Video Flow Classification
A Runtime Performance Study

Filip Västlund

The Faculty of Health, Science and Technology

Degree Project for Master of Science in Engineering

30 HP

Supervisor Johan Garcia

Examiner: Kerstin Andersson

2017-06-16
Abstract
Due to it being increasingly common that users’ data is encrypted, the Internet service providers today find it difficult to adapt their service for the users’ needs. Previously popular methods of classifying users data does not work as well today and new alternatives is therefore desired to give the users an optimal experience.

This study focuses specifically on classifying data flows into video and non-video flows with the use of machine learning algorithms and with a focus on runtime performance. In this study the tested algorithms are created in Python and then exported into a C code implementation, more specifically the random forest and the gradient boosting trees algorithm.

The goal is to find the algorithm with the fastest classification time relative to its accuracy, making the classification as fast as possible and the classification model to require as little space as possible.

The results show that random forest was significantly faster at classification than gradient boosting trees, with initial tests showing it to be roughly 7 times faster after compiler optimization. After optimizing the C code random forest could classify more than 250,000 data flows each second with decent accuracy. Neither of the two algorithms required a lot of space (<3 megabyte).
Acknowledgement

I would like to thank Procera Networks and Karlstad university for providing the resources and opportunity to do this project. A special thanks goes to Johan Garcia and Anders Waldenborg who has supported and helped me throughout the whole study.
# Contents

1 Introduction ........................................... 1
   1.1 Purpose of Dissertation .......................... 1
   1.2 Resources Used .................................. 2
       1.2.1 Development Environment .................... 2
       1.2.2 Anaconda2 4.2.0 .......................... 2
       1.2.3 Graphviz 2.38 ................................ 3
       1.2.4 Valgrind 3.12.0 .......................... 3
   1.3 Chapter Layout .................................. 3

2 Background and Related Work .......................... 5
   2.1 Decision Trees .................................. 5
       2.1.1 Performance Aspects ......................... 6
       2.1.2 Pros and Cons ................................ 8
   2.2 Decision Forest ................................ 9
       2.2.1 Creating a Forest ........................... 10
       2.2.2 The Random Forest Algorithm ................. 11
       2.2.3 The Gradient Boosted Trees Algorithm ...... 12
       2.2.4 Decision Tree and Decision Forest Studies ... 13
   2.3 Code Optimization ............................... 13
       2.3.1 Compiler Optimization ....................... 14
       2.3.2 Compiler Optimization Studies .............. 15

3 Code Generation Approach and Optimization ......... 17
   3.1 Representation with Existing Scikit-learn Methods .... 17
   3.2 Representation as if-statements .................. 17
       3.2.1 Classification Ambiguity ..................... 18
       3.2.2 Redundant Nested if-statements .............. 18
       3.2.3 Redundant else-statements .................. 19
       3.2.4 Performance Issues ......................... 20
   3.3 Representation as a struct Array ................. 20
       3.3.1 Exporting and Importing the Decision Tree structure from Python .... 21
       3.3.2 Optimizing the Array Structure .............. 22
       3.3.3 Optimizing the Struct Itself ................. 23
       3.3.4 Storing Meta Data ........................... 23
       3.3.5 Branch Behavior Optimization ............... 24
   3.4 Optimizing Random Forest as a Whole ............. 25
   3.5 Chapter Summary ................................ 25
4 Performance Evaluation Setup

4.1 Setup Overview ........................................... 27
  4.1.1 Creating a Decision Forest in Python ............... 27
  4.1.2 Classifying a Flow in C ............................. 28

4.2 Initial Studies .......................................... 29
  4.2.1 Collecting Consistent Measurements ................. 29
  4.2.2 To Refactor or not Refactor .......................... 30
  4.2.3 Using a One Vote Threshold or Two Votes .......... 30

4.3 Storage Size Requirement ................................. 31

4.4 Chapter Summary ......................................... 31

5 Performance Evaluation .................................... 33

5.1 Evaluation Details ........................................ 33

5.2 Execution Time Comparison between Algorithms .......... 34
  5.2.1 Base performance ...................................... 34
  5.2.2 Algorithms at Compiler Optimization Level: -O3 ... 37
  5.2.3 Algorithms at Compiler Optimization Level: -Ofast 39

5.3 Improving the Runtime Performance ........................ 41
  5.3.1 Only Evaluating Enough to Prove Vote Majority ... 41
  5.3.2 Storing all Vote Results in an Array ............... 42
  5.3.3 Combining Result Array Storage and Halting at Vote Majority 43

5.4 Performance Classifying Realistic Flow Type Ratios .......... 45

5.5 Further Optimization Attempts ............................. 49
  5.5.1 Sorting According to Majority ....................... 49
  5.5.2 Adapt Code for Realistic Flow Distribution ......... 51
  5.5.3 Majority is Only Relevant After Half of the Forest 53

5.6 Chapter Summary ......................................... 54

6 Conclusion and Future Work ................................ 57

6.1 Efficiently Moving the Forest from Python to C .......... 57

6.2 The Performance Evaluation ............................... 58
  6.2.1 Gradient Boosting Trees or Random Forest .......... 58
  6.2.2 Optimization options ................................. 59
  6.2.3 C code structure ..................................... 59

6.3 Future Work .............................................. 60
  6.3.1 Decision Forest by Vector Calculation .............. 60
  6.3.2 Performance Evaluation on Collecting Flow Data ... 61
  6.3.3 Optimizing the size of the decision tree .......... 61

6.4 Concluding Remarks ...................................... 61
## List of Figures

1. Example of a binary decision tree and an equivalent decision tree with more edges. ........................................... 6
2. Visualization of how bias and variance affect correct classification between training sets. ........................................... 7
3. Top three nodes in a larger regression tree generated in Scikit-learn’s GradientBoostingClassifier. ........................................... 13
4. Example of the nested redundant if- and else-statements. .............. 19
5. Example of the undesired else-statements and subsequent redundantly nested if-statements. ........................................... 20
6. The initial and final design of the struct used in the C program. ....... 21
7. Example of the final .csv representation of a tree. (Note that the name-row is excluded in the real file and only added here for illustration purposes) ........................................... 22
8. Visualization of the transformation and optimization of a decision tree represented as a struct array. ........................................... 23
9. The struct used to create the linked list that represent a random forest in C. ........................................... 25
10. The two ways that average classification time of a flow was calculated. 30
11. Baseline measurement of time relative to the maximum depth and amount of trees for the random forest algorithm. ....................... 35
12. Baseline measurement for the gradient boosting trees algorithm. ........ 36
13. Baseline performance for random forest, but with GCC compilation optimization flag -O3. ........................................... 38
14. Baseline performance for gradient boosting trees, but with GCC compilation optimization flag -O3. ........................................... 39
15. Baseline performance for random forest, but with GCC compilation optimization flag -Ofast. ........................................... 40
16. Baseline performance for gradient boosting trees, but with GCC compilation optimization flag -Ofast. ........................................... 41
17. Random forest implementation, but breaking classifications early whenever a classification type has >50% of the votes. ....................... 42
18. The random forest algorithm, but all votes are stored in a local array and evaluated at the end. ........................................... 43
19. Performance result of combining breaking at >50% votes and storing the votes in an array. (Figure 17 and Figure 18.) ....................... 44
20. Base performance of random forest, but with unique flows. (Note that the time-axis scales to 15 instead of 10.) ........................................... 46
Optimization with -O3, but with unique flows .......................... 46
Optimization with -Ofast, but with unique flows ....................... 47
Breaking when classification type reach >50% of votes, but with unique flows .......................................................... 47
Storing the votes in a local array, but with unique flows .................. 48
Combining Local array storage and breaking at 50% of votes, but with unique flows .................................................. 48
Example of how to reorder the samples and vote results once a flow classification vote has reached majority .................. 50
Sorting the array according to if a flow type has >50% votes, but with unique flows ..................................................... 51
Combining Local array storage and breaking at 50% of votes and sorting of the array .................................................. 53
Combining Local array storage and breaking at 50% of votes and sorting of the array .................................................. 54
1 Introduction

In today's world, the Internet has taken an increasingly large importance in the life of many people. It is far different from the situation when it first was conceptualized in the late half of the 20th century. The rapidly evolving environment of the Internet has allowed several successful businesses to emerge such as Facebook and Google among others. The Internet can for many be seen as one big business opportunity when there are millions of possible Internet users all over the world. It does not necessarily have to be a monetary gain, other things such as the information obtained about Internet users is also of high value.

One of the many ways information can be obtained on the Internet from its users is by inspecting users' traffic. A common way to analyze traffic earlier was to use deep packet inspection, also known as DPI for short, to look at the internal data of the packets in an Internet traffic flow. This method does, however, rely on that the internal packet data is not encrypted, which is much more common today [21].

The increased difficulty in analyzing user data does not only cause problems for those with malicious intent, however. One legitimate business that is hurt by the data encryption is the Internet service providers. They do not necessarily need exact details of users' data packets, but have great use in knowing the type of traffic the users have. By knowing whether the users currently are watching videos or downloading games among other things, they can better adapt their services to the needs of different users. Doing this can improve user experience, but it is now considerably harder to do with the use of deep packet inspection.

Today it is therefore desirable to find other alternatives to classifying people's Internet traffic, as deep packet inspection no longer is as viable. Machine learning is one such method that is explored, as it can in many situations draw relatively accurate conclusions without the need of inner contents from packets. There exist several general pieces of information in Internet traffic flows and packets that are not encrypted. This type of information can therefore be used to classify packet and flows without looking at their inner content through machine learning. Some examples of such data information are packet size and flow duration.

1.1 Purpose of Dissertation

Procera Networks is one of the businesses that offer services that make use of information gained by inspecting users' Internet traffic [52]. They have the previous year participated in studies regarding the performance of different machine learning methods. The methods examined was decision forests [36] and support vector machines [59]. This year it is desired to continue forward with such research
by further examining the classification performance and runtime performance in machine learning. This dissertation is focusing on creating an efficient decision forest implementation in C code following Procera Networks’ criterias.

There are two important goals for their runtime performance:

- The required storage size of the implemented decision tree storage should be as small as possible.
- The decision forest should be able to evaluate each flow as fast as possible.

