest unlike the accuracy. We can see that random forest starts with an F-score of almost 0.74 at 10% of flow packets, but it eventually reaches around 0.93 at 100% of packets. This plot also shows the wide gap of performance between random forest and its next competitor, KNN, which has its maximum F-score of around 0.76 at 100% of packets, which is very close to the worst performance of random forest of 0.74 at only 10% of packets. This shocking result further shows that random forest is by far the best traffic classifier when applied to the UNIBS dataset. Perhaps the most important finding of this plot is the best packet percentage given the trade-off between performance and waiting time to receive enough packets to perform the classification. We can notice from this plot that the F-scores of the best performers, random forest and KNN, saturate

at almost 60% of packets in a flow with KNN reaching an F-score of about 0.75 and random forest reaching an F-score of approximately 0.91. Therefore, it seems reasonable to pick 60% of the packets within a flow to be able to classify the flow with high accuracy and F-scores.

When we inspect the results obtained using the feature set selected using random forest feature selection shown in Figure 7.17, we can observe the similarity between them and the all features no ports results. This is depicted in the accuracy obtained by random forest which reaches a maximum value of almost 98%. This confirms the conclusion reached in our previous experiment that says that feature selection was successful at reducing the dimensionality while maintaining the performance of the classifiers. Again, a very shy rise in accuracy is observed with both random forest and SVM. As usual, naïve Bayes also performs the poorest.

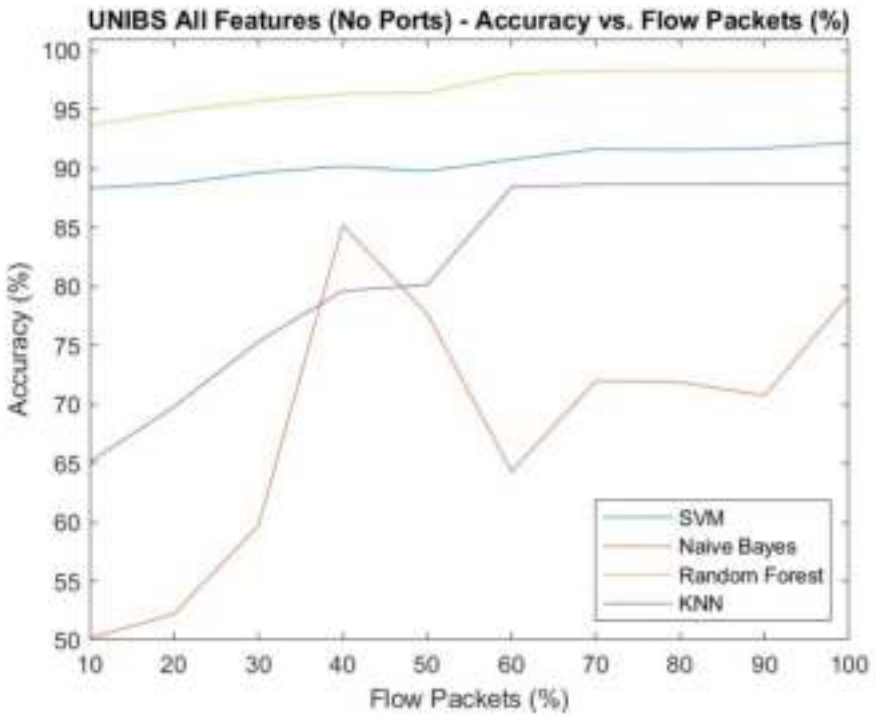


Figure 7.15: UNIBS All Features (No Ports) – Accuracy vs. Flow Packets (%)

Finally, upon inspecting the F-scores of this feature set, in Figure 7.18, we can also conclude that 60% of packets in a flow is a very reasonable number that yields a sound performance level since the F-scores of all classifiers tend to saturate after 60% of packets have been investigated. The value of 60% has been obtained using the all features (no ports), SWR (no ports), and RF (no ports) feature sets. This looks like a

very valid reason to declare 60% of packets as a very logical value for the percentage of considered packets within a flow. After that, adding more number of packets does not yield the anticipated improvement in classifier performance, but rather tends to slow down the the classifier due to the need to wait for more, unhelpful packets.

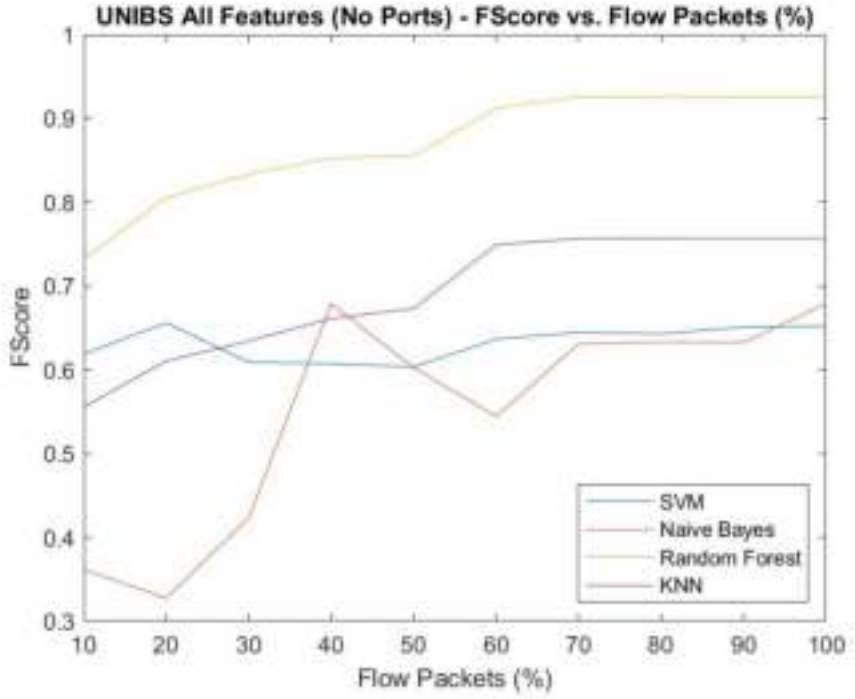


Figure 7.16: UNIBS All Features (No Ports) – F-score vs. Flow Packets (%)

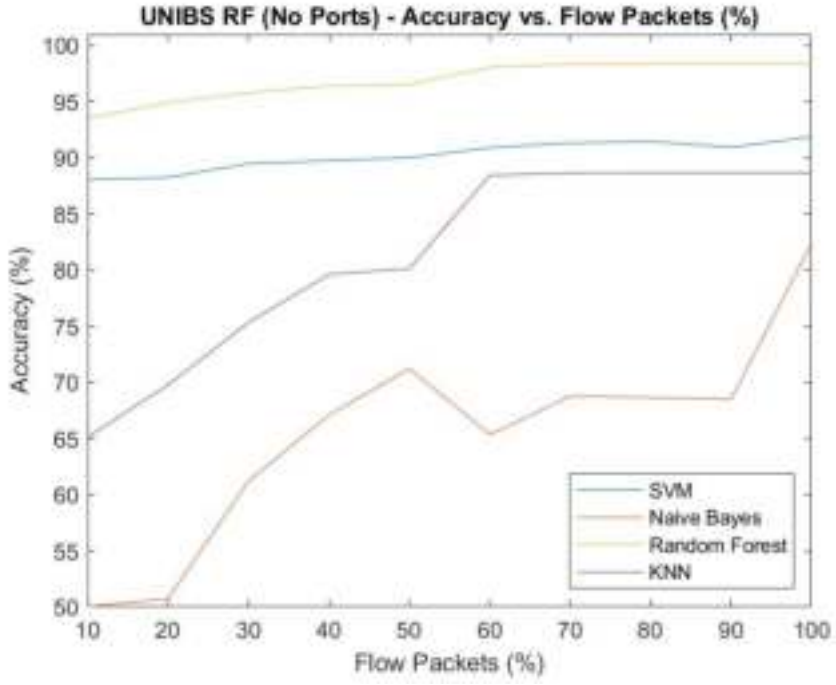


Figure 7.17: UNIBS RF (No Ports) – Accuracy vs. Flow Packets (%)

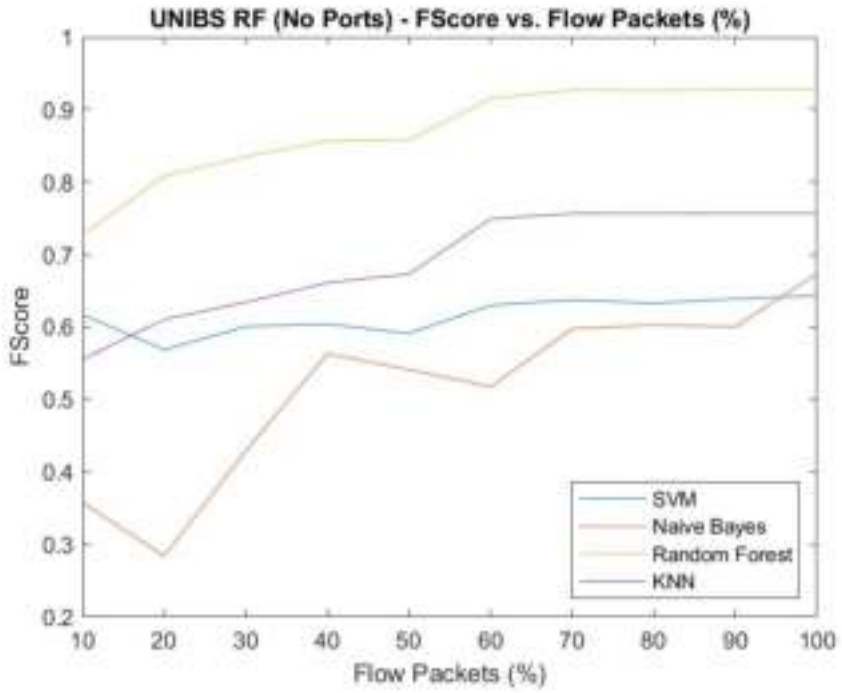


Figure 7.18: UNIBS RF (No Ports) – F-score vs. Flow Packets (%)

7.2.3.2. UNB results. Figure 7.19 shows the accuracies of the four classifiers when applied to the UNB dataset and using all features for classification. As expected, naïve Bayes falls behind while the other three classifiers are very close to 100% accuracy due to the presence of port numbers in this feature set.

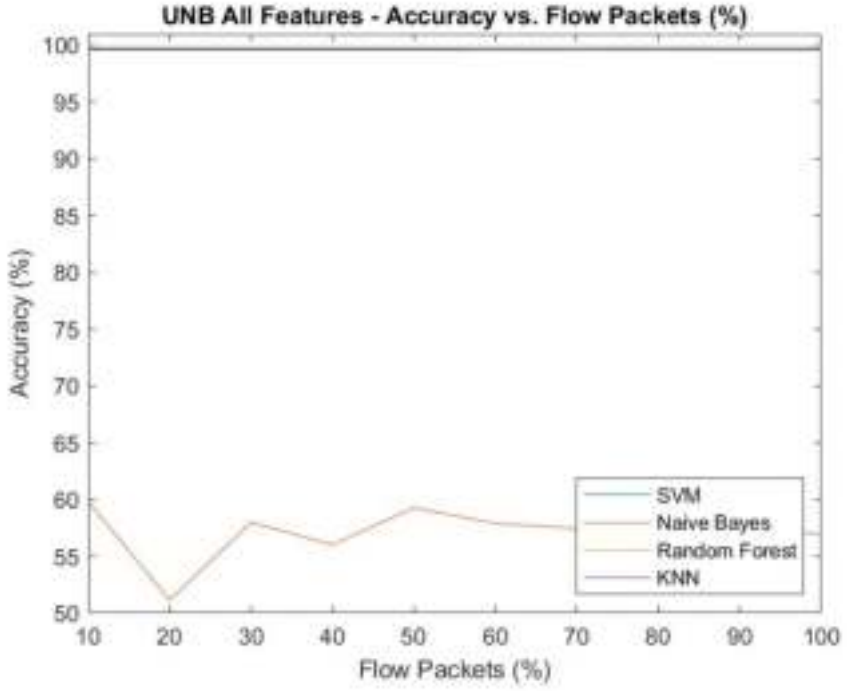


Figure 7.19: UNB All Features – Accuracy vs. Flow Packets (%)

Figure 7.20 shows the F-scores of this feature set which has a similar outline to that the accuracy.

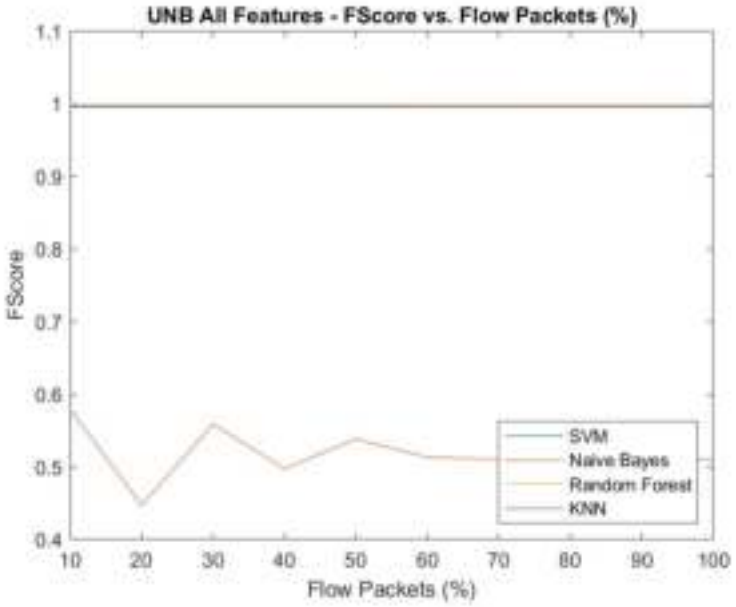


Figure 7.20: UNB All Features – F-score vs. Flow Packets (%)

Upon removing the port numbers from the previous feature set we obtained an accuracy plot that looks as shown in Figure 7.21. We observe the significant drop in accuracy from the previous results which solidifies the idea of port numbers being the most dominant features as we suggested earlier. We also have random forest outperforming all other classifiers, and KNN being the closest in terms of accuracy.

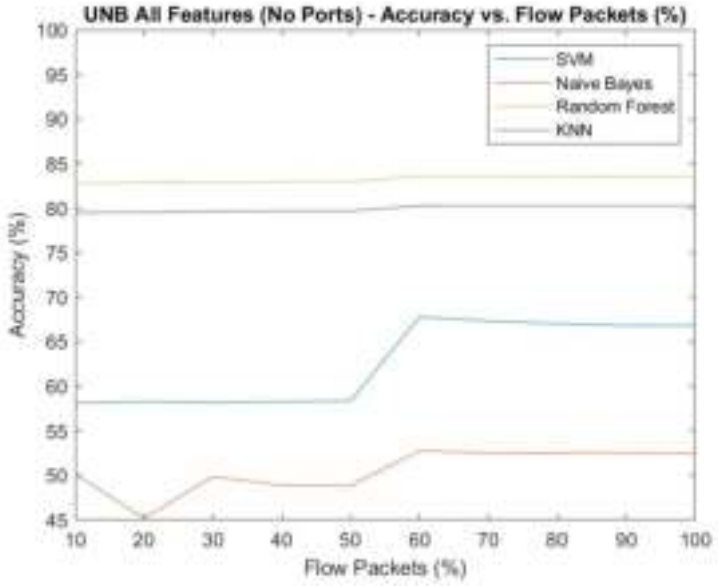


Figure 7.21: UNB All Features (No Ports) – Accuracy vs. Flow Packets (%)

By inspecting the F-score shown in Figure 7.22 we notice that classifiers like SVM and naïve Bayes saturate at 60% of packet percentage just like the case with the UNIBS dataset. However, the plots for random forest and KNN look almost invariant to the change in packet percentage which suggests that packet percentage does not impact the performance of such classifiers on the UNB dataset.

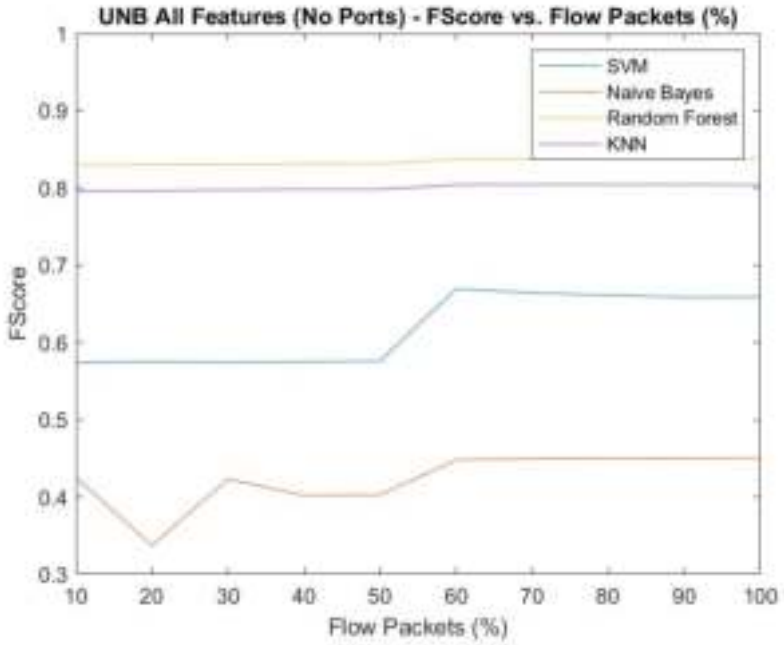


Figure 7.22: UNB All Features (No Ports) – F-score vs. Flow Packets (%)

To investigate the effect of different feature sets on the classification accuracy using different packet percentages, we plot the accuracy of the SWR (no ports) feature set in Figure 7.23. It shows a very similar behaviour to that of all features no ports with very close accuracies, which again shows the ability of SWR to reduce the problem dimensionality while maintaining a very high classification accuracy.

The F-scores of such a feature set, shown in Figure 7.24, show saturation at 60% of packets for both SVM and naïve Bayes, while showing almost no change for random forest and KNN classifiers. This bizarre behaviour might be explained by the fact that the UNB dataset, unlike UNIBS, was collected in a very controlled environment that does not resemble a real network behaviour which should indeed show a relationship between the packet percentage and the performance of a classifier. That is why the UNIBS dataset shows a clear relationship between the two parameters as it resembles a more realistic network behaviour.

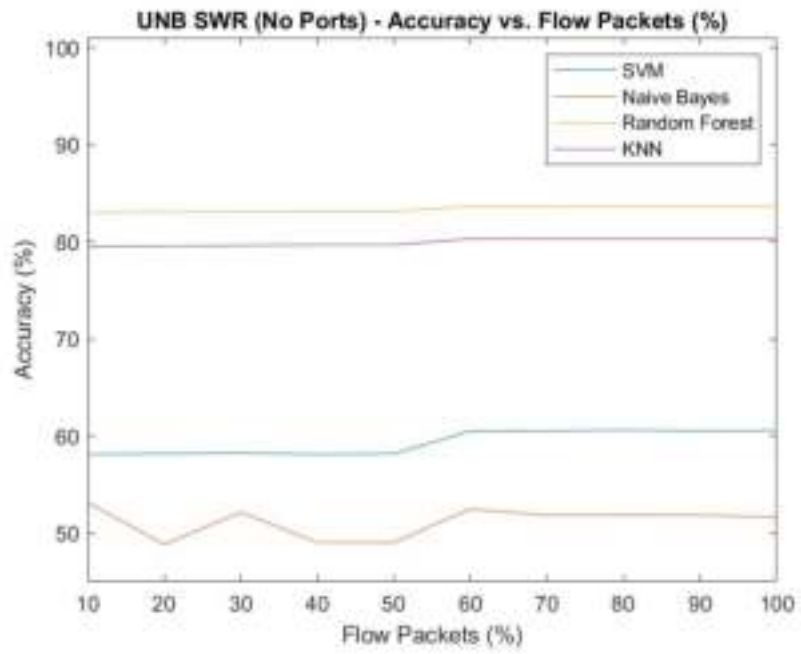


Figure 7.23: UNB SWR (No Ports) – Accuracy vs. Flow Packets (%)

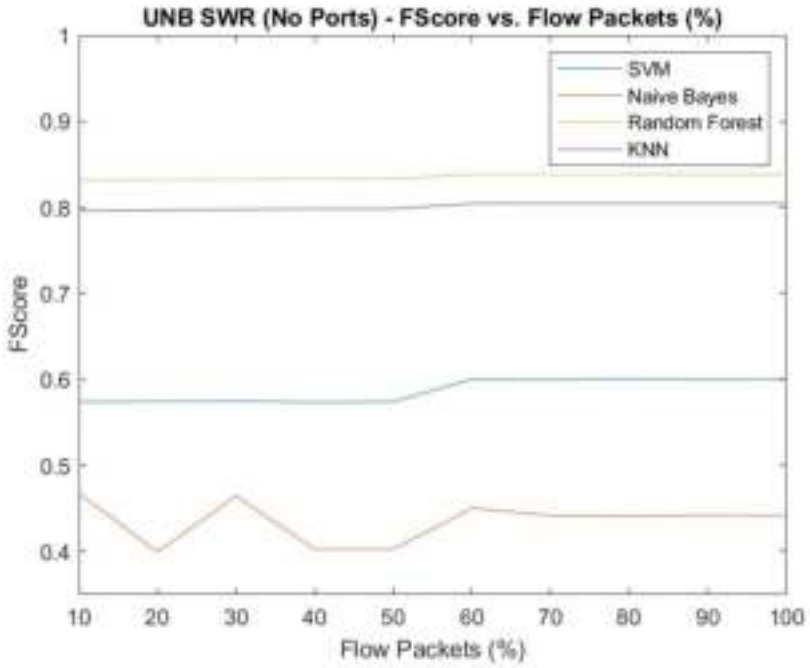


Figure 7.24: UNB SWR (No Ports) – F-score vs. Flow Packets (%)

In order to investigate the effect of packet percentage on the waiting time required to obtain the necessary packet percentage before classifying a network flow, we plot the average flow duration against the percentage of considered packets within a flow. This plot is shown in Figure 7.25. As discussed earlier, 60% of the UNIBS

packets within a flow looks like an adequate number to classify traffic flows with a high accuracy and F-score. Therefore, we find that the average required time to wait for 60% of the flow packets is found out to be around 21ms. This means that with this setup we guarantee that the classification will be done within approximately 21ms after receiving the first packet of the flow. On the other hand, the strange, compact graph of the UNB dataset looks more like two flat straight lines with a very minor jump in between. The behaviour of the UNB dataset is further questioned by this plot since a normal network would not usually have such a straight line as increasing the number of packets within a flow would usually yield a much longer time. Therefore, this endorses our previous arguments that with such a controlled and fabricated dataset we cannot usually simulate the behaviour of a real network.

In conclusion, UNIBS tends to behave more like a normal network due to the uncontrolled environment that was created while capturing the traces, unlike the artificially created UNB dataset. Nevertheless, UNB remains a good dataset to study traffic classification.

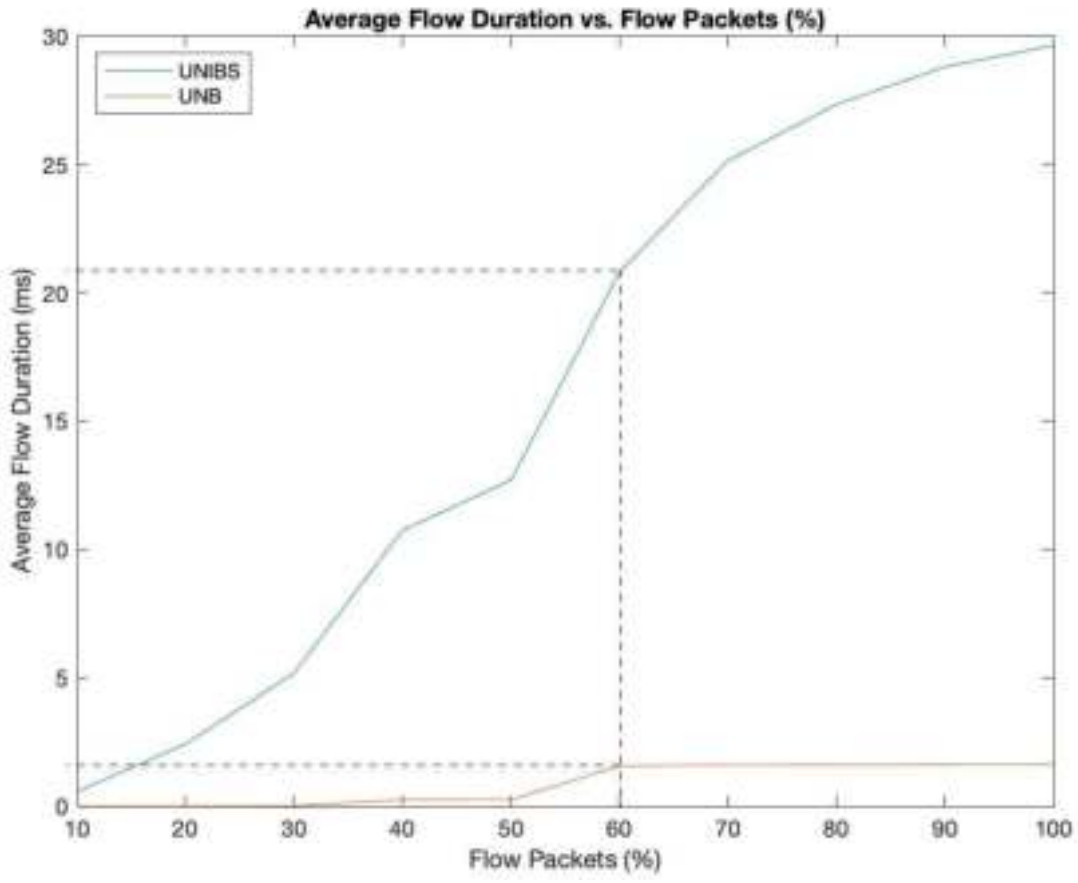


Figure 7.25: Average Flow Duration vs. Flow Packets (%)

7.2.4. Various training set sizes. In this experiment, we are concerned with the effect of varying training set size on the performance of the different classifiers. As a result, we vary the training set size from 10 to 90% while fixing packet percentage at 100% to eliminate any effect due to packet percentage on the performance of the classifiers. In order to facilitate the proceedings of such an experiment, we use the holdout method that assigns n% of the instances to the training set and (100-n) % to the test set. After that, we plot the accuracies and the F-scores of the different feature sets against the training set size. As before, we show the most important plots, however, to have a look at the remaining plots of this experiment refer to Appendix D.

7.2.4.1. UNIBS results. Figure 7.26 shows the accuracies of different training set sizes for all features including the port numbers. By closely inspecting the plots we observe that SVM, KNN, and random forest are quite resilient to any change in training set sizes since the increase in accuracy is very small when comparing the accuracies at 90% training and 10% training since the highest jump is that of KNN with about 2.5% improvement only. Moreover, naïve Bayes proves one more time that it is the worst classifier out of all four classifiers.

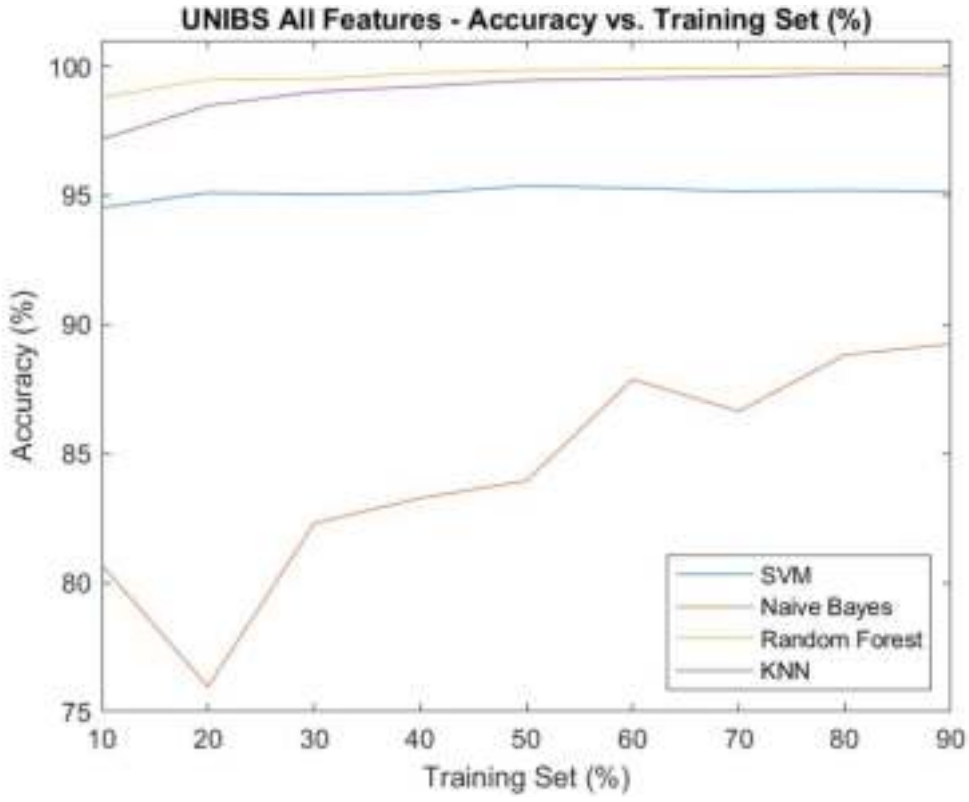


Figure 7.26: UNIBS All Features – Accuracy vs. Training Set (%)

On the other hand, the F-scores shown in Figure 7.27 display a slightly better depiction of the performance improvement as the increase in F-score goes from 0.89 to around 0.98 for random forest, for example. In addition, SVM seems to be completely resilient to the change in training set sizes. Moreover, both random forest and KNN tend to also saturate at 60% training set and 40% test set.

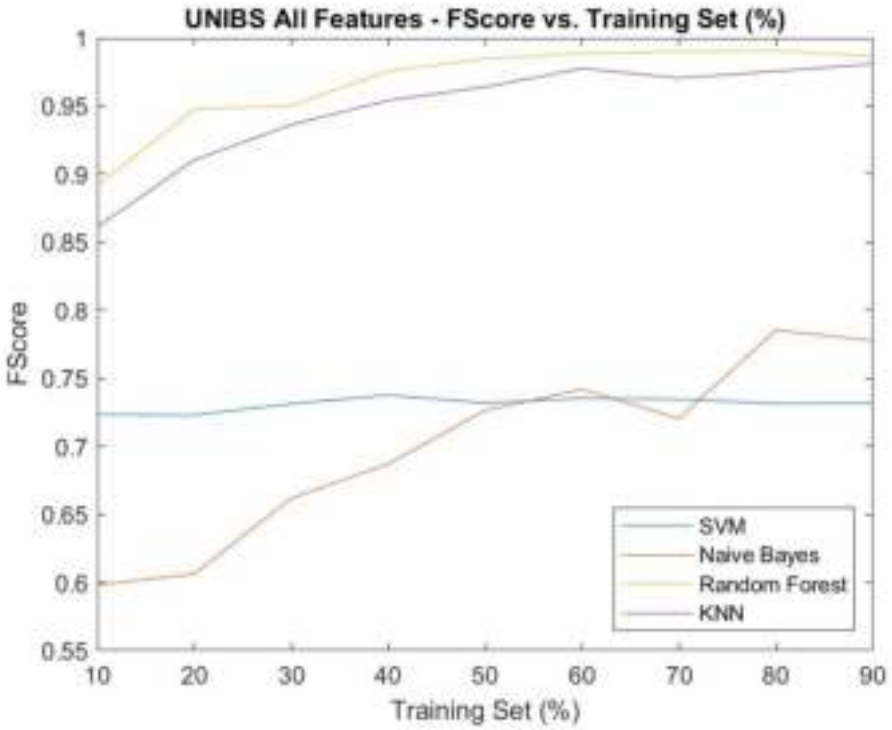


Figure 7.27: UNIBS All Features – F-score vs. Training Set (%)

To inspect the effect of disguised port numbers, we now look at all features after removing the port numbers shown in Figure 7.28. It is apparent that random forest is not highly affected by the removal of port numbers which indicates that random forest will be a very good model to classify the UNIBS dataset even in the absence of port numbers. However, SVM did drop significantly from before removing the port numbers (97.5% to almost 92% accuracy).

Again, F-score is usually a better performance measure as it depicts the huge difference between random forest and any other classifier. This is shown in Figure 7.29 that displays the F-scores of the four classifiers while varying the training set sizes. It is apparent that random forest constantly outperforms its next competitor, KNN, by almost 0.15. SVM shows its resiliency to changing the training set size one more time.

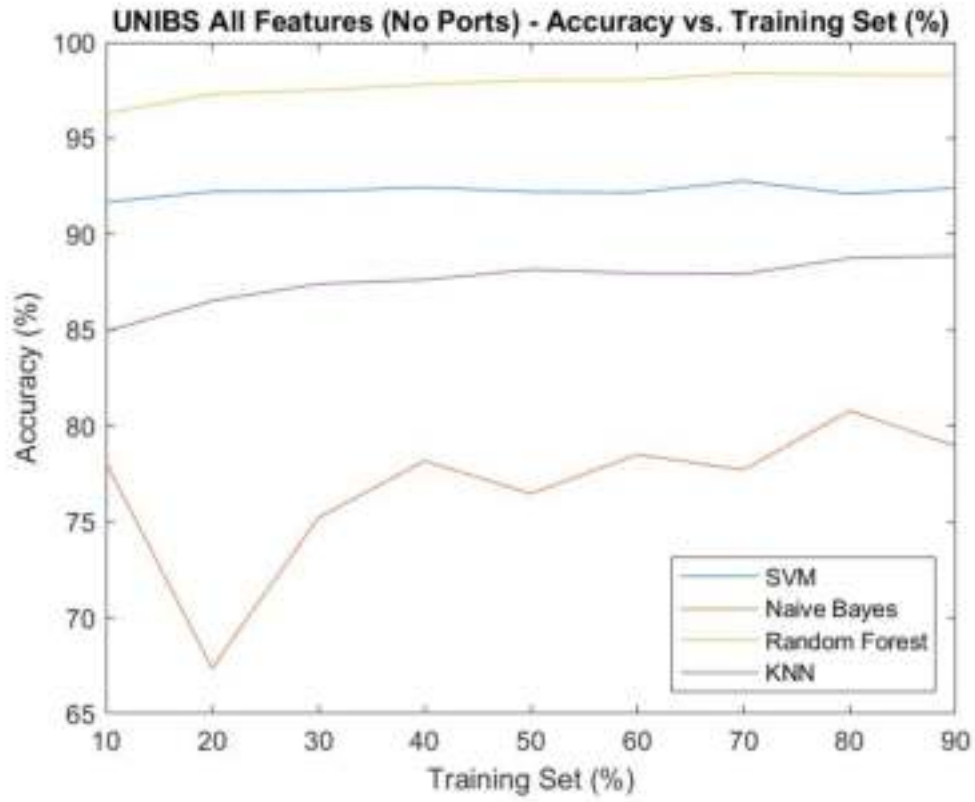


Figure 7.28: UNIBS All Features (No Ports) – Accuracy vs. Training Set (%)

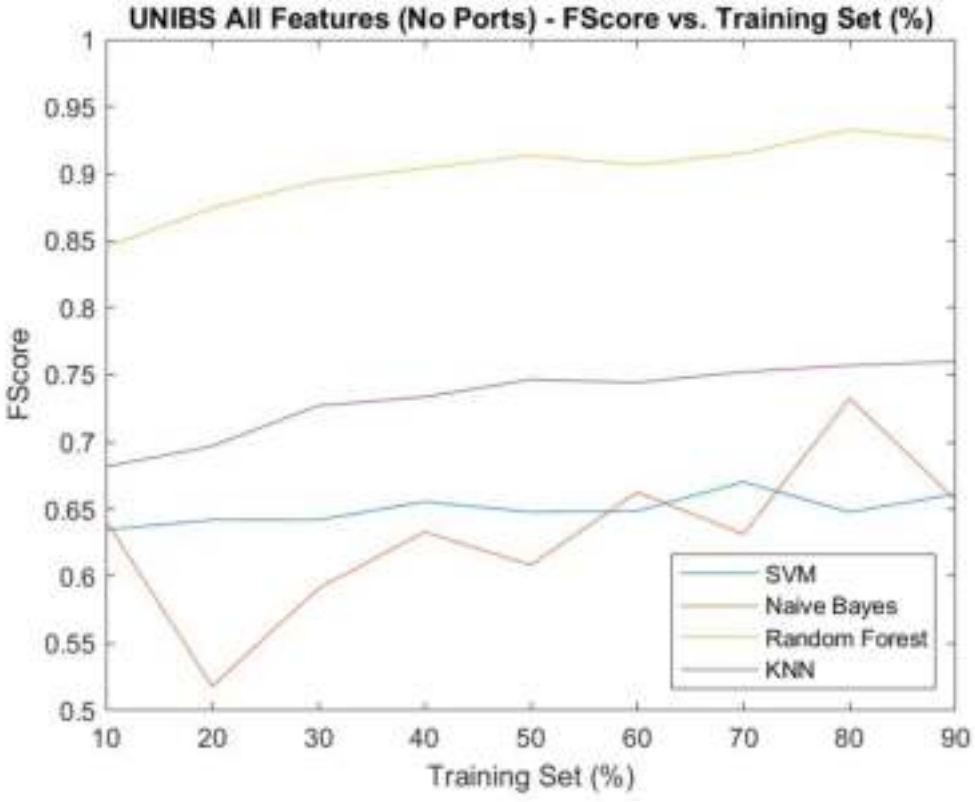


Figure 7.29: UNIBS All Features (No Ports) – F-score vs. Training Set (%)

The performance of the classifiers using the RF (no ports) feature set looks quite similar to that of all features (no ports) feature sets which proves the excellence of random forest in choosing the most relevant attributes for classification. This is shown by the accuracy plot and the F-score plot of Figure 7.30 and Figure 7.31, respectively.

