Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Improving algorithms in phylogenetics using machine learning

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

Bryan Versendaal

Delft, the Netherlands
September 2020

MSc THESIS APPLIED MATHEMATICS

"Improving algorithms in phylogenetics using machine learning"

BRYAN VERSENDAAL

Delft University of Technology

Responsible professor

Prof. dr. K.I. Aardal

Daily Supervisor

Dr. ir. L.J.J. van Iersel

Other thesis committee members

Dr. J.L.A Dubbeldam

Dr. M. Jones

September, 2020

Delft

Summary

This thesis is about the application of machine learning within phylogenetics. This was a proof of concept to see if problems within phylogenetics could benefit from machine learning. We looked at three different problems that could benefit from machine learning, namely the maximum agreement forest problem, the tree-child hybridization number problem and finally the tail move problem. To figure out if any of these problems could benefit from machine learning we used a decision tree to create a simple model with predictive capabilities and data that we generated ourselves using Python and combined this with existing algorithms.

The first section gives a brief introduction on phylogenetics in general. This is to give a better understanding of the field of study, with some concrete examples. The second section introduces the concept of machine learning. It gives a brief historic introduction and shows some of the learners that could be used to solve decision based problems. The following three sections each introduces one of the three problems. Namely, the agreement forest problem, the tree-child hybridization number problem and finally the tail move problem. Each of the problems shows a different aspect of phylogenetics. In each case, the results show that the problems could possibly benefit from machine learning.

The results on the maximum agreement problem showed that it did not really benefit from our current application of machine learning, however, intermediate results show some promise that lead us to believe that there might be some benefit from machine learning perhaps using different machine learning techniques. The tree-child hybridization number problem showed better results. In all the instances presented to the decision base algorithm we saw that it performed as well or better than the current fixed parameter tractable algorithm. For this problem it was crystal clear that it could benefit from machine learning, because it already benefited greatly from a simple decision tree. The results for the tail move problem were not as good as we hoped. The heuristic is still plagued by some randomness that we have not solved yet. However, the result did look promising in the sense that if we could overcome some of the hurdles still present we certainly believe that the problem can benefit from machine learning. The problems in phylogenetics do seem to benefit from machine learning, but it requires more complicated learners to be able to achieve true consistent improvement.

Preface

This thesis is the product of hard work, dedication and most of all enjoyment in the subject. When I started this project I was not really familiar with the concept of machine learning, but working on this thesis has broadened my horizon greatly and showed me much about machine learning.

I have learned a lot the past months and I have found it very enjoyable to see progress being made. However, I could have never done it all by myself and that is why I want to thank my daily supervisor Leo van Iersel and Mark Jones who also became somewhat of a daily supervisor. The weekly meetings were a valuable source of insights and ideas which helped me greatly in progressing in my thesis. Finally I would also like to thank Remie Janssen who shared some value insights for the last problem that I have studied.

I hope that the thesis will be an enjoyable and interesting read!

My best regards,

Bryan Versendaal

Contents

Introduction

Phylogenetics is the study of evolutionary relationships among biological entities, which are often species or genes. Within the field of phylogenetics these are often referred to as taxa. The evolutionary relation information is often captured in a tree, which is referred to as a phylogenetic tree. The most basic tree that can be considered is a bifurcating tree. This is a tree where every branch splits into two subbranches. When the relations get more complicated the tree is replaced by a phylogenetic network.

Phylogenetics holds many interesting mathematical problems, one of which is the maximum agreement forest problem. Many fixed parameter tractable algorithms are known to solve this problem, for example [11], which solves it in $\mathcal{O}(3^k n)$. In this report we will also consider the tree-child sequence problem and the tail move problem. Machine learning will be introduced to try to improve the performance of the algorithm mentioned in [11].

The tree-child sequence problem also has fixed parameter tractable algorithms. For example, existing algorithms are HYBROSCALE [1, 2], TREETISTIC [7] and PIRN [12]. These algorithms can only handle a small number of input trees and reticulations events. The algorithm that is studied in more detail in this report is created by Van Iersel et al. [5], which has a runtime of $mathcalO((8k)^k \cdot poly(n,t))$ and can handle a larger input size. Here machine learning is introduced to improve both the runtime of the algorithm and the size of the search space.

For the tail move problem a heuristic is used, which was introduced by Janssen et al. [6]. This heuristic gives an upper bound on the length of the tail move sequence. Currently, there exists no algorithm that solves this problem. Here we introduce machine learning to improve the length of the tail move sequences found by the heuristic.

The way all three problems present themselves gives rise to the idea of trying to introduce machine learning into the algorithms and heuristic. The aim of this thesis is to create a proof of concept. We want to show that the problems can benefit from machine learning by creating a simple decision tree model that will already improve the run time, search space or solution. In no way is it our goal to create very intricate machine learning models that take days to train. We want this to be the stepping stone for further research into this area.

The rest of the report is structured as follows. The first section will introduce the concepts of phylogenetics and show some examples of what phylogenies are. The second section will give a brief history on machine learning. We take a look at what classifiers are and look at some very early decision models before we introduce the decision tree that we used. Finally in that section we will also discuss how to handle data that is used to train the model. The third section will introduce the maximum agreement forest problem. The data generation and growing of the decision tree and the results will also be discussed. The fourth section will introduce the tree-child hybridization sequence problem and will also discuss the results regarding the problem. The fifth section will introduce the final problem, namely the tail move problem and will also discuss the results regarding this problem. Finally, a closing conclusion will be given with a short summary on all the problems and a few final remarks. All the code used in the report can be found on Github: https://github.com/TUbryan/PhyloThesis

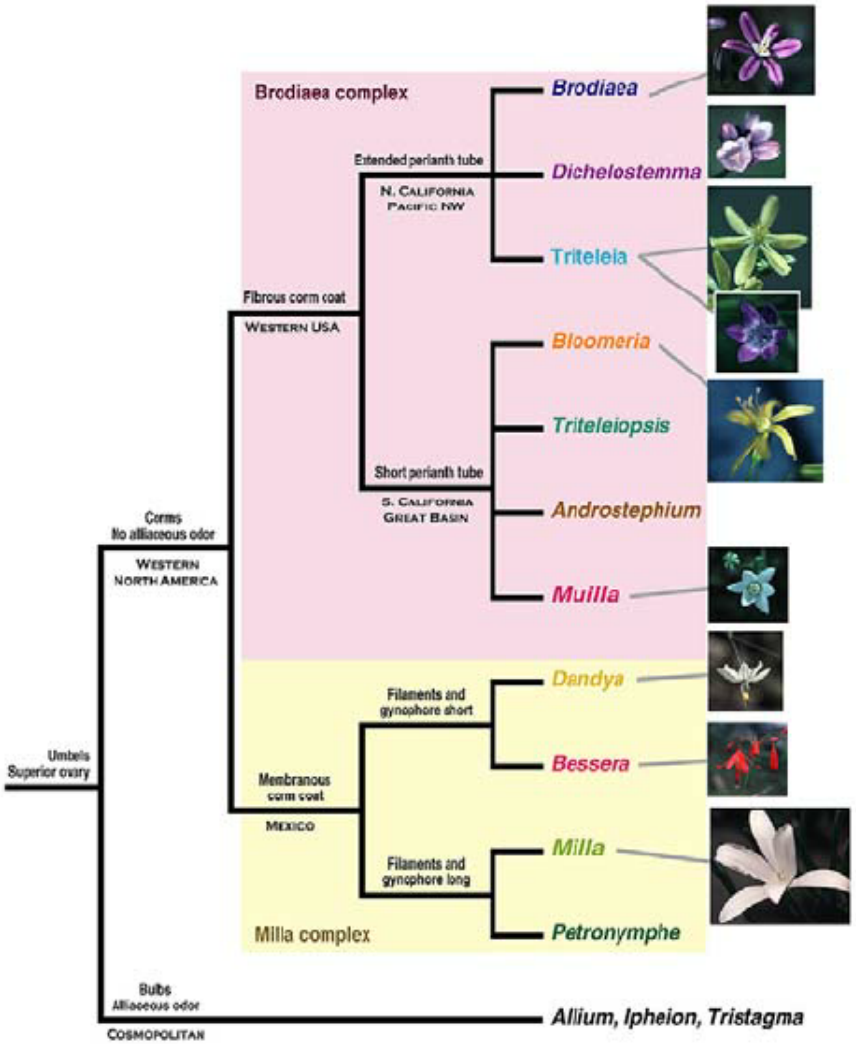

FIGURE 1. A phylogenetic tree depicting the Brodiaea and Milla complex within the Themidaceae flowering plant family. Adapted from Pyres and Sytsma, 2002.