The intent with researching efficient ways to classify flows into video and non-video flows is to in the future help Internet service providers adapt their services to their users current need. For example they could separate the video flows into their own queues so they will not be as affected by other traffic or lower the risk that the video will need to pause for buffering during a stream.

Optimally, the Internet service would prefer to allocate as few resources as possible when doing this. That is the reason why the goals of this study is left without hard boundaries. If one really had to specify a hard number as an upper limit, it would be a storage size of <50MB and a classification speed of >250,000 flows each second.

1.2 Resources Used

In order to evaluate the runtime performance properly, several computer resources was required

1.2.1 Development Environment

- Ubuntu OS 16.04, a Linux based operating system [16].
- Python 3.5 [53] with the additional packages:
  - Scikit-learn 0.18.1, packages for machine learning in Python [48].
  - SciPy Stack 1.0, a bundle of useful science packages [38].
- Training data set provided by Procera [52].
- Geany IDE 1.27, a lightweight basic IDE for C and other languages [58].

1.2.2 Anaconda2 4.2.0

Anaconda2 is an open data science platform [19]. This platform has a lot of useful tools for Python which is one of the two programming languages used. A major advantage of using a platform instead if installing several different components
individually is that you are guaranteed compatibility between different software used.

1.2.3 Graphviz 2.38

Graphviz is an open source graph visualization software [11]. It was used to generate some initial visualizations of decision trees. For example Figure 3 is made with its aid.

1.2.4 Valgrind 3.12.0

Valgrind is a GNU General Public License (GPL) system for debugging and profiling Linux programs [8]. In this study it was used for calculating memory requirement of the C program and detect memory leaks, but it has other uses as well.

1.3 Chapter Layout

This dissertation start by going through the important basics of what a decision tree and decision forest is as well as the general principle of compiler optimization in Chapter 2.

The experimental details themselves are found in chapters 3, 4 and 5. Chapter 3 documents the trials of turning a decision tree created by Python into a workable C implementation in an efficient manner. Chapter 4 briefly explains how the different programming components created works and initial runtime studies. The more in-depth studies can be found in Chapter 5. This Chapter focuses completely on improving the time it takes to classify a flow with the intent to make it as fast as possible.

More in depth conclusions and opportunities for future work are presented in Chapter 6.
2 Background and Related Work

This section will start with explaining the algorithms used from a detailed, but general perspective and then move onto an overview on compiler optimization, which is used in the project.

2.1 Decision Trees

Decision trees are a sub group of the many different types of algorithms used in machine learning. It is classified as a logic based supervised learning method [40]. Logic based here means that the algorithm in some way create a clearly defined rule set. With supervised learning, it means that the training set(s) used to teach the algorithm is organized into mapping pairs by humans. Generally these pairs are referred to as X and Y, where Y is a set of the possible classification labels and X is a collection of related data elements. The mapping is one to one, so a data element $x_i$ will correspond to classification label $y_i$ [18, pp. 1-2],[42]. There also exist unsupervised learning, which is when the algorithm has to try and define classifications on its own from unmodified raw data samples.

A simple example of supervised learning can be demonstrated by using the Iris flower data set, which was collected by Edgar Shannon Anderson [4]. (Although it was another person [30] that published it slightly later [10].) In this data set there are the four measuring parameters: sepal length, sepal width, petal length and petal width and three flower classifications: Setosa, Versicolor and Virginica. If this was implemented as a training set, the measurement would create a collection X so that every sample $x_i$ in $x_1$ to $x_n$ equals:

$$x_i = \{\text{sepal length}, \text{sepal width}, \text{petal length}, \text{petal width}\}$$

and every corresponding $y_i$ in Y equals:

$$y_i = \text{setosa|versicolor|virginica}$$

The core concept of decision trees themselves is to generate a set of conditional statements from the training data. The reason it is called a decision tree is because the statements are organized like a directed graph in the shape of a tree graph [40]. Most decision trees are binary, but there can also exist nodes with many edges as illustrated in Figure 1. In order to generate a decision tree, a training set is created in accordance to the previous paragraphs. The used training set is then read and analyzed by the underlying software responsible for creating the decision tree constraints, a process also called tree induction [45]. This is done by searching for an identifiable relations between the attribute values in X and the different classes.
in Y and define constraints accordingly. When an acceptable requirement has been identified at a node, the algorithm will decide if it is enough to classify the data or if additional nodes are needed.

![Binary decision tree example](image)

Figure 1: Example of a binary decision tree and an equivalent decision tree with more edges.

The induction algorithm will make its decision depending on how it is configured, as there is not yet a general solution that work optimally for every situation [45]. The conditional split at a node in a decision tree is most often binary when handling numbers [55], but trees can apply more than two edges at a single node. When the decision tree later tries to identify new data elements, it will follow the generated constraints at each node. Once the algorithm reaches an end node of the tree, it will have decided the classification of the new data.

### 2.1.1 Performance Aspects

Due to how important the initial definition of a decision tree is for its further performance, many studies has been done in regard to tree induction and its optimization. However, the problem of creating optimal decision trees seem to be NP complete even for binary trees and is therefore hard to solve [35, 41]. One of the most basic ways to make a tree more efficient is ordering the classification criteria to begin with the most significant constraint and work its way down to the least significant. Such split conditions can be referred to as the best divide [41].

As there are many types of data, there also exist many different types of decision tree implementations. For example identifying risks related to heart failure [61], analyzing road-accident injuries [29] or as in this report analyze data flows. Additionally, there exist several preferable ways to measure how efficient a decision tree is as well [56]. Some of these measurements, such as minimum error rate and maximum average mutual information gain can be applied to more machine learning methods than decision trees. While others, such as minimum nodes in the tree, can not and are more specific to decision trees. In general when it comes to optimizing
the efficiency of decision tree specific parameters, the focus seem to lie in making the
tree as compact as possible while still getting the most out of each node criteria. The
fewer the nodes are in the decision tree, the faster it can classify data on average.

The usefulness of a decision tree and its parameters is not only measured at
its initial creation. A decision trees performance is, like other machine learning
methods, dependent on how it performs while identifying new data. It is important
that the applied model can also perform well over several different iterations
(different training sets). As different sets are not guaranteed to fit with the real
data equally well, there can occur various degrees of errors.

One such error is overfitting, which is when the algorithm consistently groups
the identification too much in tune with the training set. By doing so, it can not
accurately identify actual data samples from similar looking noise by the training
sets [25, 13]. There also exist the possibility of underfitting, which simply is when
the model is not suitable at all instead of too fit to the training data. It does not
find a clear distinctions at all between classification [48, 37].

These two problems have a strong connection the variance and bias that occur
in the model to different degrees [31]. Variance refers to the accuracy spread, a too
high variance means the model will inconsistently apply classifications learned from
the noise in the training set used. Bias on the other hand refer to when the model
used tilts its classification towards a certain kind of data regardless of training set.
If the model consistently tilt towards incorrect data, it has high bias. The above
description can be observed visually in Figure 2. Overfitting is what happens when
the bias is low but the variance is high and underfitting happens during high bias.

Figure 2: Visualization of how bias and variance affect correct classification between training sets.

Up to this point, all the performance aspects described has been related to
classification performance. The classification performance refer to how well the decision tree will classify the data samples. It is however not the only relevant performance aspect. There also exist the runtime performance, which is how efficiently a decision tree will run from a programming perspective. This is the performance in regard to storage size and execution time of the decision tree. For example, the decision tree can have a good classification performance (high classification accuracy), but still be very bad because it is very slow or takes too much storage space. The runtime performance is what will be the focus of this report.

2.1.2 Pros and Cons

There are several general pros and cons with the decision tree algorithms in addition to those that can occur in the many different implementation variations. The absolute biggest advantage for the algorithm is its simplistic graph like structure. This has made the algorithm relatively easy to understand even for those not deeply invested in the machine learning field. It also makes decision trees easy to configure, analyze and implement compared to some other methods as well. Because of these factors, decision trees has been implemented in many open source packages for a variety of different usages. Additionally, they scale well with big data while still putting a relatively low strain on the system [55].

The structure itself is can also be a big benefit, as there can be many different factors regarding the examined data. Some other algorithms can have difficulty to handle all the given information. This is however not a problem for decision trees, as they can filter out different types of irrelevant information and classification at each node [56] if combined with an appropriate method [60].

Decision trees can also be used in combination with other similar methods and can handle classifications with problematic data relatively well [55].

There are however several drawback with using the decision tree algorithm. For example, its simplicity can also be a big drawback in some cases. The biggest problem being that it can easily acquire redundancies in cases where many classes and attributes exist [56]. Especially if the attributes are close together in values between classes [55]. More complex situations are also a problem area for decision trees. A decision tree will only look at very closely related data in the training set and therefore lack the ability for more complex classes requiring the larger picture [55]. One could argue that complex situations could be resolved in the tree if it was made more advanced, but that removes from its simplicity. Therefore, a decision tree can not be perfectly accurate and optimally effective at once [56].

The decision tree can also have a relatively sensitive structure where even smaller
changes can accumulate down its structure. If such errors are not corrected, they can create a chain effect and render the tree sub optimal [55, 56]. It is however possible to counteract this in some situations by allowing the tree to look ahead further at the expense of increased computational requirements [28].

Another approach to counteract some of the weaknesses in a single decision trees is the decision forest approach described in the next section.

2.2 Decision Forest

The decision forest approach has since its creation in the 1990s obtained wide popularity [55]. It is based on ensemble learning, which was proposed a few decades earlier [22]. Ensemble learning is when one is combining several models to solve a single task. In the context of the decision forest algorithms, ensemble learning is applied by making use of several decision trees [55].

As just stated, the decision forest is a machine learning representation using ensemble learning. It therefore works on the same underlaying theory to work, the Condorcet’s jury theorem. This is a theorem created around the late 1700s and it explains how voting will lead to a certain outcome in a majority vote. It states that as long as N number of voters overall have above 50% chance to vote for a specific choice, the chance of that choice getting majority of the votes will go toward 100% as N goes toward infinity. Similarly, the choice will lean towards a limit of 0% votes if voters overall have below 50% chance of voting that choice [44]. Instead of a single decision tree making the whole decision on its own, there are several different trees voting on the classification. It can most simply be described as a democracy, where every decision tree vote according to their own understanding (unique training set) of the situation. [55].