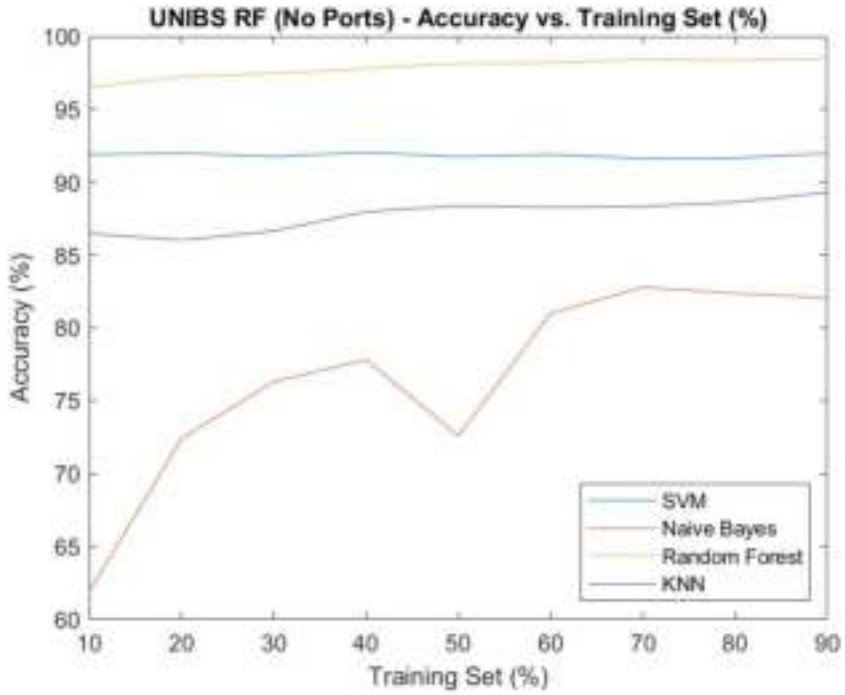


Figure 7.30: UNIBS RF (No Ports) – Accuracy vs. Training Set (%)

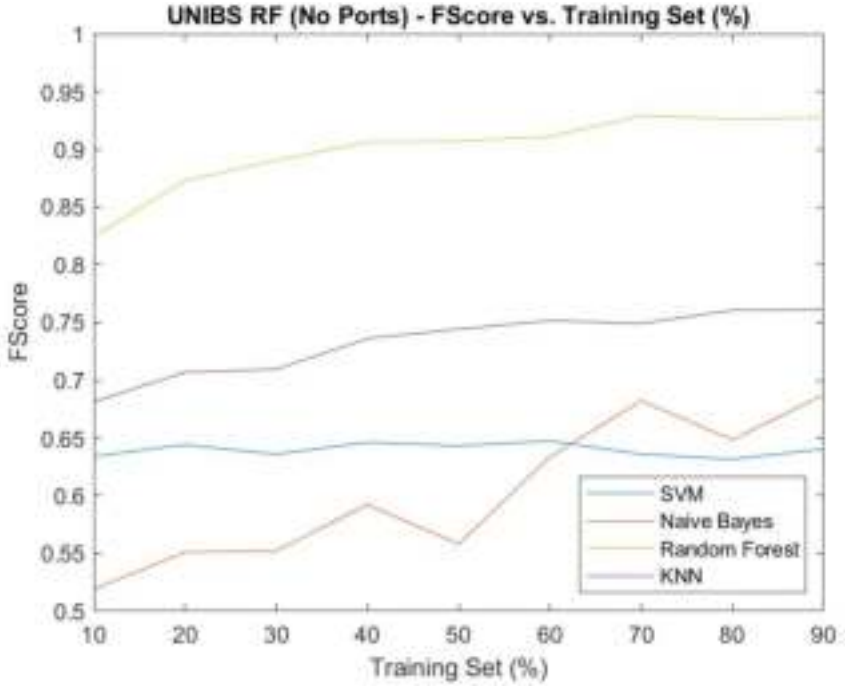


Figure 7.31: UNIBS RF (No Ports) – F-score vs. Training Set (%)

7.2.4.2. UNB results. Figure 7.32 shows the accuracies of the four classifiers using different training set sizes with all features. It is apparent that all classifiers, ignoring the sharp drop in naïve Bayes, are almost completely insensitive to variations in training set size. This indicates that the nature of the UNB dataset allows classifiers to learn the behaviour of the five classes with very minimal training data. This could be an advantage since it might indicate that classifiers do not need a huge amount of data to learn how to classify such classes. However, this was not the case with the UNIBS dataset. Therefore, we can also explain this by the artificial nature of the UNB dataset since it was not collected from a real-life network but rather was captured in a very controlled environment which does not mimic the actual operation of a congested network.

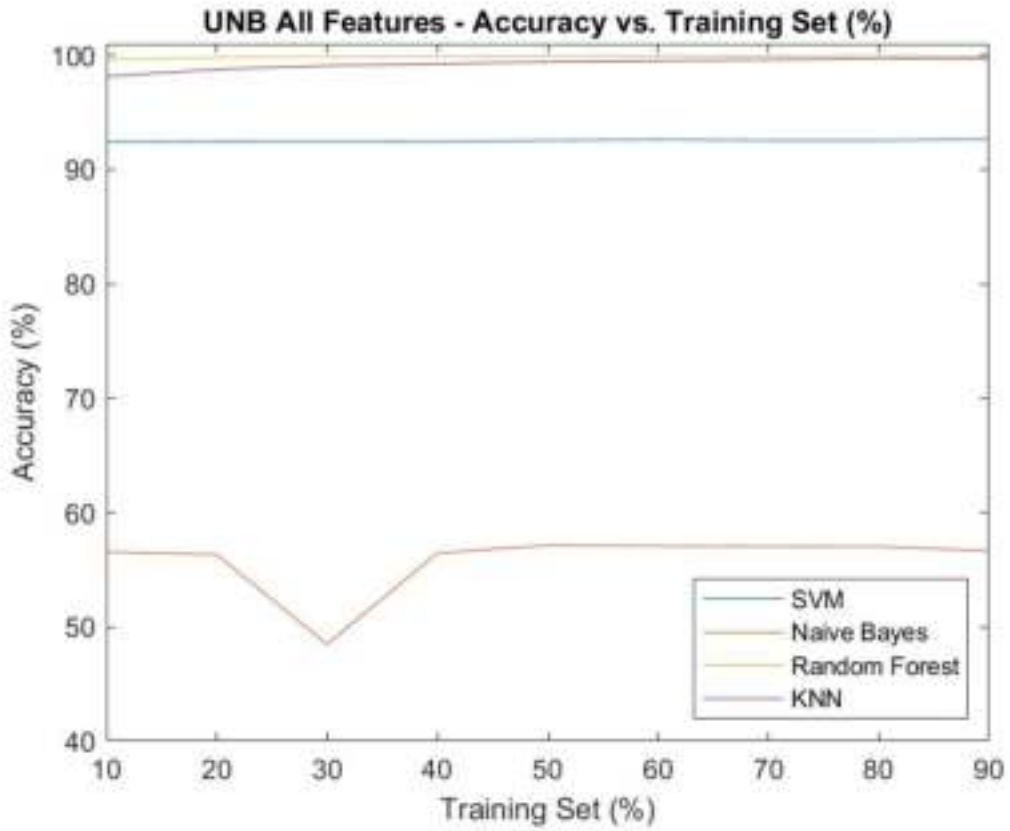


Figure 7.32: UNB All Features – Accuracy vs. Training Set (%)

The concept of insensitivity to variations in training size is further illustrated by the F-score plots which also look horizontal as shown in Figure 7.33.

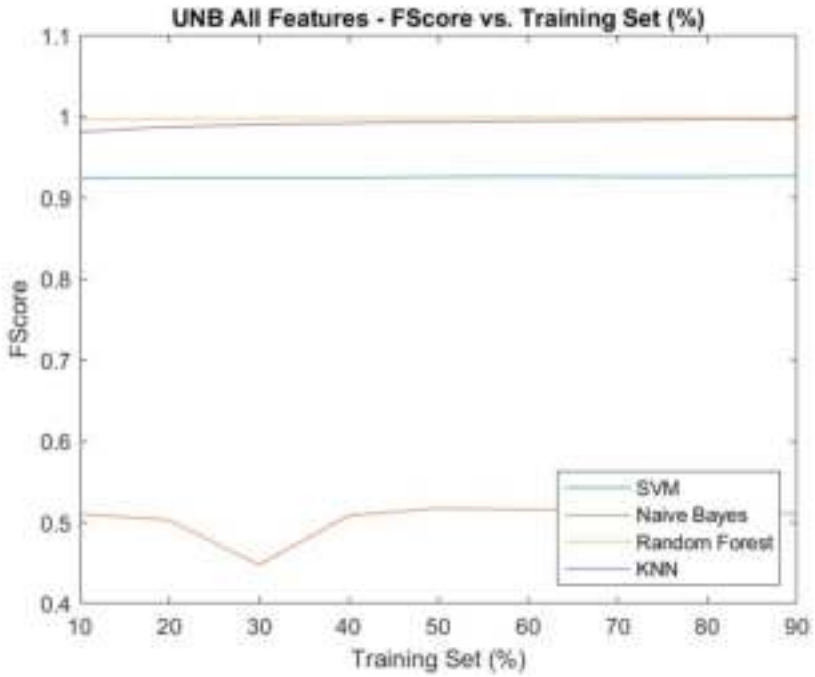


Figure 7.33: UNB All Features – F-score vs. Training Set (%)

In order to investigate whether this behaviour showed up just because of the inclusion of port numbers, we plot the accuracies of all features (no ports) feature set in Figure 7.34. The plot shows no difference compared to the earlier situation, except for the expected drop in accuracies, which further proves our point about the UNB dataset being artificially captured.

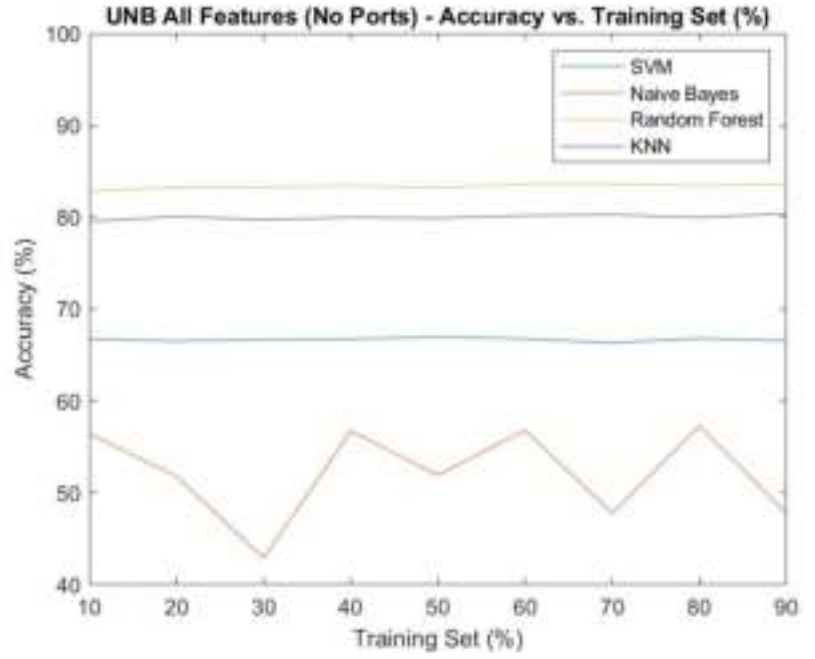


Figure 7.34: UNB All Features (No Ports) – Accuracy vs. Training Set (%)

Finally, the F-score of the all features (no ports) feature set is shown in Figure 7.35, which illustrates the same artificial behaviour.

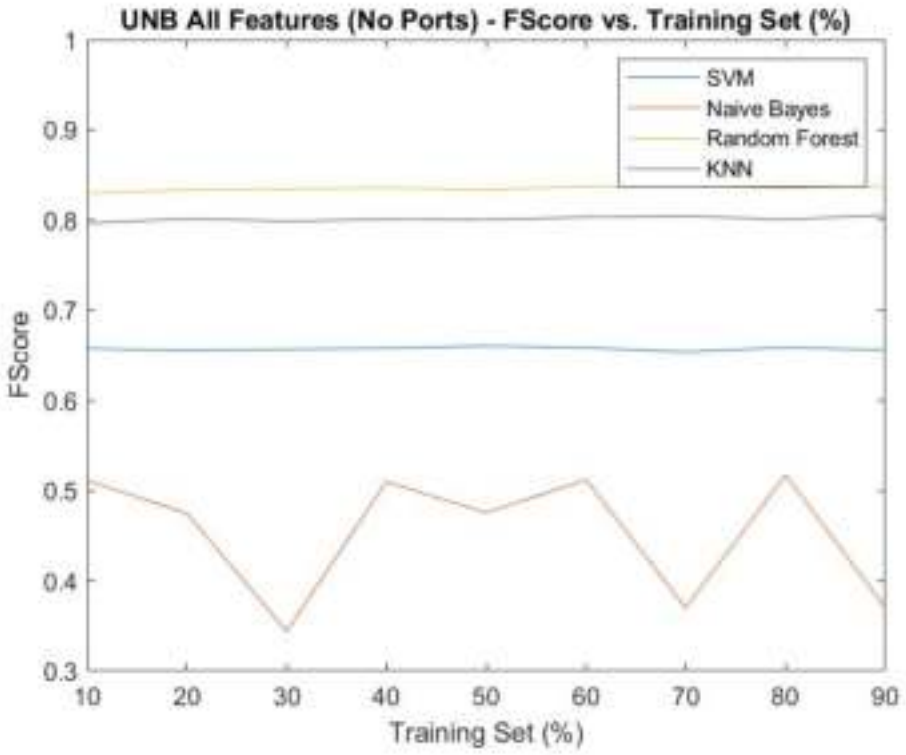


Figure 7.35: UNB All Features (No Ports) – F-score vs. Training Set (%)

7.3. FPGA Implementation and Results

Hardware designers usually need to go through the hardware design process which consists of several steps that need to be followed while designing a new digital circuit on an FPGA. Figure 7.36 shows the hardware design process. The first step in the design process is the design entry step through which the designer needs to express their design in a format that can be understood by the machine. There are multiple design entry tools including Hardware Description Languages (HDLs), drag-and-drop tools and many more. HDLs are perhaps the most sophisticated way to express a hardware design as it allows the designer to describe their circuit at a more granular and low level hence enabling them to have more control over their design. Therefore, in this work we will be using the Verilog HDL as our design entry tool. Verilog is a hardware description language used to design digital circuits, verify their functionality through simulation, perform timing analysis to ensure all timing constraints are not violated, and finally synthesize hardware logic.

Figure 7.36: Hardware Design Process

Upon describing the hardware design in Verilog, the Quartus Prime software from Intel is used for the next step in the design process, compiling and synthesis [37]. In this step, the Quartus compiler will verify the validity of the Verilog syntax and try to synthesize the Verilog code into actual hardware components. In other words, the synthesis tool in Quartus translates the code into a low-level specification that can be realized on the FPGA chip. The third step is to simulate the design. Simulation is divided into functional simulation and timing simulation. Functional simulation is usually performed to verify that the designed circuit is functionally correct and that it performs according to its specifications. On the other hand, timing simulation takes into account the signal propagation time between the logic gates and hence ensures that all timing constraints are met. Once the circuit is simulated successfully, the hardware designer can now configure the FPGA chip through downloading the bitstream file that contains the synthesized logic onto the chip. Finally, the hardware designer can now test their design on real hardware and verify whether the design performs as expected.

7.3.1. Random forest training. Upon completion of the design entry step mentioned earlier, whereby we described the random forest algorithm in Chapter 6 using the Verilog HDL, and before going into the compiling/synthesis step, we load the node information onto the FPGA's on-chip memory. In order to do so, Quartus enables the hardware designer to initialize the on-chip memory of an FPGA using a file format known as Memory Initialization File (.mif). Since training is done offline in all cases, we used the sklearn library in Python to train the random forest classifier offline using both a majority-based model and a probability-based model. Unfortunately, the sklearn library does not produce complete trees, therefore, some levels in the trees might be missing nodes since leaf nodes appear at an earlier level of the tree. This is definitely problematic with the hardware design, since as we mentioned earlier, we need all trees to have the same number of levels and the levels need to be complete in order to ease the hardware design. Therefore, a Python script was developed to train a random forest classifier while making sure that the produced tree is a complete tree. This was done

through extending the leaf nodes using dummy nodes that will simply replicate their parents such that the two children of a previous leaf node will have the same output as their parent. By doing so, we ensure that the generated trees are complete. After that, we export the generated model into several .mif files where each tree level has its own .mif file for its memory initialization. Eventually, the exported .mif files were then loaded into Quartus before compiling and synthesizing the Verilog code. By this, our design is now ready to be simulated and tested. Figure 7.37 shows the first three levels of the first tree in the generated random forest for the UNIBS dataset as an example.

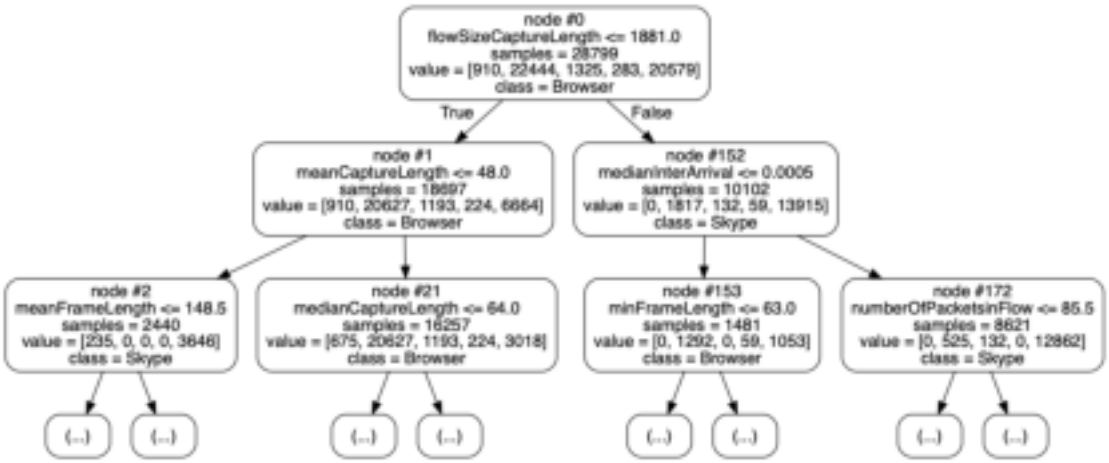


Figure 7.37: First Three Levels of the First Tree in the Random Forest

Figure 7.38 depicts the contents of the .mif file for level 2 of the previous tree. Note that the numbers before the ":" are the effective addresses of the four nodes within level 2 of the tree, whereas the numbers after the ":" are the tree level memory contents that follow the structure discussed earlier in Figure 6.5.

```
WIDTH=64;
DEPTH=4;
ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;
CONTENT BEGIN
        0       :       1800000948000000;
        1       :       3800000400000000;
        2       :       10000003F0000000;
        3       :       6000000558000000;
END;
```

Figure 7.38: Memory Initialization File for Level 2 of Tree 1

When training the random forest classifier, we must take into account the memory requirements of the generated model. Consider the case where we translate the best random forest model generated in Section 7.2 using the UNIBS dataset into its hardware counterpart. The best random forest model consists of 50 trees, where the maximum number of levels per tree is 32 levels and each node requires 64 bits to be encoded. Keeping in mind that all trees must be extended to 32 levels since one of the design constraints necessitates the need for trees with the same number of levels, in addition to the need to ensure that all trees are complete, we are faced by a difficult problem. Upon extending all trees to 32 levels, the number of nodes per tree is 2^{32} - 1 = 4294967296 nodes per tree. Therefore, the required memory to encode one full tree is 64 * 4294967296 = 32 GB. Finally, the total required memory is 32 * 50 = 1600 GB.

This means that if we were to implement the exact tree generated by software on an FPGA, we would require 1600 GB of on-chip memory to implement the entire tree. This is definitely impractical and impossible to implement on any FPGA chip. This brings about the need to compromise some of the capabilities of the tree generated in software in order to make it suitable for hardware implementation. As a result, we decided to reduce the number of trees in the random forest and the number of levels per tree in order to be able to fit the entire random forest on the Cyclone IV E chip used. This will probably reduce the accuracy of the random forest model in hardware, but it will certainly help us speed up the classification performance on hardware to a great extent. This is a compromise we are willing to make since, as will be explained later, the reduction in performance due to random forest pruning is very minimal when compared to the enormous gain in classification speed using the FPGA implementation.

7.3.2. Variable trees and levels. As mentioned in the previous subsection, fitting the entire random forest on the FPGA-chip will be very difficult due to memory limitations, in addition to the limited number of CLBs available on the chip. As a result, we wanted to perform an experiment whereby we vary the number of trees in the forest from 1 tree all the way to 50 trees (the optimal number of trees according to our experiments). Meanwhile, we observe the maximum number of levels per tree that our FPGA chip can sustain. After that, we check the classification accuracy and the F-score of the generated model in order to find out the combination of trees and levels that yield the best classification performance when implemented in hardware. Of course, we carry

out this experiment on both datasets as we might get a different optimal combination for each dataset.

Figure 7.39 and Figure 7.40 show the classification accuracy and the F-score for the UNIBS dataset, respectively. Upon inspecting the two graphs we can certainly confirm our previous assumption that when the number of levels in a tree reduces the classification performance drops. Therefore, the two graphs show that if we want to fit a greater number of trees on the FPGA chip, we would have to sacrifice the number of levels in each tree. Moreover, we would also sacrifice the classification performance to a great extent. Therefore, one of our objectives is to find the optimal combination of the number of trees and the number of levels that yields the best classification performance. The two graphs show that the accuracy and F-score peak at 2 trees each tree having 14 tree levels.

Similarly, we repeated the previous experiment using the UNB dataset to find out the optimal combination of the number of trees and levels that yields the best performance on the UNB dataset. Figure 7.41 and Figure 7.42 show the classification accuracy and the F-score for the UNB dataset, respectively. We can see that the UNB graphs show a very similar behaviour to that of the UNIBS graphs, where the performance degrades as we increase the number of trees due to the decrease in the number of levels per tree. We can also conclude from the graphs that, using the UNB dataset, 9 trees each with 12 tree levels lead to the best classification performance in terms of both accuracy and F-score.

7.3.3. The FPGA model. When deciding on the combination of number of trees and number of levels in our final FPGA model, we consider a number of factors, mainly, we want to improve the utilization of the FPGA resources by fitting as many trees and as many levels as the FPGA can handle. Additionally, and most importantly, we want to select a combination of trees and levels that would allow context switching between the UNIBS and UNB models in a very short time. Therefore, we wanted a common combination that would yield a close to optimal performance on both datasets while enabling us to quickly switch the models without modifying the hardware design, but only loading the node information of the two models onto the on-chip memory. By looking at the four graphs shown in Figures 7.39, 7.40, 7.41, 7.42 we predict that a combination of 20 trees and 10 levels would be a good option to satisfy both criteria discussed above. Such a combination utilizes the maximum possible number of CLBs

and on-chip memory blocks, while allowing us to simply load the tree level memory contents without modifying the hardware design.

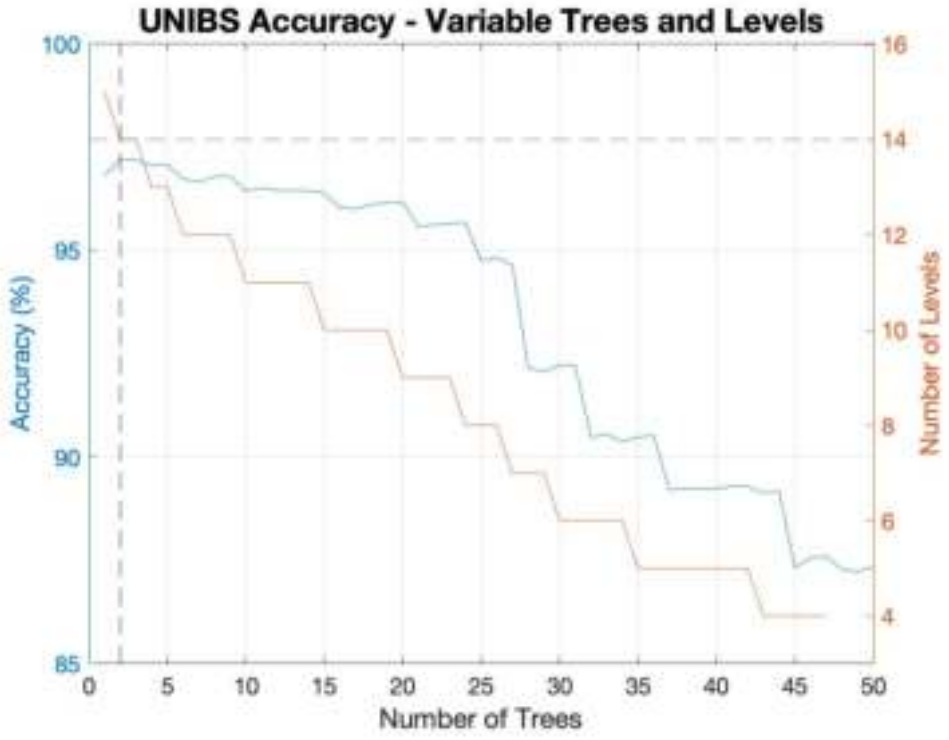


Figure 7.39: UNIBS Accuracy - Variable Trees and Levels

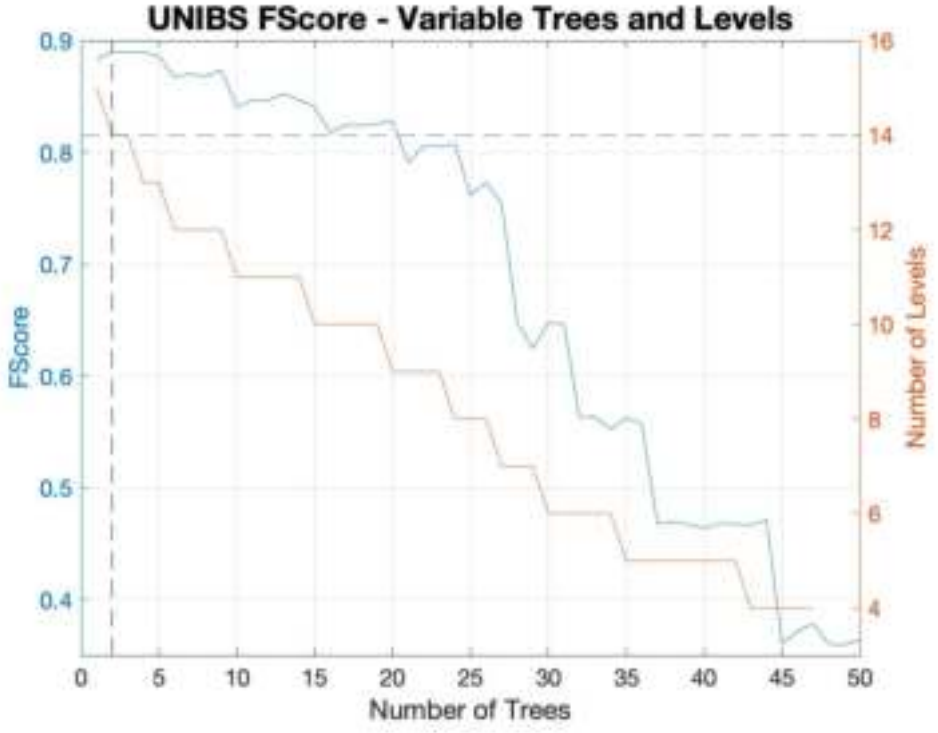


Figure 7.40: UNIBS F-score - Variable Trees and Levels

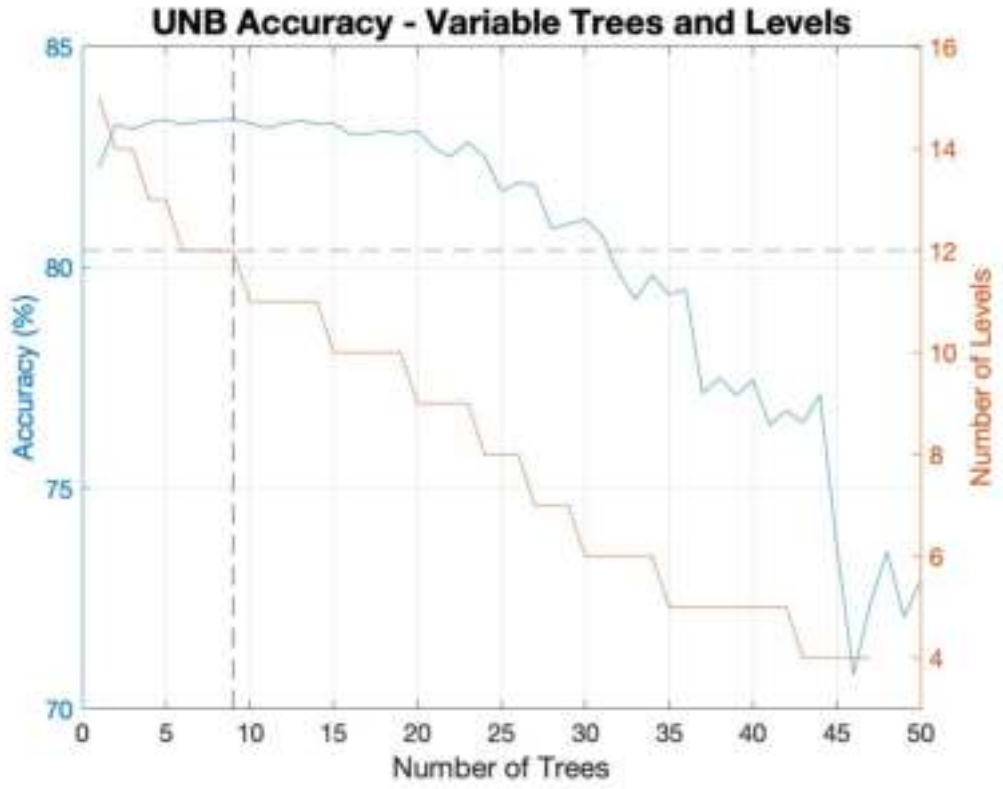


Figure 7.41: UNB Accuracy - Variable Trees and Levels

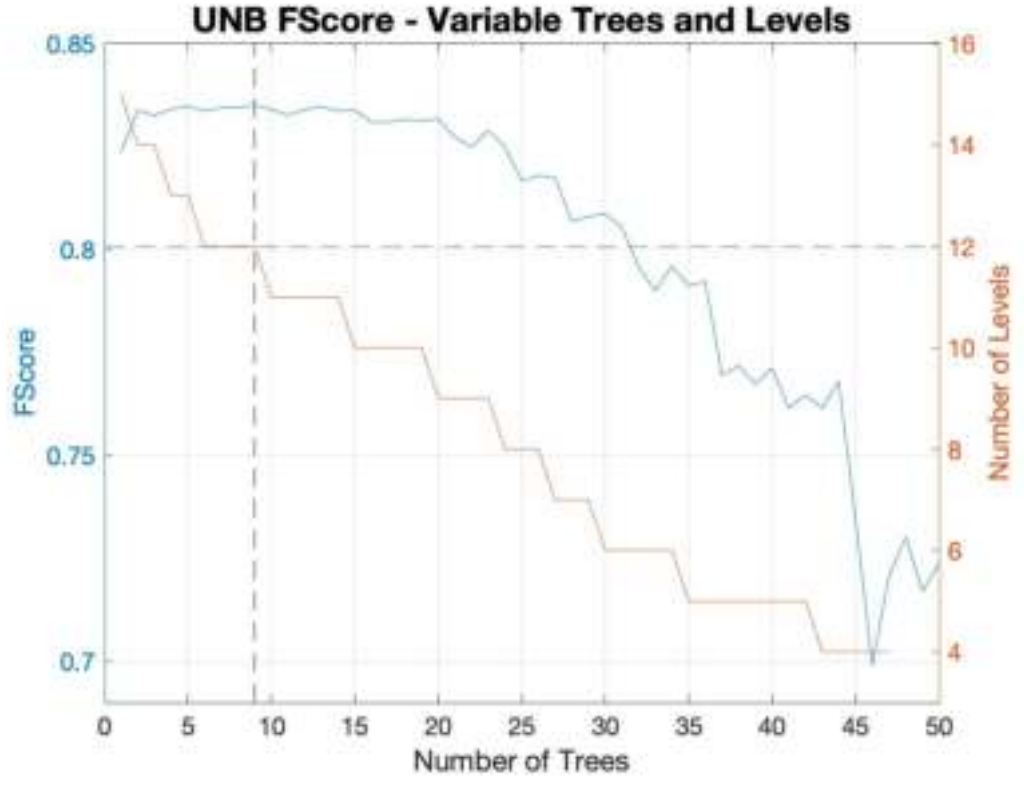


Figure 7.42: UNB F-score - Variable Trees and Levels

Calculating the memory consumption when using a (20-trees, 10-levels) model, would result in 2^{10} - 1 = 1023 nodes per tree. This means that the required memory to encode one full tree is 64 * 1023 = 8 kB. Lastly, the total required memory to encode the entire Random Forest is 8 * 20 = 160 kB. This is definitely a great saving compared to the original required memory of 1600 GB with a (50-trees, 32-levels) model.

As a conclusion of the above discussion, we decided to implement a (20-trees, 10-levels) model on the FPGA chip, we compare the performance of our FPGA model to that of the most optimal model obtained using our software simulations. To show the differences between the majority-based and the probability-based models we also record the performances of the two different models both on hardware and in software. The hardware-based models were restricted to 20 trees and 10 levels as mentioned earlier, whereas the software-based models were based on 50 trees while we had no limit on the number of levels per tree. This means that, in software, models were allowed to fully grow. This resulted in models with the parameters given in Table 7.11.

Table 7.11: Model Parameters

Model	Number of Trees	Number of Levels
UNIBS Hardware	20	10
UNB Hardware	20	10
UNIBS Software (Majority)	50	32
UNIBS Software (Probability)	50	32
UNB Software (Majority)	50	27
UNB Software (Probability)	50	30

Table 7.12 shows the comparison in terms of classification accuracy and F-score between the hardware-based models and the software-based models for the UNIBS dataset. We can draw several conclusions from this table. First of all, probability-based models usually perform better than majority-based ones. This is true when we compare the accuracies and F-scores of the software majority-based model to that of the software probability-based one. Similarly, the accuracies and F-scores of the hardware implementation show that probability-based random forests are usually superior to majority-based forests. Another observation is the fact that, even though going from software to hardware we reduced the number of trees from 50 to 20 and the number of tree levels from 32 to 10, the accuracies dropped by only 2%, while the F-score dropped by only 0.1. Indeed, this means more misclassifications on hardware, however, we do gain considerably in classification speed as will be discussed later.

Table 7.12: FPGA Model vs. Software Optimal Model for the UNIBS Dataset

Measure	Hardware		Software	
	Majority	Probability	Majority	Probability
Accuracy (%)	96.3	96.5	98.3	98.5
F-score	0.823	0.834	0.927	0.932

Similarly, Table 7.13 shows the comparison in terms of classification accuracy and F-score between the hardware-based models and the software-based models for the UNB dataset. The results of the UNB dataset confirm the conclusions obtained from the UNIBS dataset. We can see that in all cases the probability-based random forest model outperforms the majority-based models. We can also see that even though going from software to hardware the number of trees reduced from 50 to 20 and the number of levels reduced from 27 (majority-based) and 30 (probability-based) to 10, the change in accuracies and F-scores was even smaller than the change obtained using the UNIBS dataset.

Table 7.13: FPGA Model vs. Software Optimal Model for the UNB Dataset

Measure	Hardware		Software	
	Majority	Probability	Majority	Probability
Accuracy (%)	82.4	83.2	83.6	83.7
F-score	0.824	0.833	0.837	0.838

Furthermore, to gain a better understanding of how well hardware competes against software, we check the accuracy of the (20-trees, 10-levels) hardware model against the (20-trees, 10-levels) software model. The reason behind this is the need to compare the performance of hardware and software under the same circumstances. Moreover, we also recorded the performance results of a 20 trees-fully grown random forest in software to check the effect of pruning the trees to be able to fit them on the FPGA chip. Table 7.14 shows the comparison between the hardware model (20-trees, 10-levels), the pruned software model (20-trees, 10-levels) and the fully-grown software model (20 trees-fully grown) using the UNIBS dataset. The results reveal that the hardware performance is identical to that of the pruned software model. This is because the Python script that exports the trained random forest to .mif files is capable of exporting the model with a great precision such that we can map the exact model generated in software to hardware. On the other hand, if we compare the hardware results to the performance of the fully-grown tree, we can see that the comparison goes hand-in-hand with that shown in Table 7.12. The accuracy drops by approximately 2%

while the F-score drops by almost 0.1. Again, this will be justified further when we discuss the gain in classification speed later.

Table 7.14: FPGA Model vs. Pruned Software Model vs. Fully-Grown Software Model for the UNIBS Dataset

Measure	Hardware		Pruned Software		Fully-Grown Software	
	Majority	Probability	Majority	Probability	Majority	Probability
Accuracy (%)	96.3	96.5	96.3	96.5	98.3	98.4
F-score	0.823	0.834	0.823	0.834	0.924	0.933

Similarly, Table 7.15 shows the comparison between the hardware model (20-trees, 10-levels), the pruned software model (20-trees, 10-levels) and the fully-grown software model (20 trees-fully grown) using the UNB dataset. Again, the hardware results are identical to the pruned software results due to the precision of the Python script in converting the trained software model to its hardware counterpart. In addition, the comparison between the hardware performance and the fully-grown software performance reassures that not much performance is lost when using a pruned tree.

Table 7.15: FPGA Model vs. Pruned Software Model vs. Fully-Grown Software Model for the UNB Dataset

Measure	Hardware		Pruned Software		Fully-Grown Software	
	Majority	Probability	Majority	Probability	Majority	Probability
Accuracy (%)	82.4	83.2	82.4	83.2	83.5	83.6
F-score	0.824	0.833	0.824	0.833	0.837	0.838

7.3.4. Timing analysis. Upon implementing the random forest algorithm in hardware, it is essential that we perform a timing analysis of the synthesized circuit in order to ensure that it meets all timing requirements. Timing analysis is performed using a tool in Quartus known as Timing Analyzer. Using Timing Analyzer, all input and output paths in the random forest design must be constrained and the clock must be defined. In our analysis, the minimum delay on both input and output paths was set to 2 ns, while the maximum delay was set to 3 ns. This helps Timing Analyzer in detecting paths which suffer from negative slacks and clock skews. After running Timing Analyzer it usually gives the maximum frequency at which the digital circuit can run on the specified FPGA. Figure 7.43 shows the result of the timing analysis performed

on the (20-trees, 10-levels) model. As we can see, the maximum frequency at which we can operate our random forest design is approximately 35 MHz.

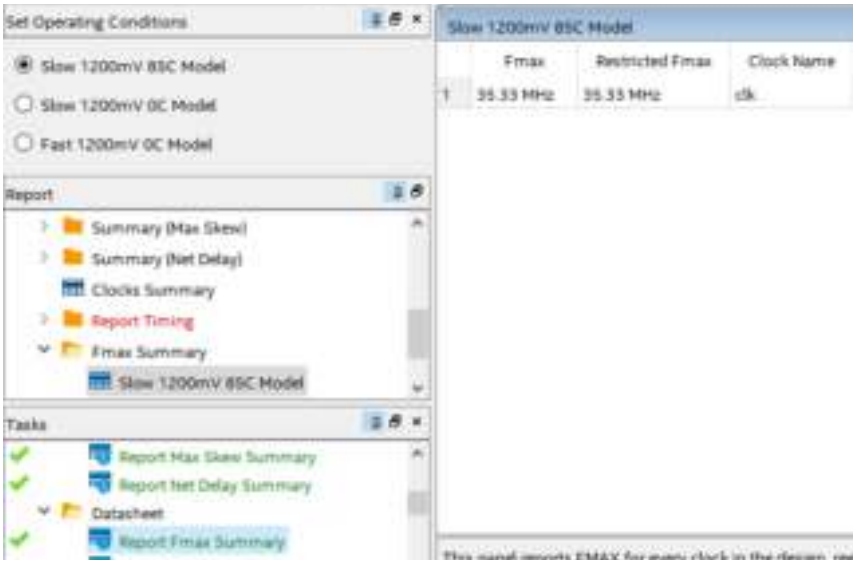


Figure 7.43: Timing Analyzer Report

Next, we calculate the time required to classify one packet using our random forest hardware accelerator. If the maximum frequency is 35 MHz, then the time period of the master clock is 28.571 ns. Recall that our hardware design of a random forest is highly pipelined, whereby it consisted of 1 input pipeline stage, 1 output pipeline stage, and 1 pipeline stage for each tree level regardless of the number of trees in the forest. This means that if we were to implement our (20-trees, 10-levels) model on the FPGA, the number of pipeline stages in the design would be 12 pipeline stages. Keeping in mind that each pipeline stage requires only one cycle to execute, therefore, the delay until we obtain the first classification from this design is 12 * 28.571 = 0.3429 μs.