1. Introduction to phylogenetics

As mentioned in the introduction, phylogenetics is the study of evolutionary relationships among biological entities. We want to capture these relations in clear structures. To this end we will use phylogenetic trees. An example of a phylogenetic tree can be found in Figure 1. Most of the times a tree is not good enough to represent evolutionary relations. For example, it can occur that two different species together create an entirely different species or within genes there exists something called lateral gene transfer. This results in a new type of phylogeny, namely a phylogenetic network.

This section will more formally introduce phylogenetics. We will start by defining terminology that will be used throughout this thesis. Afterwards we will introduce some moves that can be performed on phylogenetic trees and networks. Finally we will show commonly used way to represent phylogenetic trees, namely the newick string representation.

The species or genes that are studied are often referred to as taxa. We denote by X a finite non-empty set of taxa. A phylogeny or evolutionary tree/network represents the evolutionary relations between taxa. We will now introduce some basic definitions.

1.1. **Basic terminology.** The following definitions can be found in any graph theoretic book.

Definition 1.1. A *graph* is a pair $G = (V, E)$ where V is a set of elements called vertices and E is a set of pairs of distinct elements from V called edges. A *directed graph* is a graph in which the edges have an orientation.

Definition 1.2. A *tree* is a graph in which any two vertices are connected by exactly one path. A *forest* is a disjoint union of trees.

Definition 1.3. A *leaf node* is a node in a tree, which has in-degree one and out-degree zero. A *root node* is a tree node with in-degree zero and positive out-degree. A tree with exactly one root is called a *rooted tree*.

In many problems we consider a special kind of tree, namely the binary tree. A binary tree is defined as follows.

Definition 1.4. A *binary tree* is a tree where each node has at most in-degree one and at most out-degree two.

These terms will be used throughout the report and form the basics we need to explore the world of phylogenetics. Using the introduced terminology, an exact definition can be given of a phylogeny. The following definitions will all regard to phylogenies and supporting terminology.

Definition 1.5. A *phylogenetic tree* or *phylogenetic network* on a set of taxa $X' \subset X$, is a tree or network N for which there exists a bijection between the leafset of N and X'.

In many cases a bifurcating or binary phylogenetic tree is considered. This is simply a phylogenetic tree with the properties of a binary tree. Some algorithms regarding phylogenetic trees that will be discussed in this report require some extra terminology to be introduced.

Definition 1.6. The *depth* of a node s in a phylogenetic tree N is the length of the path from the root node to s.

Definition 1.7. We call $t \in N$ a *child* of node $s \in N$ if t is attached to s with $\text{depth}(t) = \text{depth}(s)+1$. We call t a parent of s if s is a child node of t. More generally, we call t an ancestor of s if there exists a path from t to s using only child nodes of subsequent nodes.

Definition 1.8. A *cherry* of a phylogenetic tree N is a leaf pair (a, b) where a and b have the same parent.

These are the most basic definitions regarding phylogenetics that will be used throughout this report. Other terminology will be explained when necessary. In the rest of this section some important moves that can be performed on trees or networks will be introduced that will be used in the studied problems.

1.2. **rSPR moves and Tail moves.** The Subtree-Prune-Regraft and Tail moves form only a small part of possible moves that can be performed on phylogenies, but are commonly used. After the moves are introduced some examples will be given to get the reader comfortable with the techniques.

1.2.1. *rSPR move.* The Subtree-Prune-Regraft-move revolves around detaching, also called pruning, a subtree for the phylogeny and reattaching, also called regrafting, it back in the tree in another location. Though it should be clear what a subtree is, for completeness the definition is as follows.

Definition 1.9. A *subtree* T of phylogenetic tree N, is a tree such that $T \subset N$ holds. Where $T \subset N$ means that T can be obtained from N by removing nodes from N.

The rSPR move starts by selecting a subtree within the phylogenetic tree. After the subtree is pruned it is moved to a new edge within the phylogenetic tree. A new node is created to which the subtree is attached. Figure 2 shows how to perform an rSPR moves.

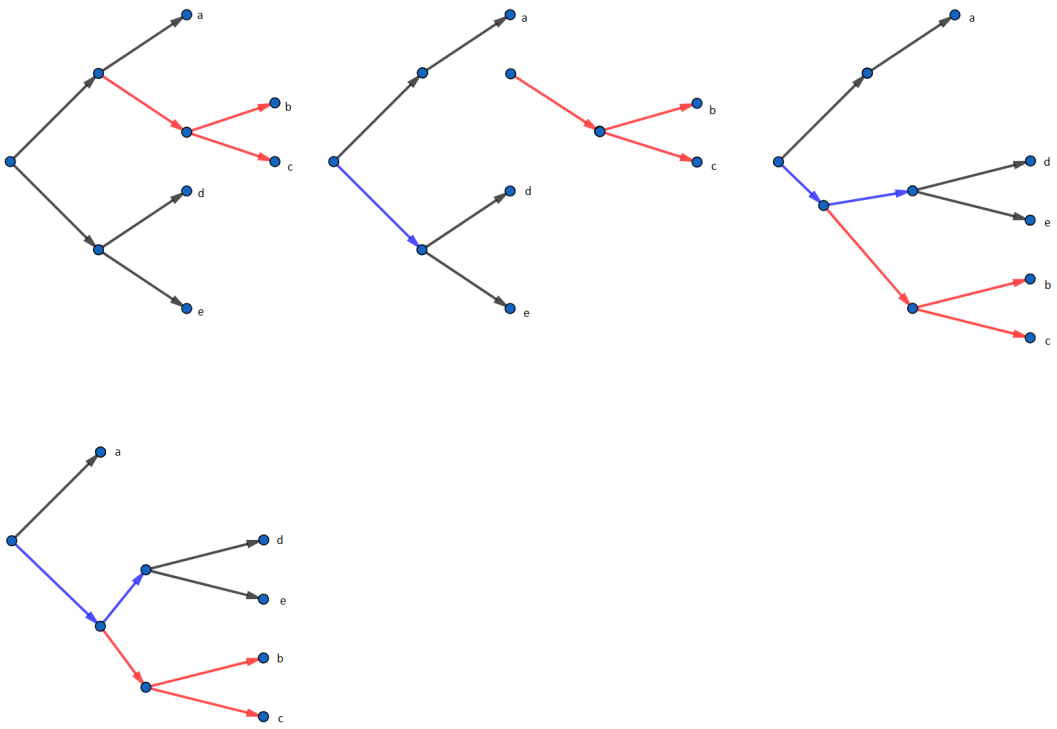

FIGURE 2. rSPR-move on a rooted phylogenetic tree

The first three images in Figure 2 show the SPR-move. The final image shows a cleaned up representation of the phylogenetic tree. We can collapse any internal node with in-degree and out-degree one, because the evolutionary information is captured in the leaves. The following move that will be introduced is a tail-move.

1.2.2. *Tail move.* Before the tail move can be introduced, the head and tail of an edge need to be formally introduced.

Definition 1.10. Given a directed graph $G = (V, E)$ and an edge $(u, v) \in E$ directed from u to v, then u is called the *tail* and v is called the *head* of edge (u, v).

The name of the tail move already suggests what the move entails, namely moving the tail of an edge to a new edge within the network. An example of a tail move is given in Figure 3.