There are also other advantages with using decision forests in general and not only just because Condorcet’s jury theorem states so. For example, if the training set as a whole is too limited to give a good approximation with a single tree. It is possible that there can be several different trees derived from the training set and each has some parts correct. If used together it is more likely that the different decision trees can make up for their individual deficiencies. It makes the quirks of individual decision trees less significant to the overall result [24, 51].

Another possibility is that there is a lot of attributes to take into consideration, thus making a single large model very complex to create. With the use of ensemble learning, it is possible to have several simplified subsets of all attributes. Each individual classifier will have a lowered performance, but if the subset splits are done well, the performance as a whole can be good. In this way the complexity has been reduced [51]. Decision forests in this case work similarly to how it often is
easier to create a cluster of computers, instead of a single gigantic super computer.

2.2.1 Creating a Forest

An important point as stated previously is to create the forest from several unique decision trees so they can make up for individual trees’ classification flaws. There are a few common ways to define unique trees. The simplest of them is using equally relevant, but unique training sets [55]. This can either be done by creating several training sets for the same purpose, or more commonly by dividing a single training set into smaller subsets. The later is also called partitioning and is more common, since it can create diverse decision trees without the need of additional training data.

A training set can be divided in several different ways, it can either be done by dividing the training set by samples or by features. These concepts of training set division works exactly the same as when dividing databases with horizontal or vertical partitioning [27]. In cases where one divides by samples, each new sub training set will still include all features from the original training set, but not all the samples. This corresponds to horizontal partitioning. Dividing by feature on the other hand is like vertical partitioning. It means keeping the original amount of samples in each training subset, but each new training set will only keep the values of some of the features [54].

In addition to the created diversity in decision trees, partitioning can also be manipulated to create decision trees for a certain purpose. An example of this would be better classification of a subset of classes, which the original group of classifiers find difficult to properly classify. Instead of looking at the training set as a whole it can be more specified towards a subset of classes where classification is extra problematic [17].

Manipulating the algorithm itself is another simple and easily applied method. In this method, the different configurations influencing the tree induction algorithms is changed. By doing so it completely changes how the algorithm will decide the applied node division in the tree. The changes in the algorithm can be done by, for example, specifically redefine certain parameters such as tree depth or by inserting randomness into the algorithm.

The big difference between these two methods of manipulating the algorithm is how they allow a training set to generate several unique decision trees. When using parameter modification, the tree induction algorithm will still generate the same tree if both training set and configurations are the same. The reason the algorithm need to change its parameters to create another tree is because it still choose the decision considered optimal at each node. Using the randomness method on the other hand can create several different decision trees from the same training
set without additional modification as it does not always choose from some sort of optimization criteria [55].

There are also other less apparent methods to generate unique decision trees for the forest, such as heterogeneous tree induction. In this method the tree induction algorithm itself is changed between trees. According to research regarding this type of decision forest algorithm, the advantage of this method is an increased performance when irrelevant attributes are included into the training set [32].

Finally, there is also the possibility to combine several different ensemble methods called ensemble hybridization. This method sounds very similar to heterogeneous tree induction, but it is not. Ensemble hybridization is not necessarily doing anything to the tree induction itself. Instead of using different algorithms for the trees in the forest, it uses several different forests in combination with each other. Basically, it is creating a bigger forest out of smaller ones. The individual forests can however still be applying the use of heterogeneous tree induction or other tree induction methods if so is desired.

After having created a set of unique trees to use, they need to be combined to a forest in some way. The process of making the forest is called decision fusion. The two main categories of decision fusion are refereed to as weighting and meta-learning. In the weighted method it simply gives all the individual trees a vote, which in some instances is weighted relative to an certain criteria (such as accuracy [46]) or simply using majority voting. In meta-learning it does not uses the individual trees to directly decide a final classification. Instead the trees’ individual classification are used in a meta-model that generates the final classification [55].

### 2.2.2 The Random Forest Algorithm

The random forest is one of many different algorithms used when creating a decision forests and has become one of the most popular methods today, since its introduction in the 90s [3, 34]. The reason for its high popularity is due to its simplicity and performance. Its popularity has also lead to it being implemented in a variety of different ways (for example Scikit-learn and C#) [55]. The reason it is called random forest is that every tree is created by a randomized subset of the whole training sample and the usage of a random subset of all attributes to define the split [12]. It uses a simple majority vote among its trees to decide the final classification.

When it comes to picking out the random subsample of the training set for each tree, there is no special way that the randomly chosen data elements in the training set are picked. This is done with replacement. The act of generating a random training subset can also be refereed to as bootstrapping or bagging [55, 23].

It is however not quite as simple when doing the randomized node split decisions
inside the individual trees of the forest. A relatively simple way to pick the node split is to just pick an attribute to split with completely at random. Doing so is however not very desirable as some attributes will simply be to inefficient in certain situations. The tree induction algorithm will, for most cases, instead try to randomize between the more relevant attributes. This can for example be done by having the odds of choosing an attribute as the split attribute relative to its perceived usefulness [55]. Another way is to pick uniformly from all the attributes that is considered most useful [23]. Some less obvious ways to insert randomness also exist, such as using histograms [39] among others.

Extremely randomized trees is a similar concept to random forest in the context that it uses randomness to create different trees. This algorithm does however do the randomization of the node split in a different manner and it also does not use bootstrapping [33].

2.2.3 The Gradient Boosted Trees Algorithm

Gradient boosted trees algorithm is another decision forest algorithm, but it do not rely on some sort of randomness to generate its forest. The gradient boost algorithm instead tries to create an accurate forest classifier by making each new tree a complement to the previously created trees in the forest. It is relatively vulnerable to over and under fitting as it can easily happen if too many or few trees are generated. One way to lower the risk of over and under fitting the boosted forest is to draw random subsamples as training set for each tree [55].

The algorithm does also not normally use majority vote in order to classify the samples it evaluates like random forest does. It is more common for the gradient boosting trees algorithm to use regression trees [55]. When using regression trees to classify data, each tree will instead uses real number ranges at each node to define the likelihood of different class labels. The values from the different individual trees will be added together to give the final classification result. An example of a gradient boosted tree be seen in Figure 3. In this Figure the value variable at each node indicate its current classification, negative values indicate one class and positive values another for this specific regression tree. The classification values can be represented by a completely different value range in another type of gradient boosted tree.
2.2.4 Decision Tree and Decision Forest Studies

When it comes to studies regarding decision tree, decision forests and machine learning in general, very few focuses on the actual runtime performance of the algorithms. Since the runtime performance is dependent on the implementation, it can not be applied in a more general sense like classification performance. The work and publishing of studies regarding runtime performance thus has limited value outside of the specific business(es) using the exact implementation.

The absolutely most relevant study for this project is the concurrent study also given by Procera Networks. It uses the same training set as this study, but has a focus in the classification performance (accuracy of the algorithms) as opposed to runtime performance like this study [5].

If one want to further read about the current studies concerning decision forests and/or decision trees, there exist article designed to give people a general overview of the subject. For decision forests, the article Decision forest: Twenty years of research by Lior Rokach is good starting point as it broadly explain decision trees, decision forests and popular decision forest algorithms [55]. This research is up to date as of 2015.

For a study focusing more in details on decision trees, the article Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey by Sreerama K. Murthy [45] a good read. It is from 1998, but still covers the topic of decision trees well.

2.3 Code Optimization

While the increased capacity of computers today has eased the pressure on efficient coding in many commercially used products, it is still very important in certain situations. This is apparent in environments where every millisecond counts, such as space ship calibrations. Another example is in working environments with limited
resources. Such situations include, but is not limited to calculators and older outdated computers. Programmers should also not take the higher capabilities of current computers as an excuse to not attempt optimizing their code. Even regular commercially made software can be done so badly that normal computers can find it difficult to handle in a timely manner [57].

There are several levels where softwares can be optimized today and they vary greatly. For the problem at hand, compiler optimizations are of most importance.

### 2.3.1 Compiler Optimization

The compiler is an important step in the programming process. Its purpose is to translate instructions in a programming languages to the equivalent instructions in another programming language [2, pp. 4]. This is primarily used for translating high level languages (such as C++ and C#) to low level languages (such as assembly and machine code) so the computer can understand the instructions in the software. As high level languages are made to be easily understood by humans, they have several redundancies compared to low level ones. These redundancies in high level languages therefore need to be removed before the computer runs the code to create the best performance.

The code optimization process can be divided in two parts. The local code optimization, which handle smaller instruction blocks and the global code optimization, which make the interaction between blocks run smoother. A simple example of local code optimization would be reducing

\[
B = A
\]

\[
C = B
\]

to

\[
C = A
\]

assuming B is not in use elsewhere in the scope. In cases where it is not used later, it is referred to as dead. It is of course more complex than this in reality where different expression, including pointers and arrays, can occur [2, pp. 533-542].

When it comes to global code optimization, the compiler can look at how the different block interact with each other in many different ways. The inner workings of these methods that the compiler uses is however not mandatory to understand for this dissertation and will therefore not be explained in further detail. They can however be read about in detail in books regarding compilation [2].
2.3.2 Compiler Optimization Studies

As programming has existed for several decades, there has been much research on how to make it as efficient as possible. Studies has been done on both how to do it from a general perspective as well as language and algorithm specific optimizations.

Since the optimization of code can be considered a subpart in compilation, the literature that thoroughly explains compilers usually also has a section about optimizing the conversion of high level languages to low level ones. Such literature can be a good starting point if no prior experience in compiler optimization exist. A recommendation to refresh knowledge about the compiler in general and its optimization is the book Compilers: Principles, Techniques and Tools [2].

If one want to read more in detail about some aspect of compiler optimization, they can turn to for example *An axiomatic approach to code optimization for expressions*. This paper by James C. Beatty focuses on optimizations of expressions by trying to find other more efficient, but equivalent expressions [9].