Nevertheless, the beauty of a pipelined design, is the fact that after we obtain the first classification result, we start getting one classification each clock cycle since the pipeline is now full. Hence, we are able to classify one packet every 28.571 ns in hardware. On the other hand, when we find the average classification time of one packet using the same model in software, it turns out that we can classify one UNIBS packet in 2.6469 μs and one UNB packet in 1.3624 μs. These numbers reflect the enormous speedup obtained using the hardware acceleration of the random forest algorithm. Speedup can be defined as the increase in execution speed when using one system in

place of another system. Mathematically, speedup can be calculated using Equation (16):

$$Speedup = \frac{Software - Based\ Classification\ Time}{Hardware - Based\ Classification\ Time} \quad (16)$$

$$UNIBS\ Speedup = \frac{2.6469\ \mu s}{28.571\ ns} = 92.64$$

$$UNB\ Speedup = \frac{1.3624\ \mu s}{28.571\ ns} = 47.68$$

Therefore, compared to the software implementation of a random forest algorithm, the hardware accelerator can be up to 92 times faster when classifying a packet in the UNIBS dataset, and 47 times faster when classifying a packet from the UNB dataset. This is an encouraging result considering the fact that our objective is to deploy such an accelerator at datacentres that handle millions of network packets per second. This is the reason why we mentioned earlier that a compromise in the accuracy or the F-score of only 2% or 0.1 respectively is justified by the fact that we are gaining a huge amount of classification speed that enables datacentres to operate at a much higher throughput that enables online traffic classification in congested networks. To find out the average throughput achieved by our design, we first find the average packet size of the UNIBS and UNB datasets using Wireshark to be 626 Bytes per packet. Using a 35 MHz clock we can approximately classify 35 million packets per second. This is because we can classify one packet every clock cycle, as mentioned earlier. This results in a total throughput of 35000000 (packets classified per second) multiplied by 626 (average packet size) which translates to a throughput of 163.24 Gbps. Hence, the average throughput achieved by our design is 163.24 Gbps.

7.3.5. Simulation results. Figure 7.44 shows the simulation results of one tree in the random forest. In this figure, we can clearly see the progression of the feature vector of the packet from one pipeline stage to the other. Moreover, we can also see the progression of the next address from one pipeline stage to the other every clock cycle.

Figure 7.45 shows the simulation of the top-level module. The first line in the simulation is the network packet content, the second line is the clock running at 35 MHz, the third line is a reset signal, and the last line is the class label of each packet. Note that, the packets used in this simulation are known to be of classes 1, 2, 3, 4, and 5, respectively. During the first 12 clock cycles the output class does not reflect any

output (don’t care). We can clearly see that we obtain the first classification after a 12 clock cycles delay, which further confirms our calculations earlier. After that, we classify one packet every clock cycle.

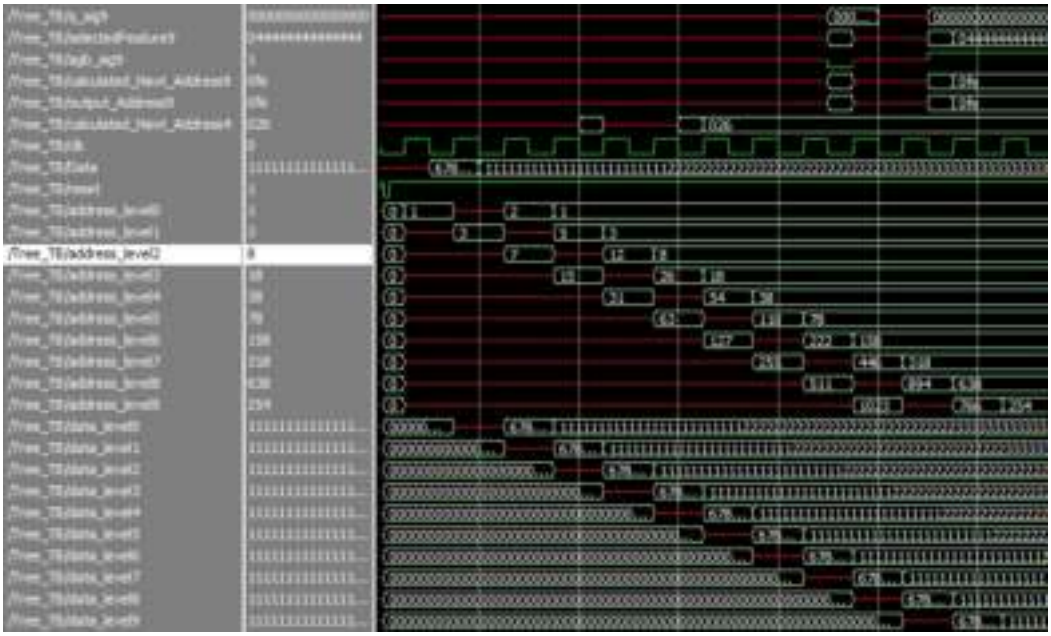


Figure 7.44: Pipelined Tree Simulation

Figure 7.45: Top-Level Module Simulation

To further verify the operation of the random forest classifier, we use the Tektronix LA6401 logic analyzer to inspect the signals of the random forest module on the FPGA. This helps test and debug the circuit in case of any misbehaviour. To do so, we project the important signals of the forest module onto the GPIO pins of the DE2-115 board. Those signals include the master clock, the reset, and the class label signals. After that, the logic analyzer’s probes are connected to the GPIO pins in order to record the behaviour of those signals. Note that, the packets used are known to be of classes 1, 2, 3, 4, and 5, respectively. Figure 7.46 shows the logic analyzer’s setup when connected to the FPGA running the random forest classifier.

Figure 7.47 shows the waveform obtained using the logic analyzer. After we hit the reset signal, the first packet is passed to the random forest classifier at the first negative edge of the clock. After 12 clock cycles, which is the time it takes the packet

to pass through all pipeline stages in our design, we start getting the class label of the first packet followed by the successive classes. Notice that, the output during the first 12 clock cycles is treated as a don't care. The logic analyzer's output matches that of the simulation results obtained earlier, which confirms the correct operation of our random forest design.

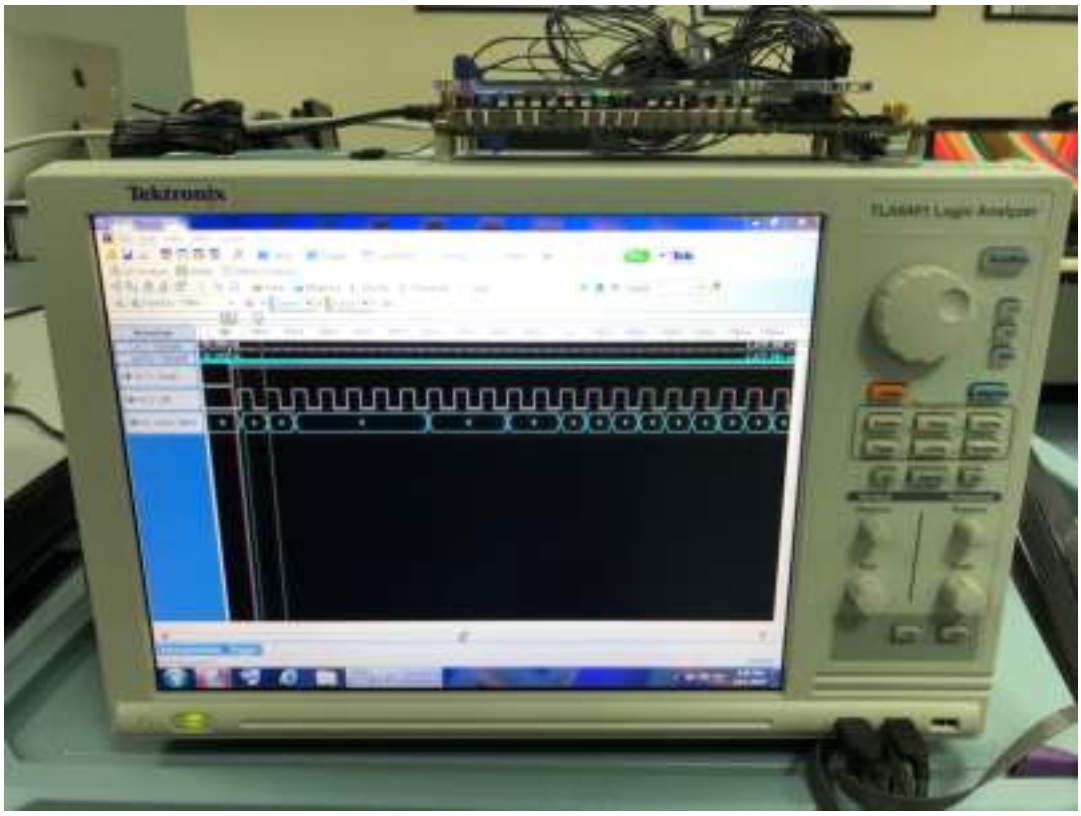


Figure 7.46: Logic Analyzer's Setup

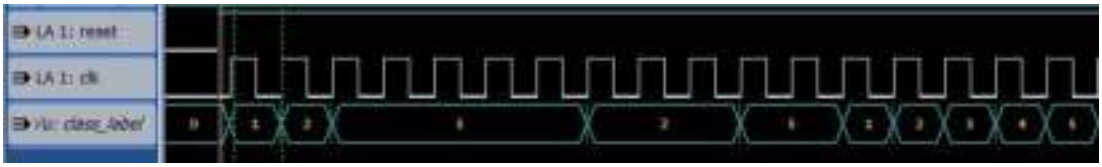


Figure 7.47: Logic Analyzer's Waveform

7.3.6. The final prototype. For the final demonstration of the random forest accelerator on the DE2-115 board, we used KEY0 as the reset input, and we displayed the class label on HEX0. Figures 7.48, 7.49, 7.50, 7.51, 7.52 show the DE2-115 board when the class label of the packet is 1, 2, 3, 4, and 5, respectively.

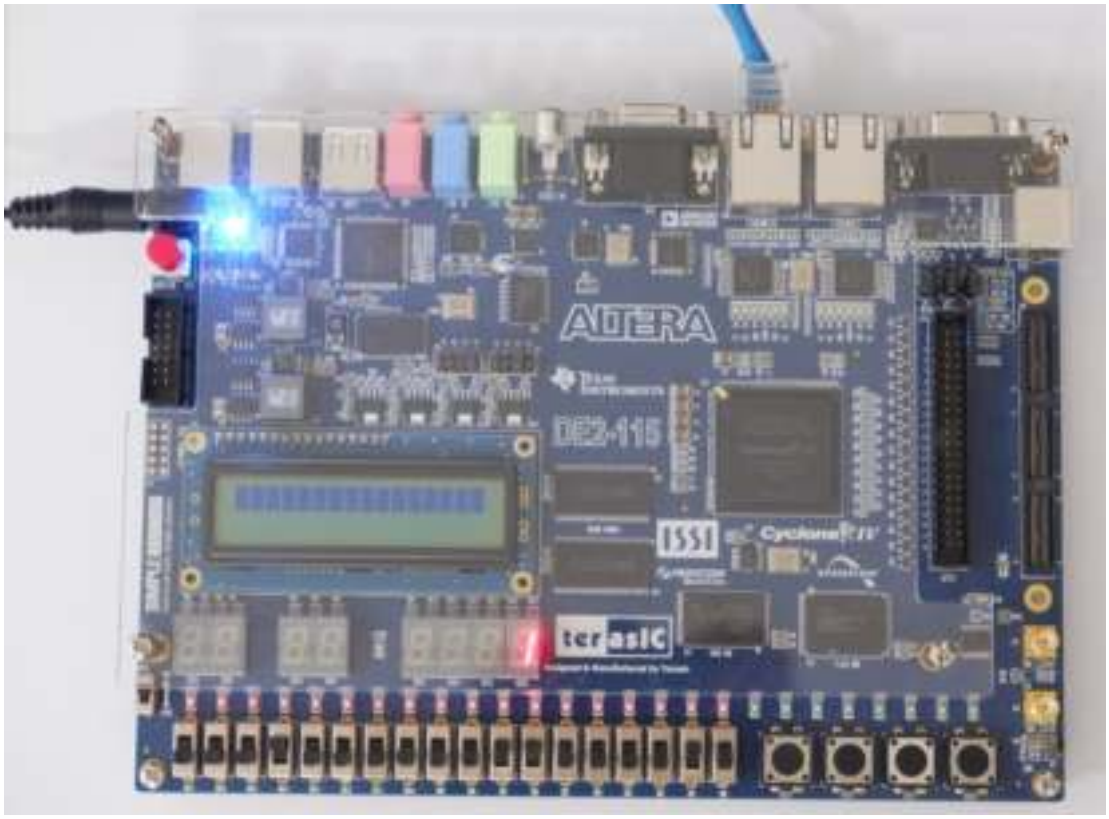


Figure 7.48: Class 1 on the DE2-115 Board

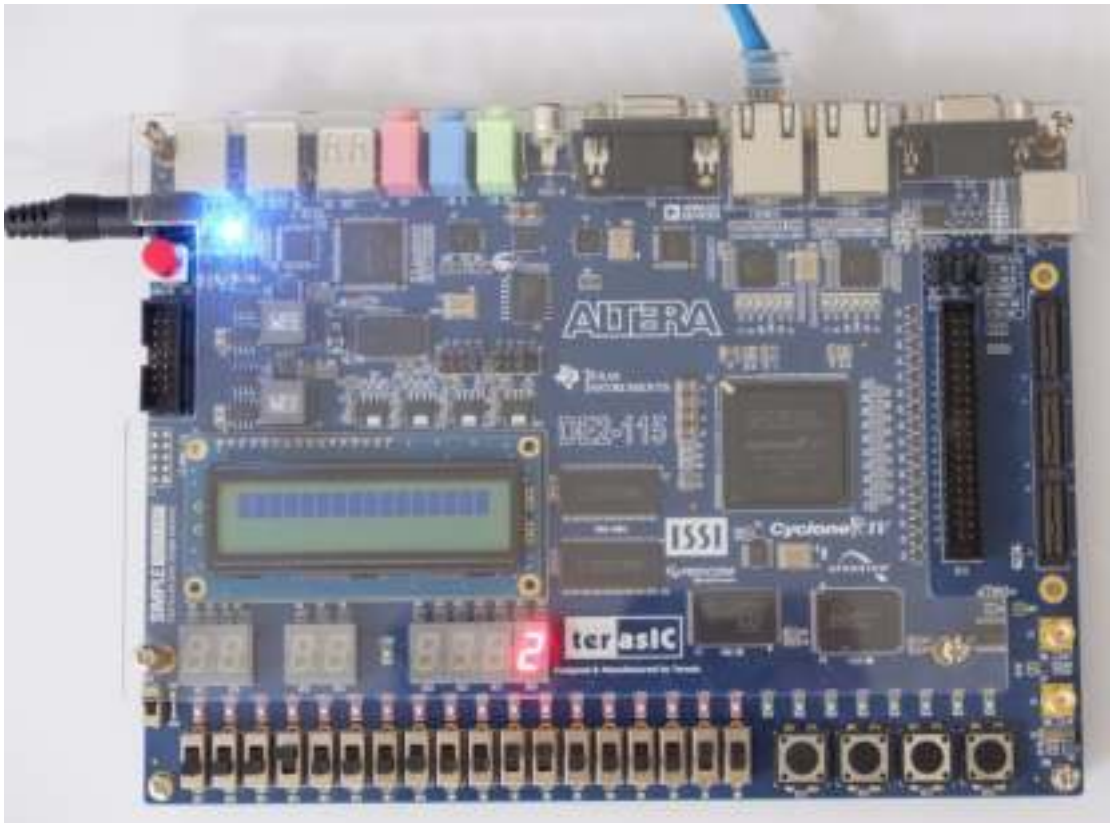


Figure 7.49: Class 2 on the DE2-115 Board

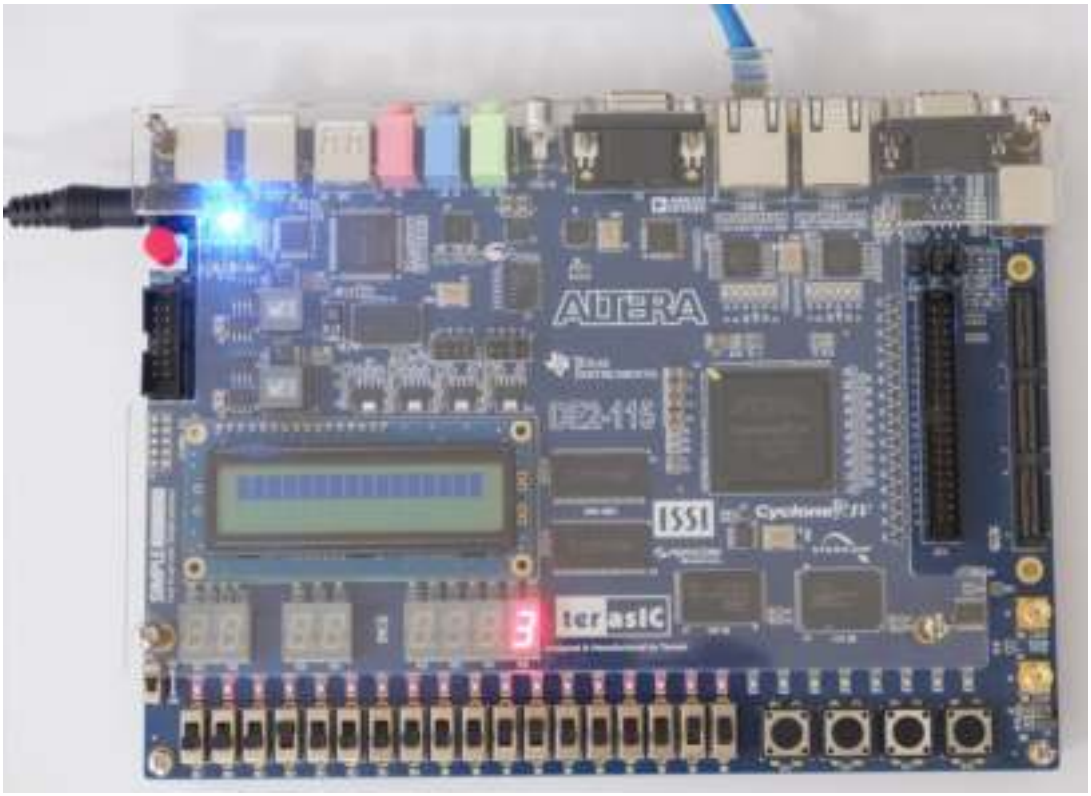


Figure 7.50: Class 3 on the DE2-115 Board

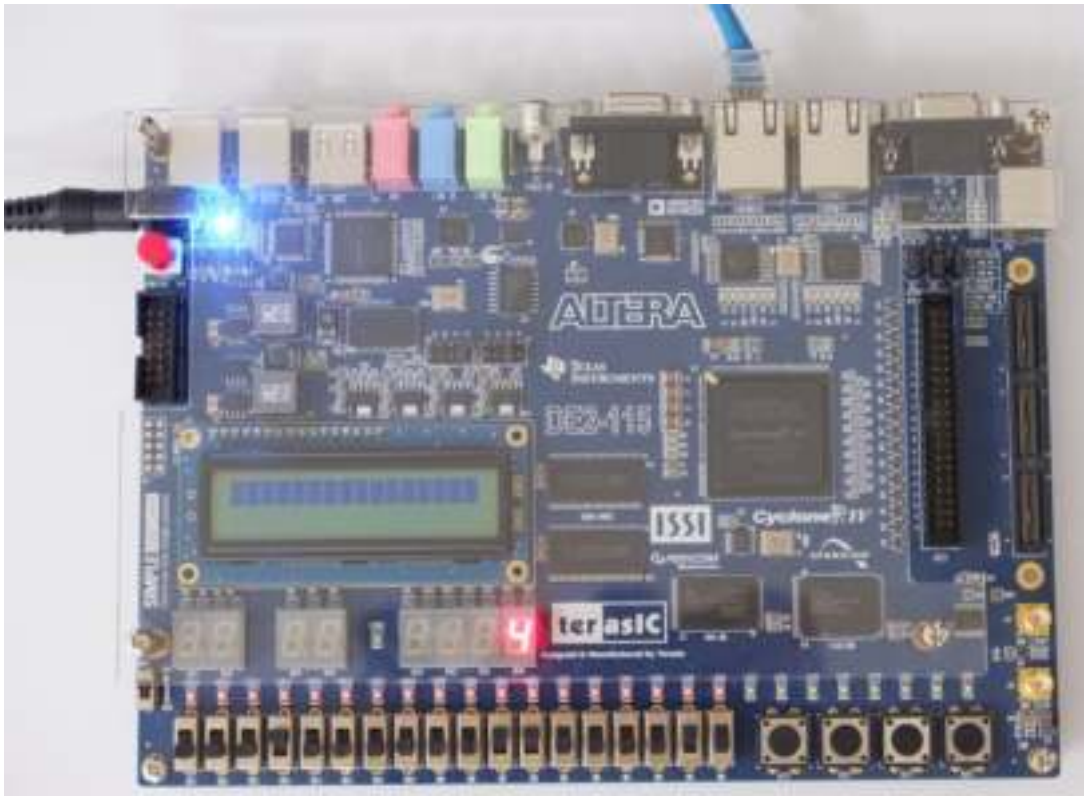


Figure 7.51: Class 4 on the DE2-115 Board

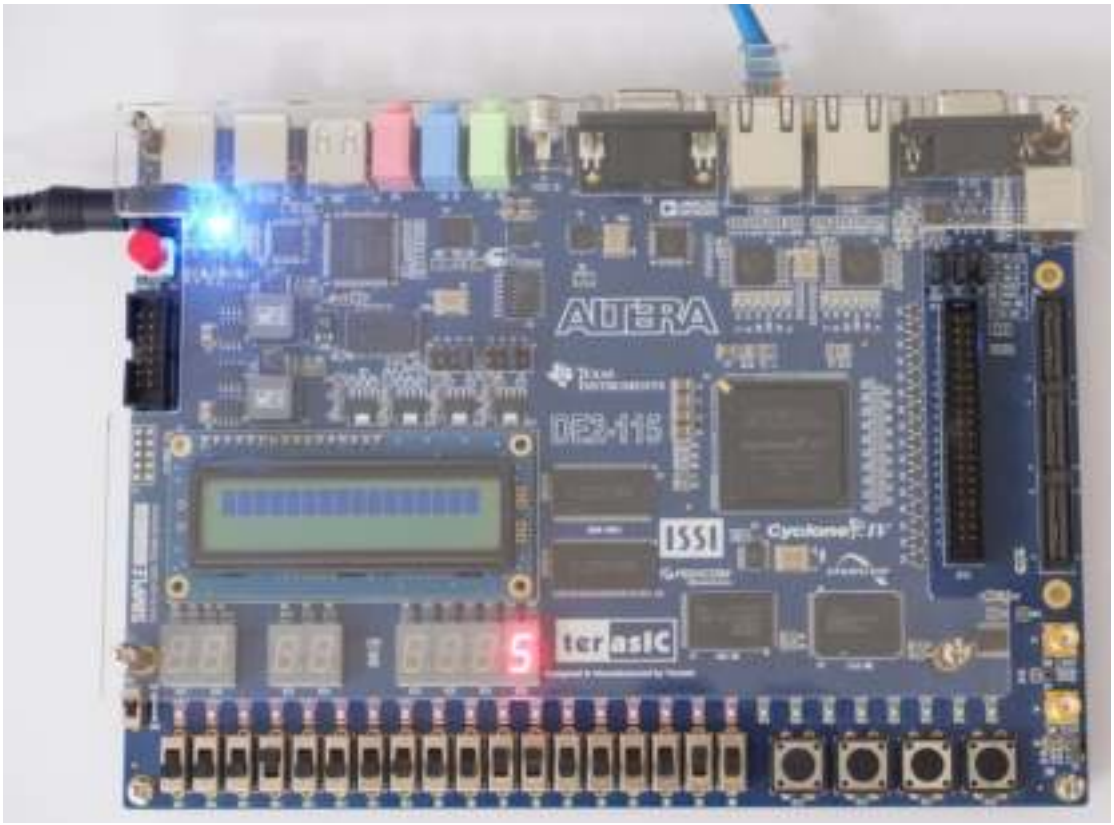


Figure 7.52: Class 5 on the DE2-115 Board

Table 7.16 shows a summary of the FPGA resource utilization after implementing the (20-trees, 10-levels) random forest model.

Table 7.16: FPGA Resource Utilization

Resource	Total	Used	% Utilization
Configurable Logic Blocks (CLBs)	114480	79207	69%
Total Registers	N/A	1200	N/A
Total Pins	529	9	2%
Total Virtual Pins	N/A	0	0%
Total Memory Bits	3981312	1258260	32%
Embedded Multiplier 9-bit Elements	532	0	0%
Total PLLs	4	0	0%

7.4. Discussion of Results

Having implemented the random forest accelerator on the FPGA, we compare our results to existing work in the field of network traffic classification. Two main performance measures are of interest; classification accuracy and F-scores. Additionally, we compare our hardware packet throughput to that reported in the literature.

The highest accuracy and F-score obtained on the UNIBS dataset in our experiments were 98.5% and 0.932, respectively. By inspecting the previous work published on traffic classification using the UNIBS dataset, we found that confidence intervals of classification accuracies are not reported. They tend to report only their highest accuracy result while ignoring the importance of mentioning the size of the training set and the testing set. This is because obtaining a 98% accuracy using a dataset which consists of only 100 instances, for example, cannot guarantee that the model would achieve the same results on new instances. On the other hand, obtaining a 98% accuracy on a dataset that consists of 10000 instances is more reliable due to the fact that more instances were used for training and testing. Therefore, we opt to calculate the confidence intervals of our reported accuracies and F-scores. The confidence interval can be calculated using Equation (17):

$$p = \frac{\left(f + \frac{z^2}{2N} \mp z\sqrt{\frac{f}{N} - \frac{f^2}{N} + \frac{z^2}{4N^2}}\right)}{\left(1 + \frac{z^2}{N}\right)} \tag{17}$$

where f is the obtained accuracy or F-score, z is 2.33 which corresponds to a 98% confidence level. N is the number of test instances in the experiment. In our case, the accuracy is 98.5% while the F-score is 0.932. We used a 10-fold cross-validation experiment to obtain such results from a dataset of 45541 instances, therefore, N is 4554 which is approximately the number of test instances in each fold.

Therefore, the confidence interval for the classification accuracy is found out to be:

$$p \in [98.0\%, 98.9\%]$$

Whereas, the confidence interval for the F-score is found out to be:

$$p \in [0.923, 0.940]$$

These results show that we can guarantee that if this experiment was to be repeated on the same dataset using different seeds, the accuracy will lie between 98.0% and 98.9% with a 98% confidence. Similarly, the F-score will lie between 0.923 and 0.940 with a 98% confidence.

As mentioned earlier, publications on traffic classification using the UNIBS dataset tend to overlook the need to report the confidence intervals, and hence, we will simply compare our mean accuracy and F-score with their reported accuracies and F-scores. Table 7.17 shows a summary of the obtained accuracies and F-scores from the

literature compared to the ones obtained in our experiments. Few observations are drawn from the results in Table 7.17. First, some papers did not report the F-score results, which made the comparison slightly more difficult as two papers surpassed our F-score, while one paper fell behind. On the other hand, we can see that we surpass all the papers in terms of classification accuracy.

Table 7.17: Summary of Accuracies and F-scores in the Literature vs. Proposed Design

Design	Accuracy	F-score
Proposed	98.5 %	0.932
[13]	98.0 %	0.980
[25]	96.3 %	N/A
[26]	95.5 %	N/A
[30]	90.6 %	0.964
[31]	91.3 %	0.916
[38]	97.5 %	N/A

We also examine the maximum throughput achieved due to implementing the random forest algorithm on an FPGA. Although this work, according to the best of our knowledge, is the first attempt to accelerate a random forest-based network traffic classifier on an FPGA, nonetheless, we compare our achieved throughput to that of other implementations including C4.5 decision tree and SVM based classifiers. Table 7.18 shows a summary of the obtained throughputs from the literature. As mentioned in Section 7.3, the maximum throughput achieved by our random forest accelerator is 163.24 Gbps. This is more than twice as fast as the maximum reported throughput in Table 7.18.

Table 7.18: Summary of Throughputs in the Literature vs. Proposed Design

Design	Throughput (Gbps)
Proposed	163.24
[39]	8
[40]	28.6
[41]	40
[42]	80

Chapter 8. Conclusion and Future Work

Traffic classification is the process of assigning traffic flows to the applications that generated them. Machine learning is the most successful method of traffic classification as it makes use of flow-level features that do not affect the privacy of the communicating parties and it overcomes the issues of encryption whereby a classifier does not decipher the contents of the packets.

In this work, we made use of two publicly available datasets known as UNIBS and UNB dataset. Five different classes of traffic were used from each dataset, where Skype, Browser, BitTorrent, Mail, and RSS were chosen from the UNIBS dataset, while Skype, Torrent, Netflix, Spotify, and YouTube were used from the UNB dataset. In this research, we started by extracting a list of features which were then reduced using two feature selection algorithms; stepwise regression and random forest feature selection. We investigated the effect of discretization on the performance of the classification algorithms using an entropy-based and a Gini index-based discretization algorithms. The improvement in the classifiers' performance due to the use of discretization was negligible compared to the significant increase in the training and testing time of the classifiers. As a result, we continued our experiments using the non-discretized datasets to allow for a much faster classification process.

We conducted three main experiments to test multiple parameters. The first experiment was to build five different classifiers, namely, naïve Bayes, linear SVM, polynomial SVM, KNN, and random forest using the cross-validation testing method. This experiment was mainly concerned with the performance of the different classifiers in terms of overfitting, classification accuracy and F-scores. In general, random forest outperformed every other classifier in almost all aspects reaching a maximum accuracy of 98.5% and F-score of 0.932. In the second experiment, we concluded that the most optimal percentage of packets within a flow that need to be considered when extracting flow-level features is around 60% which required a waiting time of about 21ms. Furthermore, the last experiment revealed that the training set size was not of a great importance to the different classifiers since they do not significantly boost the classification performance.

In addition to the software implementation, we designed a hardware-based random forest traffic classifier using a highly pipelined architecture that makes use of the parallel execution capabilities of Field Programmable Gate Arrays (FPGAs). We

used the DE2-115 development board in the implementation of a (20-trees, 10-levels) classifier. The results obtained using the hardware implementation show that we can map a software-trained model precisely to a hardware implementation. In order to fit an entire random forest tree on the FPGA chip we needed to reduce the number of trees and the number of levels per tree compared to the software models. This resulted in a minor reduction in classification accuracy (96.5%) and F-score (0.834) while boosting the classification speed significantly resulting in an average throughput of 163.24 Gbps. The achieved throughput was more than twice the maximum throughput reported in the literature. Moreover, the hardware acceleration of the Random Forest classifier enabled us to achieve a speedup of 92.64 and 47.68 using the UNIBS and UNB datasets, respectively, when compared to their software-based models. This helps datacentres and internet service providers implement an online traffic classifier that can cope with the increasing network speeds.

In future work, we will port our implementation to a more sophisticated FPGA chip that can allow us to fit more trees with more levels on it, as well as, enhancing the throughput further. Moreover, we will seek to deploy and test our FPGA accelerator in a real-life network that allows the flow of network packets into the FPGA accelerator and observe its performance. Finally, we would like to modify our design to enable online training of the random forest model in order to allow our design to learn new changes in the behaviour of the network on the spot.

References

[1] J. Clement, “Number of Internet Users Worldwide 2005-2018,” *Statista*, Jan. 9, 2019. [Online]. Available: https://www.statista.com/statistics/273018/number-of-internet-users-worldwide. [Accessed: Feb. 2, 2019].

[2] “Worldwide Broadband Speed League 2018,” *Cable*, 2018. [Online]. Available: https://www.cable.co.uk/broadband/speed/worldwide-speed-league. [Accessed: Jan. 4, 2019].

[3] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin and J. Aguilar, "Towards the Deployment of Machine Learning Solutions in Network Traffic Classification: A Systematic Survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1988-2014, Secondquarter 2019.

[4] Y. R. Qu and V. K. Prasanna, "Compact hash tables for high-performance traffic classification on multi-core processors," in *2014 IEEE 26th International Symposium on Computer Architecture and High-Performance Computing*, Jussieu, 2014, pp. 17-24.

[5] “What is an FPGA? Field Programmable Gate Array,” *Xilinx*. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. [Accessed: Nov. 15, 2018].

[6] C. Maxfield, *The design warrior's guide to FPGAs: Devices, Tools and Flows*, 1st ed. Elsevier, 2004. [E-book] Available: https://www.sciencedirect.com. [Accessed: Mar. 29, 2019].

[7] A. Dainotti, A. Pescape and K. C. Claffy, "Issues and Future Directions in Traffic Classification," *IEEE Network*, vol. 26, no. 1, pp. 35-40, January-February 2012.

[8] A. Madhukar and C. Williamson, "A longitudinal study of p2p traffic classification," in *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, Monterey, CA, 2006, pp. 179-88.

[9] "Application Layer Packet Classifier for Linux," *SourceForge*, Jan. 7, 2009. [Online]. Available: http://l7-filter.sourceforge.net. [Accessed: Feb. 23, 2019].

[10] T. Liu, Y. Sun and L. Guo, "Fast and memory-efficient traffic classification with deep packet inspection in CMP architecture," in *2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*, Macau, 2010, pp. 208-17.

[11] Y. Xue, D. Wang and L. Zhang, "Traffic classification: Issues and challenges," in *2013 International Conference on Computing, Networking and Communications (ICNC)*, San Diego, CA, 2013, pp. 545-9.

[12] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, “Blinc: Multilevel traffic classification in the dark,” in *Proceedings of the Special Interest Group on Data Communication conference (SIGCOMM)*, Philadelphia, PA, 2005, ACM.

[13] G. Lu, R. Guo, Y. Zhou and J. Du, "An Accurate and Extensible Machine Learning Classifier for Flow-Level Traffic Classification," *China Communications*, vol. 15, no. 6, pp. 125-38, June 2018.

[14] Y. Wang, Y. Xiang, J. Zhang and S. Yu, "Internet traffic clustering with constraints," in *2012 8th International Wireless Communications and Mobile Computing Conference (IWCMC)*, Limassol, 2012, pp. 619-24.

[15] A. Saeed and M. Kolberg, "Towards Optimizing WLANs Power Saving: Novel Context-Aware Network Traffic Classification Based on a Machine Learning Approach," *IEEE Access*, vol. 7, no. 1, pp. 3122-35, 2019.

[16] B. Yamansavascilar, M. A. Guvensan, A. G. Yavuz and M. E. Karsligil, "Application identification via network traffic classification," in *2017 International Conference on Computing, Networking and Communications (ICNC)*, Santa Clara, CA, 2017, pp. 843-8.

[17] A. Este, F. Gringoli and L. Salgarelli, "On-line SVM traffic classification," in *2011 7th International Wireless Communications and Mobile Computing Conference*, Istanbul, 2011, pp. 1778-83.

[18] D. Tong, Y. R. Qu and V. K. Prasanna, "Accelerating Decision Tree Based Traffic Classification on FPGA and Multicore Platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, pp. 3046-59, Nov. 2017.

[19] T. Groléat, M. Arzel and S. Vaton, "Stretching the Edges of SVM Traffic Classification with FPGA Acceleration," *IEEE Transactions on Network and Service Management*, vol. 11, no. 3, pp. 278-91, Sept. 2014.

[20] T. Soylu, O. Erdem, A. Carus and E. S. Güner, "Simple CART based real-time traffic classification engine on FPGAs," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, 2017, pp. 1-8.

[21] *Wireshark.* [Online]. Available: https://www.wireshark.org. [Accessed: Oct. 10, 2018].

[22] *Tcpdump.* [Online]. Available: https://www.tcpdump.org. [Accessed: Oct. 20, 2018].

[23] R. Fontugne, P. Abry, K. Fukuda, D. Veitch, K. Cho, P. Borgnat and H. Wendt, "Scaling in Internet Traffic: A 14 Year and 3 Day Longitudinal Study, With Multiscale Analyses and Random Projections," *IEEE/ACM Transactions on Networking*, vol. 25, no. 4, pp. 2152-65, Aug. 2017.

[24] "UNIBS: Data sharing," *UNIBS*, Jul. 12, 2011. [Online]. Available: http://netweb.ing.unibs.it/~ntw/tools/traces. [Accessed: Feb. 14, 2019].

[25] M. Dusi, F. Gringoli, and L. Salgarelli, "Quantifying the Accuracy of the Ground Truth Associated with Internet Traffic Traces," *International Journal of Computer and Telecommunications Networking*, vol. 55, no. 5, pp. 1158-67, Apr. 2011.

[26] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso and K. C. Claffy, "GT: Picking Up the Truth from the Ground for Internet Traffic," *SIGCOMM Computer Communication Review*, vol. 39, no. 5, pp. 13-8, Jan. 2009.

[27] "GT: the Ground Truth software suite," *UNIBS*, Oct. 8, 2009. [Online]. Available: http://netweb.ing.unibs.it/~ntw/tools/gt. [Accessed: Feb. 15, 2019].

[28] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun and A. A. Ghorbani, "Characterization of encrypted and VPN traffic using time-related features," in *Proceedings of the 2nd International Conference on Information Systems Security and Privacy(ICISSP)*, 2016, SCITEPRESS.

[29] T. Shanableh and K. Assaleh, "Feature Modeling Using Polynomial Classifiers and Stepwise Regression," *Neurocomputing*, vol. 73, nos. 10– 12, pp. 1752–9, Jun. 2010.

[30] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining Practical Machine Learning Tools and Techniques*, 4th ed. Cambridge, MA : Morgan Kaufmann Publisher, 2017. [E-book] Available: https://www.sciencedirect.com. [Accessed: May. 23, 2019].

[31] U. M. Fayyad and K. B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. Chambéry, 1993, pp. 1022-7.

[32] I. Kononenko, "On biases in estimating multi-valued attributes," in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. Montreal, Quebec, 1995, pp. 1034-40.

[33] X. Lin, R. D. S. Blanton and D. E. Thomas, "Random forest architectures on FPGA for multiple applications," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*. Banff, Alberta, 2017, pp. 415-8.

[34] "DE2-115 User Manual," *Intel*, 2017. [Online] Available: https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf. [Accessed: Jul. 1, 2019].

[35] H. Wei, B. Sun and M. Jing, "BalancedBoost: A hybrid approach for real-time network traffic classification," in *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, Shanghai, 2014, pp. 1-6.

[36] Z. Nascimento, D. Sadok, S. Fernandes and J. Kelner, "Multi-objective optimization of a hybrid model for network traffic classification by combining machine learning techniques," in *2014 International Joint Conference on Neural Networks (IJCNN)*, Beijing, 2014, pp. 2116-22.

[37] "Quartus Prime Pro Edition," *Intel*. [Online]. Available: http://fpgasoftware.intel.com. [Accessed: Jan. 15, 2019].