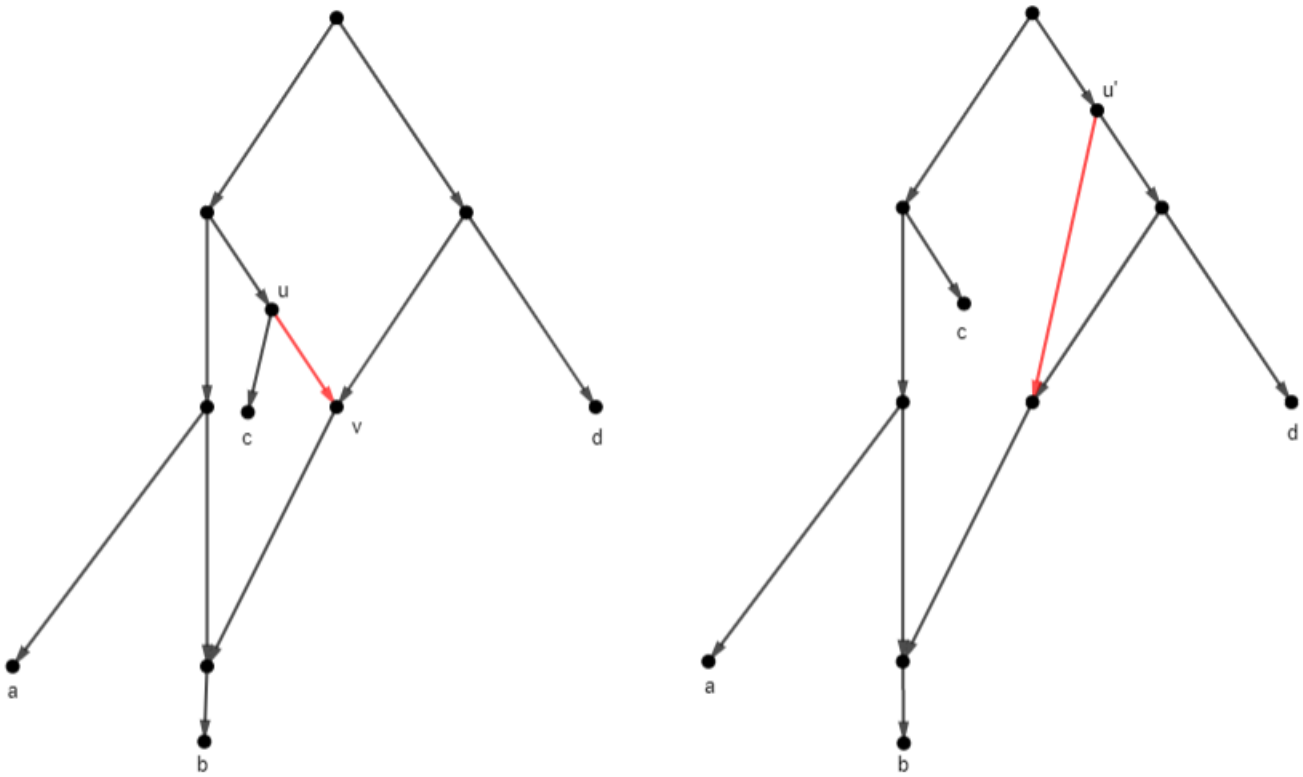

FIGURE 3. Tail-move on a rooted phylogenetic network.

There exist more moves that can be performed on networks. For example, the head move. In most cases though a head move can be replicated by a number of tail moves. These will be the moves that will be used throughout the report, any other moves will be explained when encountered.

1.3. **Newick string representation.** Trees can be represented in many ways. One of the most commonly used formats is the newick string format. The newick format was first adopted by James Archie, William H. E. Day, Joseph Felsenstein, Wayne Maddison, Christopher Meacham, F. James Rohlf, and David Swofford in 1986 during an informal meeting of Society for the Study of Evolution at Newick's lobster restaurant in Dover, New Hampshire [4]. This subsection will give a quick insight in how to interpret the newick string to recreate the corresponding tree. The phylogenetic tree in Figure 4 will be used as an example. A newick string is represented using brackets. What is inside the brackets shows what is beneath the node that is directly outside the considered brackets. We will construct the newick string for the example in Figure 4 step by step.

If we wanted to generate the newick string using a program we would start at the leaves and work our way to the root node, but to get an idea of how the newick representation works we will start at the root node. This means we start at node A and we see that the nodes directly beneath A are B and C and thus our start string becomes (C,B)A;. We have checked of node A an can move inside the brackets of A. For each node inside the brackets we have to check if they have any child nodes. We start with C and we see that C has two child nodes namely D and E. This means that we can update our string to ((E,D)C,B)A;. If we now do the same for B we end up with the newick string ((E,D)C,(G,F)B)A;. There can also be instances where the length of the edges is known. Suppose that we know that the edge AB has length 2, then we can represent this in the following way: ((E,D)C,(G,F)B:2)A;. Having the representation that also shows all the edge lengths is often the most popular representation. However, it is not required to represent internal nodes or any nodes if the node names hold no value for the tree. Therefore, depending on context the following two strings could also correctly display the tree from Figure 4 : ((E,D),(G,F)); and ((,),(,));.

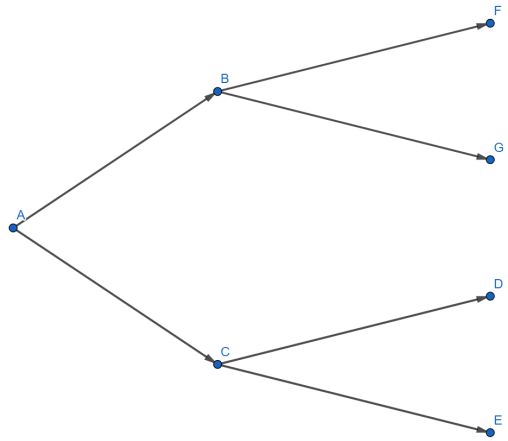

FIGURE 4. A simple phylogenetic tree. The tree can be represented with the following newick string: ((E,D)C,(G,F)B)A;.

2. Introduction to machine learning

Machine learning (ML) is growing into a widely used tool in many practical applications and it has become a strong classifier tool. The most well known machine learner is the chess-master-beating machine Deep Blue. This was a chessbot that was able to defeat the chess grandmaster Kasparov. Machine Learning has evolved to very complicated neural networks which can achieve great feats. This section will introduce the basics of machine learning and will give an in-depth explanation of decision trees. First we show what classifiers are and give a brief example. Secondly we discuss one of the very first decision models that was created to give the reader a practical introduction on the basics of machine learning. Afterwards we will introduce the model that we used throughout the report, namely the decision tree and some key concepts required to understand the model. Lastly we will discuss how to handle the data that we need to train the model and how we can evaluate a model that has been trained. These are all concepts that will be used throughout the report.

2.1. **Classifiers.** The simplest form of classifier can be compared to a neuron in the brain. A single neuron has the option to fire a signal or to not fire a signal. This can be interpreted as a yes or no response. A classifier works in the same way. Bound to a set of rules it will return a yes or no response. This can be used to classify data. Consider that we have two different types of plants and we would like to create a program that can see the difference between the two types. There are many ways this can be achieved. For instance, a program can be created that makes decisions based on imagery. This is however far more complicated than using something called the features of the data. Features are characteristics that can be represented as numerical values. Examples of features for the flower-example are petal length, petal width and the number of petals. The program can then train on these features to be able to distinguish between the different type of flowers. A very well known data set is the iris flower data set. Table 1 shows possible characteristics that can be used as features for the training set that can be used for the classifier.
We can also call Table 1 a data frame which captures some features of the iris flower family. A data frame is one of the most common used objects to store features and train learners. The figure shows that there is data for four features and the last column holds the class. The most simple classifiers, if presented with usable data, can distinguish two different classes because they are only able to produce a zero or one statement. Frank Rosenblatt was the first to presented the first concept for a learner rule that could simulate a neuron. A learner rule is a rule that is used in the corresponding learner program to train it. Formally we can classify the iris flower example as a binary classification problem where we can classify one

Instance	Sepal Length	Sepal Width	Petal Length	Petal Width	Class
145	6.7	3.0	5.2	2.3	Iris-Virginica
146	6.3	2.5	5.0	1.9	Iris-Virginica
147	6.5	3.0	5.2	2.0	Iris-Virginica
148	6.2	3.4	5.4	2.3	Iris-Virginica
149	5.9	3.0	5.1	1.8	Iris-Virginica

Table 1. Tail of the Iris flower data set from the UCI Machine Learning Repository.

flower as a 1 and the other flower as 0 or -1. The data is always split into a training set and a test set. Let us now introduce a very simple neuron model as can be found in [10].

2.2. **Early classification model.** We want to simulate a neuron, meaning that it can either fire or not. Therefor we introduce something called an activation function $\phi(z)$ [[10], Chapter 2.1], where it takes a linear combination of the given features $\boldsymbol{x}$ and a weight vector $\boldsymbol{w}$. Here z is simply the given linear combination, $z = w_1x_1 + \cdots + w_nx_n$. The length of the vectors is determined by the number of features that are used during the training of the learner. Having $x = [x_1, \ldots, x_n]$ and $w = [w_1, \ldots, w_n]$ implies that there are n features being used. For each sample we want to find out to which of the classes it belongs. We can define a threshold θ which determines which output value belongs to the sample. If $\phi(z)$ is greater than θ we predict class 1, otherwise we predict class -1. This function $\phi(\cdot)$ can be anything, but in this case we will consider a simple heavyside step function. This gives us the following function.

$$\phi(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ \text{-1} & \text{otherwise} \end{cases}$$

We could also move the threshold to the left side. If we then introduce $x_0 = 1$ and $w_0 = -\theta$ we get $z = w_0x_0 + w_1x_1 \cdots w_nx_n = \boldsymbol{w}^T\boldsymbol{x}$ and

$$\phi(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ \text{-1} & \text{otherwise} \end{cases} \tag{1}$$

Figure 5 beautifully shows how the input data is formed into a binary output. The left figure shows how the input is transformed in binary data and the right figure shows how this then can be used to distinguish two linear separable classes. This also shows why this is a rather simple model, as it only works if the data is linearly separable. The rule for Rosenblatt's first model was quite simple and can be captured into two steps.