Another article of possible interest would be the *Some Topics in Code Optimization* by Christopher Earnest. This article discuss a method to find redundant expression calculations. It describes how the method itself work and talk about some possible improvement to it [26].

For those that are interested to look into optimization of the C programming language specifically and not compiler optimization, they can start with *Manual and Fast C Code Optimization* by Mohammed Fadle Abdulla. This article does not focus on compiler optimization, but manual optimization and way to improve your programming in C as well as tests a few ways to optimize the code. Among the different ways, the article mentions: Variable data types, Register Variable allocation, Loop Termination Condition, Branch Removal, Functional inlining and AND/OR operator conversion. [1].
3 Code Generation Approach and Optimization

This section will show a few ways to export Python generated decision trees to equivalent C-code and explain different types of optimization that was applied to the different export methods. The reason that several methods to do this was explored was to demonstrate initial proof-of-concept functionality, and then finding a method that would be well suited from this project.

In order for an implementation to be suitable for the study, it should both allow for an easy basic implementation as well as allowing quick modifications so many variations can be explored for optimization purposes. Ultimately, it was experiments using these variations that lead to the final choice of implementation.

That is however no to say that the implementations mentioned here are the best or only ones that exist. No other C-code representations was pursued however, since the struct array representation seemed to have the desirable traits of easy basic implementation and quick modification.

Since the random forest algorithm simply is a collection of several decision trees, the initial tests consist of generating and optimizing code for a single decision tree. The used training set was obtained from samples by Procera Networks \[52\], which was then fed into the Scikit-learns DecisionTreeClassifier. It was then needed to get the data structure from the tree generated by Scikit-learn into an easily parsed format.

3.1 Representation with Existing Scikit-learn Methods

The first attempts to do this was by using the already existing export_graphviz \[15\] method in Scikit-learn by creating the file in .txt format instead of the recommended .dot format. While this method gave completely parse-able text, there was plenty of redundancies, which would require an additional step of cleaning. The approach was not pursued further.

3.2 Representation as if-statements

After finding the existing decision trees representation lacking, an alternate approach was quickly found \[47\]. It uses the underlying tree__attribute that every Scikit-learn DecisionTreeRegressor is given and parses it to a Python code generator. The code generator was copied, modified and re-purposed into a C code generator as that was the desired format instead of Python code. The reason that C was the desired language for generating the decision tree is because its relatively close to assembly language and therefore quite good as an initial performance
standard. Afterwards, the code was compiled to make sure no direct syntax errors had occurred.

3.2.1 Classification Ambiguity

The next step in the code generation process was to optimize the created C code. The reason this was necessary was due to how the Scikit-learn object represent decision tree end nodes. When an end node is reached, it will not give a definitive classification. Instead Scikit-learn represent how many of each classification type that fit into the node’s classification. For example, an end node answer of [10 3] could represent 10 flows that are not video and three that are. In the generated code, such ambiguous results are not desirable and instead turn into a return false; (more non-video than video packets).

The translation from Scikit-learn object to C gave raise to several new problems when generating the code. The first problem even occurred before the code was even complete. In some cases, the end node array cases where for example [5 5] or more generally stated:

\[ N_{\text{video}} = N_{\text{not-video}} \]

This made the generator completely unable to decide whether to define the flow as video related or not. It could however be easily solved due to the fact several decision trees will be used as a random forest. In cases when an individual node can not decide which flow type to pick, it will randomize the decision.

This could give inconsistencies for a single tree, but the forest as a whole would be very unlikely to have significant amount of trees reaching such randomized nodes during the same iteration. That in turn would mean that the tree’s randomized vote(s) only would matter if it was the deciding vote. Such a result would already imply that the other trees can not get a conclusion either as they had voted evenly for all the alternatives.

3.2.2 Redundant Nested if-statements

Another problem that was found had to do with optimization because of redundant statements. The problem would occur when both the if and else case favor the same packet type. The reason that this could occur was that two end nodes (of the same parent) in the random forest can decide on the same classification, but to different degree. For example the left node gives [10 0] and the right one gives [100 5]. When such a situation is translated to the generated code, it will give redundant if- and else-statement as seen below in Figure 4.
As clearly seen in Figure 4, even if the innermost statement is reduced to a simple `return 0;`, there will still be a redundant statement. This problem is handled by iterating the same code repeatedly and reduce the code one layer at a time. According to the specification of the project, there was no time limit in regard to the optimization of the trees before implementing them. Due to this, it was not important to make this optimization fast.

The function used for this optimization will look for situations where both the if- and else-statement contain identical returns. If such a code section is found, it is deleted. The algorithm will then be run again in case the optimization can now be done one layer higher. In order to make the function know when it can’t simplify the statement any more, it will compare the last two iterations and if there is no size change. When it detect no positive change, the function loop will stop.

### 3.2.3 Redundant else-statements

There exists other redundancies too as seen in Figure 5. Instead of having all the shown else-statements, it could be reduced to a single return call after the nestled ifs. Since the return will occur as long as not all the if-statement go through, it is not relevant where in the nestling it happens. This pattern can be cleaned in smoother way than the previous problem as the else pattern is not nestled, but repeating. This allows for all the redundant else statements to be removed in a single iteration.
Figure 5: Example of the undesired else-statements and subsequent redundantly nestled if-statements.

Yet another possible optimization of the C code actually appeared as a direct effect of the previous removal of redundant else-statements. In this case it can be seen that the removal of else-statements has caused unnecessarily nested if-statements (Figure 5). This makes the code look bloated as the same can be achieved by having all the conditions in a single if-statement and achieve the same result. The nestling might seem like an issue, but from a technical perspective it is unlikely. In compilation to a assembly languages (Mips for example) it is already common to divide larger statements into smaller parts [43]. The reason for doing so is the simplicity of the instructions used in low level languages. As there exist no way of executing advanced statements all at once, they are broken down. This need to divide statements regardless combined with the more efficient compilers of today makes the if-statements unlikely to cause any inefficiencies compared to a combined statement.

3.2.4 Performance Issues

There is however an issue with using a C code representation of the decision trees as a whole. The initial tree tested takes a space of 221.8 kilobyte, which can add up once the tests move on to implementing a random forest. To further optimize the decision tree and random forest implementations, it is also explored if the same tree/forest can be represented in other ways. As previously stated, the allotted space for the final implementation is very limited.

3.3 Representation as a struct Array

The next decision tree representation explored was by reading it into an array of structs. Both the initial and final design of the struct format used can be seen in Figure 6. In the initial representation, unsigned long val represent the threshold seen in the if-statements from the previous section. If an examined flow’s value is less than or equal to the unsigned long val variable in the struct, it will go to the
array index stored in unsigned int LC and otherwise it will go to the array index in unsigned int RC. In order for the function to know which variable in the flow to evaluate, it uses the unsigned short fea variable.

This type of testing will happen every time the function moves to a new node in the array (decision tree) until an end node is reached. In the initial design, the unsigned short type variable was required for the program to know when an end node was reached. That functionality was later integrated into the unsigned int LC and unsigned int RC variables as will be explained later in this section.

![Figure 6: The initial and final design of the struct used in the C program.](image)

3.3.1 Exporting and Importing the Decision Tree structure from Python

Unlike the if-statement representation, this type of representation can however not be directly transformed from the initial Scikit-learn representation in Python to automatically generated code in C. In order for the information to be accessible in the C program, it instead requires manually created C code that can obtain the data from a common middle ground. The used data structure in Python has no equivalent data structure in C and therefore needed its information to be stored in a format C can import.

An easy format to both write and read to is the .csv format, which can represent the equivalence of each node as a single line. Figure 7 is an example of how this mapping between the data needed in the struct and the .csv representation would look. One of the reasons this file format is easily used is its resemblance to a list, which is an used data structure in Python and allow very flexible modification of the data in a quick manner.

In Figure 7, Feature is the feature to evaluate, Value is the threshold for the features value and TrueID/FalseID are the index of the nodes you should go to depending on the evaluated flow’s feature value. (Take note that the two first row in Figure 7 store meta data, as explained in Section 3.3.4.)

The .csv-file can then be read into the struct array defined in C with the aid of the stream reading function fgets and the C string token functionality strtok.
3.3.2 Optimizing the Array Structure

Doing a plain unmodified .csv-file representation did however immediately surface a few flaws with a 1:1 representation. The first apparent problem in the .csv representation was redundant lines. In the if-statement representation, every end node needed to be represented with a return statement inside the relevant if-statement. Such a thing would however not be necessary in a struct array representation where every node is just an index in an array. Unlike the if-statements, it would be no difference between a video classified end node at index 666 or 1337 in an array. As seen in Figure 3.3.2 B, there are several redundant elements in the array’s first layout. There are duplicate classification nodes at two locations. The indexation of the decision tree was therefore changed so all end/classification nodes would be stored in a single location instead. The beginning of the array is the optimal location for this, since it can not change between different instances of trees and is easily referred to. When a classification was found, it would refer to one of those nodes instead as seen in Figure 3.3.2 C. By doing so, the array’s size could be reduced by up to 50% compared to having an array element for each classification node as in Figure 3.3.2 B. (Note that transformation can go directly from A to C in Figure 3.3.2 and step B can therefore be skipped completely.)
3.3.3 Optimizing the Struct Itself

This array structure in Figure 3.3.2 C also lead to another possible optimization. As seen in Figure 6, there is a variable called unsigned short type in the struct. The intended use for this variable was to define whether the tree had reached an end node or not and in such a case, which classification. The C program would test the value of this variable in each node and return it when it reached a value within the classification range (between 0-1 in this case, since the only classes are video/not video). This variable could now be completely removed because the lowest indexes could now be used directly for classification instead. Instead of first going to a specific node and then look at the unsigned short type variable to find if it is an end node, it just need to look at the value of unsigned int LC/unsigned int RC. If the index found is within classification range, a class had been found and the examination can stop.

3.3.4 Storing Meta Data

With this method, the actual content of the first nodes also became irrelevant, since they are just mark placeholders for when to end a classification. This lead to another possible usage of the structs stored at these indexes. The program need to know certain information during initialization in order to represent and evaluate new packets correctly. For example the amount of rows is needed to specify the size of the struct array and the amount of features is needed to specify the elements to
read from a packet.