[38] W. Linlin, L. Peng, M. Su, B. Yang and X. Zhou, "On the impact of packet inter arrival time for early stage traffic identification," in *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Chengdu, 2016, pp. 510-5.

[39] A. Monemi, R. Zarei and M. N. Marsono, "Online NetFPGA Decision Tree Statistical Traffic Classifier," *Computer Communications*, vol. 36, no. 12, pp. 1329-40, Jul. 2013.

[40] T. Groleat, M. Arzel and S. Vaton, "Hardware acceleration of SVM-based traffic classification on FPGA," in *2012 8th International Wireless Communications and Mobile Computing Conference (IWCMC)*, Limassol, 2012, pp. 443-9.

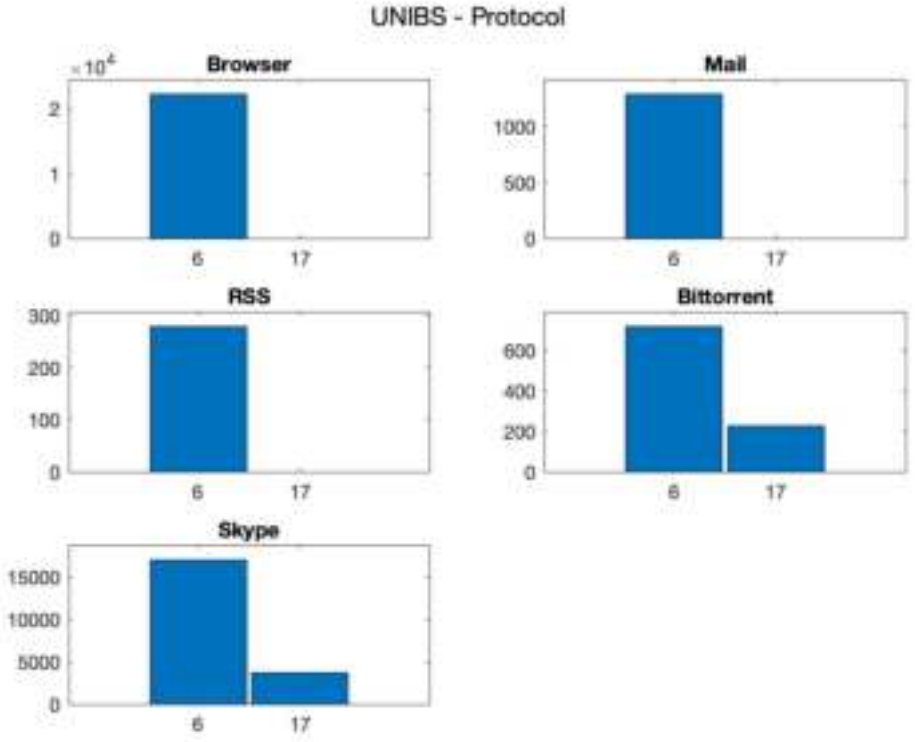
[41] S. Valenti, D. Rossi, A. Dainotti, A. Pescapè, A. Finamore and M. Mellia, "Reviewing Traffic Classification," *Data Traffic Monitoring and Analysis,* vol. 7754, no. 1, pp. 123-47, 2013.

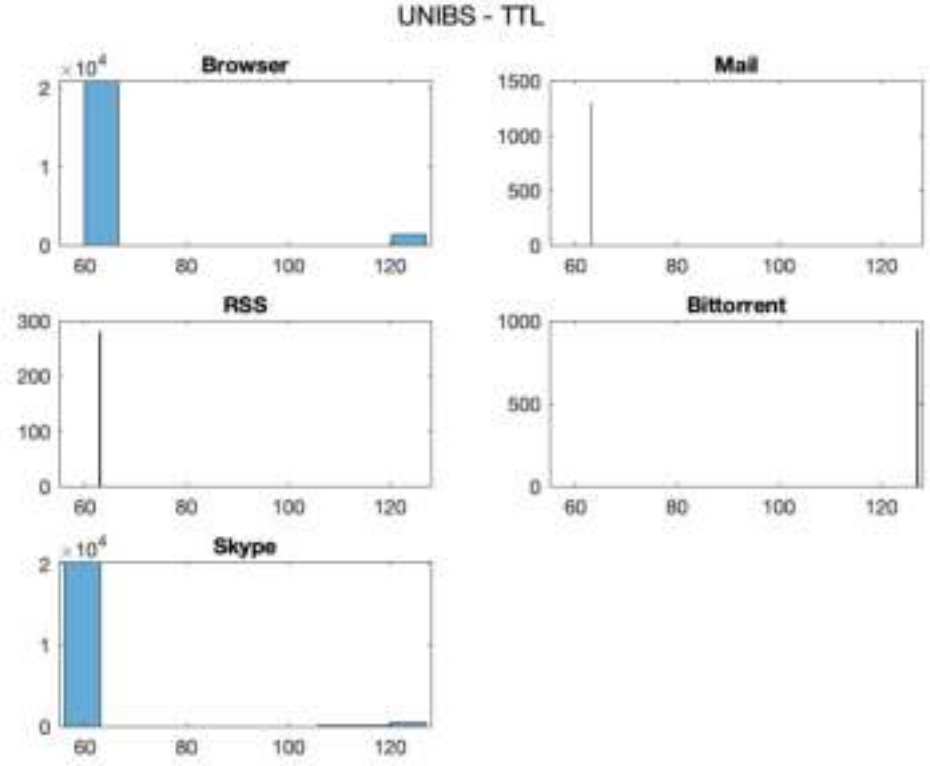
[42] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on FPGAs," in *Proceedings of the ACM/SIGDA International Symposium On Field Programmable Gate Arrays*, California, 2009, pp. 219-28.

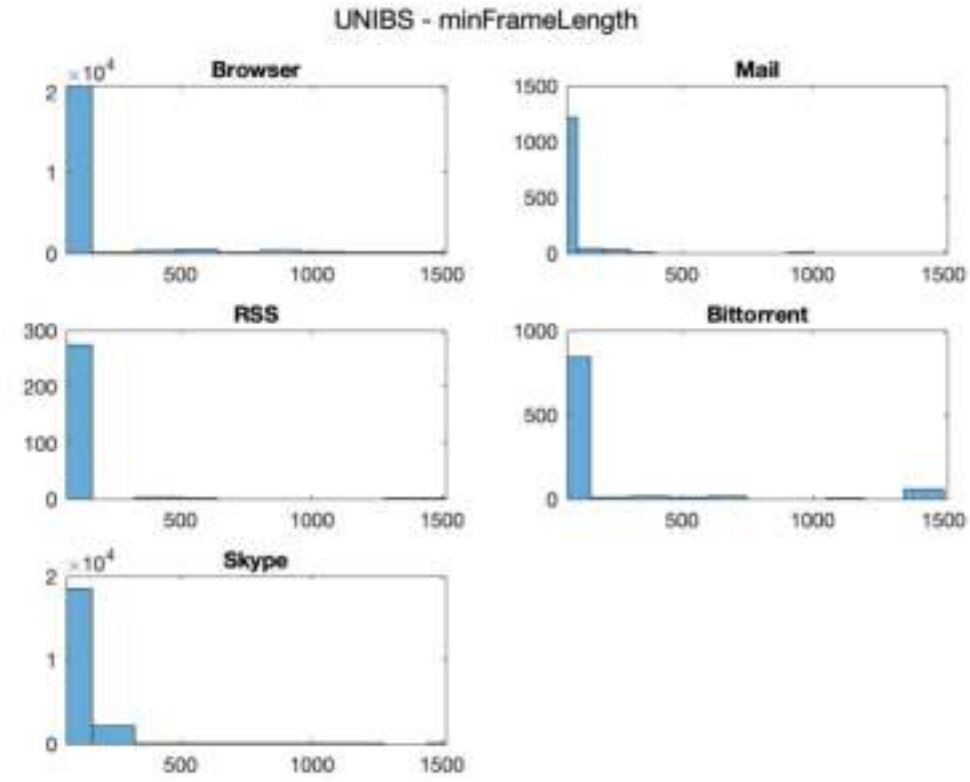
Appendix A – Feature Glossary

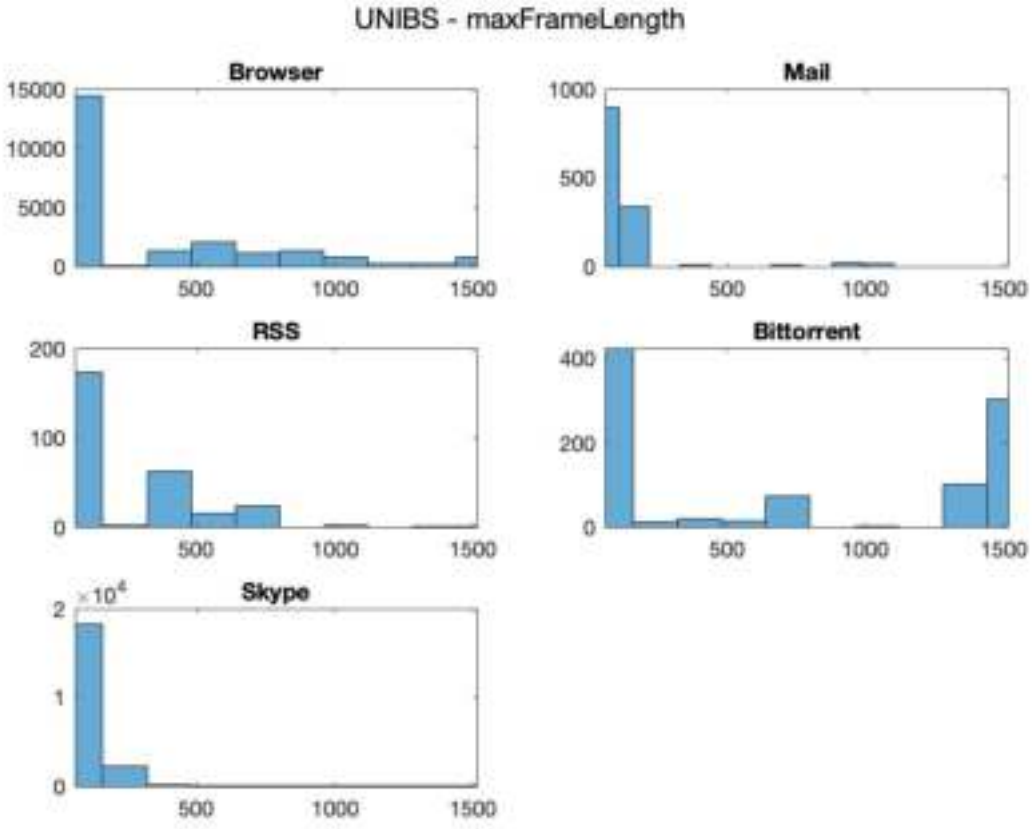
Feature	Definition
Source Port	16-bit unsigned integer that ranges from 0 to 65535 which represents the transport layer port number of the source computer
Destination Port	16-bit unsigned integer that ranges from 0 to 65535 which represents the transport layer port number of the destination computer
Frame Length	The size of the packet
Capture Length	The size of the packet in addition to all the headers that precede the packet
Interarrival Time	Time between the arrival of two consecutive packets within the same flow
Flow Duration	Total time to receive all packets within a flow
Minimum	Least within the flow of packets
Maximum	Highest within the flow of packets
Mean	Average within the flow of packets
Median	Median within the flow of packets
Variance	Variance calculated using all packets within the flow
Entropy	Entropy calculated using all packets within the flow

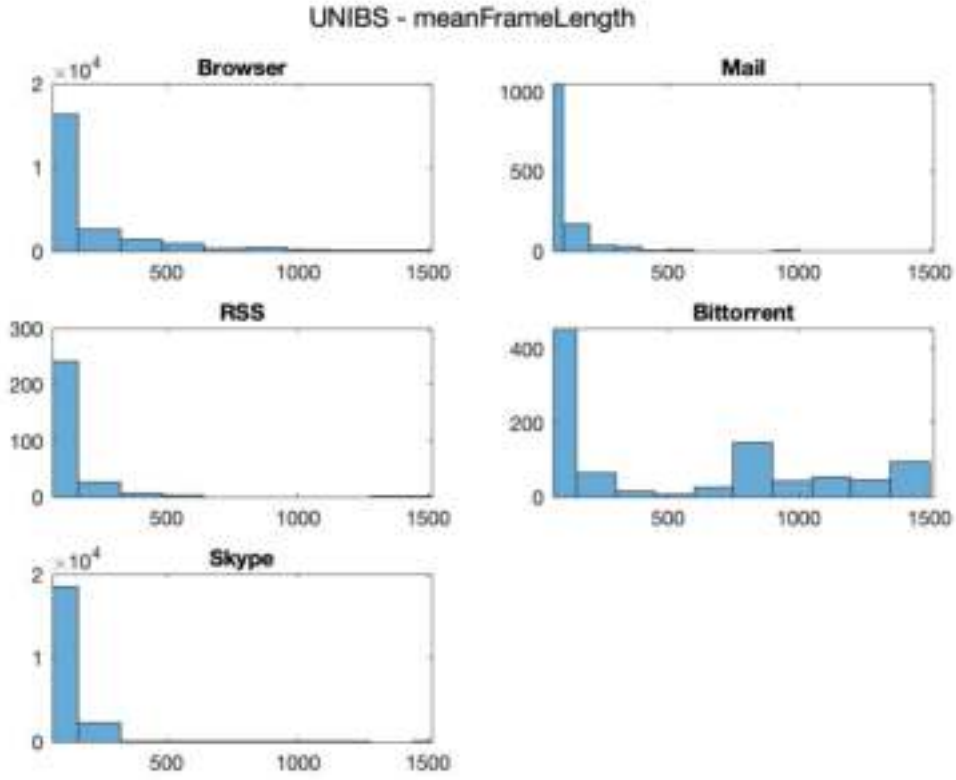
Appendix B – Feature Histograms

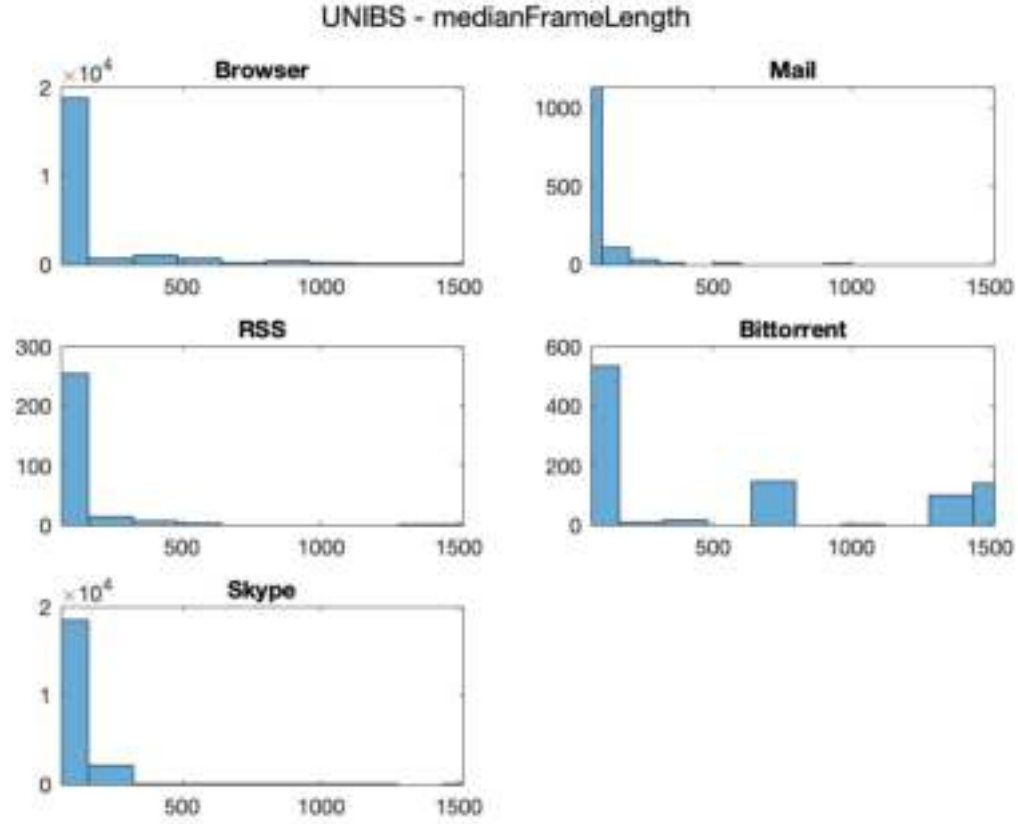


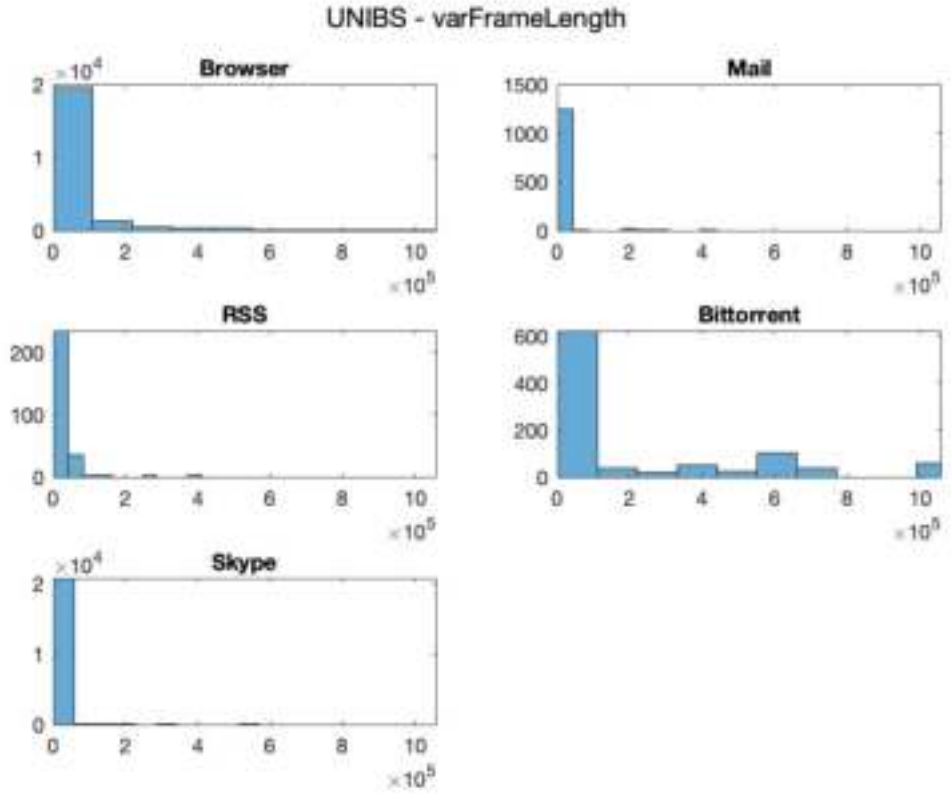


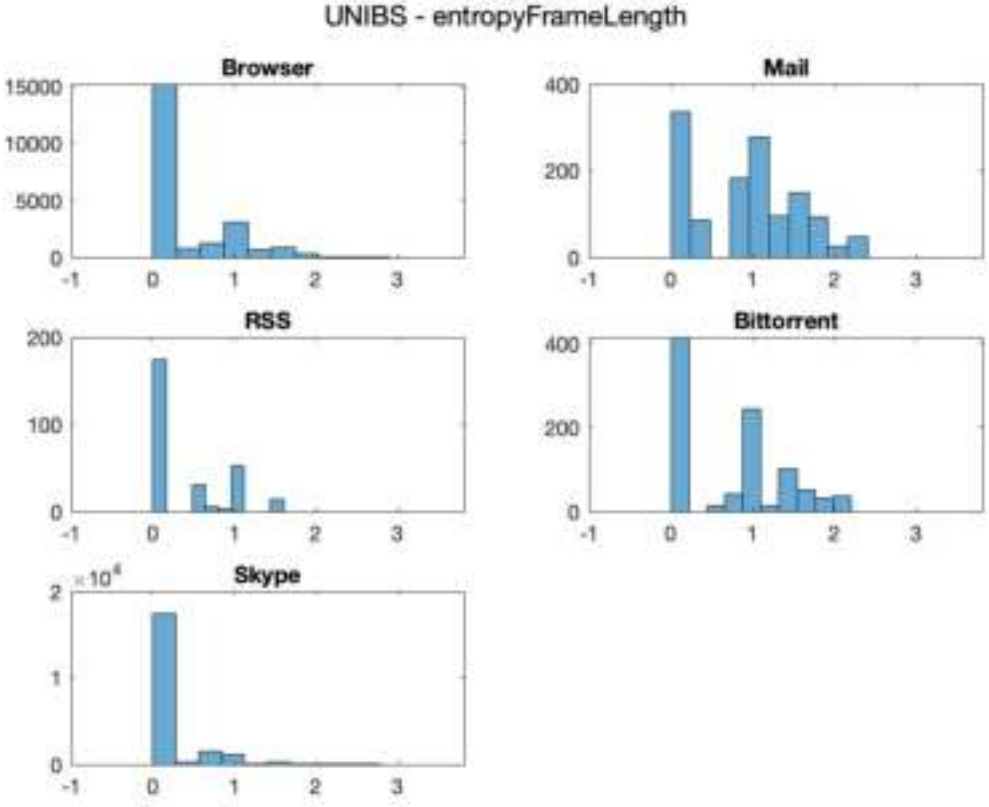

UNIBS - minFrameLength
Browser
Mail
RSS
Bittorrent
Skype
×10^4

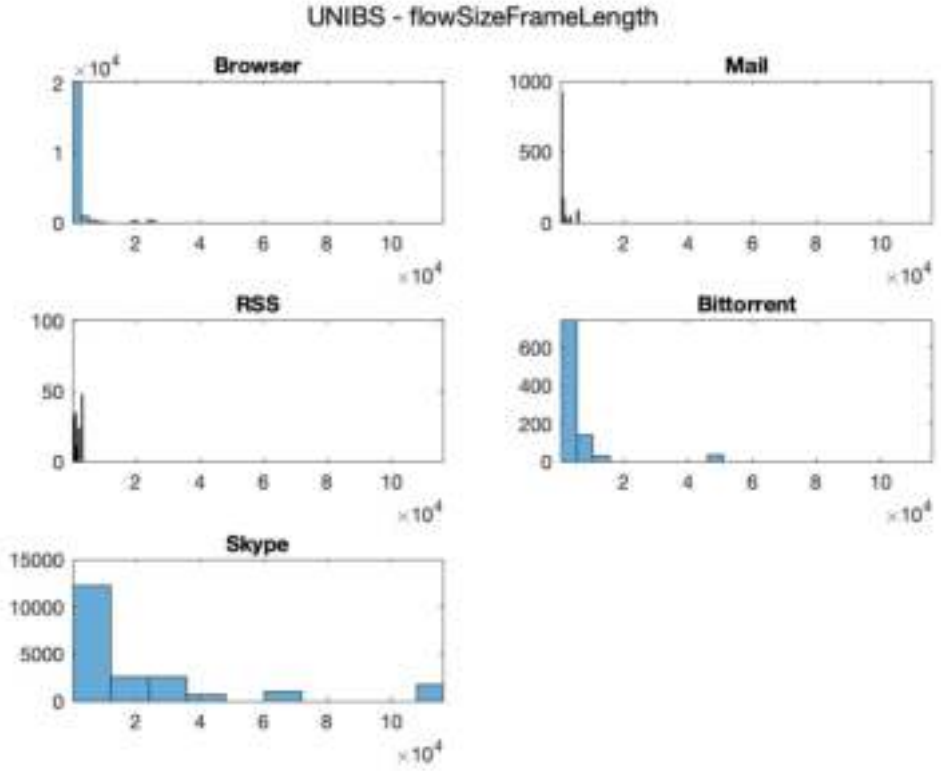

UNIBS - maxFrameLength
Browser
Mail
RSS
Bittorrent
Skype
×10^4

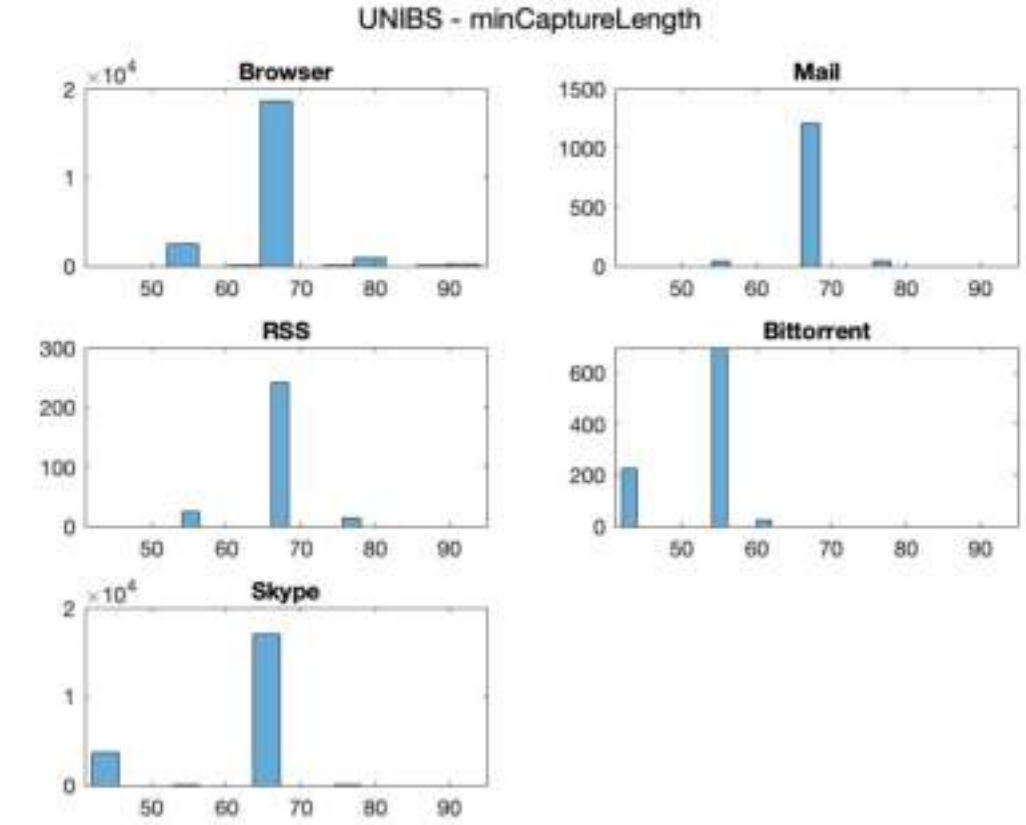

UNIBS - meanFrameLength
Browser
Mail
RSS
Bittorrent
Skype
×10^4
0
1
2
100
200
300
400
500
1000
1500

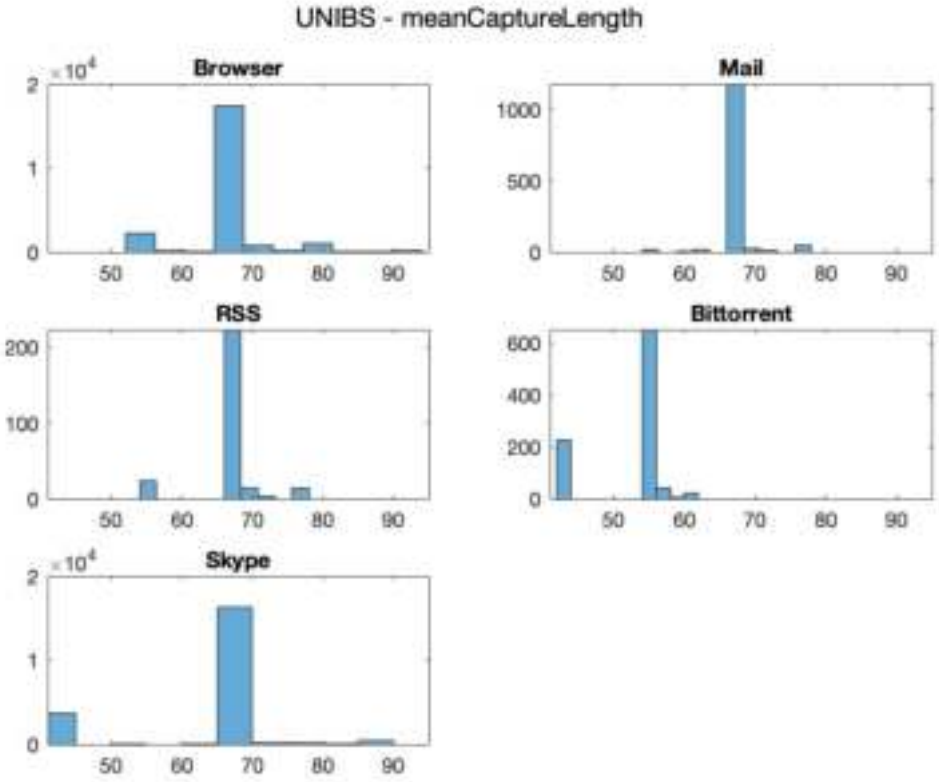

UNIBS - medianFrameLength
Browser
Mail
RSS
Bittorrent
Skype
×10^4
0
1
2
100
200
300
400
500
600
1000
1500