(1) Set the weights to zero.
(2) For each sample in the training data perform two steps:
    (a) Compute the output values $\hat{y}$
    (b) Update the weight vector $\boldsymbol{w}$

All the weights w_i are updated simultaneously. We can write the updating step as follows: $w_i = w_i + \Delta w_i$. Now Δw_i is calculated by the learning rule that has been set. The learning rule determines how the model is trained and can differ for each problem. In the case of Rosenblatt he used a simple rule.

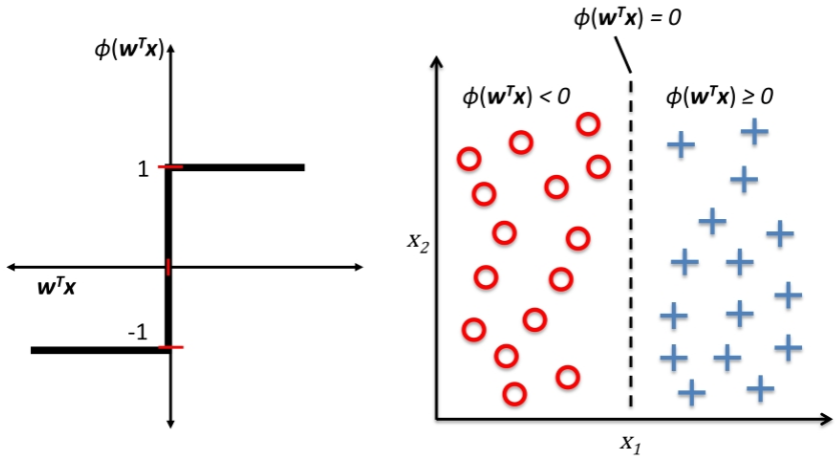

FIGURE 5. Visualisation of binary output, [[10], Chapter 2.1].

Consider we are using the kth sample of the training set, then the learning rule is as follows:

$$\Delta w_i = \eta(y^{(k)} - \hat{y}^{(k)})x_i^{(k)}$$

Here η is the learning rate, this is simply a constant between 0 and 1. $y^{(k)}$ is the true class of the kth sample and $\hat{y}^{(k)}$ is the predicted class. As mentioned before, we update all weights simultaneously, this means that $\hat{y}^{(k)}$ gets only computed once for all the weights. For the interested reader a python implementation in [[10], Chapter 2.2]. As mentioned before we need to have linearly separable data. To formalize this concept we will now give a definition, which can be found in many optimisation or analysis books.

Definition 2.1. Let S and T be two sets in an n-dimensional Euclidean space. Then S and T are *linearly separable* if there exist $n+1$ real numbers $w_1, w_2, \ldots, w_n, k$, such that every point $s \in S$ satisfies $\sum_{i=1}^{n} w_i s_i > k$ and every point $t \in T$ satisfies $\sum_{i=1}^{n} w_i t_i < k$.

Even though this is the simplest machine learning program that can be created, it shows the basic concepts very well. There are many more ways to create classifiers and this report will not cover them all. Also any data wrangling techniques that will be used will not be discussed in detail. Concepts will only be fleshed out if it is required to grasp the overarching methods that are discussed. Should anyone be interested in techniques to create clean data for python machine learning we would like to recommend [9]. There is one more classifier we would like to discuss in more detail, namely the decision tree.

2.3. **Decision Tree.** Sometimes it can be very difficult to interpret classifier models. If we care about interpretability the decision tree can be a very good choice. The name of the model describes perfectly how it works. We break down the data into categories by making decisions based on asking a series of questions. Figure 6 shows a very simple concept of a decision tree, but captures the basics of the model quite nicely. If we had a data set that would hold a set of days with the features Weather, Humidity and Wind, then we could use this decision tree to break down the data into two classes by answering a series of questions. However we will consider a less complicated decision tree. In the figure the feature weather splits of into three children which is possible, but for now we will limit ourselves to binary decision trees.

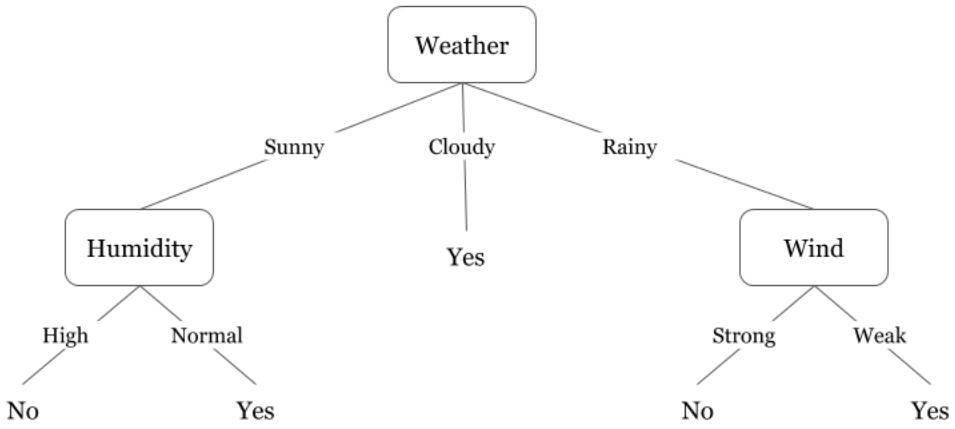

FIGURE 6. A decision tree that determines if we can play football.

We again start with a dataset, which contain features of some objects. Think, for example, of the Iris flower set. Now the decision tree uses the features of the training set to learn questions on which it will split the samples. The goal of the tree is to find the class that belongs to each sample correctly. The decision tree algorithm starts in the root of the tree. Here the data will be split on the feature that gives us the most information gain (IG), this will be formalized later. Using iteration we can then split the data again in each child, until we have found pure leaves.

Definition 2.2. A leaf is *pure* if it only contains samples from one of the considered classes.

However, in many instances this will result in a very deep tree and that will lead into overfitting of the data. Overfitting of a model means that the model fits the training data too well, i.e., it picks up on noise of the training data. This means that the model will perform extremely well on the training data but not on any new and unseen data. The only question that remains is which feature we should split on. For this we define an objective function that will be maximized (or minimized) via the decision tree algorithm. The function that we use in the algorithm is a standard one and can be found in, for example, [[10], Chapter 3.6]. The function is defined as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{n} \frac{N_j}{N_p} I(D_j) \tag{2}$$

In this function, f is the feature that will split the data, D_p and D_j are the data sets of the parent p and the jth child node. I is the impurity measure, defined below, and lastly N_p and N_j are the number of samples present in the parent node and child node respectively.

As we can see in equation (2), the information gain is the difference between the impurity of the parent node and the sum of the impurities of the children. Using equation (2), we can quickly see that the information gain is largest if the impurity of the children is low. If we would create a multifurnicating tree however, the search space could be enormous. For simplicity most libraries in Python use binary decision trees, using this fact and defining D_{left} and D_{right} for the data sets of the left and right child respectively we get the following representation for the information gain.

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}) \tag{3}$$

The only thing that is left is to define what the impurities are. In binary decision tree algorithms there are three commonly used impurity measures. These measures are the *Gini* impurity, the *Entropy* and the *Classification Error*. We will first introduce the Gini impurity.

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2 \tag{4}$$

The term $p(i|t)$ is the fraction of samples that belong to class i at node t. We could consider this as a probability. So, intuitively the Gini impurity can be seen as a function that minimizes the chance of misclassification. The impurity is maximal if we have uniformly distributed data set regarding the classes, i.e., if the data set is a perfect mix of all the classes. The second impurity measure we will introduce is the entropy.