Such information could of course with relative ease be calculated while processing the .csv-file in C, but it would be more convenient if they were already calculated. Since there now exists unused elements at the start of the struct array, they could be used to store this information. While the execution time is of little importance during initialization, making it more efficient is not detrimental. With two positions in the array guaranteed to be unused, there is room for at least 8 meta data values without additional cost. The information would be stored during the creation of the .csv-file in Python and quickly accessed during data importation to the C program. Currently the stored information is the amount of rows, classification types and features in the tree. All these values were previously needed to be calculated by examining the .csv-file in a variety of ways to gather this information. An example of meta data information stored can be seen on the .csv-file row 1 in Figure 7. From left to right they represent: nodes in the tree, classification types and amount of features. The last zero is simply an unused element.

### 3.3.5 Branch Behavior Optimization

Lastly, some of the same issues as when using the if-statement representation could be found to some degree. The tree could still get in a situation when it can not decide the classification and there could occur redundant statements like Figure 4. Both of these issues could easily be cleared during the initial Python generation of the .csv-file. This was also much easier to implement than the if-statement version as well, since it was done on a list data structure instead of a text file.

The issue of uncertain classification was once again solved by randomizing which classification to pick. This was however done immediately when creating the .csv in python, as opposed to during execution of the C program. Randomization was done in this manner due to the limited evaluation time allowed during classification of a flow. Since a randomization call in addition to a return statement is more expensive than just a return statement, this would lower the amount of clock cycles used during the C code execution.

Redundant statements could also be removed during the creation of the .csv-file, since the nodes are done recursively. This was done by simply not creating a new line when the TrueID and FalseID list elements would be the same value. For such situations, it would instead return the ID-value to the previous recursive call. This also makes the optimization more efficient than the if-statement version as it did not require several iterations.

After all these tweaks were done to the struct array representation of a decision tree, some basic measurements was done for the needed storage size.
3.4 Optimizing Random Forest as a Whole

Once the decision tree optimization felt sufficiently good, the next step would be to create a representation for a whole random forest algorithm and optimizing it as well. The first step being to translate a random forest created in Scikit-learn to an equivalent representation in C. This was done in a similar manner to the single tree representation done in Section 3.3.

The random forest itself was not significantly more complex to represent than a single decision tree. There is no significant difference between a single decision tree and a random forest except that there are several trees in a forest. Due to this, the random forest could therefore be easily represented as a linked list [14] as seen in Figure 9. The reason the random forest was represented in this way in C was because of the size of the individual decision trees. As mentioned in the background chapter, the trees in the random forest are not necessarily of equal node size. If something like a matrix was used to represent the trees instead, they would have to accommodate the largest generated tree in the forest. Such representations would therefore leave unused elements for all trees with less nodes and therefore waste precious storage space.

It should be noted that the linked list could also have been done as an array, but that was not the initial choice.

```c
struct forest {
    struct forest * next;
    unsigned int rows;
    struct node tree[];
};
```

Figure 9: The struct used to create the linked list that represent a random forest in C.

In order to easily read the different trees in the random forest, they were all stored in the same .csv-file, but all trees were treated as if having their own indexation, classification nodes and stored meta data as seen in Figure 7. When the C software later imports the data, it can quickly distinguish the different trees from each other when the meta data is storing the amount of rows (nodes) in each tree. Instead of reading the .csv-file until the end, it reads the specified number of rows and then create a new tree in the linked list after each interval.

3.5 Chapter Summary

There were a few methods that was explored to test exporting and importing decision forest data from Python’s Scikit-learn to C. The existing method in Python was
found inadequate as it was made for smaller graphical representations as explained in Section 3.1.

It was then attempted to export a decision tree to C code as if-statements in Section 3.2. This proved to be a very good way to discover inefficiencies in the conversion from Python to C that could be cleared up, this include redundant if-statements and undetermined classification results. The implementation was too bulky to use practically.

The final representation that was tested and ultimately the one used was representing the decision forest as a linked list of tree arrays described in Section 3.3 and Section 3.4. This structure had the benefit of being very lightweight and the forest structure could easily be transferred from Python to C with a .csv-file as a middle man. In addition to the optimizations done in Section 3.2, the size could be drastically reduced by storing the end nodes in only one array location and exploiting their unused data to store meta data.
4 Performance Evaluation Setup

As the random forest is the machine learning method that should be used to evaluate the final results of this project, it was important that the random forest’s parameters were configured properly. Evaluating which parameter settings are the most optimal is not part of this study however. The parameters was therefore obtained through the concurrent report mentioned in related work [5]. The initial test parameters used in the random forest classifier was as follows: **RandomForestClassifier(n_estimators = 40, max_features = 10, max_depth = 20, n_jobs=-1, random_state=0)**.

According to the study mentioned, there were two parameters that could be considered more important for both random forest and gradient boosting trees. The most important parameters were **n_estimators**, which define how many trees the forest will have and **max_depth**, which is the maximum amount of node depth individual trees can have. The other parameters was not as important for the overall result and was therefore not a significant focus.

4.1 Setup Overview

The complete setup in this study can be divided into two components, one in C and one in Python. The Python program is only required for the initial decision forest generation and for translating it into a format readable by C. As for the C code, that is where the runtime performance matter and there the performance evaluation is done.

4.1.1 Creating a Decision Forest in Python

This task of creating the decision forest model in python is completely irrelevant for the runtime performance. It only provides the initial decision tree that is used, therefore it does not affect the performance of actually classifying real data flows.

The python component in this study is very simple. It will start by reading the training set provided in either .csv-file or .hdf-file format and then use either the **read_csv** or the **read_hdf** function imported from the pandas library extension included in the SciPy Stack [38].

The training set will then be used with either Scikit-learn’s **GradientBoostingClassifier** or **RandomForestClassifier** depending on which decision forest is desired. The different parameters configuring the decision forest can be defined, but also be left blank to apply the default setting for tree induction.

Once all the relevant decision forest parameters are configured, the created decision forest’s **estimators** attribute is translated into a list. During the creation
of the list, the decision trees are also optimized in accordance to Section 3.3 to better suit the C code implementation. The list is finally exported to a .csv-file with Python’s csv module (all trees are stored in the same .csv-file). An example of the file generated can be seen in Figure 7.

4.1.2 Classifying a Flow in C

This is the code where the runtime performance is relevant, but not for the whole program. Only the actual classification of flows is where there is any runtime performance constraint. It does however require several more steps than the classification itself to classify one or several data flows.

Everything starts by opening the .csv-file generated in Python that represent the decision tree. The file is opened with C’s fopen() function and parsed line by line. Meta data is stored in the .csv-file thanks to some of the optimizations done during the python exportation of the forest (explained in Section 3.3.4). This allow the memory allocation of the whole data structure to go a lot smoother. The C program can thanks to this very easily understand when to switch from one tree to the next as well as easily allocate the exact storage size needed on the heap for each tree with malloc(). If the meta data was not already calculated by Python and stored in the .csv-file it would have to be calculated before any storage space could be allocated.

The data structure itself is a linked list of the struct forest seen in Figure 9. Each individual tree is represented as an instance of the struct where the tree itself is stored in the struct node tree[] array. The internal index positions of the struct node tree[] array represent a node in the tree. Each element struct node in the struct node tree[] array contains the relevant feature, its constraint and the index addresses for the branch nodes. This can be read in more detail in Section 3.3.

Reading the .csv-file with the flow samples that are evaluated is significantly less complex than reading the decision forest. All the flows have the exact same structure and can therefore be stored in a Flows x Features-matrix of the long data type (or just an array if a single flow is to be classified).

Once both the decision forest and the flow sample(s) are read into the C program the classification process can start. To evaluate runtime performance, one measures the time at the start of the classification and the time at the end of it. The initial studies focuses on how to best get a reliable measurement and baseline from where the real performance evaluation can start.
4.2 Initial Studies

An important factor for the algorithms was the execution time, since it is very limited. The execution time was roughly estimated by evaluating 100 data packet flows from the training set. It was concluded that the classification time was done in between 4 and 20 microseconds. Further testings of classification time measurements was done in order to get a more conclusive execution time span.

All initial studies was done with the RandomForestClassifier configuration mentioned in the beginning of Section 4.

4.2.1 Collecting Consistent Measurements

It was concluded that testing 1,000,000 flows would be sufficient to give an accurate average execution time while also factoring in variation between flows and execution instances of the forest. There was however not 1,000,000 unique flows available in the training set, so the test sample of roughly 50,000 flows was repeated several times until 1,000,000 time measurements had been done. In addition to running the 1,000,000 flows, the test was done several times in order to see how it differentiated between executions of the classification application. This lead to a consistent average of values. They were values of 3.6-3.9 microseconds with unsigned longs and 3.8-4.2 microseconds for signed long values when evaluating a single flow. Since the training sets used will contain negative values, the signed longs methods will be the one used.

Originally there was an attempt to measure the maximum and minimum duration it took to evaluate a single packet flow as seen in Figure 10 A. This was done by measuring each flow evaluation individually and yielded results of a 1 microsecond minimum and 20-80 micro seconds maximum. The method in this test was however considered flawed due to 1 microsecond being the smallest measurable unit. It is possible that each test took slightly less or more than 1 microsecond and therefore did not give exact time measurements. Such unavoidable rounding errors would stack up in the long run and possible give a very incorrect total time for the 1000000 flows when each time was rounded. As a way to solve this, the attempt to measure minimum and maximum for individual flows was abandoned and further tests was done by first reading all the flows into memory first and measuring the total evaluation time in one bulk as seen in Figure 10 B. Doing the measurements in this way did indeed improve the accuracy of the measurements. Instead of the unstable time measurements hovering between 3.8-4.2 micro seconds, the time was consistently measured to be around 3.91-3.97 microseconds on average. There was some rare instances of 4+ microseconds, but it is likely the time was affected by external factors. Times of 4+ microseconds occurred less when unessential applications such as the Internet browser was closed.
4.2.2 To Refactor or not Refactor

It was also experimented with different levels of refactoring the evaluation into functions to see which is most efficient. It was tested if different levels of refactoring would matter for the execution time. The result was that refactoring both the individual tree and the forest as a whole into their own functions created better performance. Compared to not refactoring these instructions whatsoever, it saved around 0.1 microseconds. The biggest gain was when refactoring the trees themselves, which usually yielded at least 0.05 microseconds. Some rare instances where refactoring did not improve the results happened in a few of the many test runs, but this is probably caused for the same reason as the 4+ microsecond averages did in the previously mentioned paragraph. The slight improvements in performance occurred both when measuring the flows in bulk as well as when individually measuring them and then summarize the individual times (figure 10 A and B).