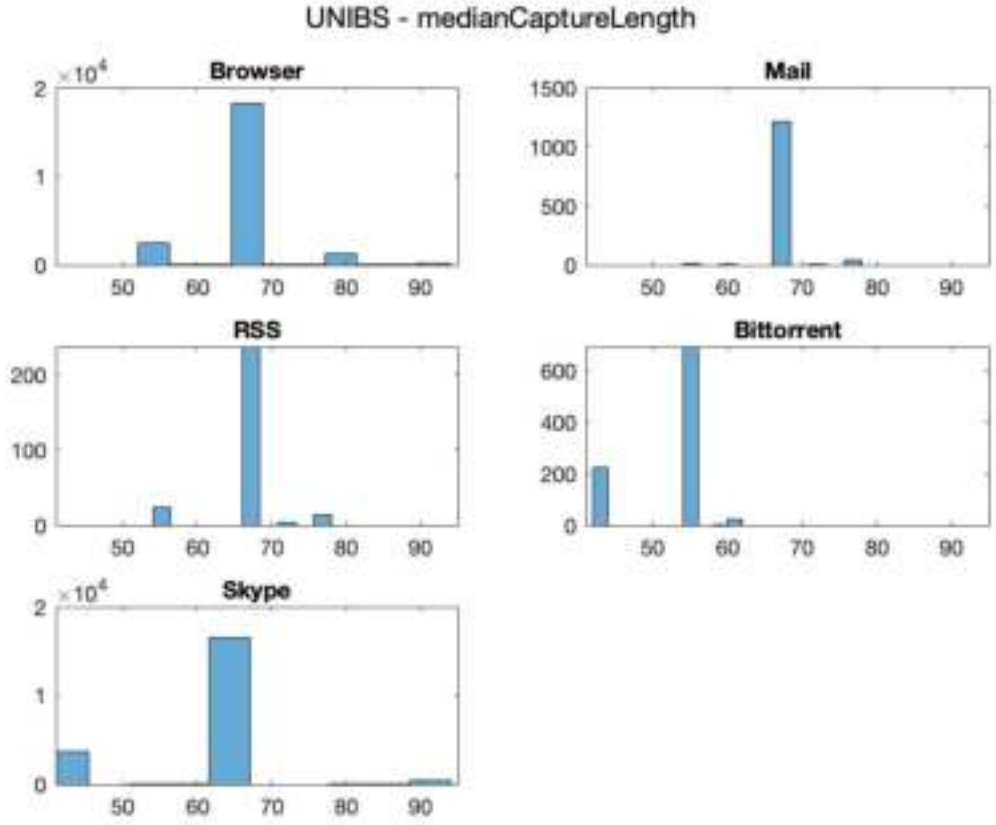

UNIBS - varFrameLength
Browser
Mail
RSS
Bittorrent
Skype

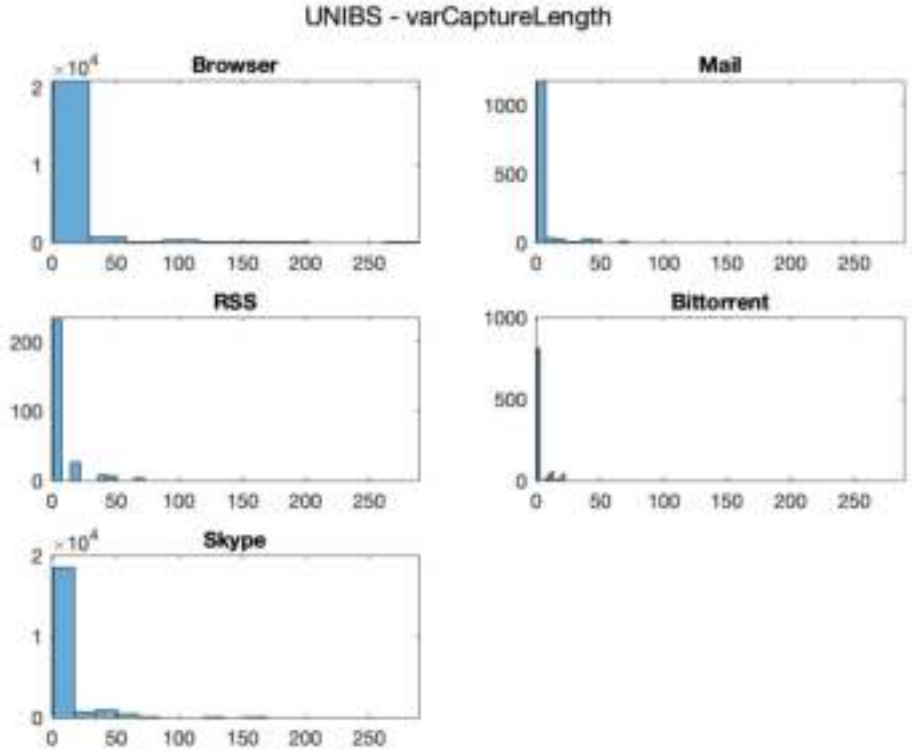

UNIBS - entropyFrameLength
Browser
Mail
RSS
Bittorrent
Skype

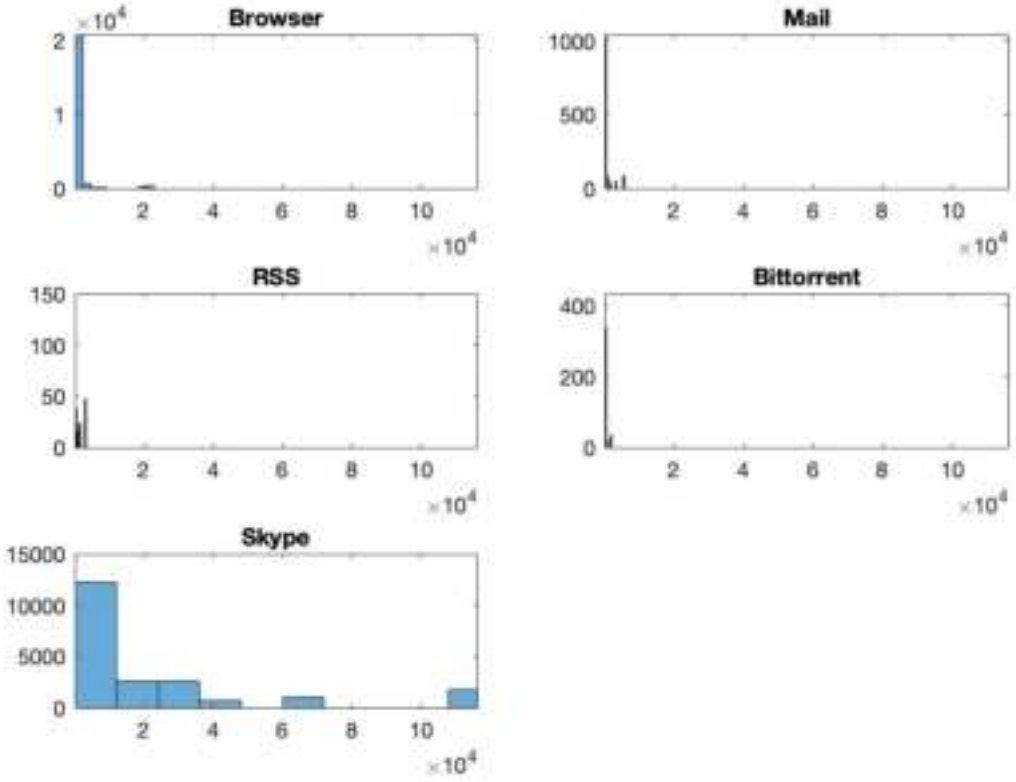

UNIBS - flowSizeFrameLength
Browser
Mail
RSS
Bittorrent
Skype

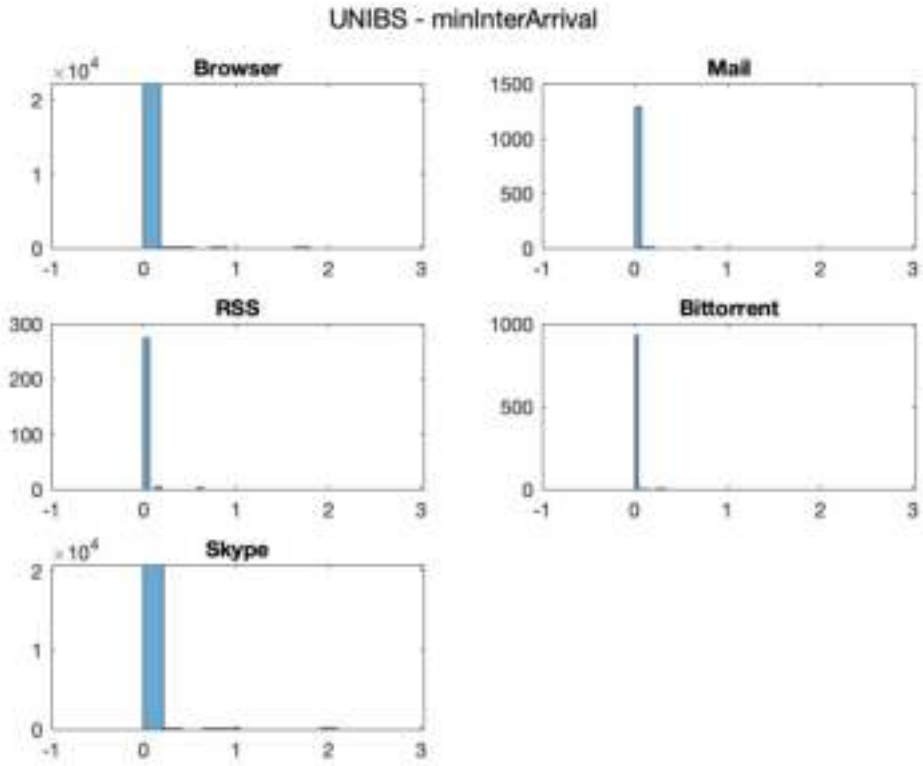

UNIBS - minCaptureLength
Browser
Mail
RSS
Bittorrent
Skype

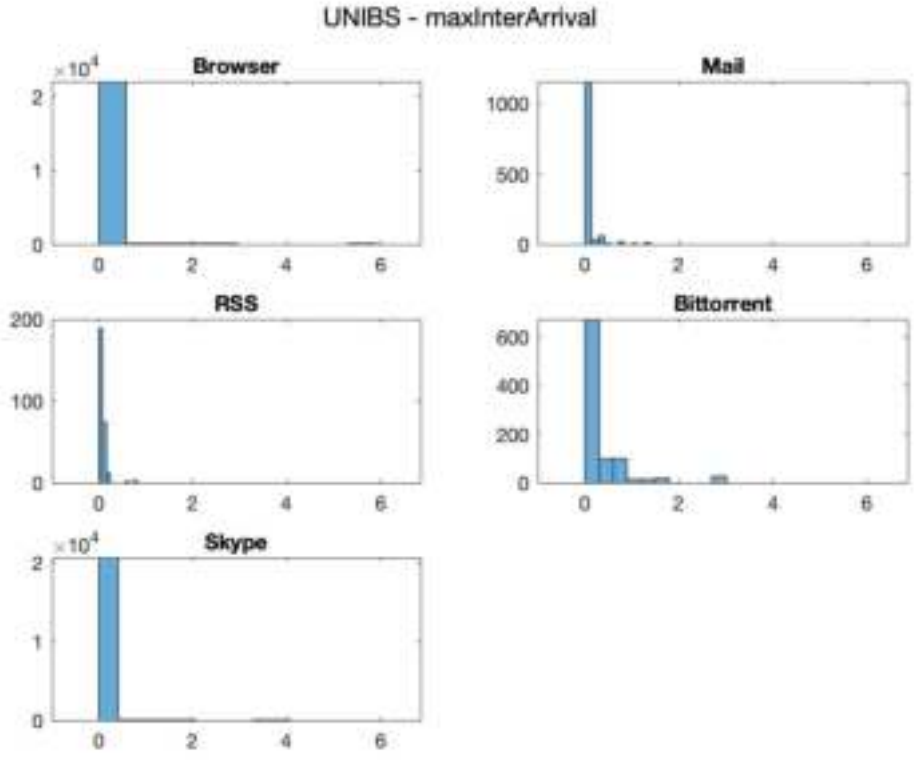

UNIBS - meanCaptureLength
Browser
Mail
RSS
Bittorrent
Skype

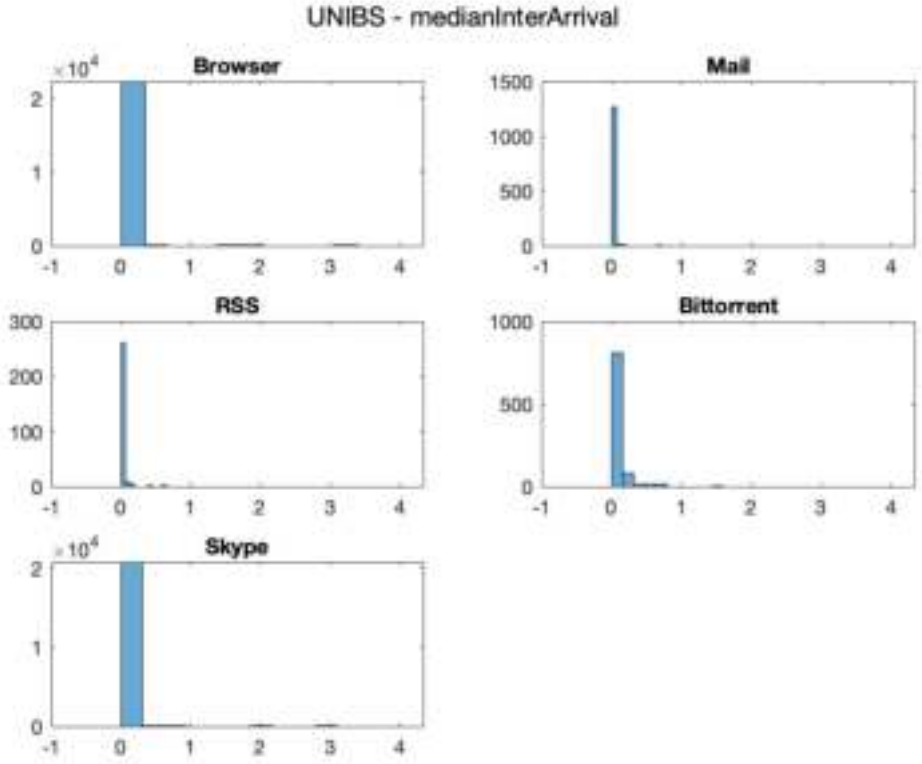

UNIBS - medianCaptureLength
Browser
Mail
RSS
Bittorrent
Skype

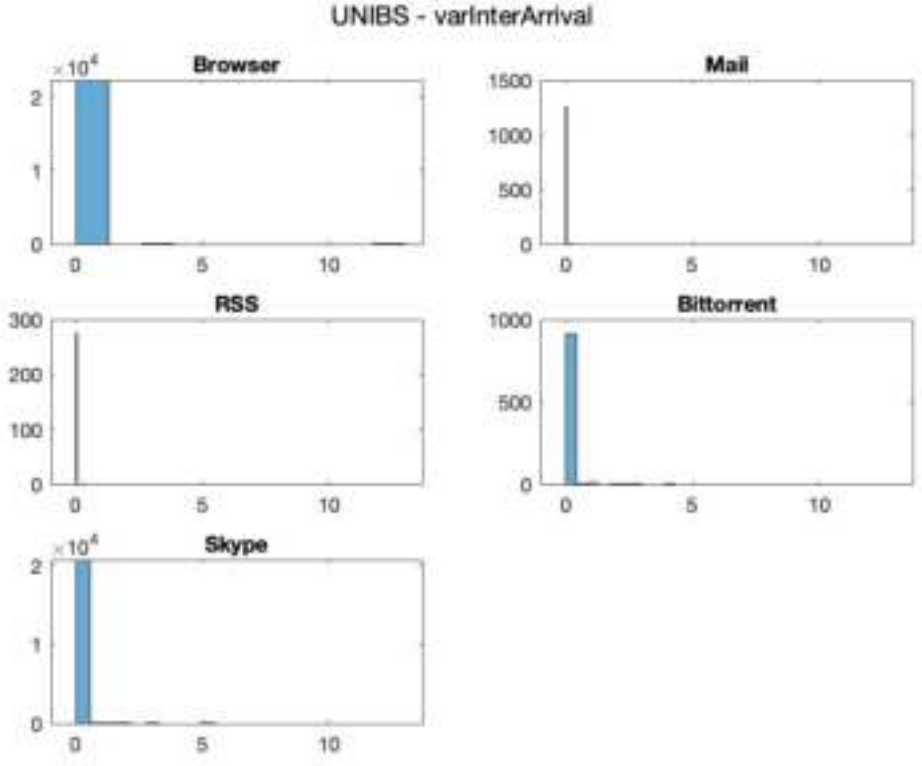

UNIBS - varCaptureLength
Browser
Mail
RSS
Bittorrent
Skype

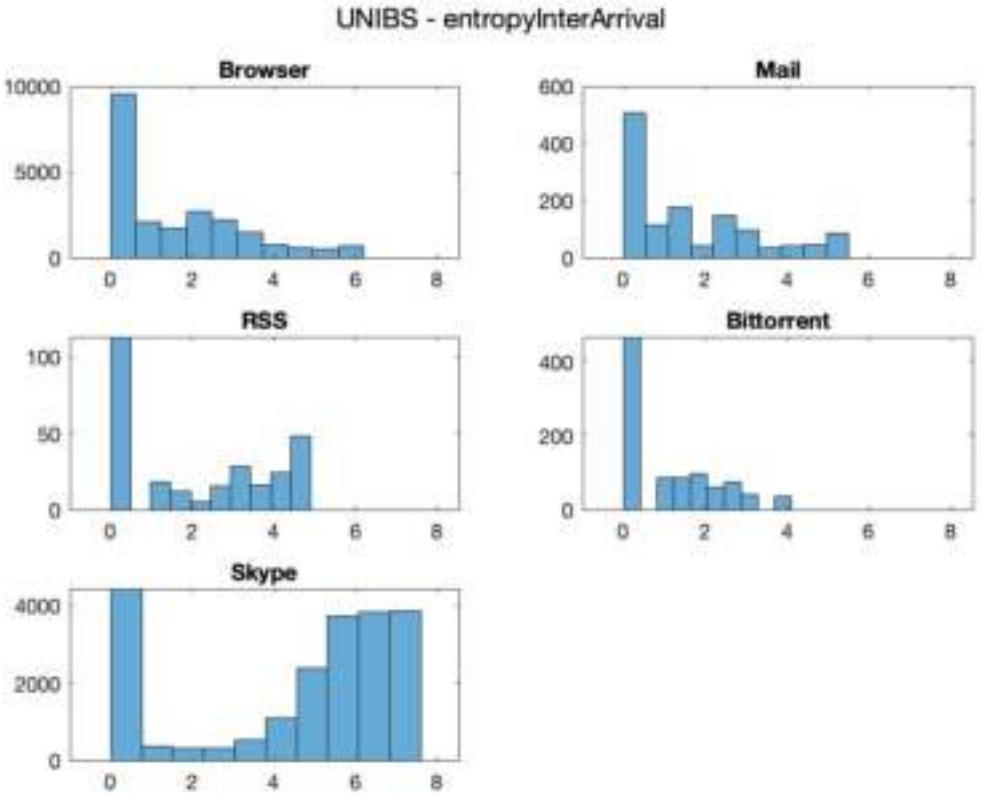

UNIBS - flowSizeCaptureLength
Browser
Mail
RSS
Bittorrent
Skype

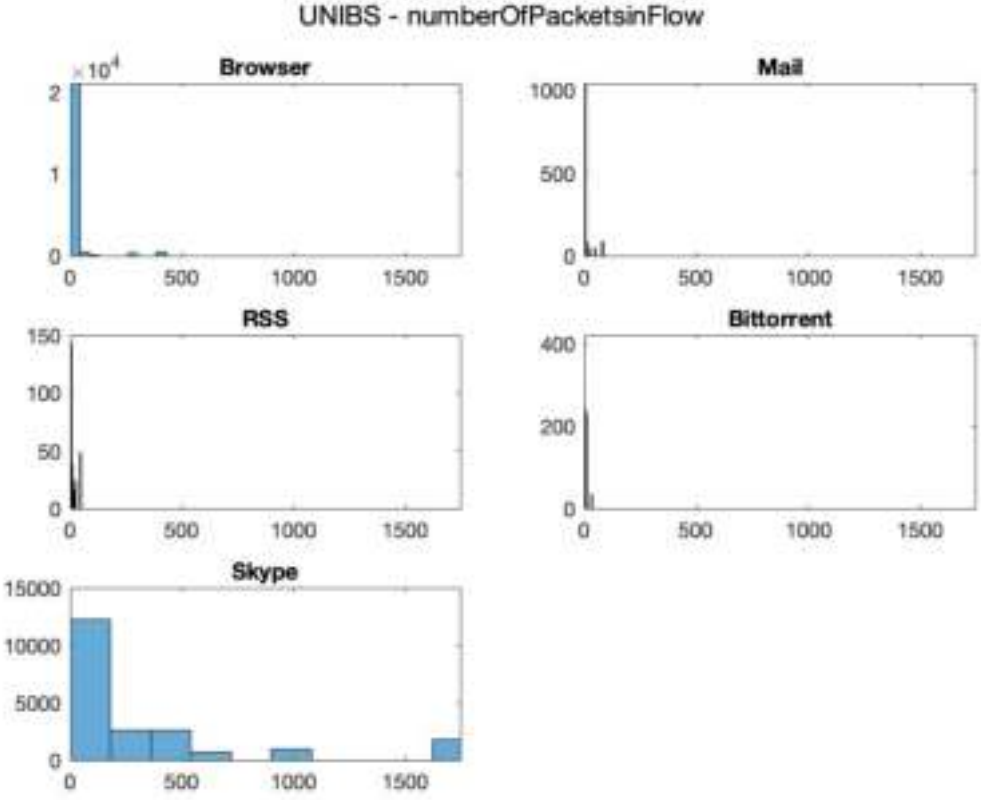

UNIBS - minInterArrival
Browser
Mail
RSS
Bittorrent
Skype

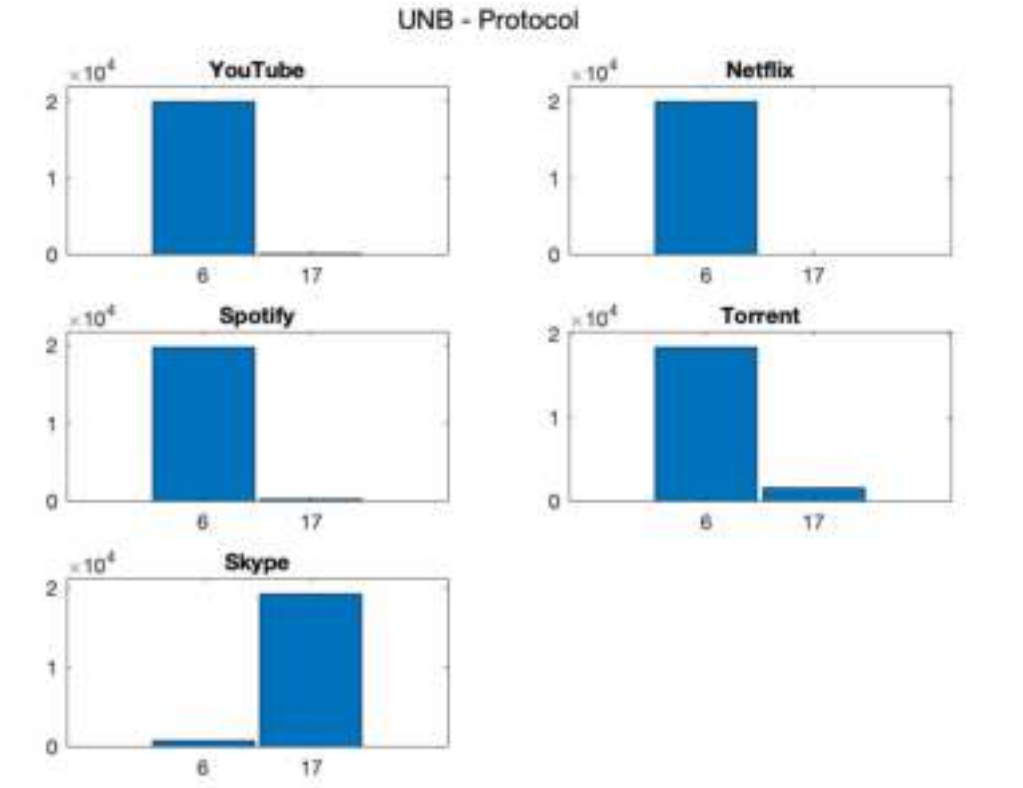

UNIBS - maxInterArrival
Browser
Mail
RSS
Bittorrent
Skype

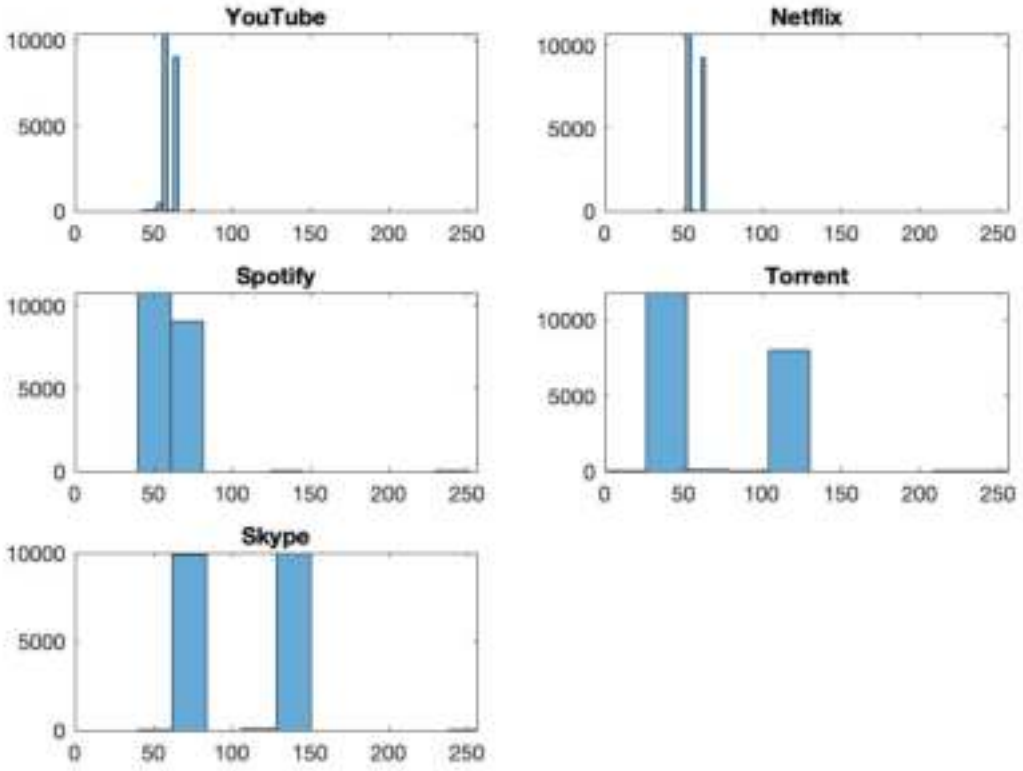

UNIBS - medianInterArrival
Browser
Mail
RSS
Bittorrent
Skype

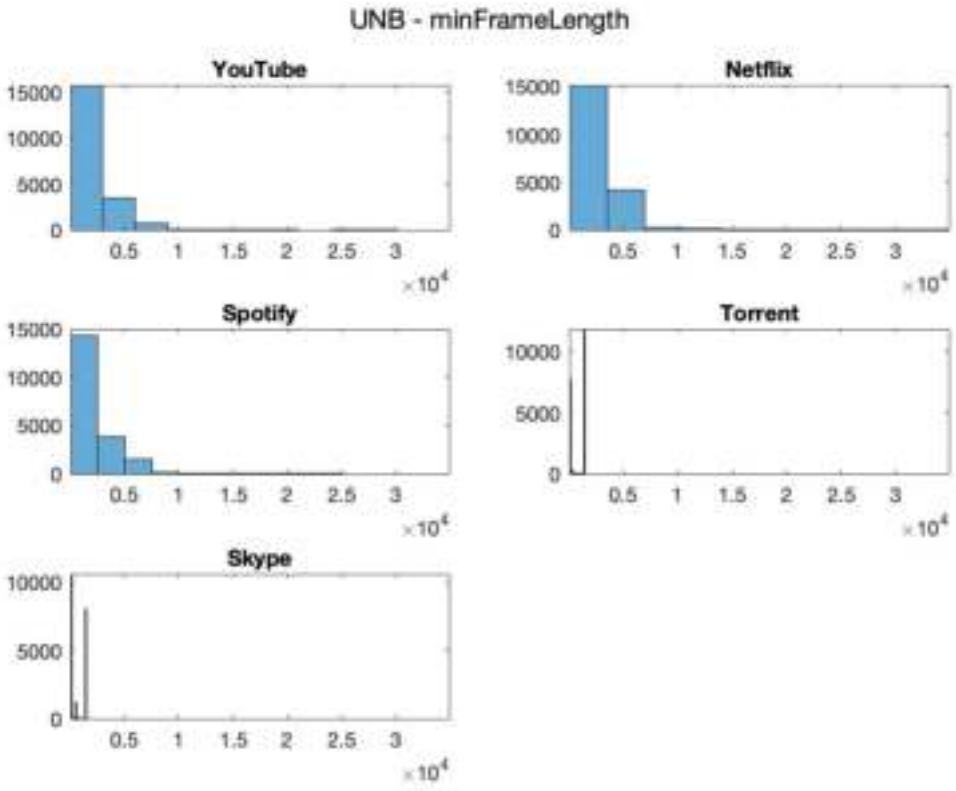

UNIBS - varInterArrival
Browser
Mail
RSS
Bittorrent
Skype

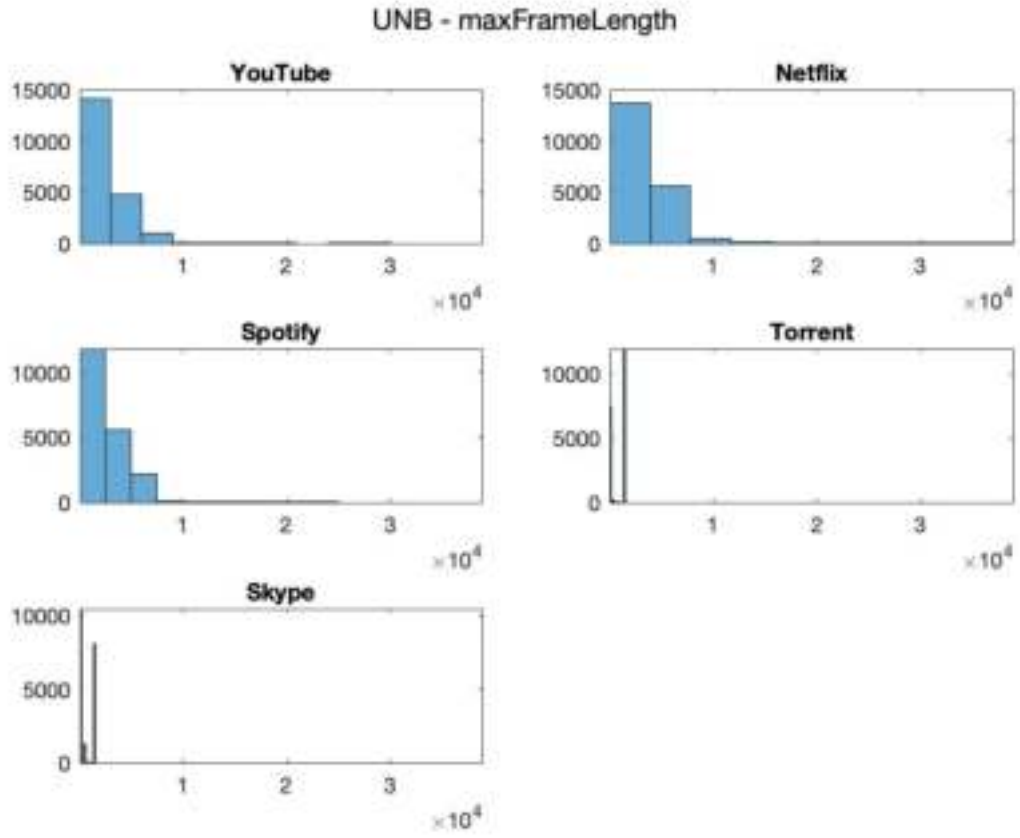

UNIBS - entropyInterArrival
Browser
Mail
RSS
Bittorrent
Skype

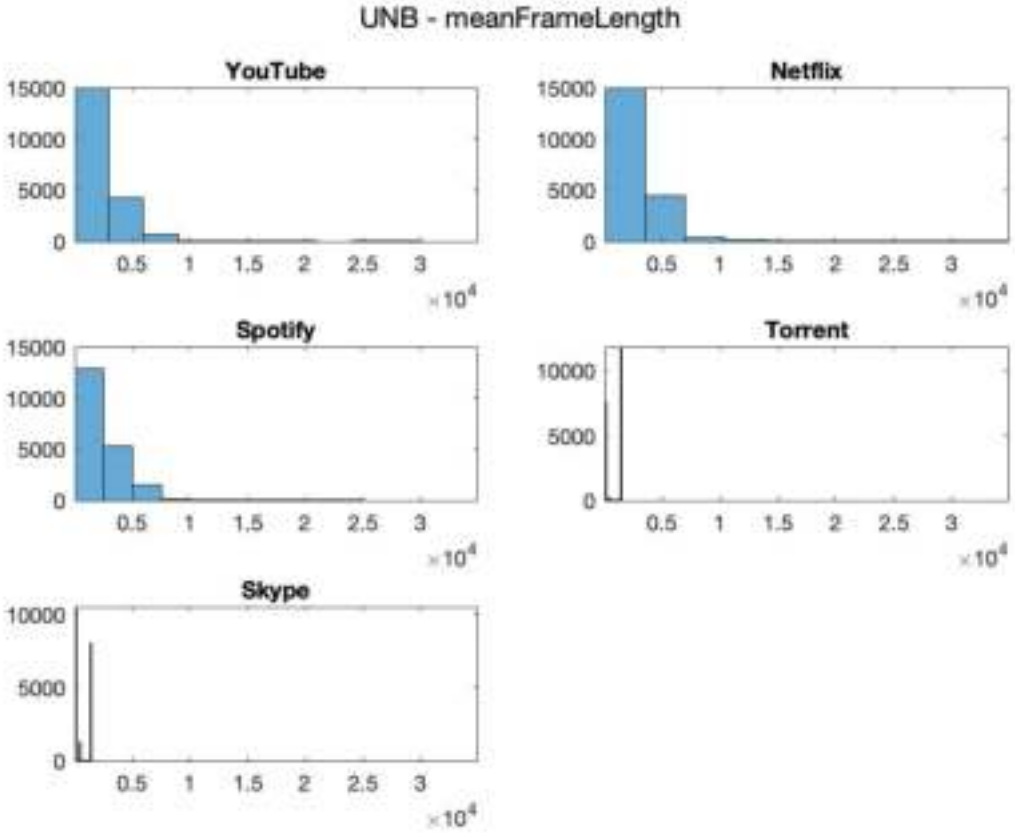

UNIBS - numberOfPacketsinFlow
Browser
Mail
RSS
Bittorrent
Skype

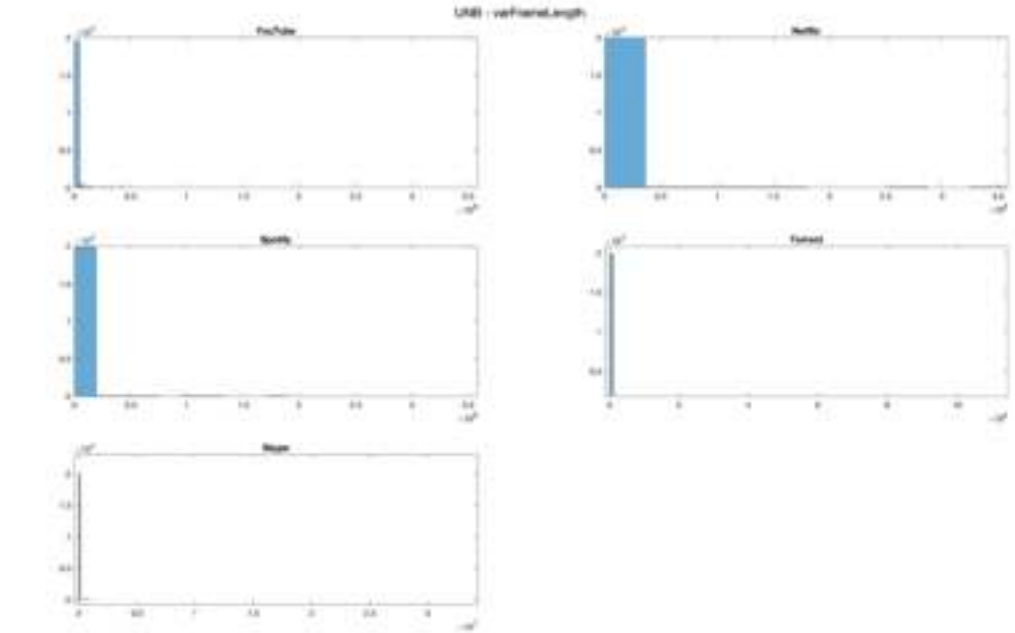

UNB - Protocol
YouTube
Netflix
Spotify
Torrent
Skype
×10^4
2
1
0
6
17

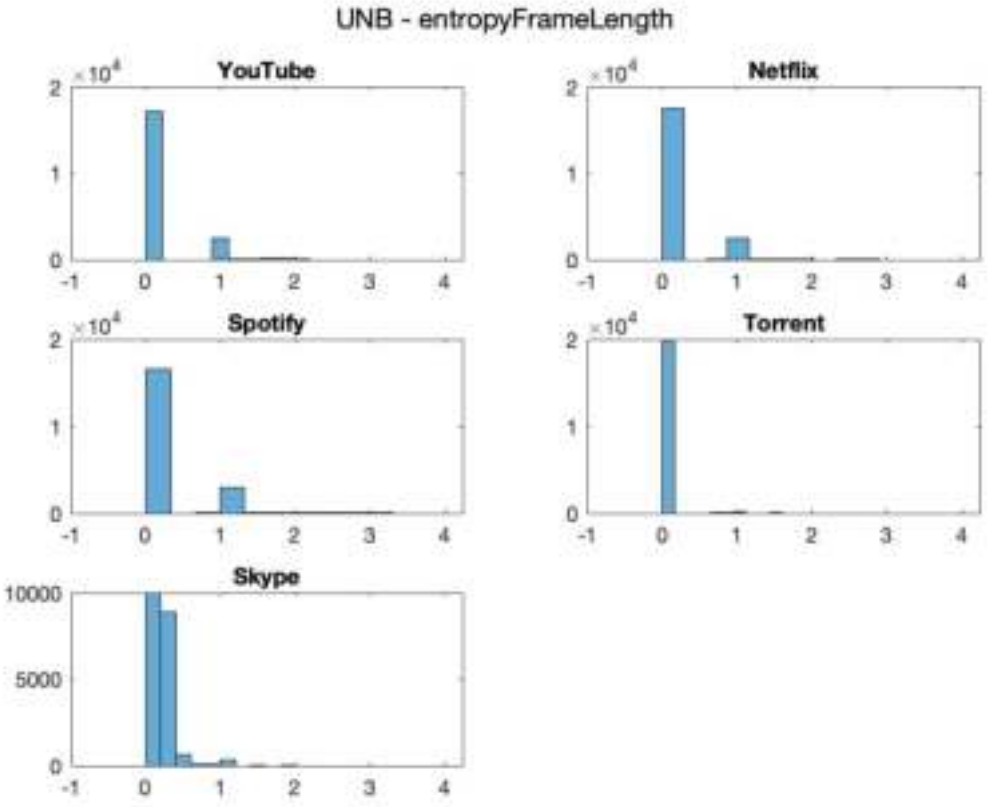

UNB - TTL
YouTube
Netflix
Spotify
Torrent
Skype
10000
5000
0
0
50
100
150
200
250

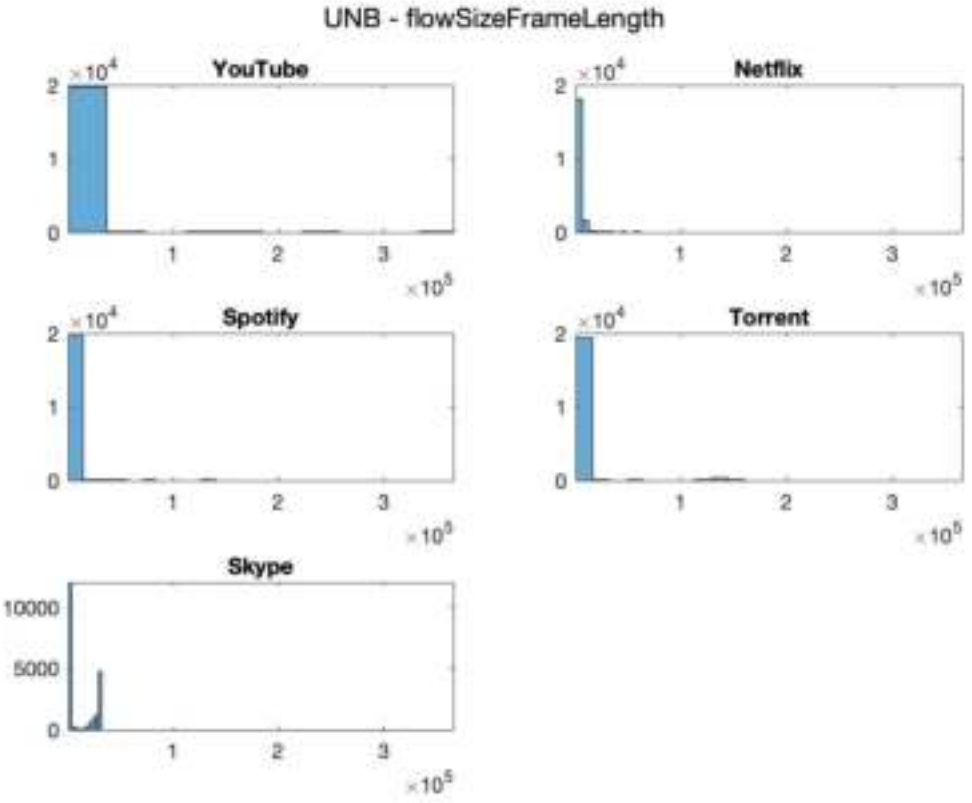

UNB - minFrameLength
YouTube
Netflix
Spotify
Torrent
Skype
15000
10000
5000
0
0.5
1
1.5
2
2.5
3
×10^4

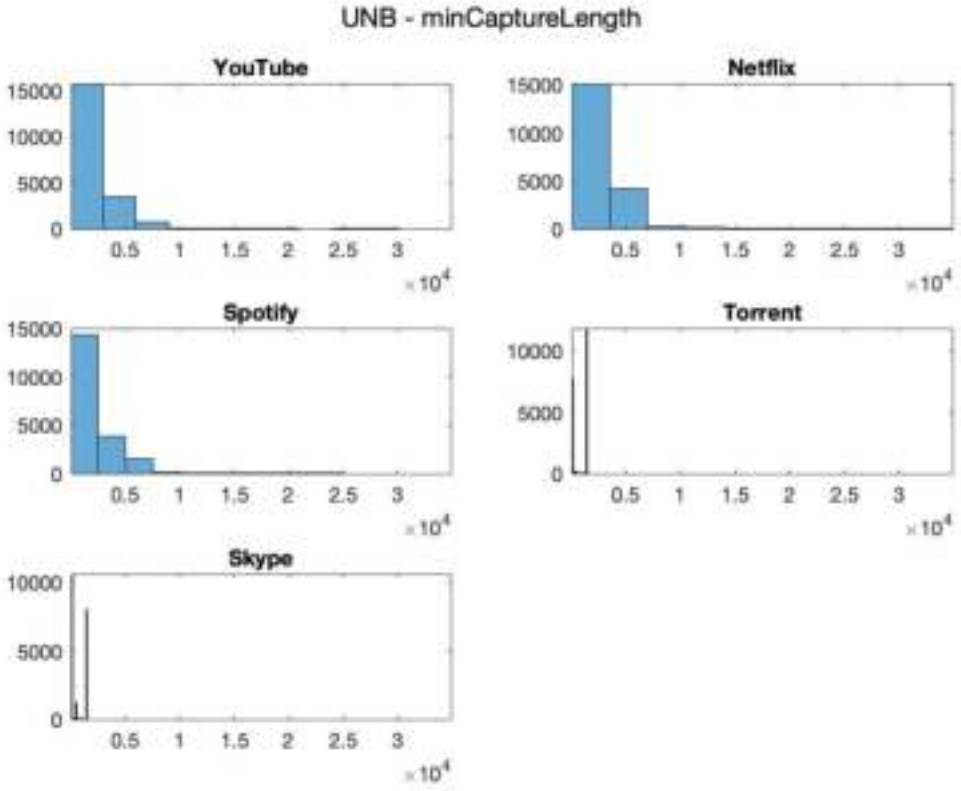

UNB - maxFrameLength
YouTube
Netflix
Spotify
Torrent
Skype
15000
10000
5000
0
1
2
3
×10^4

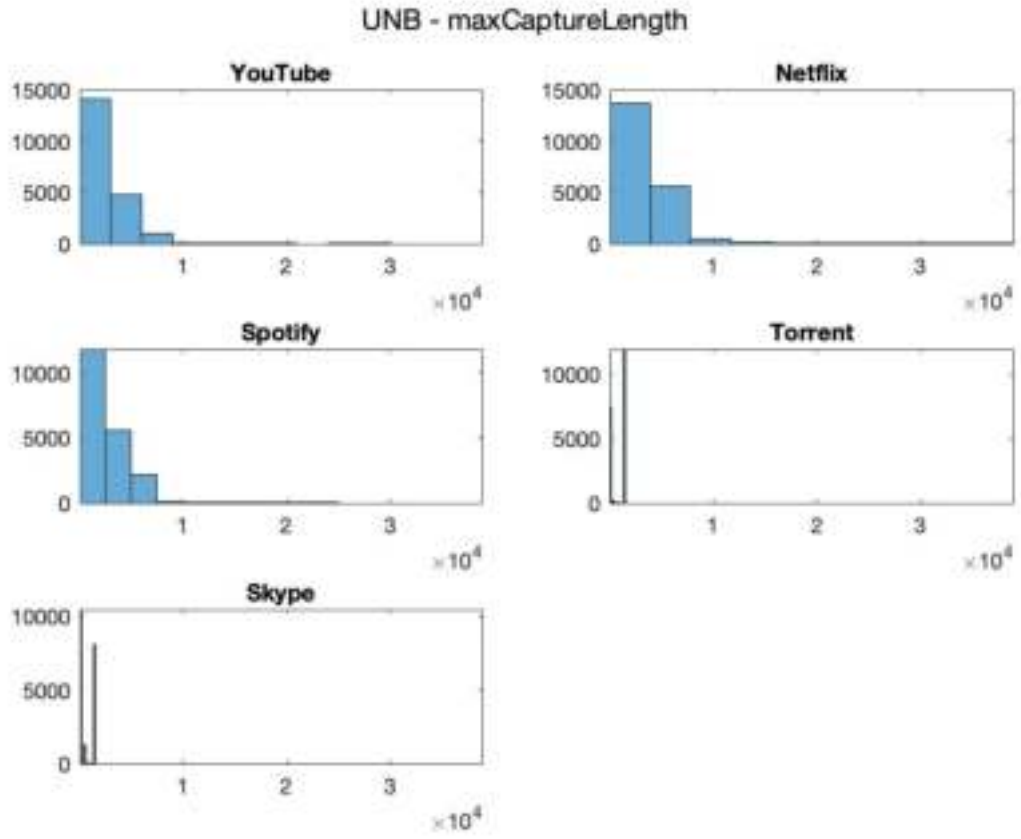

UNB - meanFrameLength
YouTube
Netflix
Spotify
Torrent
Skype
15000
10000
5000
0
0.5
1
1.5
2
2.5
3
×10^4