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) log_2 p(i|t) \tag{5}$$

We see that if all the samples belong to the same class we will have an entropy that is equal to 0 and with a uniformly distributed data set we will have an entropy of 1. Intuitively this implies that the entropy tries to maximize the mutual information in the decision tree. Finally we will take a look at the classification error.

$$I_E = 1 - max(p(i|t) \text{ for } i \text{ in } \{1, \ldots, c\}) \tag{6}$$

This measure is not useful for growing the decision tree, however it is useful to use as a pruning criterion. Practice shows that using Gini impurity and entropy result in similar results. This means that it is often not useful to create trees using different impurity measures. It is often better to experiment with different pruning options.

Sometimes it is not desirable to grow a tree that results into a decision tree that only has pure leaves. This could lead to overfitting which in turn can lead to a biased search tree. There are many pruning options, for example the classification error impurity. However which pruning option should be used during the growing of a decision tree heavily depends on the problem that is being considered. Dealing with impure leaves is quite a simple procedure. If a leaf is impure it will return the classification that has the highest fraction of occurrences in the leaf.

It is also possible to combine decision trees into a stronger learner. This is a concept that is called random forests. The idea is that many decision trees get combined into a more robust model that is less susceptible to overfitting. The idea of the algorithm is quite simple. We start by randomly selecting a sample set of size n from the training set, as seen in [[10], Chapter 3.6]. After the selection process we use the sample to grow a decision tree. However the growth decision differs from a standard decision tree. At each node we do the following:

(1) We randomly select b features without replacing them.
(2) We then split the node, using one of the b features that gives the most information gain by using an impurity measure of your choosing.

Now to generate the forest we need to repeat this process a number of times to obtain enough decision trees. Classification is now done by a majority vote between the trees. This method has its flaws however, because we lose the interpretability of a single decision tree, but the algorithm requires a lot less tuning to perform reasonably well. The only parameter that really requires tuning is the number of decision trees that are used within the random forest.

2.4. **How to handle unbalanced data.** In many instances the data that is generated can be unbalanced, meaning that there are more of one class than the other class. This can lead to models that will only predict one class and in that way achieve an incredible high accuracy. For example, if we have a data set of 100 flowers and it consist of 98 roses and 2 tulips then always predicting a rose will lead to an accuracy of 98 percent. Decision trees are less susceptible to unbalanced data but to prevent generating bad models due to unbalanced data we could use a technique called resampling. There are two ways we can resample the data, namely oversampling the minority class, i.e., the class that is underrepresented in the data or undersample the majority class, i.e., the class that is over represented in the data. We will briefly discuss both techniques.

2.4.1. *Oversampling.* Oversampling is done to increase the number of instances from the minority class that appear in the data. The concept is rather straightforward. To increase the size of the data that is of the minority class we randomly select a sample from the minority class and add it to the data. We keep doing this until we have balanced out the data. Simply put we randomly duplicate data from the underrepresented class.

2.4.2. *Undersampling.* Undersampling is the opposite of oversampling. Here we randomly delete instances from the data of the majority class until we have a balanced data set. Which technique should be used depends on the problem at hand, sometimes oversampling works better than undersampling. It is a matter of testing to see which fits the model best. Most of the time resampling will lead to a decreased accuracy of the model, but a far more reliable one.

2.5. **Evaluating a machine learning model.** A very important part of a machine learning model is its performance. A question that always should be asked while training a model is whether or not it only performs well on the training data or if it will also perform well on unseen data. This is exactly why machine learning models should be evaluated after or while training. In this subsection a few evaluation techniques will be explained, namely train-test-split, k-fold cross validation and LOOCV (Leave one out cross validation). These evaluation techniques all use a performance metric that depend on the problem at hand. The performance metrics that will be discussed in this subsection are the accuracy score and the

AUROC score (Area under the receiving operating characteristics). We start by explaining the evaluation techniques.

2.5.1. *Train-test-split.* This is the simplest way to perform some sort of evaluation of the model. After data generation we have the data set X and the set y that holds the class labels. These data sets are then split into a training set X_{train}, y_{train} which is used to train the model and a test set X_{test}, y_{test} which is used to evaluate the model. The size of the training set depends on the problem, but most commonly we make a split of $70\% - 30\%$ or $80\% - 20\%$. The reason for a test set is to provide the model with data that it has not seen before.

After training the model we can then use the test set to evaluate the model by letting it predict the classes. After the model is done predicting we can score the model by using a performance metric, for example, the accuracy which will be explained in the performance metrics section. This then gives us a score that shows how well the model is performing. There is no golden rule on scores that should be used. Different problems require different performance scores and different scores should be considered before drawing conclusions on the performance of the model. This is a very easy way to evaluate the model and get an idea of how good the model is performing on unseen data, but does not work well when the model is overfitting, because the test data is relatively small. A better way to detect overfitting in a model is to use k-fold cross validation which we will introduce next.

2.5.2. *K-fold cross validation.* As mentioned in the introduction of this subsection there are many types of cross validations. However, the overall concept of cross validation remains the same with every technique. We partition the data set that is going to be used in the model in a number of subsets. The idea is then to hold out on one set at the time and train the model on the remaining sets. After the model is finished training then the hold out set is used to test the model.

The k-fold cross validation procedure has one parameter called k that indicates the number of equal sized sets the data will be split into. This is why it is often called k-fold cross validation. The parameter k does not only determine the number of subsets that will be created, but also the number of times the following procedure will run:

(1) Select a new test set
(2) Select the remaining sets as the training set.
(3) Fit the model on the training set.
(4) Evaluate the model on the test set using a pre-selected performance metric, e.g., accuracy.
(5) Save the evaluation score and discard the model.

After the process is finished the skill of the model can be analysed using the evaluation scores by taking, for example, the mean of all the scores. The only difficult part of k-fold cross validation is to determine a correct value for the parameter k. The value of k must be chosen in such a way that each subset that is created is large enough to be statistically representative of the whole data set. A very common value for k is $k = 10$. One thing to note is that if a value of k is chosen that can not evenly split the data into k subsets, then one subset will hold the remaining data samples from the original data set. In many

cases the data gets initially split into a training and test set on which the model will be trained and afterwards tested. Then k-fold cross validation is used on the training set to evaluate the current model. This can lead to, for example, the tuning of parameters of the model if overfitting is detected. Then the initial test set is used for a final evaluation of the model. This way both k-fold crossvalidation and train-test-split are combined to evaluate the machine learning model. We will now quickly go into a bit more detail about overfitting.

K-fold cross validation is a good technique to determine whether or not the machine learning model is susceptible for overfitting. This cross validation technique guarantees that the model gets tested using k different unseen data sets. The evaluation scores then show if the model overfits on any of the used data or if it performs more or less the same on each training set - test set combination.

Figure 7 gives a graphical representation of k-fold cross validation which also shows the initial splitting of train and test data.