4.2.3 Using a One Vote Threshold or Two Votes

Another thing that was explored was how the program would calculate the total amount of votes in the random forest classification. Since it is a binary tree that is used, there are only two voting alternatives. Because of this, the voting can be calculated in two ways. It can either be done by keeping two separate variables as different votes and see which one is the biggest at the end or by having a single variable for only one of the classifications. That variable would then be compared to a previously calculated threshold that is required to get the majority of the votes.
This test was once again measured for testing all the 1,000,000 flows in bulk as well as measuring their times individually and then summarizing them. The results was not as easy to interpret in this time measurement however. Since the algorithm is called the random forest algorithm, it obviously means that a certain randomness is inserted into it. This caused the optimal system (using one or two votes) to change between different random forests despite that they had the same parameter configurations. Sometimes it was better to use the majority threshold, sometimes it was better to use a two voting system and sometimes it did not matter at all.

This test also displayed a big flaw with measuring the flow classification times one by one. The rounding errors would cause the second voting system to always be faster. For example, testing a two vote classification first would always cause the majority threshold voting to be faster. Testing the majority threshold voting first would cause the opposite to be true. Due to this unreliability of the measurements, all other result from this point onward was only measured in bulk to not cause as severe rounding errors. It was decided that the randomness between trees with the same parameter configurations was not severe enough to need additional testing and it instead should be focused on how different parameter settings would affect the overall classification time of the forest.

4.3 Storage Size Requirement

There were some initial tests done on a configured random forest algorithm to find a performance base line. Firstly, the size of the program was measured and the storage for the whole forest require roughly 2.5 MB of data. As the storage constrain in this project is around 50 MB, this was not a high priority in the initial testing stages due to the good margin.

4.4 Chapter Summary

After some initial tests, it was decided that the classification time performance should be done on 1,000,000 flows measured in bulk and than the average classification time should be calculated. The reason the classifications was measured in bulk was to minimize rounding errors. If the flows were measured individually each time, they would be at risk of having rounding errors in their time measurement and therefore lower the accuracy. It was also decided that there would be no performance evaluation of the C program size required as the initial result of roughly 2.5 MB required was sufficiently good for now.
5 Performance Evaluation

As the code used previously for the initial measurements in Section 4 was only the most basic and easily implemented decision forest evaluation possible there was much room for improvement on the code.

The code itself was however not the only important part of the performance evaluation. Since both the random forest algorithm and the gradient boosting trees algorithm was evaluated it was important to see which one would give the best ratio of runtime and classification performance. The best case scenario for this study would be that one of the two decision forest algorithms evaluated in this study perform significantly better than the other. The advantage of discovering if one algorithm being clearly superior in performance compared to the other is that all further tests can be solely done on the best one. The performance evaluation therefore starts with comparing the two algorithms and then focus on the one with the best balance between runtime performance and classification performance.

A last point of note is that the performance evaluation in this study in no way measure the classification performance of the actual algorithms and only focuses the runtime performance. Any mentions of classification performance results can be completely credited to the author of the concurrent study mentioned in Section 2.2.4 as well as read in more detail there.

5.1 Evaluation Details

During the performance evaluation the classification test itself remains constant for the algorithm, while the C code and optimization configurations does not. For each test 1,000,000 flows are classified and the average classification time is measured in relation to the decision forest algorithms’ amount of trees and maximum node depth.

According to the concurrent study exploring the algorithms’ classification performance mentioned in Section 2.2.4, there is a certain minimum value to these configuration parameters where anything lower will make the overall classification undesirably low. The results in said study also show that there is a diminishing return in accuracy increase as the parameter values get higher [5]. It was discussed with those involved in the study what a suitable range would. The goal was to define a range where the algorithms would not suffer from large inaccuracies in the classifications or longer runtime for insignificant classification accuracy increases. In short, higher or lower parameter values would compromise the quality of the classification time or accuracy to an undesirable degree.

For the random forest algorithm the amount of trees range from 10 to 80 trees
and the maximum node depth is between 10 and 40. It is important to note that only every second value is tested. This means that in mathematical terms the test ranges can be described as following:

\[
\text{Number of trees} = \{10, 12, ..., 78, 80\} \\
\text{Maximum node depth} = \{10, 12, ..., 38, 40\}
\]

The gradient boosting trees algorithm used a range of trees from 100 to 1000 in increments of 100 and every maximum node depth from 3 to 20. Similarly, the tested amount of trees and maximum node depth can be expressed as following:

\[
\text{Number of trees} = \{100, 200, ..., 900, 1000\} \\
\text{Maximum node depth} = \{3, 4, ..., 19, 20\}
\]

### 5.2 Execution Time Comparison between Algorithms

As there exist many different types of decision forest algorithms [55], it can be worthwhile to compare their runtime performance to see if the benefits of their classification performance outweigh negative impacts on runtime performance.

#### 5.2.1 Base performance

The tests started by comparing different configurations of the parameters with the most significant performance impact in a random forest. The parameters was again obtained from the other study mentioned earlier that use the same training set. Out of all the possible configuration parameters used in the Scikit-learn’s RandomForestClassifier, the most important ones was the depth of the trees and the amount of trees according to the other study. The range of the parameter values tested was to have 10-80 trees and 10-40 nodes in maximum tree depth. However, it was deemed necessary that only every second value needed to be tested to give good result. Doing tests in such manner was deemed enough to have a balance between total testing time and a good representation of the relation between classification time and the parameters. As there as only two major parameters involved, the relationship between time and parameters could be displayed in a plane graph as seen in Figure 11
The random forest was not the only algorithm tested, another tested algorithm was the gradient boosting trees algorithm. This algorithm was also implemented in Scikit-learn’s `GradientBoostingClassifier`. This algorithm’s generated data structure was slightly different than the random forest’s and therefore needed a slightly different code to be exported to a `.csv`-file. It did however not require any modifications to the C code to then read the `.csv`-file so no additional time tests or modification was needed on the code itself. Just as when measuring the execution time of a random forest, the relevant parameters were the amount of trees and the maximum amount of nodes depth. The relevant parameter range was again obtained from the other concurrent study, the constraints used was 100-1000 trees and 3-20 in maximum depth. In this test the tree parameter had an increment of 100 and all node depths in the interval was tested. As with the previous results, they were plotted into a plane graph as seen in Figure 12.
It is important to note that since both the random forest algorithm and the gradient boosting trees algorithm are types of decision trees, the underlying C code that classify the flows are exactly the same. This means that the resulting differences are more likely caused by the differences in the size of the decision trees and not the C code itself. As mentioned in the concurrent study in Section 2.2.4, the classification accuracy obtained show that they are relatively similar in performance, but gradient boosting trees are marginal better for the parameter ranges used [5]. Both the random forest and the gradient boosting trees algorithm uses ranges that give the best classification time and classification accuracy balance, but the better classification accuracy in gradient boosting trees is not significant enough to justify a runtime roughly 8 times as high.

This means that the trade of does not seem worth it in the unoptimized case, but it is also important to compare the runtime performance after higher compilation optimization is applied since that is how the final product will be constructed.
5.2.2 Algorithms at Compiler Optimization Level: -O3

The baseline classification time measurements of the random forest and the gradient boosting trees algorithm was done with the default configurations of the GCC compiler [49]. In short, there is a high potential for compilation optimization in the software, that is most likely something that will be used in the final product. Because of this, there was tests done on how some of the possible compilation optimization options affect the algorithms’ performance.

The two compilation options used was -O3 and -Ofast. The -O3 flag is the highest standard optimization in the GCC compiler. This means that it applies all the possible standard optimizations in the GCC compiler. Standard in this context means optimizations that will work on all programs using the language standard. The -O3 optimization flags are also not single optimizations, but collections of several optimization flags that all will be enabled. Using -Ofast on the other hand means that some non-standard optimizations will be done and it is not guaranteed to work in all environments.

There is far to many different optimization flags to go into individual detail, but some of them are -finline-functions, -fmerge-constants and -foptimize-strlen. The -finline-functions optimization tries to inline (unrefactor) a function to remove the need for a function call [50]. The other two optimization flag examples should be obvious due to their names. One merge constants with the same value and the other optimize the C string functions.

The improvements when using higher degrees of optimization can be seen in Figure 13-14. These figures show that regardless of algorithm, there is a performance boost in time by using higher degrees of optimization. It is however shown that the increase in performance of the gradient boosting tree algorithm random forest algorithm is greater relative to the. For the random forest algorithm the execution time is decreased by less than 10% (around 1 microsecond) while the gradient boosting trees algorithm has a decrease of around 50% (around 50 microseconds).

It is clear that there is a significant improvement in the gradient boosting trees algorithm’s performance, but it is however still not fast enough to be worth using.
Figure 13: Baseline performance for random forest, but with GCC compilation optimization flag `-O3`.
5.2.3 Algorithms at Compiler Optimization Level: -Ofast

The **-Ofast** optimization configuration is very similar to the **-O3** configuration except that it also uses non-standard C optimizations in addition to those in **-O3** optimization. Non standard in this context mean that it does not follow all the common standards of C programs and is therefore not guaranteed to always work as intended [50]. It can however be faster in some cases.

This is however not the case for either the random forest algorithm or the gradient boosting trees algorithm. The random forest results seen in Figure 15 is not distinguishable from those using the **-O3** flag seen in Figure 13. For the gradient boosting trees algorithm the execution time is even worse where the measure time in Figure 16 is actually worse than when using the regular **-O3** optimization. It is clear that the execution time can go as high as around 60 microseconds with **-Ofast** optimization, while it only reached around 55 microseconds using **-O3**.