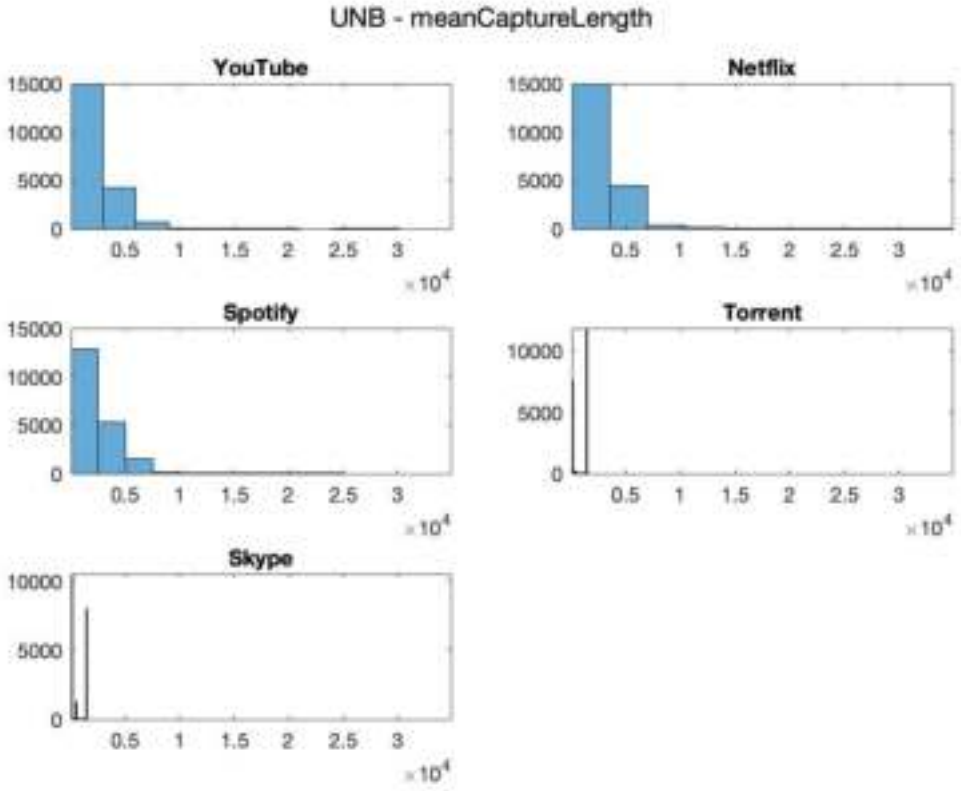


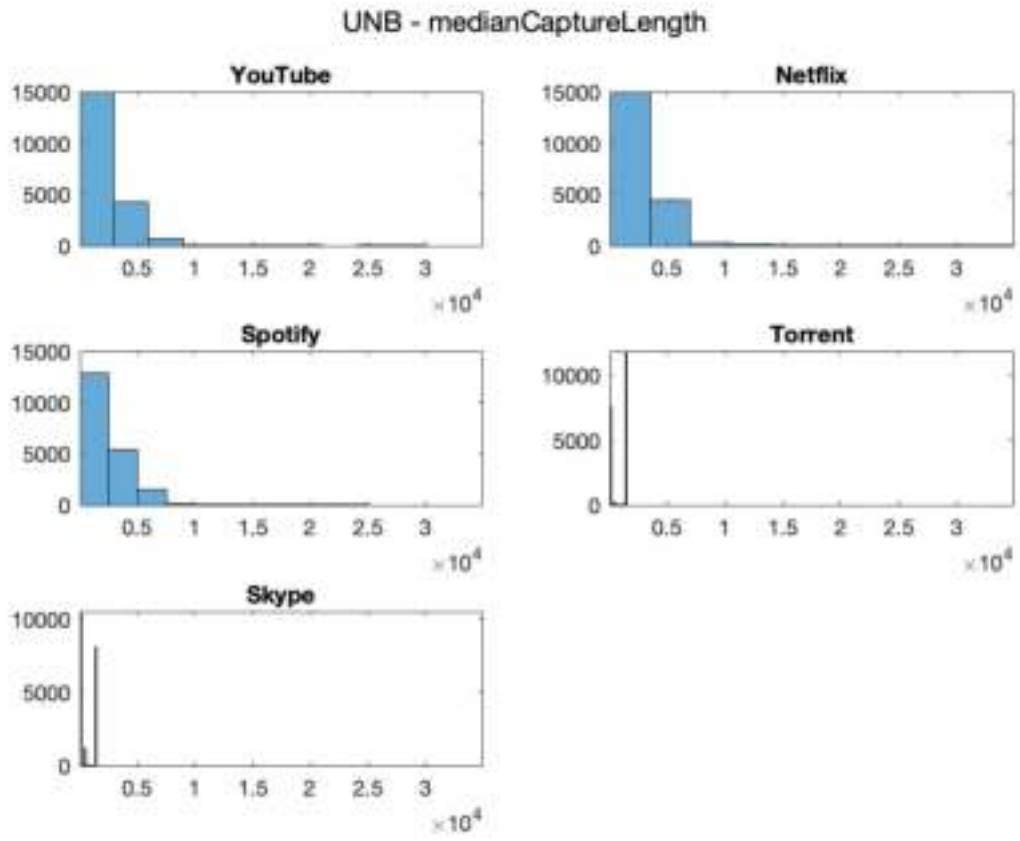

UNB - entropyFrameLength
YouTube
Netflix
Spotify
Torrent
Skype

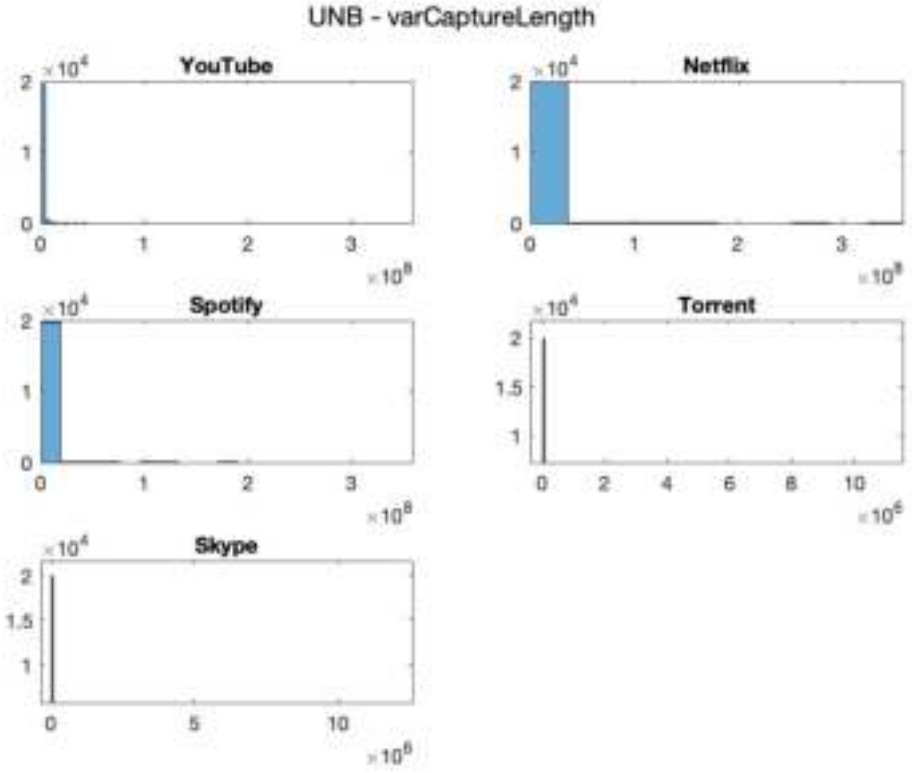

UNB - flowSizeFrameLength
YouTube
Netflix
Spotify
Torrent
Skype

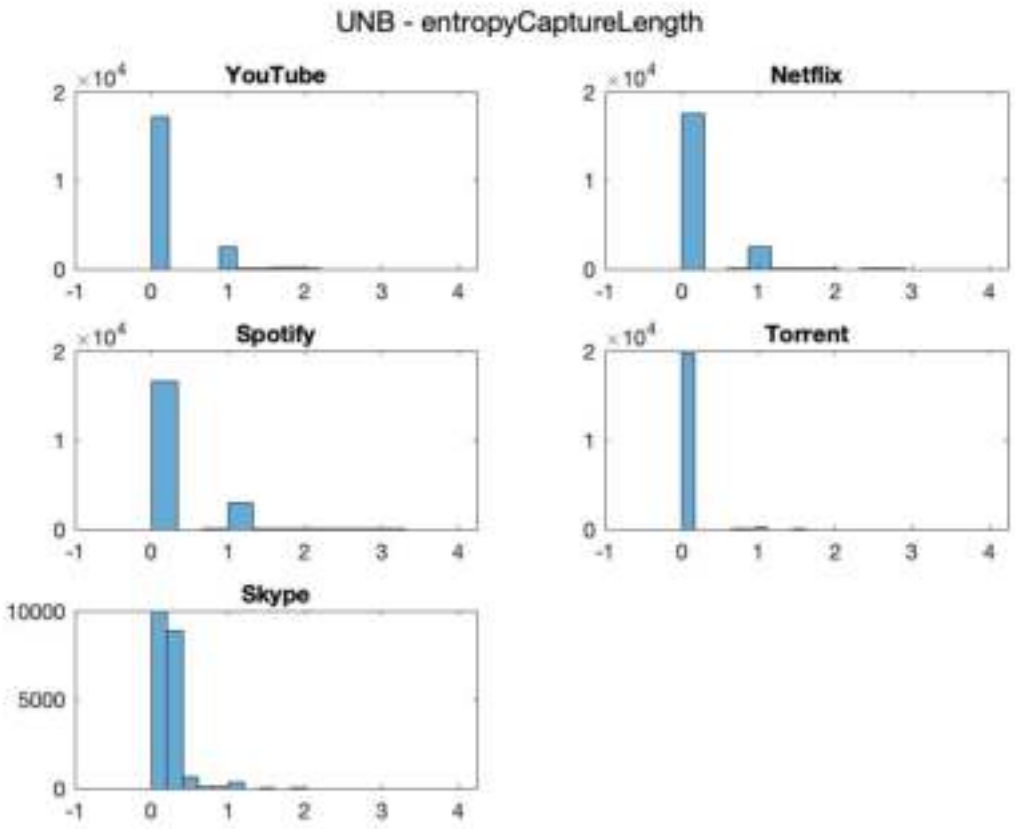

UNB - minCaptureLength
YouTube
Netflix
Spotify
Torrent
Skype
15000
10000
5000
0
0.5
1
1.5
2
2.5
3
×10^4

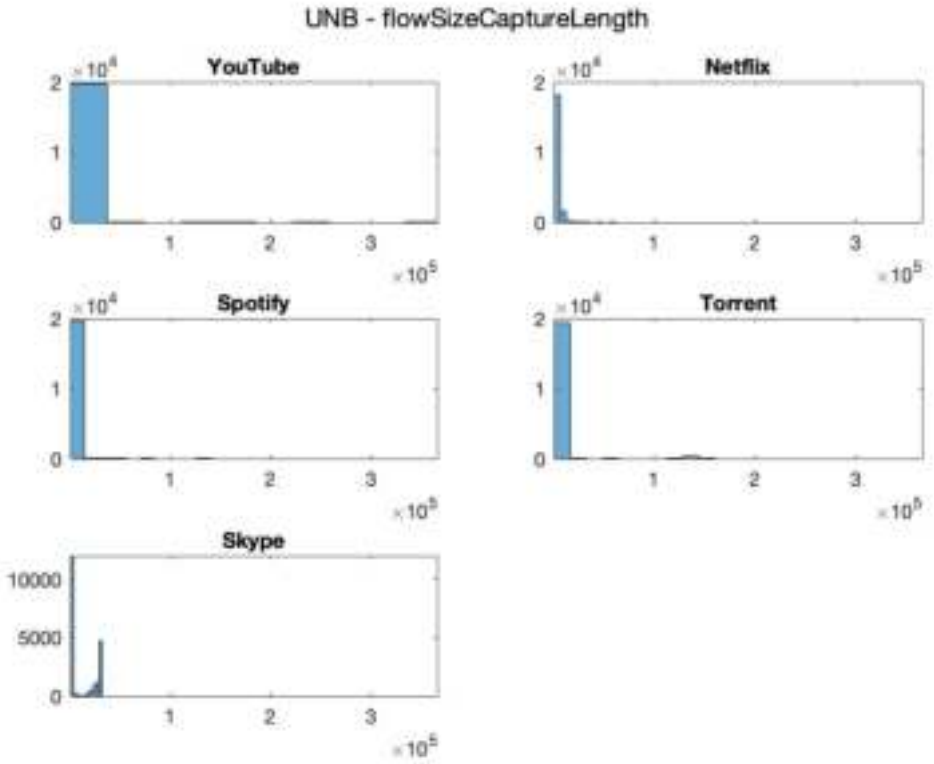

UNB - maxCaptureLength
YouTube
Netflix
Spotify
Torrent
Skype
15000
10000
5000
0
1
2
3
×10^4

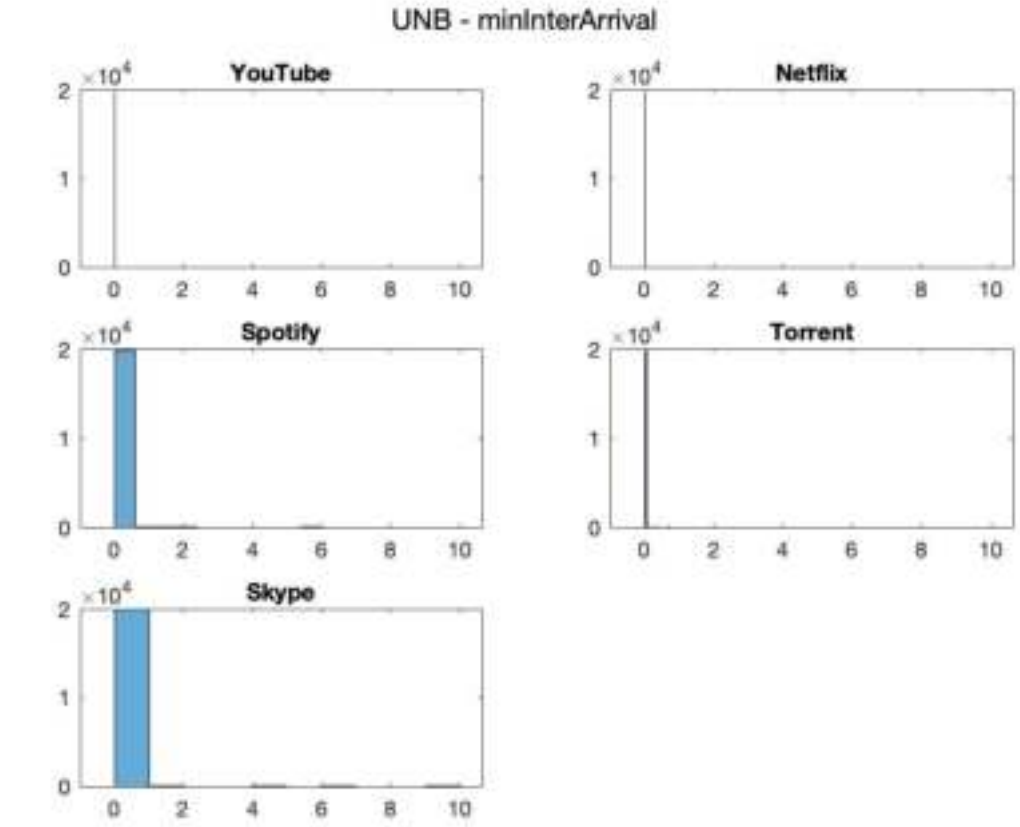

UNB - meanCaptureLength
YouTube
Netflix
Spotify
Torrent
Skype
15000
10000
5000
0
0.5
1
1.5
2
2.5
3
×10^4

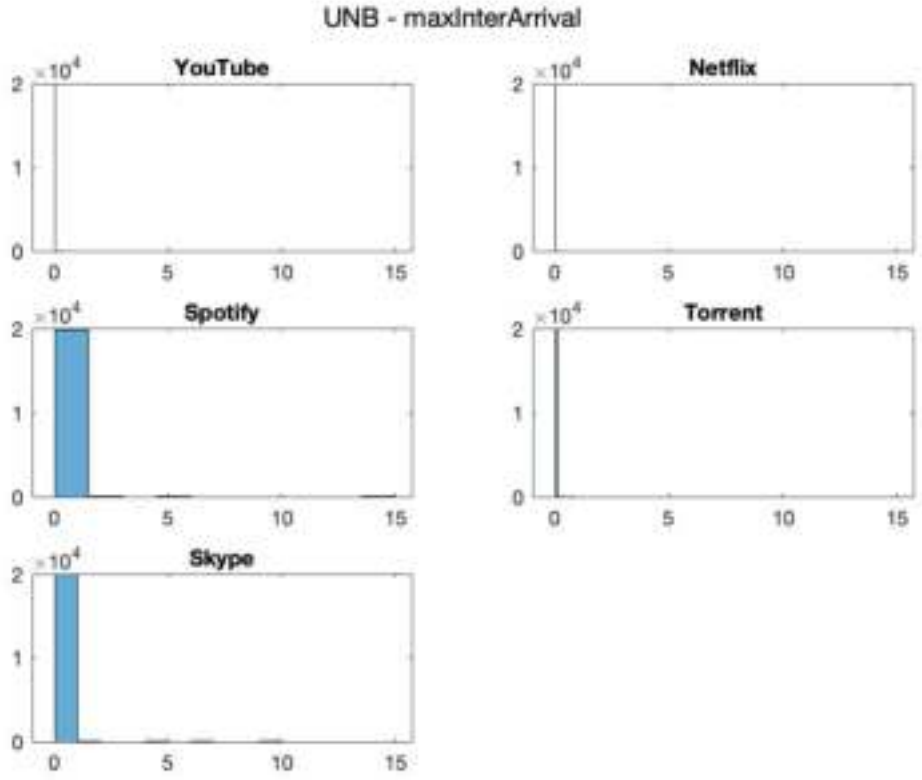

UNB - medianCaptureLength
YouTube
Netflix
Spotify
Torrent
Skype
15000
10000
5000
0
0.5
1
1.5
2
2.5
3
×10^4

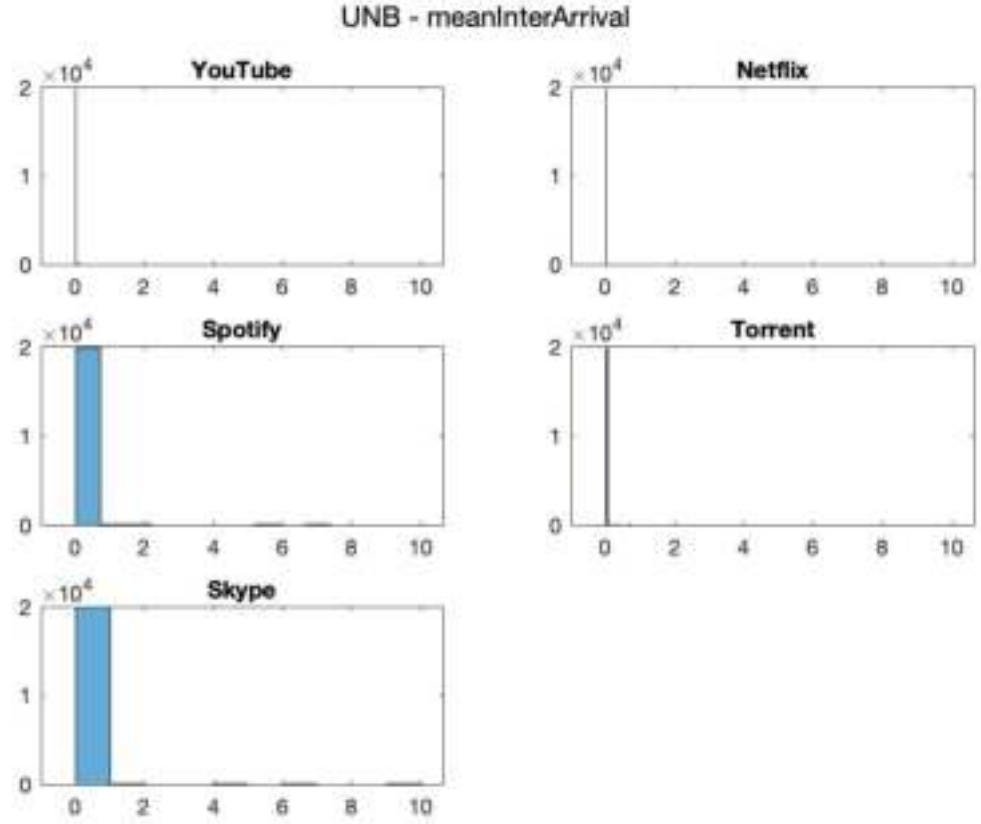

UNB - varCaptureLength
YouTube
Netflix
Spotify
Torrent
Skype

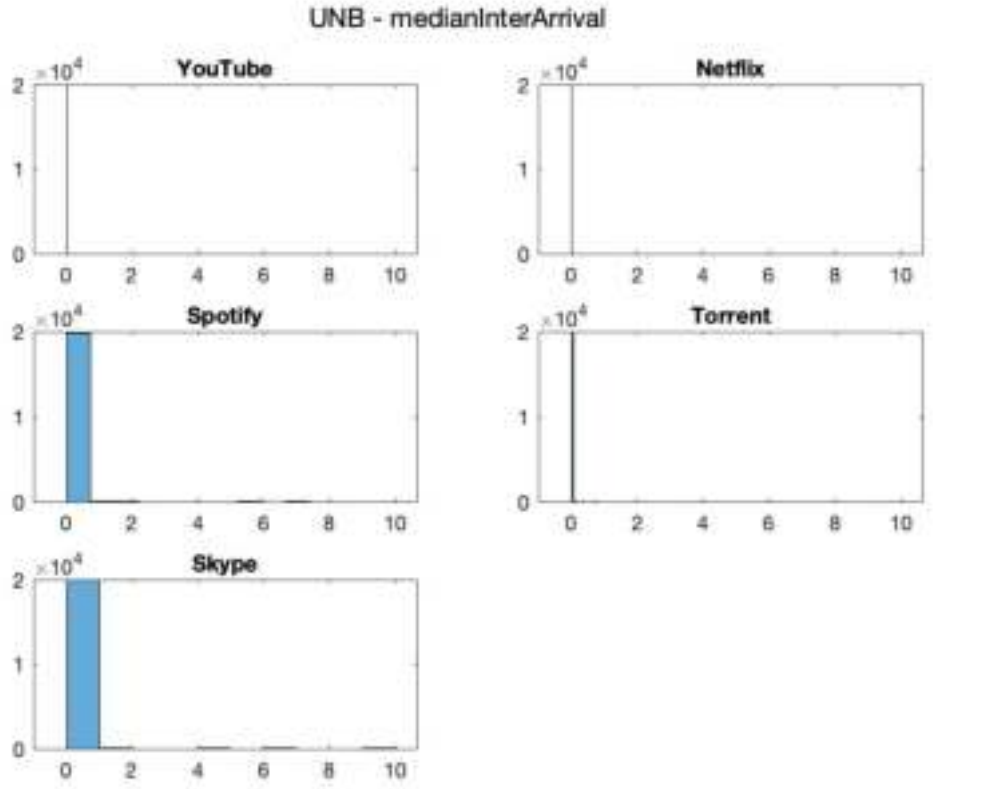

UNB - entropyCaptureLength
YouTube
Netflix
Spotify
Torrent
Skype

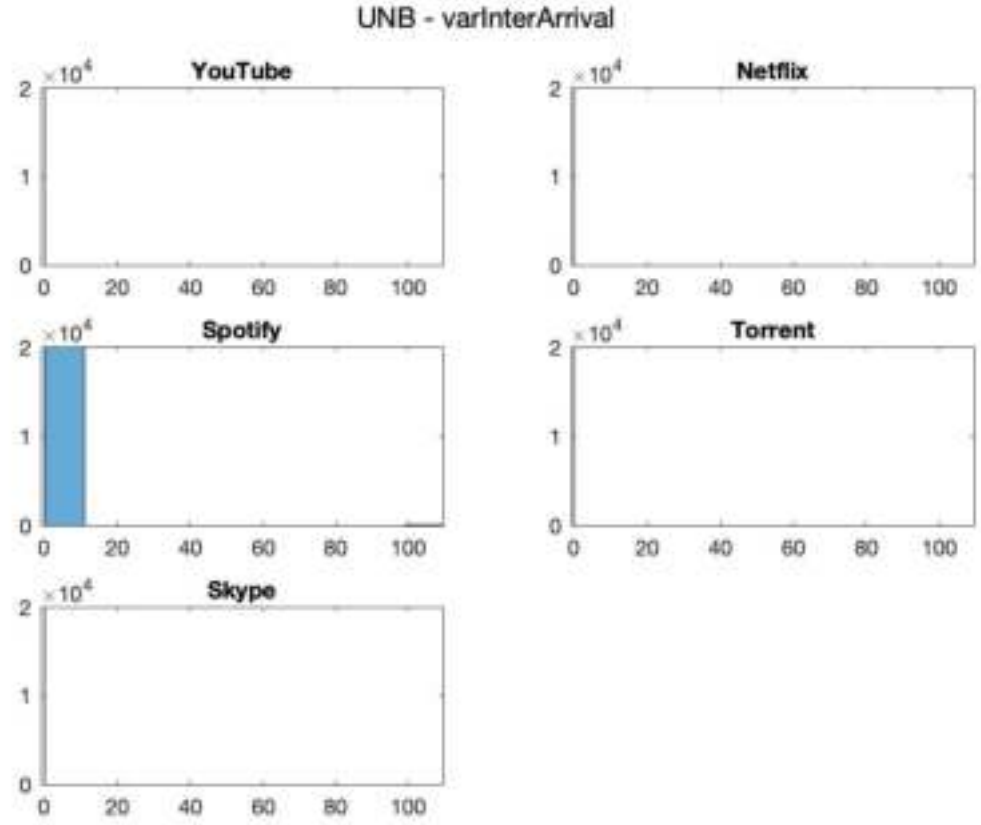

UNB - flowSizeCaptureLength
YouTube
Netflix
Spotify
Torrent
Skype

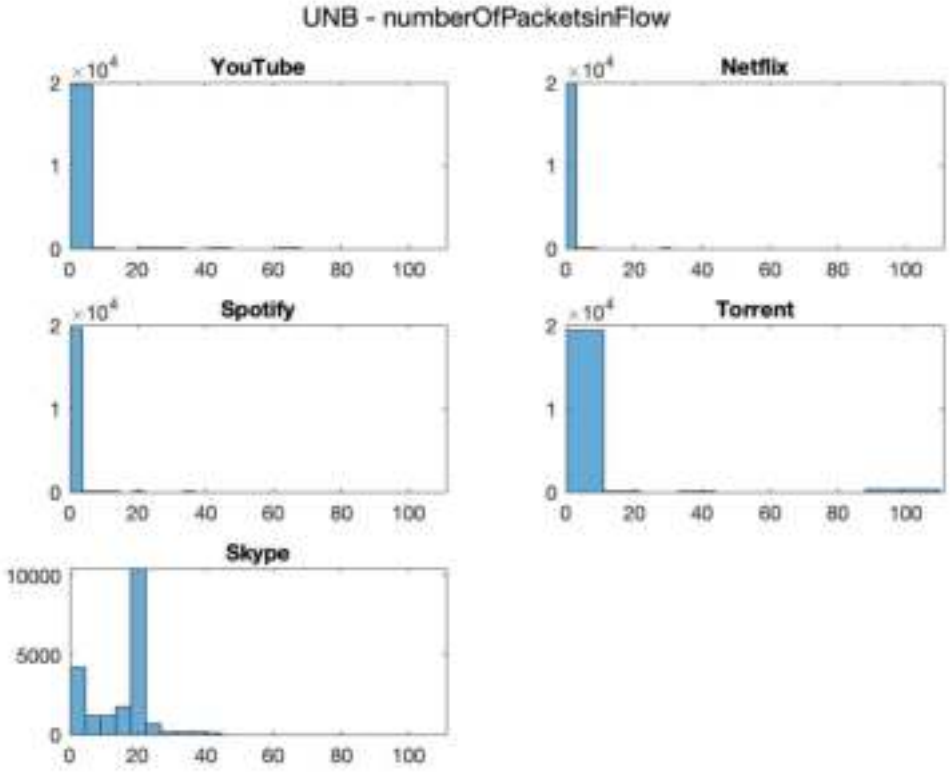

UNB - minInterArrival
YouTube
Netflix
Spotify
Torrent
Skype

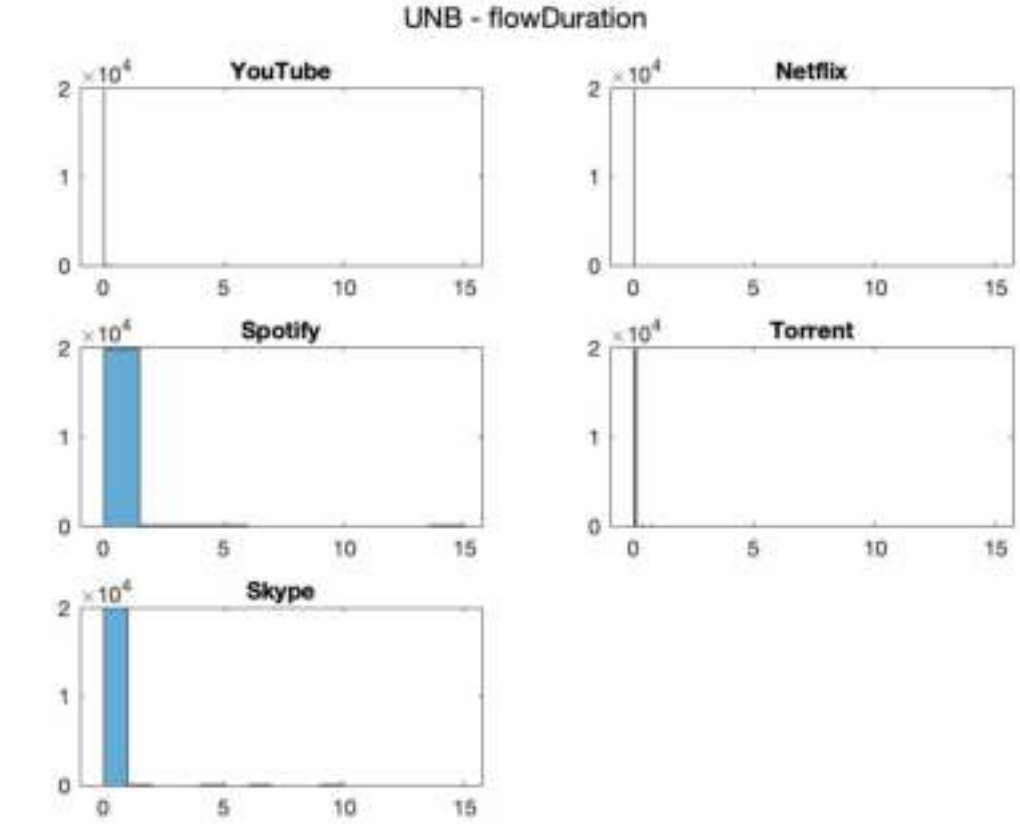

UNB - maxInterArrival
YouTube
Netflix
Spotify
Torrent
Skype
×10^4
0
1
2
0
5
10
15

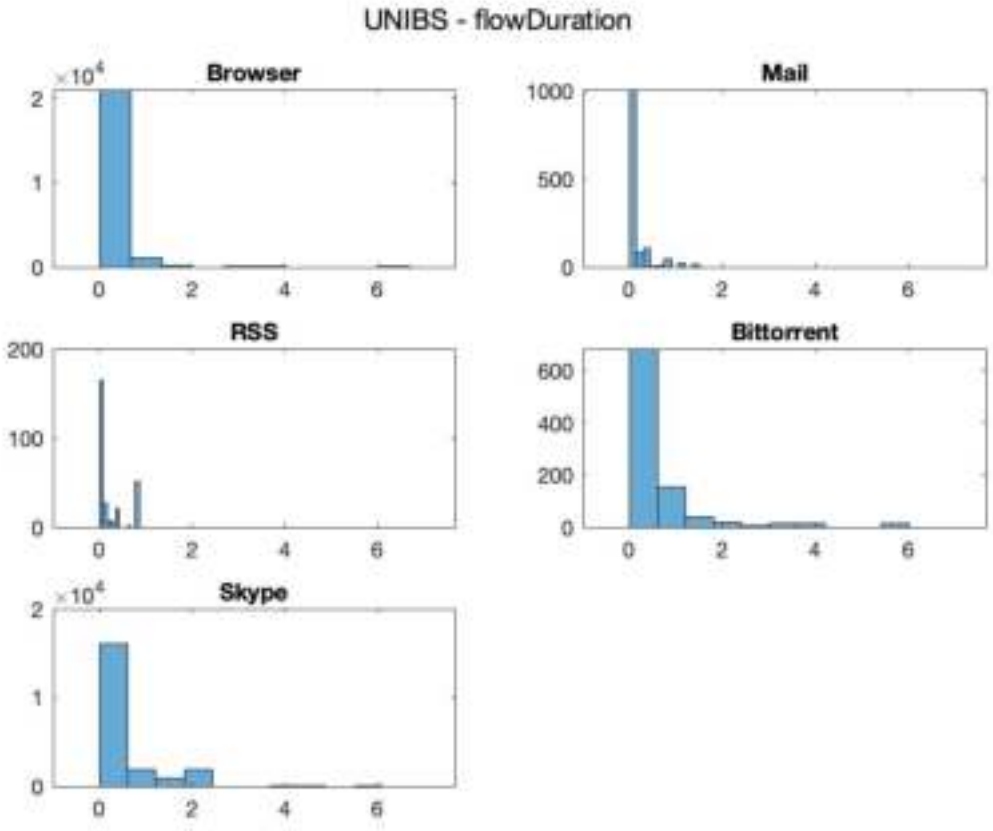

UNB - meanInterArrival
YouTube
Netflix
Spotify
Torrent
Skype
×10^4
0
1
2
0
2
4
6
8
10


UNB - medianInterArrival
YouTube
Netflix
Spotify
Torrent
Skype
×10^4
0
1
2
0
2
4
6
8
10

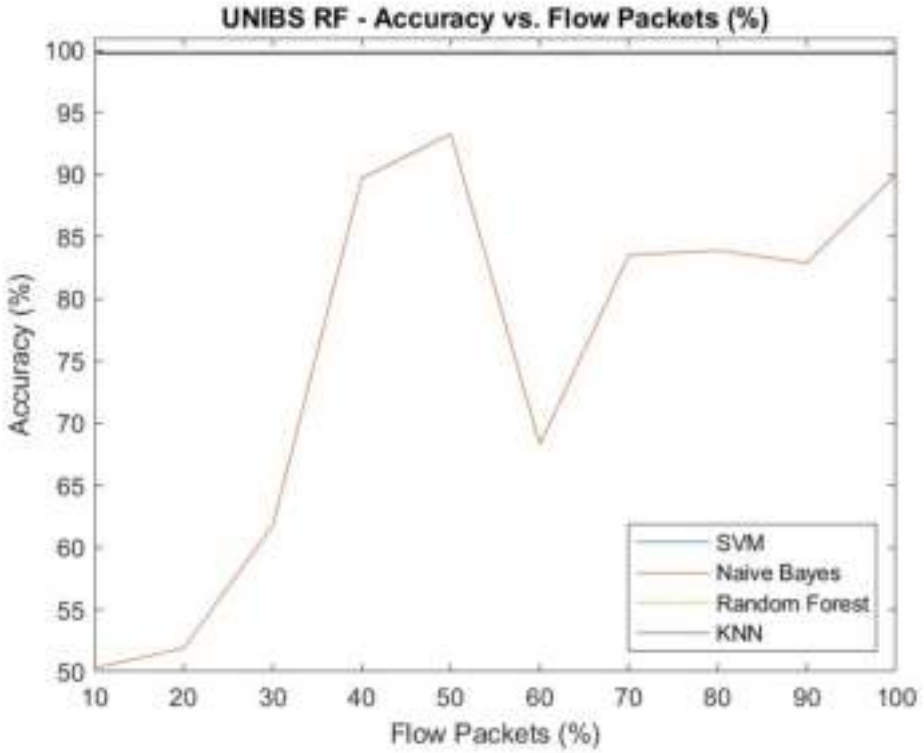

UNB - varInterArrival
YouTube
Netflix
Spotify
Torrent
Skype
×10^4
0
1
2
0
20
40
60
80
100