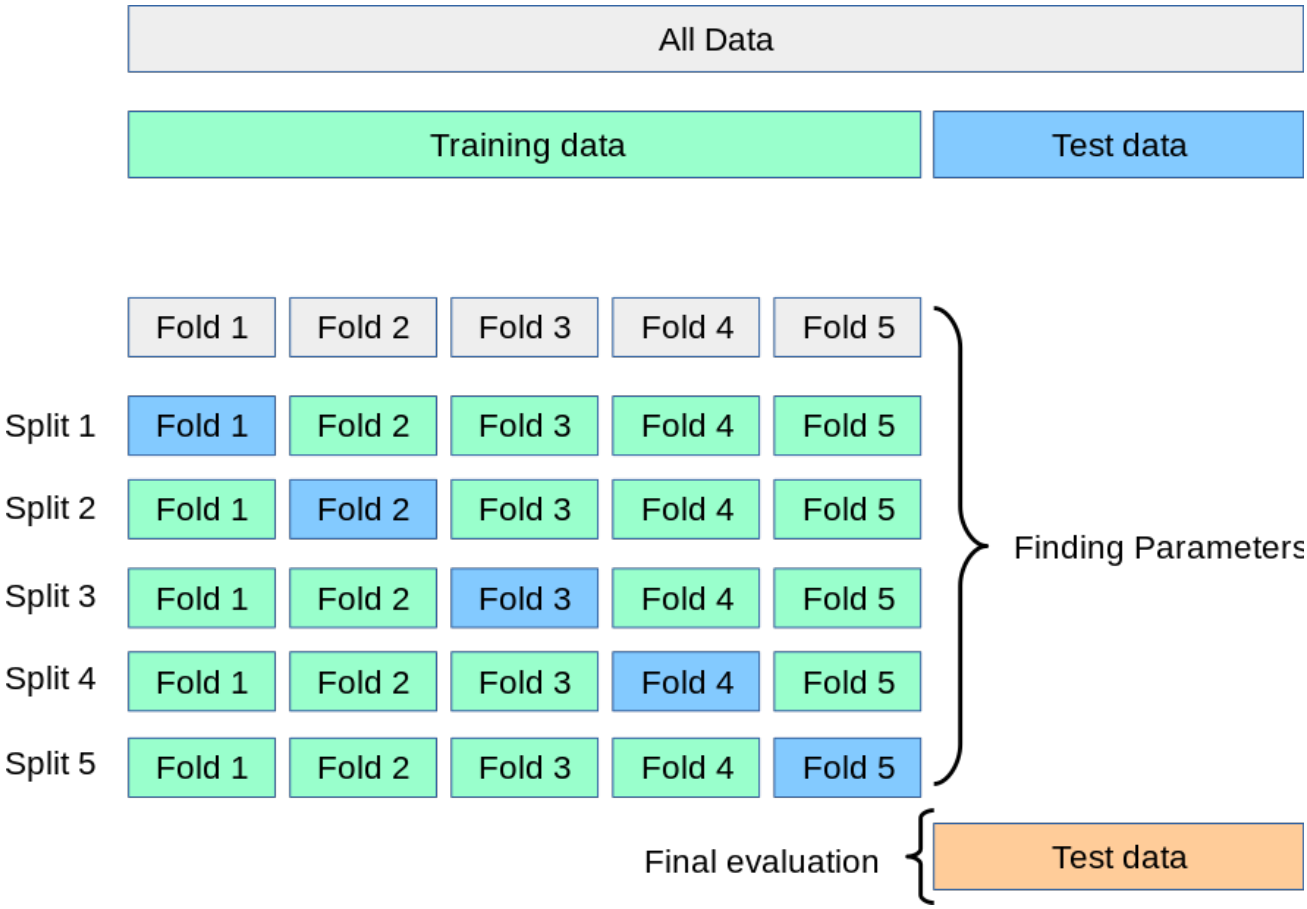

FIGURE 7. K-fold cross validation on the initial training data from scikit-learn.org.

2.5.3. *Leave one out cross validation.* LOOCV is the same as k-fold cross validation, but taken to its literal extreme. With LOOCV we split the data set in n samples, where n is the size of the initial data set. The evaluation made using this technique is very good, but is more computationally heavy than the standard k-fold cross validation. However, there exist some techniques that make predictions using leave one out just as easily as regular predictions. In this report we will often limit ourselves to the use of k-fold cross validations, due to the ease of implementation and computational effort.

We can now take a look at some of the performance metrics that can be used during the evaluation of a machine learning model.

2.5.4. *The accuracy score.* Performance metrics dictate in what way a machine learning model is evaluated. Every technique discussed so far uses a performance metric. The simplest performance metric is the accuracy score. As the name suggest, this metric determines the accuracy of the model. This is done by looking at the number of correct predictions and the total number of predictions. The performance metric can be captured in the following formula.

$$\text{accuracy score} = \frac{\text{number of correct predictions}}{\text{total number of samples}}$$

This formula can be adjusted to figure out how many false positives or false negatives the model returns. This will sometimes give more insight than the actual accuracy score, because false positives can lead to bad decision making.

2.5.5. *The AUROC score.* The reason that we only discuss two performance metrics is because the AUROC score requires some knowledge of other performance metrics and those will be discussed here. AUROC is more difficult to understand than the accuracy score and therefore we will first give a intuitive description. AUROC can be split into two parts, namely AUC and ROC. ROC is what we call a probability curve and AUC represents the degree of separability. AUC is an indicator of how well the model can distinguish between the classes present in the data. A high value of AUC means that the model is better at predicting true positives and negatives, meaning that it will predict a zero as a zero and a one as a one.

Before we introduce some of the concepts required to understand the AUROC score we have to talk about what happens "under the hood" of most machine learning models. In many cases the model returns a deterministic value to us. In the case of a binary decision tree, if we introduce a data sample to the tree to have it decide to which class it belongs it will always return one of the two classes. This is only done because most of the users desire to have deterministic values, however, the decision tree will **always** return the probability that it is the one class or the other class and then for ease of use if, for example, the probability of the class 1 is higher than $\frac{1}{2}$ the decision tree will say it must be 1. Almost all machine learning models work with probabilities and only return classes for the ease of interpretation.

The AUROC score will not look at the final predicted class, but will use these probabilities and will also look at different thresholds other than $\frac{1}{2}$, which is used as a standard. These threshold can be arbitrary numbers, but most of the times these values will lie between 0

and 1. For each of these thresholds we can then look at the rate True Positives get predicted and the rate at which False Positives get predicted.
We will first introduce the True Positive Rate(TPR), specificity and the False Positive Rate(FPR). Note that any of these values are dependent on the chosen threshold value that determines how a sample gets classified.

$$\text{TPR} = \frac{\text{True positive}}{\text{True Positive} + \text{False Negatives}}$$

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}}$$

$$\text{FPR} = \frac{\text{False Positives}}{\text{True Negatives} + \text{False Positives}} = 1 - \text{specificity}$$

The ROC curve is a probability curve that is plotted with the TPR against the FPR. An example of a ROC curve can be seen in figure (8).

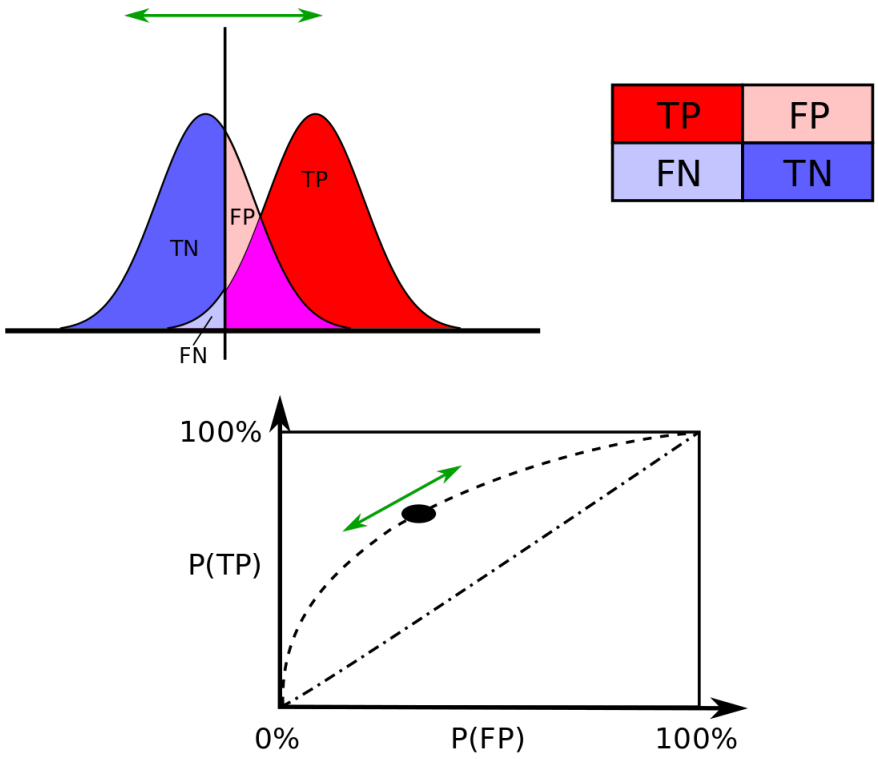

FIGURE 8. A visual explanations of AUROC where P(FP) = FPR and P(TP) = TPR, shared according to Creative Commons Attribution-Share Alike 3.0 Unported license, https://creativecommons.org/licenses/by-sa/3.0/deed.en.

If the probability distributions of the 1 class and the 0 class, as can be seen in the top left of Figure 8, are available, then they can be used to calculate the TPR and FPR. If the distributions would not overlap the model would be able to perfectly distinguish between the two classes and we would end up with an area under the ROC curve of 1. But how can we create such a ROC curve? The idea is to calculate the TPR and FPR for different threshold values. This threshold is displayed as a vertical black line in the top left graph of Figure 8. In many cases a finite number of threshold values is used, often resulting in a piecewise linear or piecewise constant function. If the distributions of both the True Positives and False Positives are known then the ROC can be created by plotting the cumulative distribution function of the True Positives along the y-axis against the cumulative distribution function of the False Positives along the x-axis. If the AUROC score is around $\frac{1}{2}$ then the model cannot distinguish between classes. If the score is 1 the model perfectly distinguishes between the two classes and if the score is 0 it means that the model is predicting zero classes as one and one classes as zero. Simply swapping the two around will result in a very good model.