Due to the results in 13-16 it was concluded that no further testing on runtime performance for gradient boosting trees should be done. The classification

![Classification time relative to trees and node depth](image)
performance gain was simply not significant enough to justify the lower runtime performance. As the C code for classification is the same regardless of if it is random forest or gradient boosting trees, the gradient boosting trees algorithm would likely always be significantly behind in classification time. This would most likely be true even after significant optimization of the C code itself.

Figure 15: Baseline performance for random forest, but with GCC compilation optimization flag -Ofast.
5.3 Improving the Runtime Performance

The following group of tests was therefore only measured using the random forest algorithm. As there was no further algorithm tweaking done, the tests was completely focused on improving the internal programming structure created in C to make the whole flow classification process run faster.

5.3.1 Only Evaluating Enough to Prove Vote Majority

The most notable thing about the vote structure in random forest is that you not necessarily need to make all the trees do a classification. It is enough to let trees cast a vote until one classification type has obtained an incontestable amount of votes. For a two class system like in this study, it would simply be enough if one class obtain above half of the votes. By testing how many votes each classification type has after a tree has casted its vote, the needed amount of classification could, while unlikely, theoretically be lowered by up to almost 50%. The classification time needed after this change can be seen in Figure 17. It can be seen that the time was...
reduced by around 2 microseconds.

Figure 17: Random forest implementation, but breaking classifications early whenever a classification type has >50% of the votes.

5.3.2 Storing all Vote Results in an Array

Another possible change in the C code was classify all the 1,000,000 data flows tree by tree instead of classifying each flow with the whole forest before moving onto the next flow. Each individual flow’s total voting results would all be stored in a large array instead. All the flows would be given a final classification at the end instead of getting them one by one.

There was two expected main advantages to this type of code redesign. Firstly, by storing all the votes in a large array instead of continuously call a function to evaluate and return the forest end result for a flow, the classification result would be more speedily accessible due to being a local array. The other advantage was that the random forest would only need to be looped through once as each tree will do all the classifications needed before the forest moves to the next tree. This is also expected to speed up the classification time. The resulting performance can be seen in Figure 18 and show similar results Figure 17.
5.3.3 Combining Result Array Storage and Halting at Vote Majority

As the results show in Figure 17 and Figure 18 both individually display improvements in the execution time performance, the next step was obviously to try combining the two improvement variations. This lead to the further improved results seen in Figure 19 below.
Figure 19: Performance result of combining breaking at >50% votes and storing the votes in an array. (Figure 17 and Figure 18.)

While the combination of majority breaking and result array storing created another clear improvement, it was also not possible to completely stop evaluating individual flows even when a classification majority was achieved. If the flows in the array was completely classified, it would still be in an unknown location in the array and therefore each following tree would still want to classify the flow. In order to prevent unneeded classifications, an if-statement had to be done at each tree iteration. This resulted in an additional amount of needed if-statement checks according to the following formula:

\[
\text{Additional\_checks} = (\text{Evaluated\_flows}) \times (\text{Trees\_in\_forest})
\]

It was still proven better than going through with the classifications as Figure 19 display better results than Figure 17 and Figure 18.
5.4 Performance Classifying Realistic Flow Type Ratios

In all the previous measurements, the 1,000,000 flows used was simply the training set repeated until 1,000,000 flows was classified as mentioned in Section 4.2.1. At this point it was decided that all the previous random forest tests should be redone with a new data set that consisted of 1,000,000 unique flows as opposed to repeating the training set until 1,000,000 classifications was done. The new set of flows would be having flow types distributed in a manner much more similar to what has been observed in reality. The training set had a ratio of around 50% video and 50% non-video, while the new would have around 5% video and 95% non-video flows.

The reason all the tests should be repeated was to better guarantee the credibility of all the results to be as high as possible. By using flows better reflecting reality the measured performance results was more trustworthy.

The new results seen in figures 20 to Figure 25 show that the default compilation configuration performed significantly worse reaching almost 15 micro seconds while the test repeatably classifying the training set only was around 10 microseconds. However, the performance results did not show significant change compared to using the training set once higher optimization configuration was enabled with the optimization flags -O3 or -Ofast. If one observe the graphs carefully, there exist a very slight overall increase in classification time. It is almost unnoticeable without going back and forth between the images.
Figure 20: Base performance of random forest, but with unique flows. (Note that the time-axis scales to 15 instead of 10.)

Figure 21: Optimization with -O3, but with unique flows
Figure 22: Optimization with -Ofast, but with unique flows.

Figure 23: Breaking when classification type reach >50% of votes, but with unique flows.
Figure 24: Storing the votes in a local array, but with unique flows.

Figure 25: Combining Local array storage and breaking at 50% of votes, but with unique flows.
5.5 Further Optimization Attempts

After the repeated tests was done, the focus once again turned to optimizing the code structure for the algorithm. Since there was no noticeable difference between classifying flow more representative of reality and classifying the test set, no further tests was needed in that regard.

The reason being that there was no noticeable problems even when classifying flow more representative of reality if high levels of complicator optimization was done.

5.5.1 Sorting According to Majority

As mentioned earlier in regards to combining the method used in the previous section gave the problem that flows could not be permanently excluded from the array once they had reached majority. Instead an if-statement had to repeatedly check if majority was already reached for each flow/tree combination. In cases where there indeed was a majority, there was no need to do a classification. The primary reason for this was that the flow results in the array was not ordered in any predictable manner where flows reaching majority could be excluded.

There was an attempt to exclude flows completely from the classification once one of its classification types obtained the majority of the votes. Since the problem was that it was unknown where in the array that results with majority was, the array was sorted in accordance to Figure 26. In Figure 26 A it display an example how the voting result with and without >50% of the votes could be stored while in the middle of evaluation flows. Figure 26 B illustrates the detection of a new flow that has reached majority and Figure 26 C displays the result after the resulting array position swap. By doing this, there would be a clearly defined region in the array that could be defined as classified and therefore would no longer need to be evaluated.
Figure 26: Example of how to reorder the samples and vote results once a flow classification vote has reached majority.

A possible issue with this is that both the flow array itself, the votes for video and votes for non-video all has to be repositioned accordingly or the mapping would be incorrect. Since there was 1,000,000 samples, this would result in a total of 3,000,000 swaps. It was ultimately proven by the results displayed in Figure 27 that the act of swapping was more harmful than simply testing if a flow should be ignored every time.
5.5.2 Adapt Code for Realistic Flow Distribution

One interesting effect of using data distributed as it is expected to be in reality is that it is only 5% video flows. This type of clearly tilted distribution can actually be used to slightly simplify the vote counting and testing for vote majority. Since there is so little video traffic it can possibly be more efficient to only count and test majority for votes going toward non-video flows.

Up until this point all tests looking has kept count of both video and non-video votes and monitoring if either achieve >50% of the votes so a tree classification can be skipped. The majority test has been done for both video and non-video alike every time a vote is casted. This is fine when the distribution is roughly 50-50 like the training set, but when one type has a clear majority compared to the other it is possible to waste more time on false positives than true positives as the below example will show.

If a tree has for example 90% accuracy, then it means that 10% of the time it will give the wrong classification. Lets assume that this is applied to our test data. The amount of incorrectly classified flows will then be 100,000 flows (10% of 1,000,000),
where around 95,000 will be non-video flows classified as video and 5000 will be video flows incorrectly classified as non-video. Out of the correctly classified flows we have 855,000 non-video classifications and 45,000 video flows.

This is of course not guaranteed to be the case as a sufficient high accuracy for the individual trees will remove this issue. The equation for when correct classifications will be equal or greater than the false classifications for a binary tree can be expressed as following:

\[ Class_1 \times True\_positive \geq Class_2 \times (1 - True\_positive) \]

Or more simply:

\[ Class_1 \times True\_positive \geq Class_2 \times False\_positive \]

By inserting the 95% and 5% distribution flow traffic data set distribution into the equation, it is found out that the true positive chance needed is \( \geq 95\% \) to get at least as many false as true positive video classifications. The individual random forest trees are not implemented for their individual accuracy however and therefore not likely to have the high accuracy needed. It should therefore in theory be more efficient to not check for video vote majority at all as it should make more checks on incorrect classifications than correct. Doing so will also not cause an issue for the final classification because it can be deduced completely by seeing how many non-video votes a flow has. The C code for this variation is almost identical to that of the non adapted code used in Section 5.3.3, with the only difference being the removal of the votes for video.

Despite the theoretical likelihood that it would save time to only count non-video votes, the practical execution time measured in Figure 28 is telling a different story. There is in fact a slight execution time increase seen in Figure 28 compared to the results in Figure 25 where both video and non-video votes are counted. To make sure this was correct, both measurements was repeated two times, which lead to indistinguishable graphs. A difference could only be seen if one carefully observed the actual point values. While not guaranteed to be the case, this most likely imply that breaking the video flow classifications early when they reach majority is saving enough time to offset the false positive video votes and majority tests that occur.
5.5.3 Majority is Only Relevant After Half of the Forest

All tests up until now has been testing for majority any time a classification from a tree has been done. This is however not necessary as a proven majority can only be achieved once 50% or above of the trees has voted. This can very easily be changed by dividing the classification process in two parts. In the first half of the classification the trees does classifications with no checks for voting majority afterwards, since it is impossible to have majority yet. As soon as the forest has gone past the mid point, the code will move into another loop instead where it will start checking for majority. By dividing the classification in two steps, the amount of times unnecessary checks for majority occur will be decreased. The result of doing this is seen in Figure 29.

This performance evaluation gives the same type of results as the two evaluations before it. In theory the change sound like it can has some promise to improve the performance, but in reality it turn out that the trade of is not worth it. In this case the time it takes to do two separate classification loops for the decision forest is taking longer than the act of doing the unnecessary checks for majority condition.
A possible factor to why no time was saved can be that the forests are relatively small and therefore the C program does not do a lot of redundant condition checks.

Figure 29: Combining Local array storage and breaking at 50% of votes and sorting of the array.

5.6 Chapter Summary

It can be seen in Section 5.2 that the random forest algorithm has clearly superior classification time regardless of compiler optimization level. Since this was the case, all performance evaluations in regard to C code changes was only done with the random forest algorithm. The decision tree structure of the two algorithms was identical so it would not be beneficial to evaluate both more than necessary.