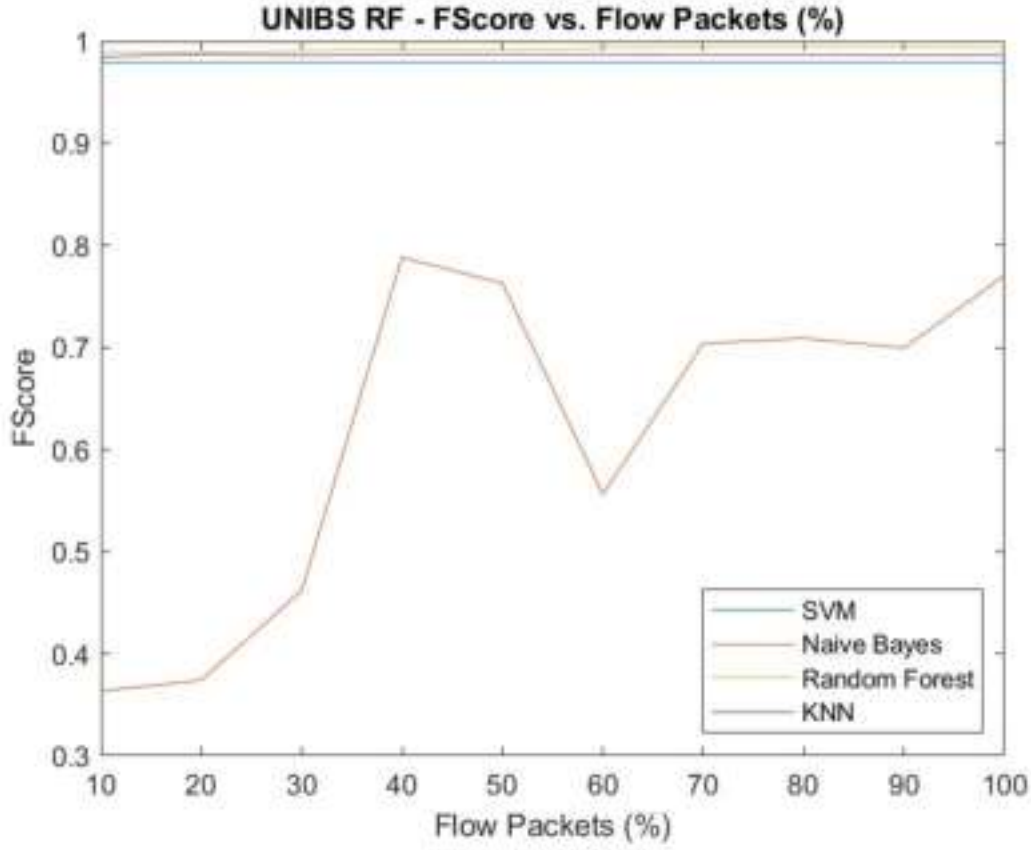

UNB - numberOfPacketsinFlow
YouTube
Netflix
Spotify
Torrent
Skype

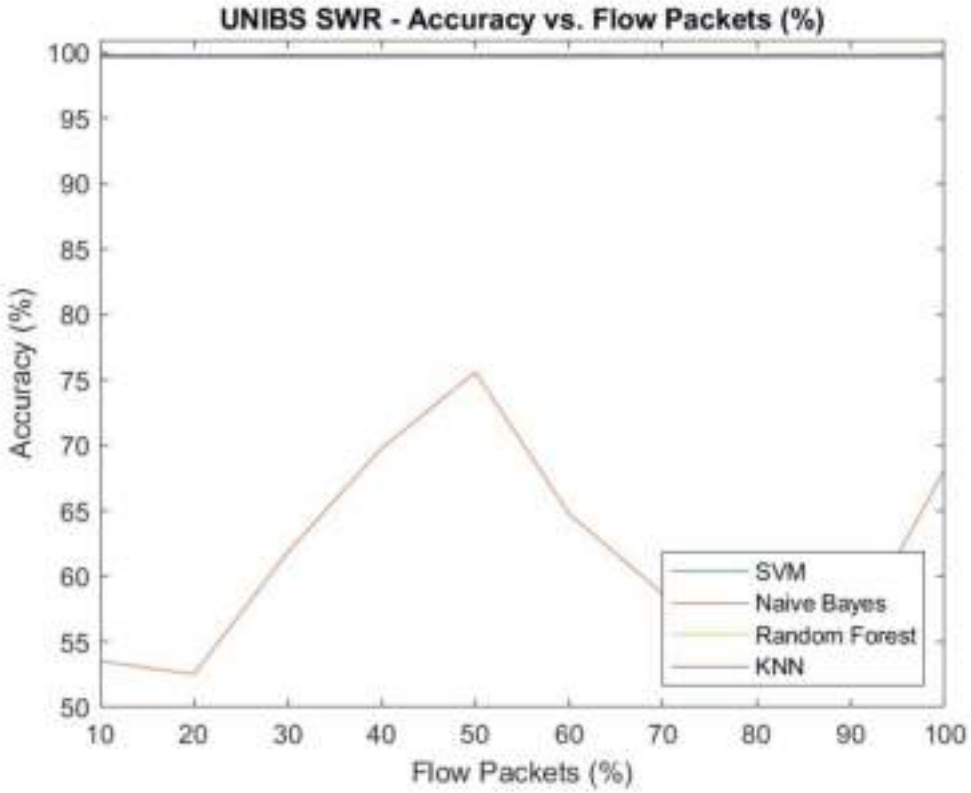

UNB - flowDuration
YouTube
Netflix
Spotify
Torrent
Skype

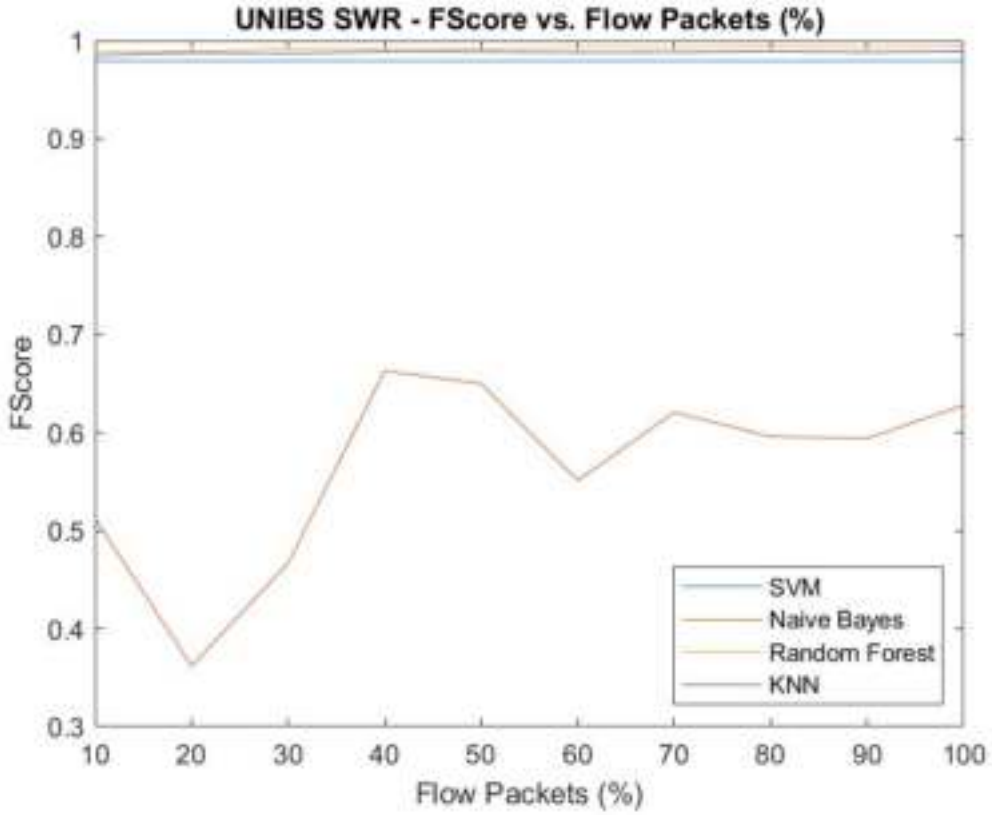

UNIBS - flowDuration
Browser
Mail
RSS
Bittorrent
Skype
×10^4

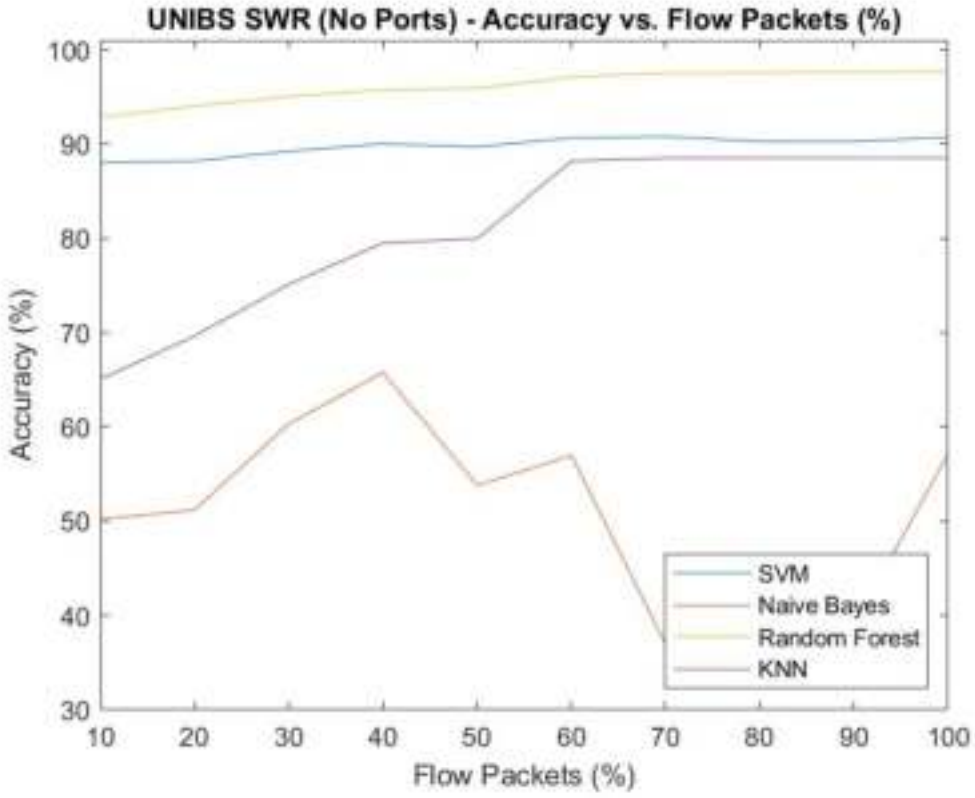
Appendix C – Various Packet Percentage Within a Flow

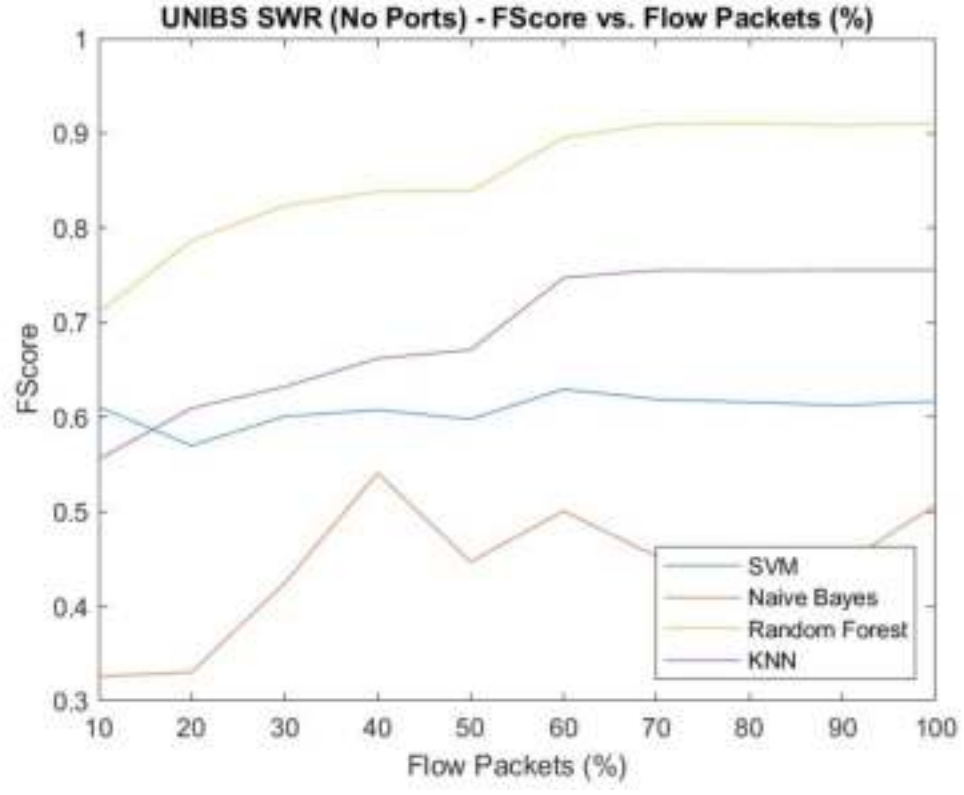


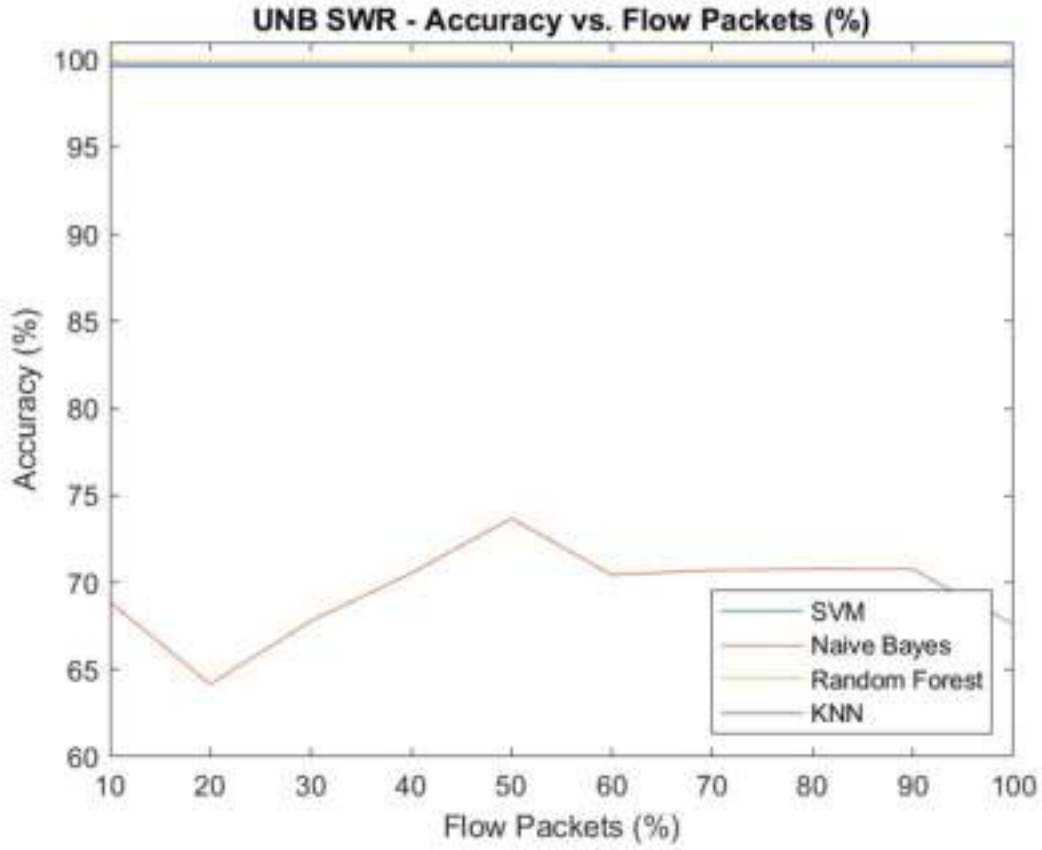


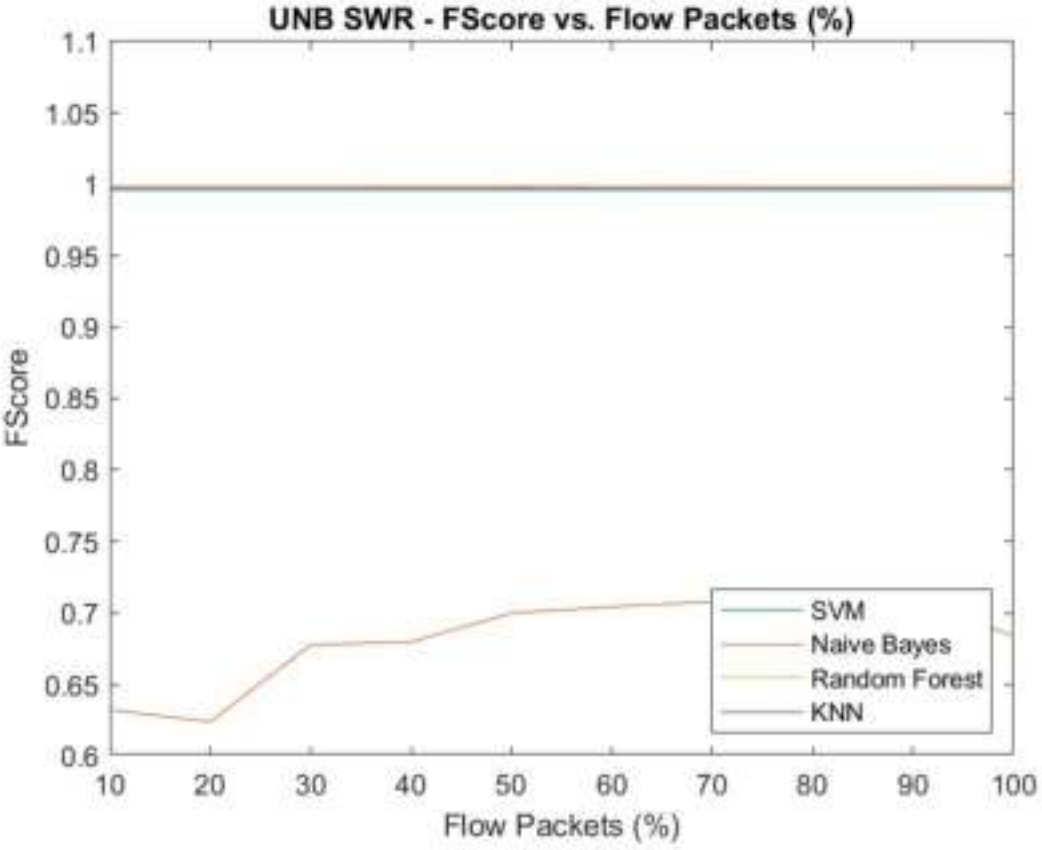

UNIBS SWR - Accuracy vs. Flow Packets (%)
Accuracy (%)
Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN

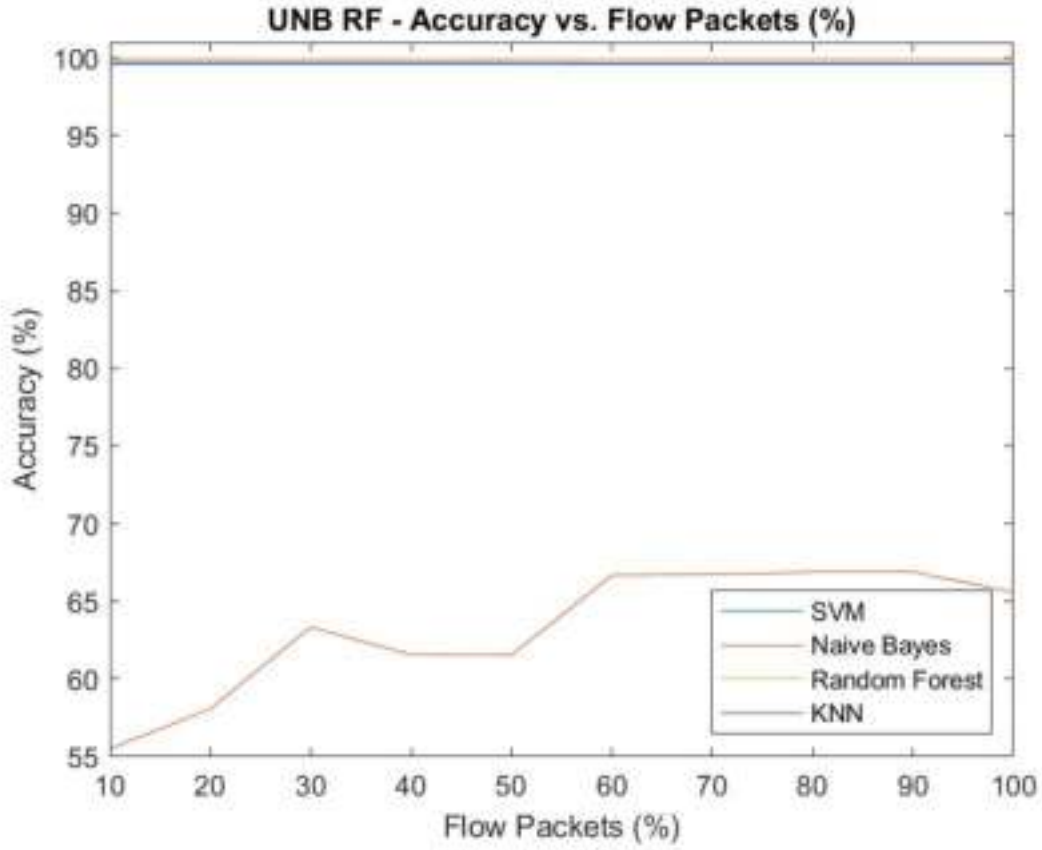

UNIBS SWR - FScore vs. Flow Packets (%)
FScore
Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN

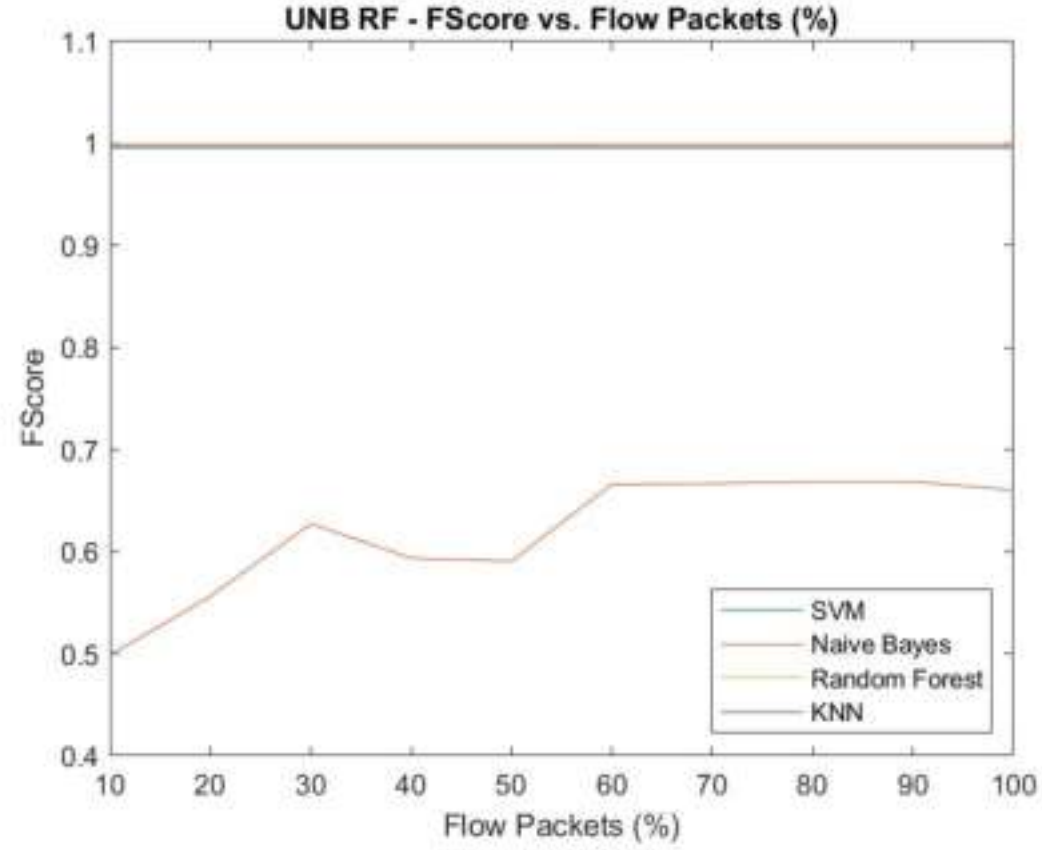

UNIBS SWR (No Ports) - Accuracy vs. Flow Packets (%)
Accuracy (%)
Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN

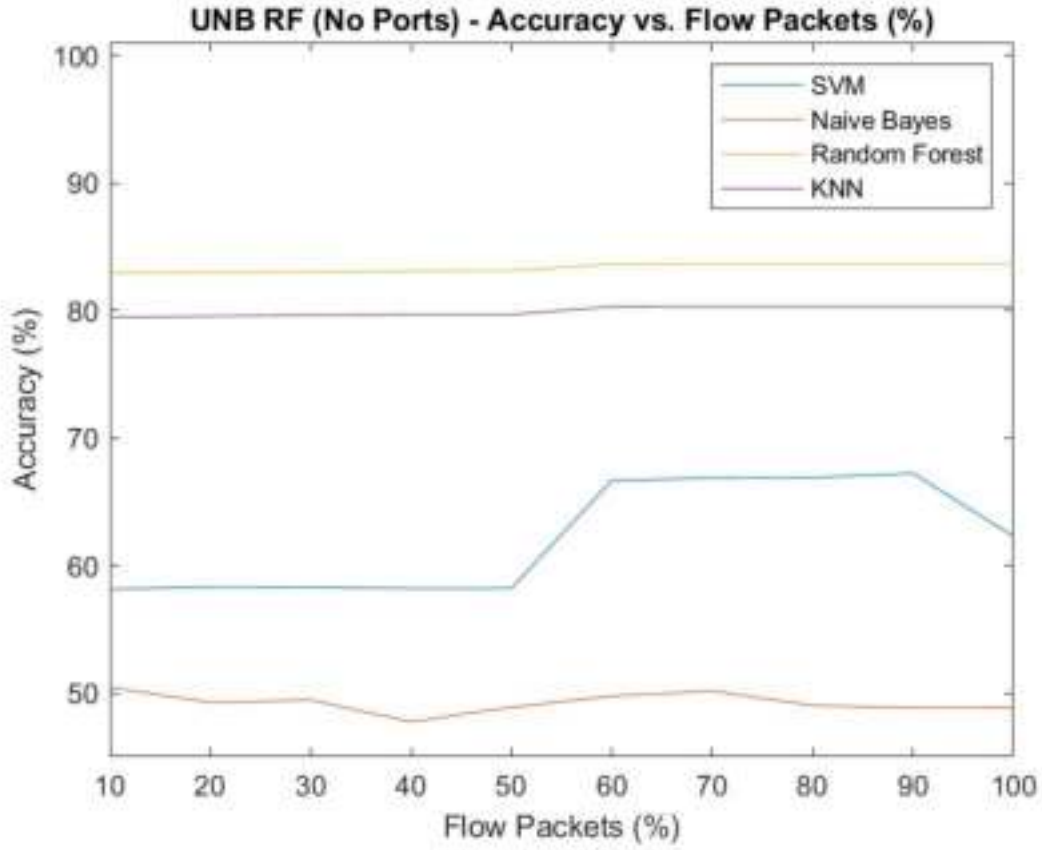

UNIBS SWR (No Ports) - FScore vs. Flow Packets (%)
FScore
Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN

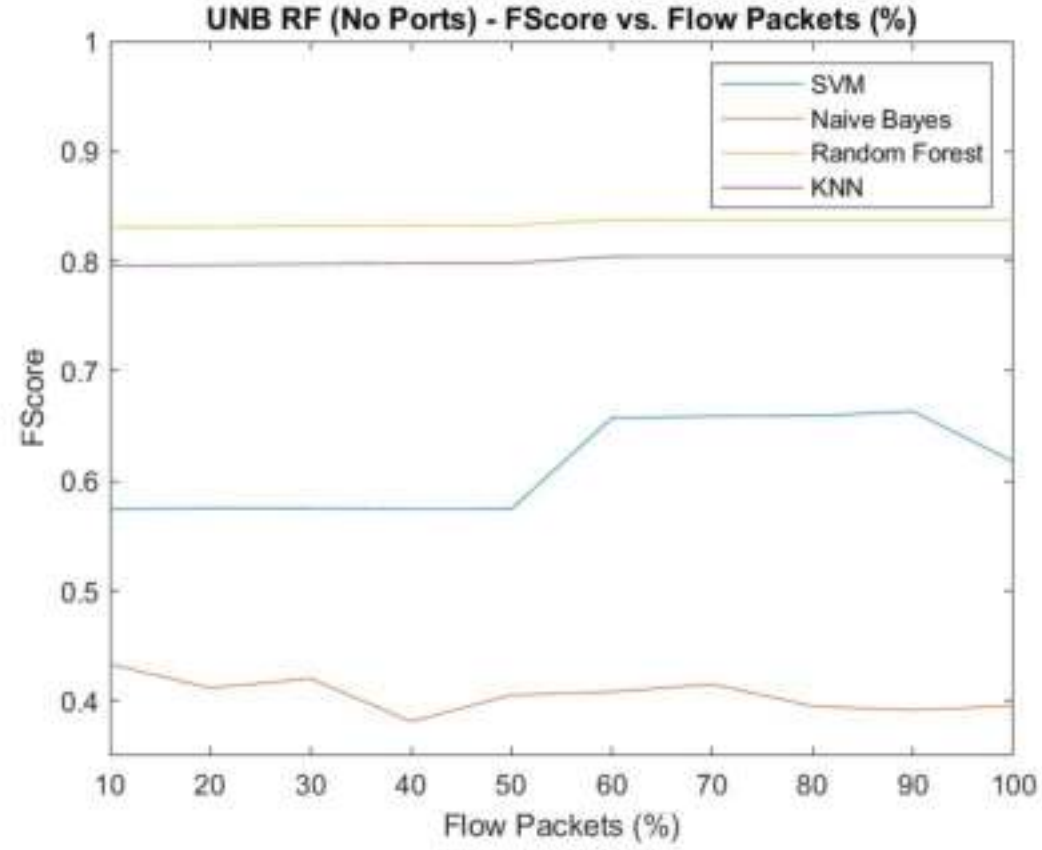

UNB SWR - Accuracy vs. Flow Packets (%)
Accuracy (%)
Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN


UNB SWR - FScore vs. Flow Packets (%)
FScore
Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN

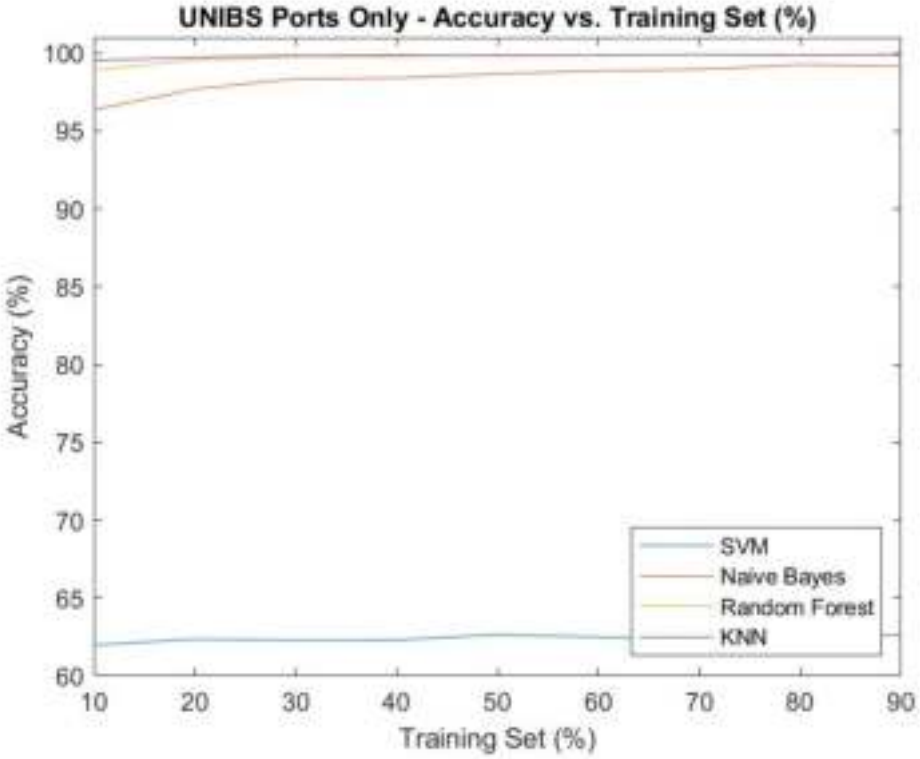

UNB RF - Accuracy vs. Flow Packets (%)
Accuracy (%)
Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN

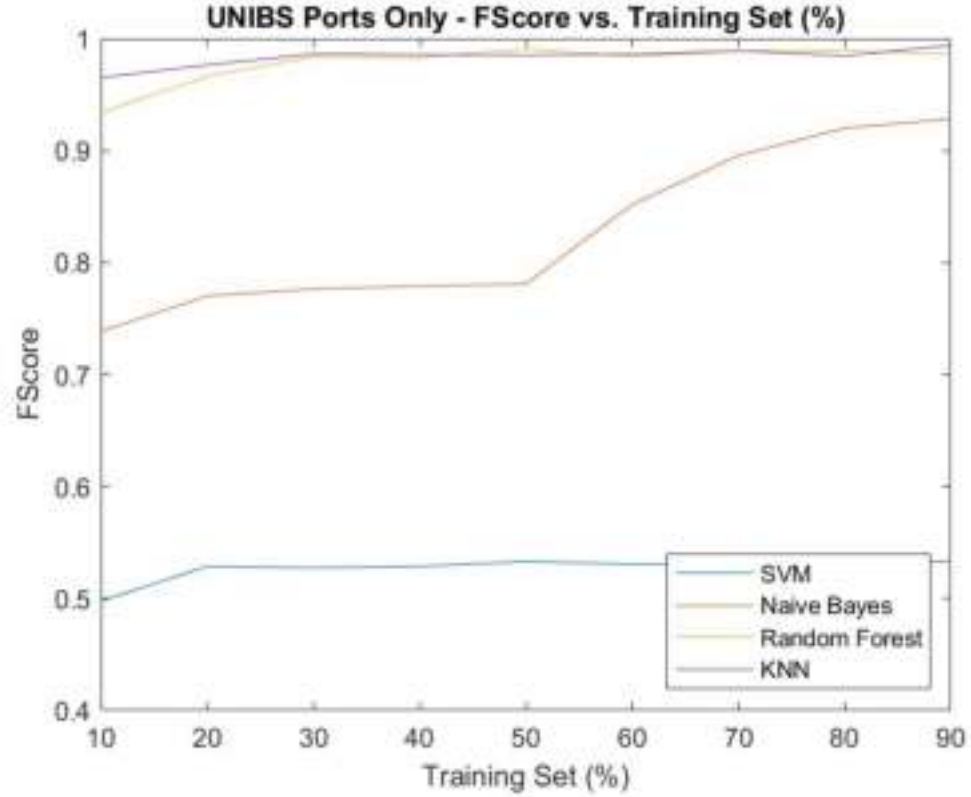

UNB RF - FScore vs. Flow Packets (%)
FScore
Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN

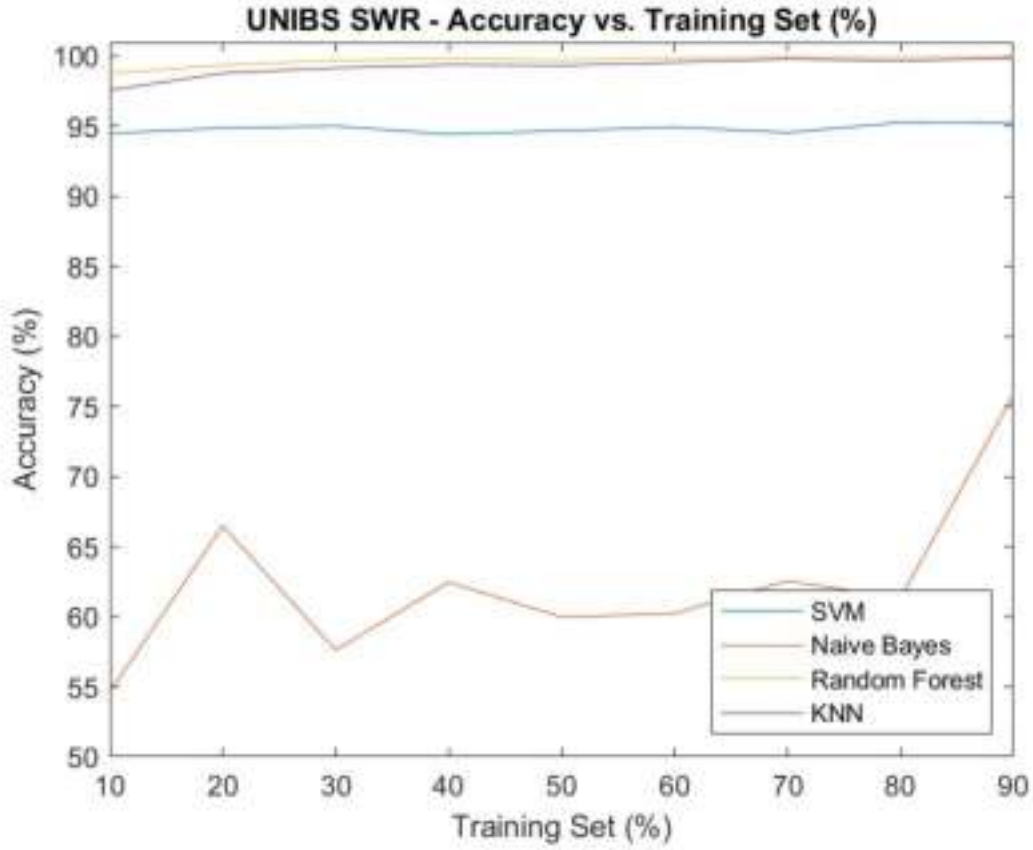

UNB RF (No Ports) - Accuracy vs. Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN
Accuracy (%)
100
90
80
70
60
50
10
20
30
40
50
60
70
80
90
100
Flow Packets (%)

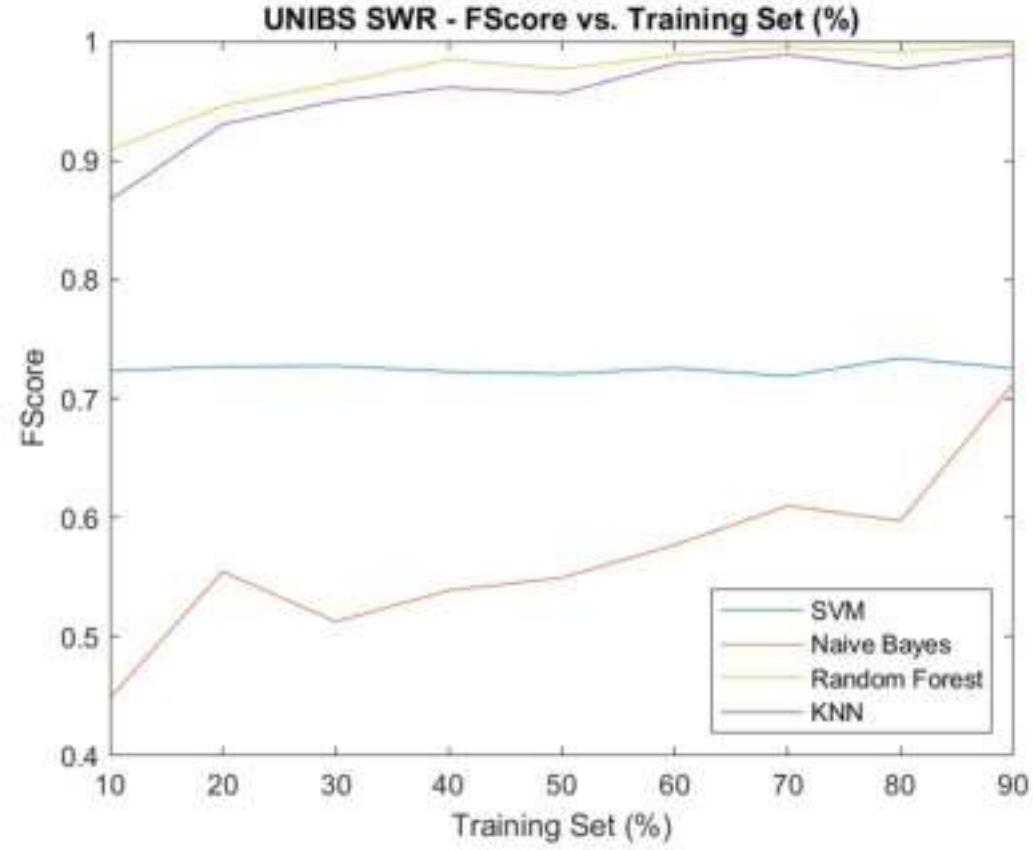

UNB RF (No Ports) - FScore vs. Flow Packets (%)
SVM
Naive Bayes
Random Forest
KNN
FScore
1
0.9
0.8
0.7
0.6
0.5
0.4
10
20
30
40
50
60
70
80
90
100
Flow Packets (%)