We will finish up by showing a quick example on how to create an ROC curve and thus find the AUROC score. Luckily for us, there exist many programs that calculate the TPR and FPR for us, given some thresholds. In Table 2 show four different thresholds for which the TPR and FPR are calculated. Using this data we can now create the corresponding ROC curve. The ROC curve can be found in Figure 9. It is always plotted with an indicator curve. This is the curve you would get if every sample gets classified randomly.

Threshold value	True positive rate	False positive rate
2	0	0
1	0.55	0.29
0.5	0.75	0.45
0	1	1

TABLE 2. Example data to create an ROC curve.

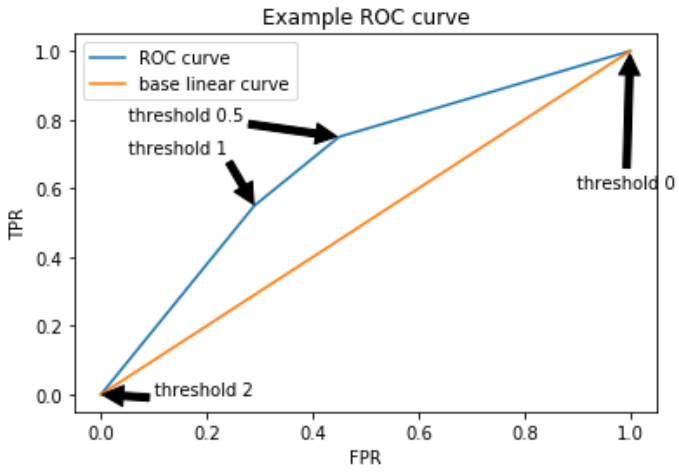

FIGURE 9. The ROC curve belonging to the example data. The blue line is the ROC curve and the orange line is the indicator curve.

Now that we have a basic understanding of some of the concepts of machine learning we are now able to move on to the first problem that will be studied in this report, namely the maximum agreement forest problem (MAF).

3. The maximum agreement forest problem

In this section we will introduce the maximum agreement forest problem. We will start by giving some theoretic definitions required to understand the problem. After this we will introduce the algorithm and we will introduce what a fixed parameter tractable algorithm is. Afterwards we will discuss how we selected the features and how the data was generated for training the model. Finally we will discuss the creation, the performance of the decision tree and the results from the implementation into the FPT algorithm. We start now by introducing some key concepts.

We will limit ourselves to finding an agreement forest between two trees. We first define what an agreement forest of two phylogenetic trees is.

Definition 3.1. An *agreement forest* of two phylogenetic trees S and T on X is a forest F that can be obtained from each of S and T by deleting a set of edges and then repeatedly:

(1) deleting all vertices with in degree 0 and out degree 1;
(2) deleting all leaves that are not labelled by an element of X;
(3) suppressing all vertices with in degree 1 and out degree 1.

Using this definition we can create a agreement forest. However there can exist multiple agreement forests for a set of two phylogenetic trees. We therefore also define what a maximum agreement forest is.

Definition 3.2. A *maximum agreement forest* (MAF) is an agreement forest with a minimum number of components. The number of components is denoted by $MAF(S, T)$.

It now remains to introduce the Maximum Agreement Forest problem. The problem is defined as follows:

- **Instance:** two rooted binary phylogenetic trees T_1, T_2 and a number $k \in \mathbb{N}$
- **Parameter:** k
- **Question:** does there exist an agreement forest of T_1 and T_2 with at most $k + 1$ components?

The formulation of this problem is known as a parameterized problem. Before we can discuss the algorithm to solve this problem we need to introduce fixed parameter tractability.

Definition 3.3. A parameterized problem is called *fixed parameter tractable* (FPT) if there exists an algorithm solving the problem with running time $f(k)n^c$ for some constant c and some computable function f, with n the size of the input.

Whidden, Beiko and Zeh [11] showed that there exists an algorithm that solves MAF in $O(3^k n)$ time. We will now introduce this algorithm.

- Initially $F = T_2$.
- If F is an agreement forest of T_1 and T_2 and $k \geq 0$ then we are done.
- If there is a leaf that is a singleton component of F, delete it from T_1.
- Find a cherry (a, b) in T_1.
- If (a, b) is a cherry in F, collapse (a, b) into a single leaf in T_1 and F.
- Otherwise, recursively solve three subproblems:
  (1) cut the edge leading to a in F, set $k' = k - 1$
  (2) cut the edge leading to b in F, set $k' = k - 1$

(3) cut all p edges leaving the path between a and b in F, set $k' = k - p$
(if a, b are not in the same component of F then we can skip this one)

There are some optimizations that can be done to the algorithm to speed it up, but for now we will use this simple algorithm and evaluate it alongside an algorithm that has a machine learning implementation. The fourth step of the algorithm asks us to find a random cherry in T_1. We are going to replace this step of the algorithm with a classifier that will decide which cherry should be selected to have the quickest results.
The first classifier that we used in the algorithm was a simple decision tree. Starting with this simple learner we could decide whether or not machine learning would have any useful impact on the program before we try more convoluted learners. As explained in the machine learning theory section we will first require a number of features on which the decision tree is going to decide. Firstly we will discuss the features that we have selected for the decision tree. We will discuss what they are and why they are the most useful to the structure of phylogenetic trees and maximum agreement forests. After the features are selected we will be able to take a look at the tree that we have grown to try to speed up the algorithm.

3.1. **Selecting the features.** During the growing of the decision tree we will be considering the following features.

(1) Depth of the cherry.
(2) The number of leaves present in T_1.
(3) Distance between the leaves of the cherry in the second tree.
(4) The number of cherries in T_1.
(5) Size of the tree beneath the lowest common ancestor in the second tree.

The choice for every feature will be explained in detail and we will also make some remarks during every explanation.

3.1.1. *Depth of the cherry.* In our data every edge had the same length, therefore we could use the distance from the cherry to the root as the depth of the cherry. This is one of the features that is very easy to get from the trees and is a clear numerical value. One remark has to be made about this feature. Consider that we have a dataset that contains many sets of trees of different sizes. This means that cherries will be on very different depths in the different tree pairs. Therefore one should consider normalising the feature by dividing the depth of the cherry by the maximum depth within the tree. This way we avoid misclassifying cherries because of their depth, but that way we can also see if the depth of the cherry within the tree determines if it is a good cherry to select. We believe that normalised data would have a better accuracy.

3.1.2. *The number of leaves present.* This feature was only used in the first tree that we created. We discovered that this was a very weak feature as larger trees result in more leaves. Also in many cases we fixed the number of leaves and therefore the impact was the same for every data entry that we used. We still mention this feature as we did perform test with this feature.

3.1.3. *Distance between the leaves of the cherry in the second tree.* Consider having a cherry (x, y) in the first tree and suppose that it is not a cherry in the second tree. We then determine the distance from x to y in the second tree and use that as a feature that determines the structure of the tree. Again the size of the tree determines the distance between the leaves, therefore we should again consider a normalisation step. This means that we will have a

Tree input + Cherry	feature 1	feature 2	feature 3	feat 4	feat 5	Search Depth	Class
(tree1,tree2,(9995,9994))	0.31	13	0.25	0.2	0.15	1650	1

TABLE 3. Example of a data sample.

data set that will hold the distance between the leaves divided by the maximum distance in the tree. This way we can safely consider different sized trees without creating a bias in the data.

3.1.4. *Size of the tree beneath the lowest common ancestor in the second tree.* The size of the tree that we will consider is determined by the number of leaves present beneath the lowest common ancestor. This feature is similar to the distance feature, but we believe that this feature shows a different kind of structure than the distance feature. This is because it does not show the distance between the leaves of the cherry in the second tree, but shows the structure of the tree that holds both the leaves. Again a remark must be made considering different sized trees. The data that is used should be normalised by the size of the whole original tree. This size will also be determined by the number of leaves present in the tree.

This are the features that will be considered while growing the decision tree.

3.2. **Data Generation.** This section will briefly discuss how the training data is generated for the maximum agreement forest problem.

We limit ourselves to pairs of phylogenetic trees. This means that we require a data set that holds pairs of phylogenetic trees.

3.2.1. *Generating a tree.* The way that we generated a binary tree was very straightforward. We are working in python and because we are using phylogenetic tree we used the Python package *Ete3*. This is a well-documented package, that has been especially created to work with phylogenies. Due to the user-friendliness of the package we could easily create trees that were populated at random. After the tree was randomly populated a simple program was used to make sure that for all the trees the nodes were labelled correctly.