The results in Section 5.3 showed that the classification time could be improved by stopping classifications early by if one of the types had obtained the majority of the votes. It could also be improved by evaluation several flow in a local array as opposed to classifying flow one after another. Combining these two methods gave even better results.

There was also some changes to the code that was not beneficial as show in Section 5.5. Not applying majority break on video flows was not beneficial despite that they only represent around 5% in real flows. Dividing the classification in two
parts to only check for majority break after half the forest was also slower and so was sorting the vote result array after majority.

Finally, Section 5.4 showed the classification time of packets was not affected by whether it was the training set used or a data set with realistic flow type distribution that was classified. It showed a difference when not using higher levels of compiler optimization, but since the software will use the more optimized compilation, this is irrelevant to the whole picture.
6 Conclusion and Future Work

This section reflects on the results from Section 3 to Section 5 to give increased insights and conclusive answers in regard to the goals of this study. The goals in this case was relatively simple, one should export a decision forest created in Python's Scikit-learn extension and make an as efficient as possible equivalent implementation in C. Since the decision forest itself is identical between representation, it was the runtime performance in C that should be optimized. There are a few key components that seem to affect the runtime performance.

An important note is that there is no deeper evaluation or discussion about the decision forest’s required size despite that is is one of the runtime performance aspect to optimize in the project. The reason for this is that it was judged to be less critical than the classification time and was therefore put in the back.

6.1 Efficiently Moving the Forest from Python to C

When it comes to exporting and importing data from Python to C there is actually an important point to make that not necessarily is apparent when reading Section 3. It is not directly affecting the runtime performance itself, but after the preparatory effort needed.

In Section 3.3 it is clearly explain how there are several changes done to the decision tree structure before it is stored into a .csv-file so it is optimized already when it is processed by C. This not the only way to do this though. The decision trees could just as well be stored unmodified in the .csv-file and then optimized in C.

This might not seem like a big deal at first glance, but have huge impact on how much work it requires. For example, how the size of a tree is calculated in Python and C. In Python it is very easy, you just put the tree nodes in a list and then call the \texttt{len()} function. In C on the other hand you have no \texttt{len()} function, The language is very hands on and you have to create the \texttt{len()} yourself. It also isn’t just to put the tree in a dynamic list as no such thing exist and the tree can obviously not be put in an array without already knowing the size. The point of this example is to show that while everything done in Section 3.3 can just as well be done in C, it would be much more time consuming programming wise and therefore undesirable.

Another thing that can be analyzed here is the difference between using Python to generate the forest in C-code and exporting the forest into a common file format that can then be imported into a C-code structure. From the experience gained during this study, it seems that using auto-generated C-code is the fastest way to
translate the decision forest structure from Python to C-code. This does however comes with the flaw that it is harder to modify the structure of the trees in the forest. To efficiently do this, the code had to be changed post generation with another piece of code and was still relatively rigid. Modifying a text file of generated C-code was a relative tedious task despite the fast initial code generation.

When using custom made C-code it was more work to both export it from Python and Importing it into C. It however also gave more much more control over the situation and allowing several modifications to the decision forest. In this project where one wants to make optimization changes in a structure modification flexibility is much more important than an easy data structure translation.

So in summary, while it is not directly affecting the runtime, it is also important to consider which language you do the preparations in. Making a custom export and import method seems to be desirable as it allow more control when it comes to modifying the code structure and therefore optimization flexibility.

6.2 The Performance Evaluation

In regard to the performance evaluation itself, there was a several aspects were evaluated regarding the classification time. These include the difference in classification time between the random forest algorithm and the gradient boosting trees algorithm, decision forest performance for different optimization options and how changes to the C code implementation affected the classification time.

6.2.1 Gradient Boosting Trees or Random Forest

As seen in Section 5, this study do a runtime performance comparison between the random forest algorithm and the gradient boosting trees algorithm. It was of interest to see if the increased classification performance of the gradient boosting trees algorithm was worth it relative to its runtime performance.

The results show clear evidence that random forest is the best choice for an environment where time is of the essence. Even with the classification performance measurements obtained from the related concurrent study in Section 2.2.4, it is clear that a small increase in classification accuracy is not worth the high increase in classification time. Even with higher optimization configurations for the GCC compiler, the performance achieved for the gradient boosting trees was not more than 40,000 flow classifications a second.

The random forest algorithm was already at least 2.5 times faster (100,000 flows a second) and had a similar, but slightly lower classification performance. For a study that attempt to find a decision forest implementation in C that is as fast and
small as possible as long as the classification does not suffer severely, it was clear that random forest would be the chosen algorithm.

It is possible to decrease the classification time further for both the gradient boosting trees and the random forest by not using the upper parameter bounds tested. For example, choosing to use only 20-30 trees in the random forest can bring the classification speed to 500,000 flows each second. This would only have a very minor effect on the accuracy, and be well within the suitable operating range as discussed in Section 5.

Lowering the gradient boosting trees algorithm’s parameters so that the classification accuracy get on par (instead of slightly better) with the random forest would however still not make it a better option. Even in the best case scenario, it would only be obtaining a similar performance as the random forest. That would not make any improvements for the runtime performance.

6.2.2 Optimization options

When observing the performance evaluation of the C code implementation itself, it can be seen that there are clear improvements obtained. The initial results using the training set showed that there was little difference between the default compiler optimization and the more advanced `-O3` and `-Ofast`. This was however not the case when using flow feature data that better reflected the real world. In this case the difference in classification time was significant.

Another important deduction from the compiler optimization is that for the lower level of optimization, there can be a huge difference in execution time depending on how the flow feature set that is classified look. There was however no significant difference between the two different flow feature sets, regardless of if it was a real collection or the training set. This can be an indication that the trees themselves are well balanced. If the highest levels of optimization display little difference in classification time it can be assumed that the tree itself is relatively stable and the previous instability was due to the C code. To be sure of this, it would of course be needed to use several more data set classifications evaluation to see how different distributions affect the classification time. It should however not be necessary as one of the flow features sets reflected that of reality and therefore should behave as it is expected to in the real world.

6.2.3 C code structure

The results for the different C code version are quite mixed. Some of the changes, such as majority break condition and storing the vote results in a local array have significant impact on the performance. Other changes did however not have any
positive impact at all and was instead detrimental to the classification time.

It is important however to understand that the two changes mentioned above have a lot more significant impact than the other changes done. There is a trait that differentiate these changes from any other changes to the C code. They affect the overall structure of the C code.

For example, when introducing the Majority break condition, the program is no longer using the whole random forest to classify data flows. With this change it will just use enough of the whole forest to draw a conclusive result. You no longer have to go through the whole forest to reach the end point. Storing the vote results in a local array also affect the overall structure of the program. The program no longer run each flow through the forest, instead it slowly marches all the flows through each tree in the forest simultaneously and keep all vote results closely accessible.

The results that gave no improvement to the runtime performance did however not cause any significant changes to the structure. All that was done was trying to optimize them. Sorting the local result array means that the program still slowly marches all the flow through it simultaneously. It just made them also march in an order. Ignoring majority break for certain intervals or flow types is the same. The program still uses majority break conditions, it just does ignore some of the situations.

So in summary, there was some code changes that affected the larger structure of the classification that greatly improved the results as seen in Section 5.3. Any further optimization to these changed did however fail to improve classification time as seen in Section 5.5.

6.3 Future Work

While quite a bit of performance evaluation has been done in this thesis, there are several aspects that have been briefly mentioned as potential additional work items.

6.3.1 Decision Forest by Vector Calculation

In this project it was briefly mentioned that the runtime performance could be possibly improved by using Vector Calculations. This is a type of calculation where the operations can be done in bulk instead of one by one.

For example, the two groups A and B of n values can be added together by running a loop n times. If one were to use vector calculation instead, it is possible to do the following:

1. Read and decode instruction

2. Get group A
3. Get group B
4. Add their numbers together
5. Place the resulting group here

Some real examples are the SSE2 [7, 20] and MMX [6, 20] instruction sets. However, it is not apparent how to integrate the use of vector calculations to parallel tree traversals.

6.3.2 Performance Evaluation on Collecting Flow Data

This dissertation focuses specifically on the classification time of the runtime performance. The flows used for performance evaluation was already calculated when they was obtained. If the system was to be used in a real environment this would not be the case. All flow in the real world is collected as a sequence of packets. These packets would need to be combined into a flow before they can be classified.

The time it takes to calculate the flow features would also need be taken into consideration to give realistic total classification time from start to finish. Even if the results in this dissertation seems to display capacity to classify with a magnitude of $10^5$ flows each second, it is the total time that counts in the end. All would be for naught if the total time required turns out barely allowing classification magnitude $10^4$. To gain insight in the runtime classification performance as a whole it is therefore important to evaluate the time it takes to collect a flow’s data in addition to the time it takes to classify a flow.

6.3.3 Optimizing the size of the decision tree

Since no deeper attempt to optimize the storage size requirement was done except an initial measurement mentioned in the small Section 4.3, this could be explored future. The upper limit given for the storage size of the decision tree was however set as 50 megabyte and only around 2.5 megabyte was used. With such a big margin it is probably better to focus on the classification time as of now.

6.4 Concluding Remarks

This study show that it is in fact possible to create a feasible implementation of a decision forest in C when it comes to runtime performance. According to the results in Section 5 the initial unoptimized performance of the algorithms was not that great. It was however apparent that the random forest algorithm as the go to method because gradient boosting trees was more than 10 times slower in the base performance part of Section 5.2.
It is however clear that higher compiler optimization options affected the gradient boosting trees more as well as random forest. The effect on random forest could however not be seen when using the training set as Section 5.4.

Finally it is also clear how important the classification structure is as Section 5.3 show classification time reductions of almost 50%. It is however also demonstrated that theoretically feasible optimization are not guaranteed to give a better result in Section 5.5.

The study does however only evaluate part of the whole picture however, thus making it possible that more optimized implementations of a decision forest exist. There are several parts of the implementation that remain unexplored as mentioned in Section 6.3.
References