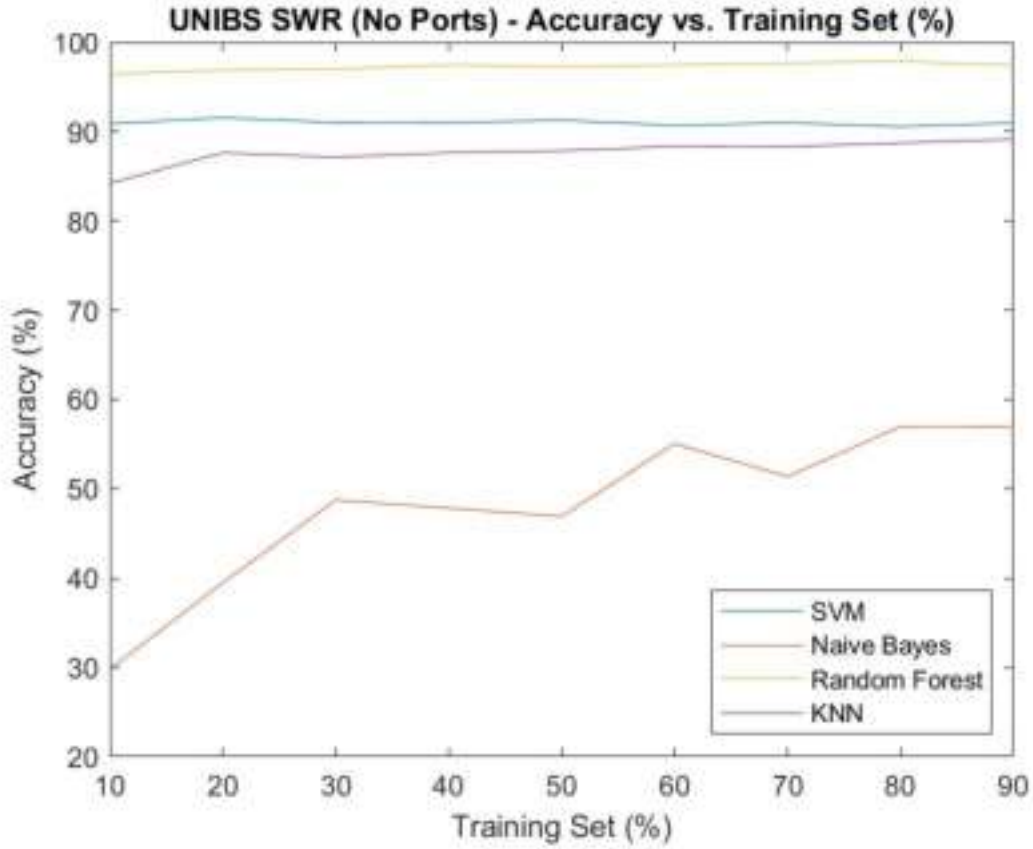
Appendix D – Various Training Set Sizes

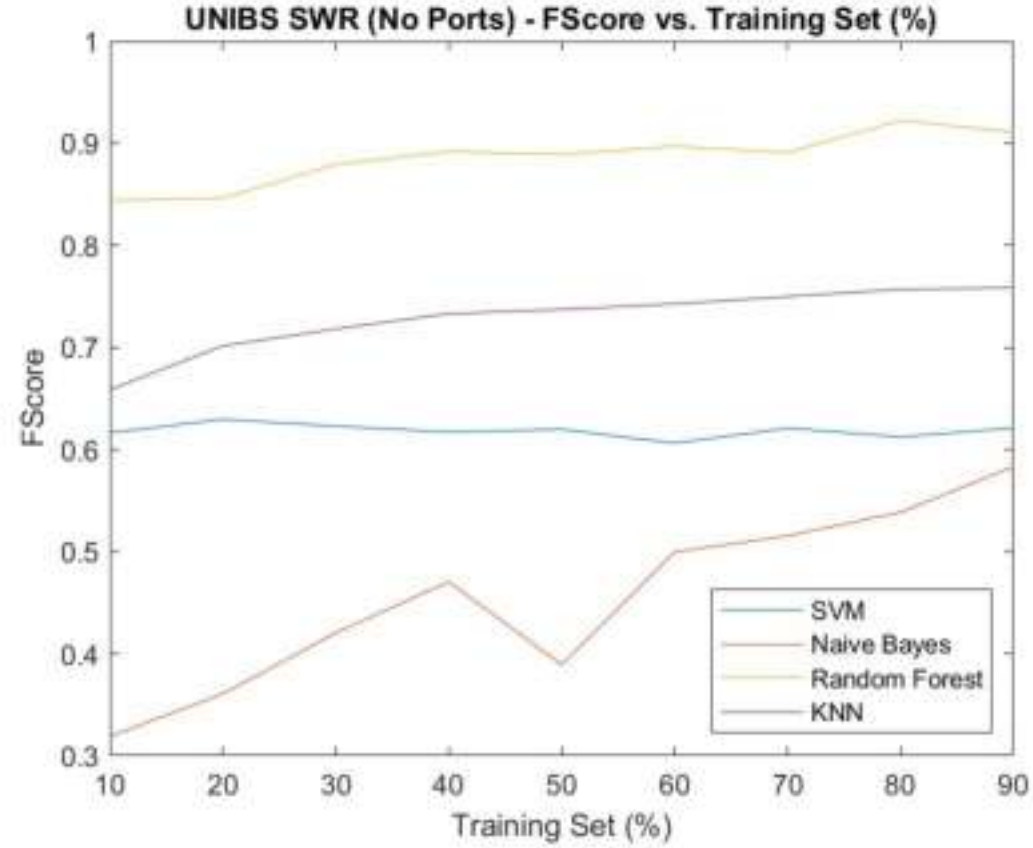


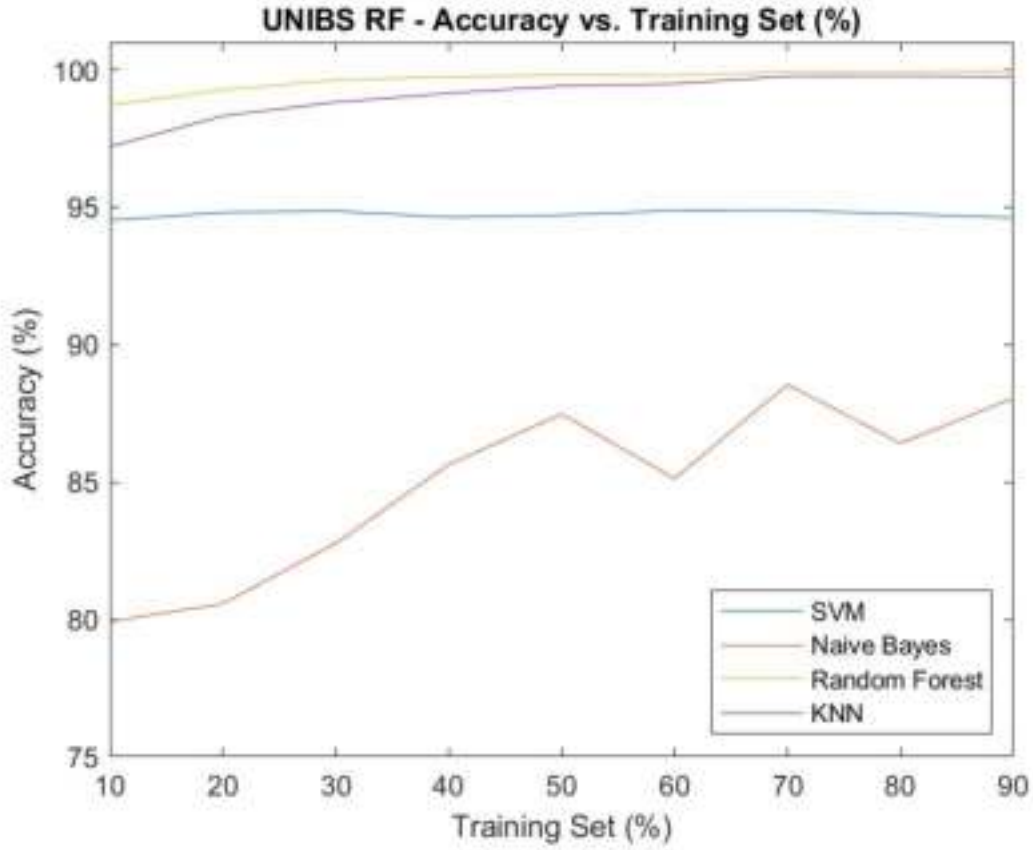


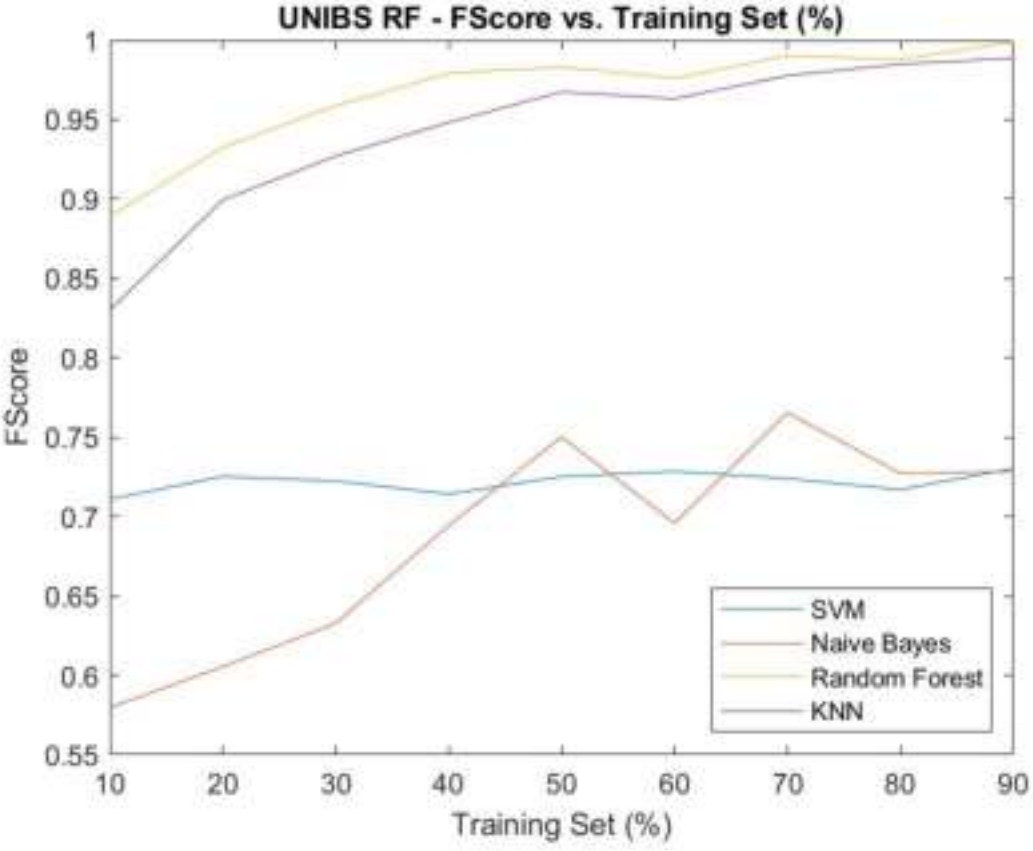

UNIBS SWR - Accuracy vs. Training Set (%)
Accuracy (%)
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

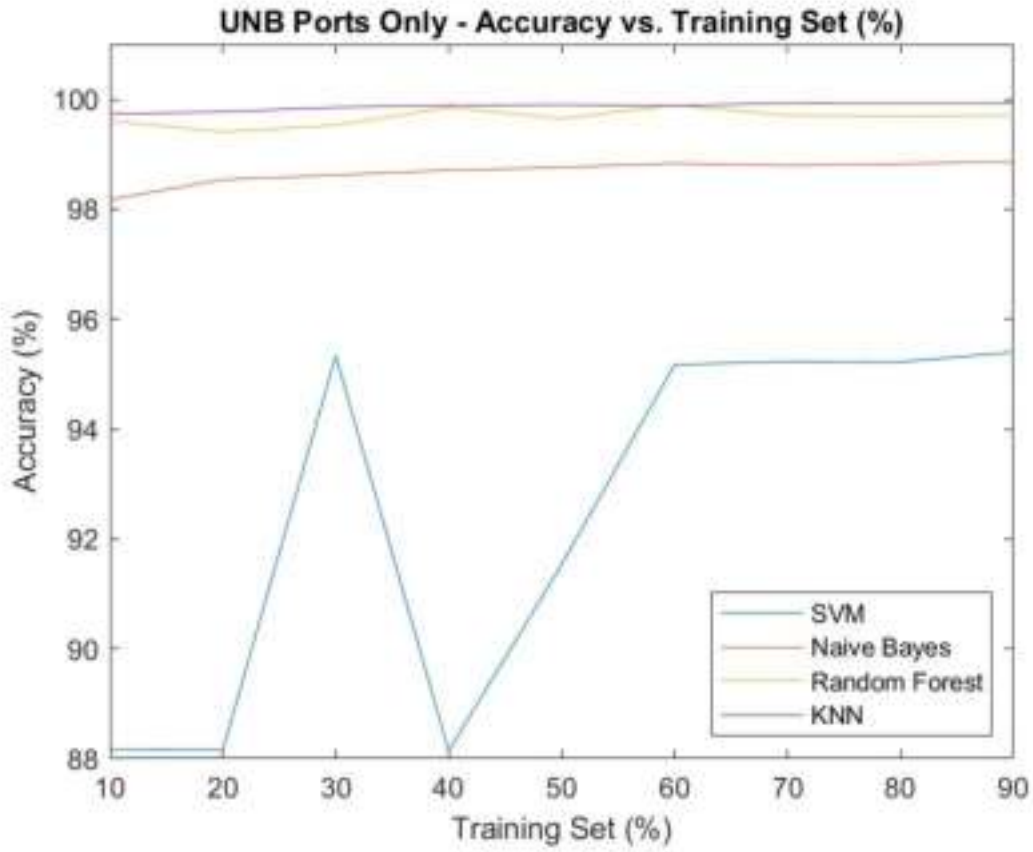

UNIBS SWR - FScore vs. Training Set (%)
FScore
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

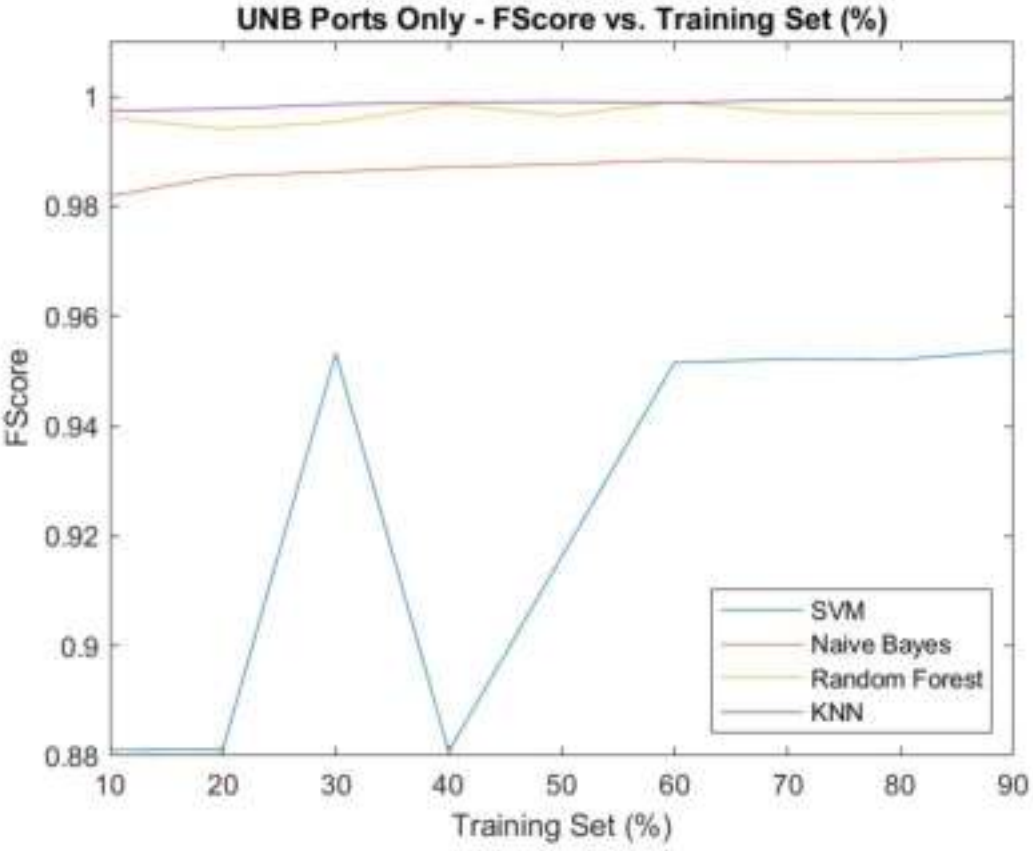

UNIBS SWR (No Ports) - Accuracy vs. Training Set (%)
Accuracy (%)
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

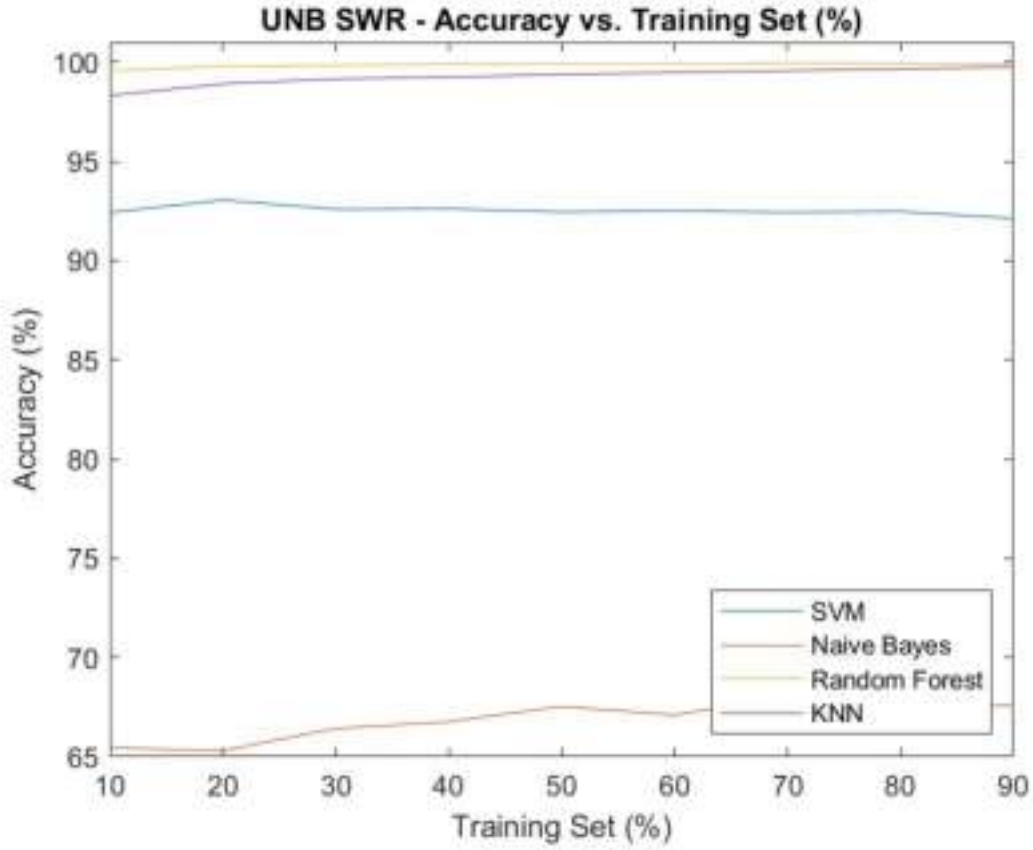

UNIBS SWR (No Ports) - FScore vs. Training Set (%)
FScore
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

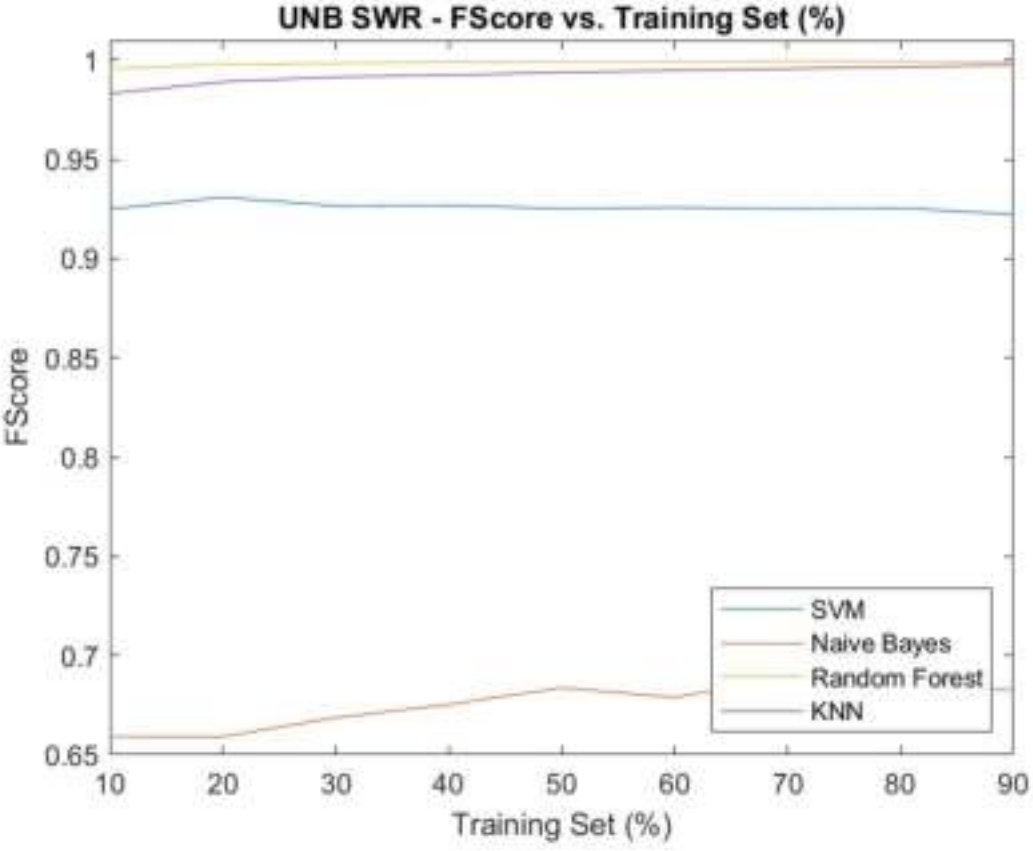

UNIBS RF - Accuracy vs. Training Set (%)
Accuracy (%)
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

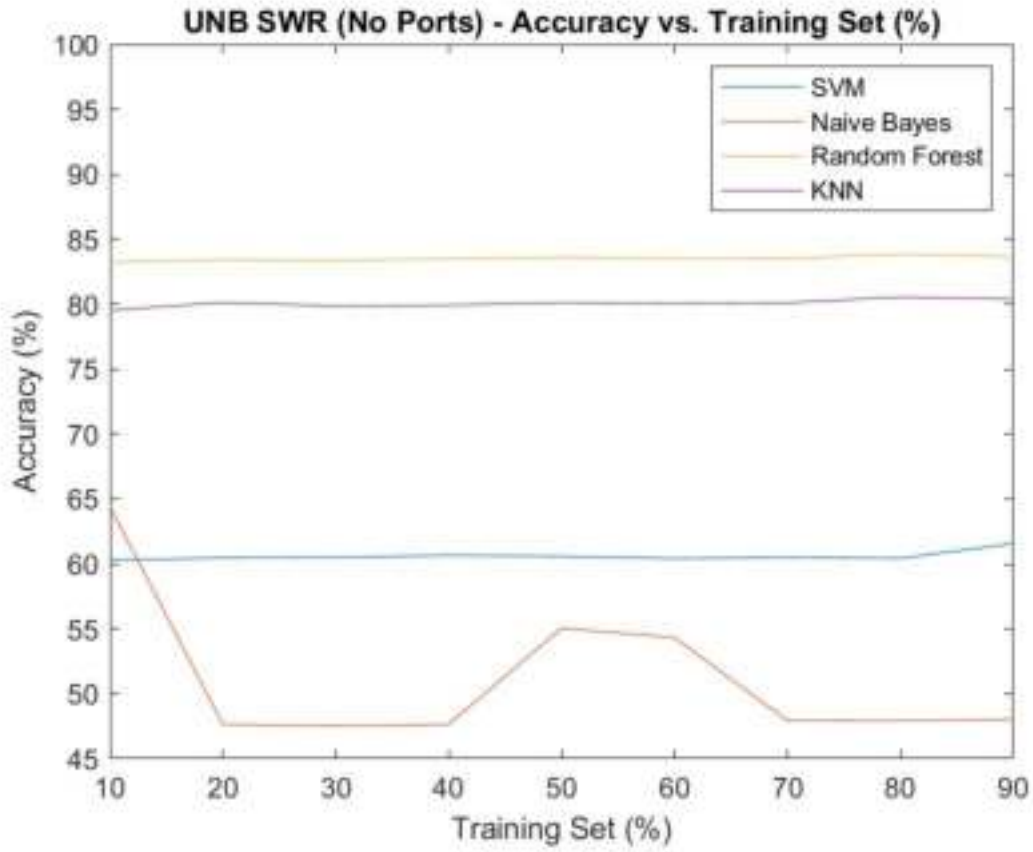

UNIBS RF - FScore vs. Training Set (%)
FScore
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

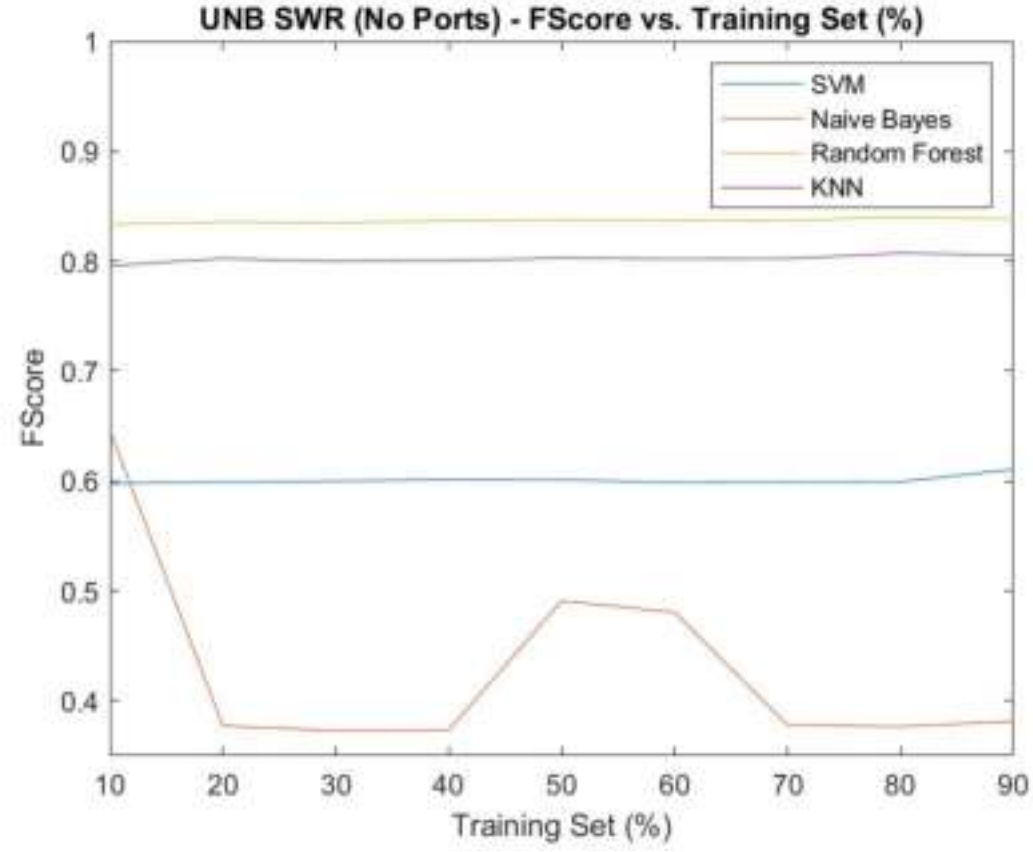

UNB Ports Only - Accuracy vs. Training Set (%)
Accuracy (%)
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

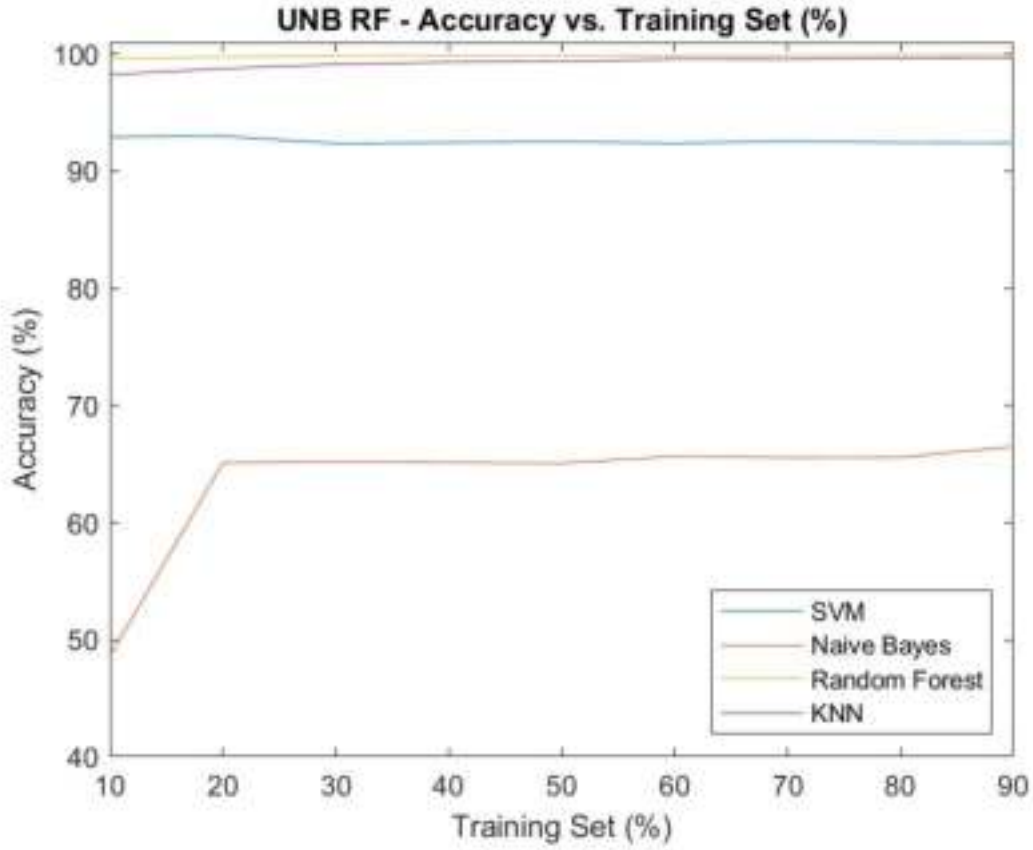

UNB Ports Only - FScore vs. Training Set (%)
FScore
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

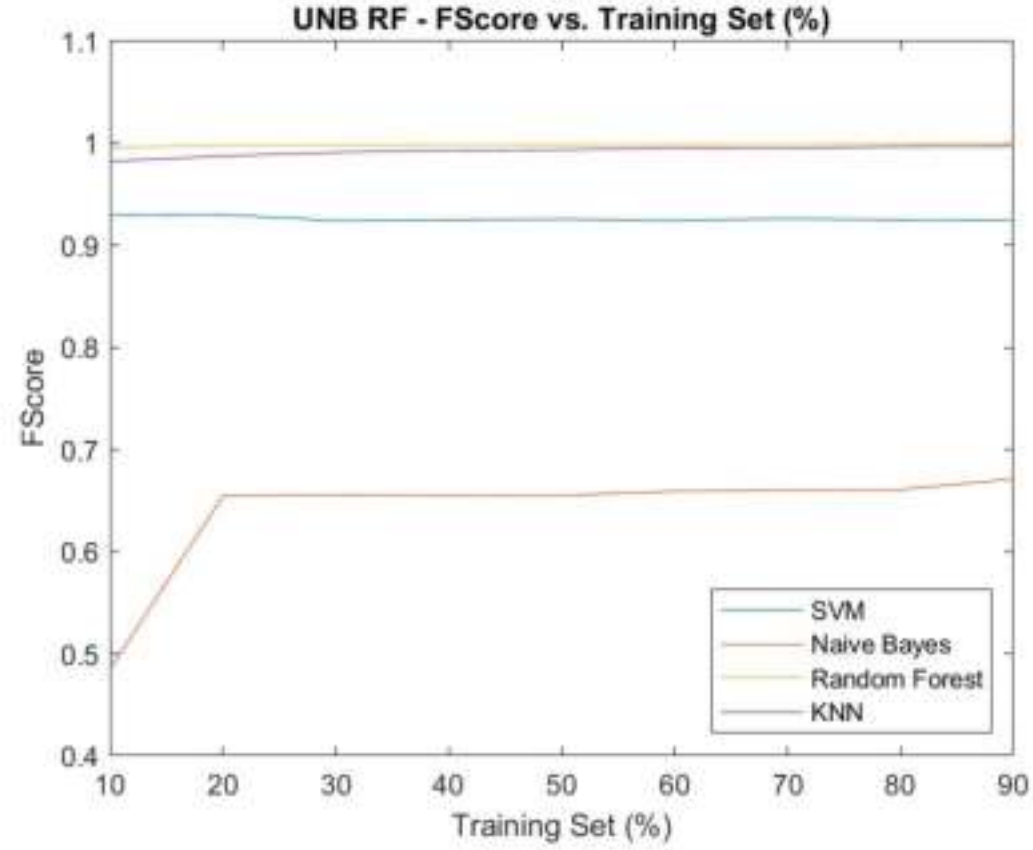

UNB SWR - Accuracy vs. Training Set (%)
Accuracy (%)
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

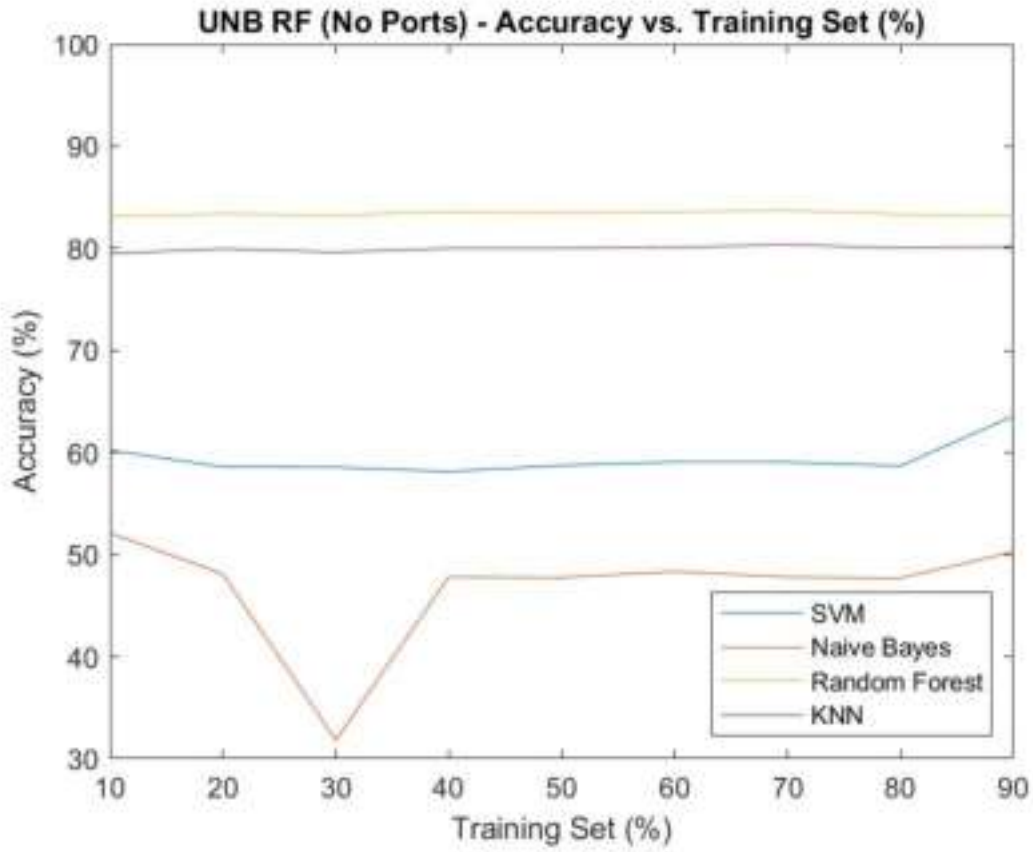

UNB SWR - FScore vs. Training Set (%)
FScore
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

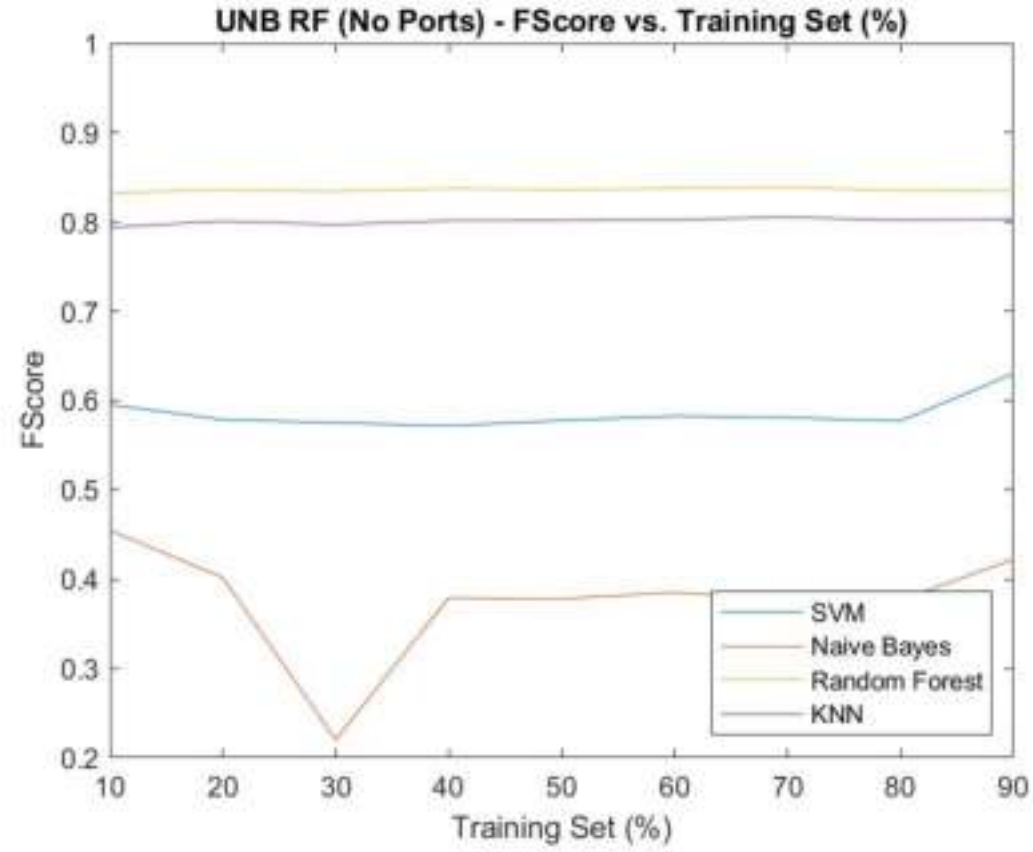

UNB SWR (No Ports) - Accuracy vs. Training Set (%)
Accuracy (%)
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN
100
95
90
85
80
75
70
65
60
55
50
45
10
20
30
40
50
60
70
80
90


UNB SWR (No Ports) - FScore vs. Training Set (%)
FScore
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN
1
0.9
0.8
0.7
0.6
0.5
0.4
10
20
30
40
50
60
70
80
90


UNB RF - Accuracy vs. Training Set (%)
Accuracy (%)
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN


UNB RF - FScore vs. Training Set (%)
FScore
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN


UNB RF (No Ports) - Accuracy vs. Training Set (%)
Accuracy (%)
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN


UNB RF (No Ports) - FScore vs. Training Set (%)
FScore
Training Set (%)
SVM
Naive Bayes
Random Forest
KNN

Vita

Mohammed Elnawawy was born in 1995, in Giza, Egypt. He received his primary education in Cairo, Egypt, and his secondary education in Dubai, UAE. He received his B.Sc. degree in Computer Engineering from the American University of Sharjah in 2017 as a summa cum laude.

In January 2018, he joined the Computer Engineering master's program in the American University of Sharjah as a graduate teaching assistant. During his master's study, he co-authored four conference papers which were presented in highly reputable international conferences, in addition to one journal paper. His research interests are in Field-Programmable Gate Array and Machine Learning.