3.2.2. *Generating the tree-pairs.* We start by a set of trees that will all be the first tree of the tree-pairs. In our case, due to the limited computing power available we limited the number of leaves for the trees to ten. We started with a set of 1000 trees. After the trees were generated the second tree for each pair was generated by using each of the 1000 trees and performing a number of rSPR moves on the trees. The number of rSPR moves used to randomize the trees were chosen uniformly at random from $\{2, 3, 4, 5\}$. By doing this we created a good mix of tree-pairs without having to spend a week to generate the data. On our GitHub page https://github.com/TUbryan/PhyloThesis the code can be found that simulates rSPR moves. Table 3 shows one entry from the generated data.

3.2.3. *Obtaining and classifying the cherries.* After generating the starting data the cherries could be selected from each tree by using part of the implementation that can be found on the GitHub page. This means that each tree pair resulted in training data which is equal to the number of cherries present in the trees. This means for trees of ten leaves we could

get at most five data points. Finally after obtaining all the cherries from the tree we needed to determine which cherries would be regarded as good cherries to pick and which cherries would be bad cherries, i.e., which cherries would be classified as a 1 (good) and which cherries would be classified as a 0(bad). This was achieved by running the basic algorithm and recording the size of the search tree we obtained after starting with each specific cherry. This shows how many recursive steps the algorithm needed to make before finding a solution. For our experiments we have chosen to set the best ten percent of the cherries for each tree-pair to good cherries, i.e., 1 and the rest to bad cherries, i.e. 0.

We are now able to grow the decision trees using the prepared data.

3.3. **Creating the decision tree.** In this section we will discuss the creation of the decision tree and afterwards we will also discuss the performance of the tree.

3.3.1. *Growing the tree using the data set.* Many tools are already available for making decision trees. One of these tools is the Python package Scikit Learn. This is a very optimised package that has many machine learning implementations, one of which is creating decision trees. This saved us the time to have to implement a decision tree classifier of our own and due to the high optimisation it created trees very quickly. A few things need to be considered before growing the tree. We need to tweak some of the parameters that will be used in creating the decision tree. One of these parameters is which measure we want to use. Some testing showed, as mentioned in the theory section, that the difference in results from using the entropy measure or gini measure are nearly negligible. The depth of the decision tree and the features that are used during the growth process have a much bigger impact on the accuracy of the tree.

Before we discuss the results of the first tree we briefly explain how the test data and the training data is prepared. This is a very simply procedure, but it does require some thought. The most commonly used technique is randomly splitting the data $80\% - 20\%$ or $70\% - 30\%$,as mentioned in the evaluation section, where the training set consists of the most data and the test set consists of the remaining data. One note should be added here, however, because randomly splitting the data could result in an over-representation of a certain class. For example, the data could be heavily favoured towards 0 because most of the training data consists of 0 entries but this can be avoided by considering resampling. There does not need to be a $50\% - 50\%$ representation of both the classes, but both classes should have a sufficiently large representation. We have chosen to upsample the good samples, so that we got a 50-50 spread when growing the decision tree.

The training set will be used to grow the initial decision tree. The decision tree becomes large rather quickly. For a detailed viewing of the decision tree the tree.dot file can be found on the GitHub page and viewed with the GVEDIT program.

3.4. **Evaluating the performance of the decision tree.** We will use three methods to evaluate how the decision tree is performing. First of all we will perform a 10-fold cross validation using the training data. After this we will look at two performance measures, namely the accuracy score and the AUROC score.

Instance	1	2	3	4	5	6	7	8	9	10	mean
Score	0.61	0.59	0.66	0.60	0.62	0.65	0.66	0.66	0.56	0.67	0.63

TABLE 4. Results from the 10-fold cross validation.

Occurrence	True positive	True negative	False positive	False negative	Correct	% Correct	%False positive	%False negative
	493	437	272	228	930	65	19	16

TABLE 5. Results from the accuracy score evaluation.

3.4.1. *10-fold cross validation.* To perform the cross validation we do not introduce new data to the model. Instead we train ten different trees using ten different portions of the training set as test set. This was explained in more detail in section 2.5.2. We perform the k-fold cross validation with $k = 10$ and we use the accuracy score as a measure. Table 4 shows the individual scores and also the mean score from all the instances. We see that the scores are not spectacularly good, but they do show that the model can somewhat predict the cherries correctly, achieving an average accuracy of 63 percent. This means that we should not be worried that the model will overfit the data, however, there is a chance that the model does not perform as well as we hope, because of the poor accuracy from the cross validation. To further evaluate the model we will take a look at the accuracy score and the false positives and false negatives and finally we will calculate the AUROC score.

3.4.2. *The accuracy score.* We will now evaluate how many instances from the test data get predicted correctly by the model and how many false positives and false negatives are produced. The test data that we used had a total of 1430 samples. The results from the accuracy score evaluation can be found in Table 5. We see that the model performs roughly the same as we saw during the cross validation. We achieve an accuracy of 65 percent. We see that there is quite the portion of false positives and false negatives. These could influence the performance of the model when implemented in the algorithm. We will now calculate the AUROC score.

3.4.3. *The AUROC score.* Calculating the AUROC score will give us an idea on how well the model distinguishes between 0 and 1. Remember that a machine learning model never really returns 0 or 1, but always works with probabilities which are then afterwards translated to deterministic values. This means that we can calculate for each sample the probability that it gets classified as a 0 and the probability that it gets classified as a 1. We will use the test data to evaluate the AUROC score. For the test data we can calculate the prediction probabilities. Table 6 shows a small part of the predicted probabilities for the test data samples. These probabilities are calculated by looking at which leaf the sample ends up and then looking at what classes appear in that leaf. If we have a leaf that is split (1,9) then we have a 0.9 probability to be classified as a 1.

We can now use the probability predictions of the 1 class, i.e. the "good" class, and the true class labels of the test set to compute the true positive rates and the false positive rates for different threshold values. These threshold values can be random values, but most of the time these values fall between 0 and 1. Using these false positive rates and true positive

True Class	Probability 0	Probability 1
1	0.17	0.83
0	0	1
0	1	0
1	0	1
0	0	1
0	1	0

TABLE 6. Part of the calculated probabilities from the test data.

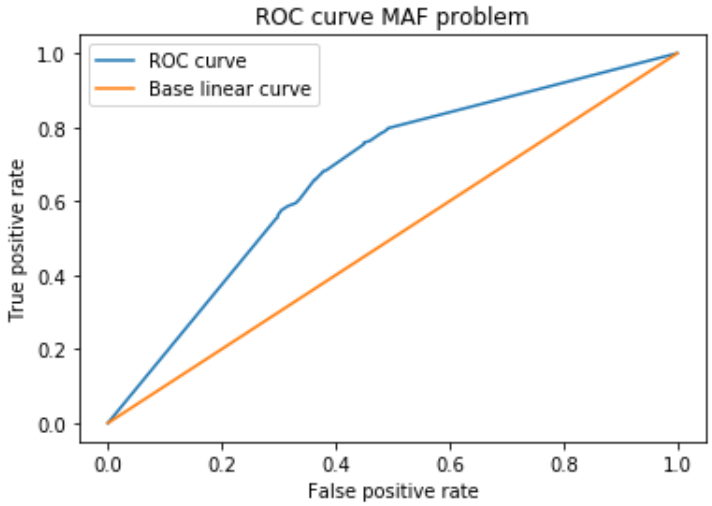

FIGURE 10. The ROC curve created from the test data.

rates we can then plot the ROC curve, which can be found in Figure 10. The linear curve that is shown in the figure is the curve we would obtain if every sample would be classified as 0 or 1 by a coin toss. The AUROC score is now the area under the ROC curve. This gives us a final AUROC score of 0.67 or 67 percent. This means that the model can distinguish between 1 and 0 somewhat.

3.5. **Evaluating the algorithm.** In this section we will evaluate the performance of the model when we implement it into the fixed parameter tractable algorithm by replacing the random choice in step four with the decision based choice. We start by explaining how we created the data we used to test the performance. Afterwards we will discuss the results and evaluate whether or not this problem can benefit from machine learning.

3.5.1. *The experiment.* The experiment was quite simple. We generated 100 different tree pairs with a random number of leaves between 10 and 15. To generate the second tree of a pair we used 5 to 12 random rSPR moves on the first tree. We took a look at the depth of the search tree. This means that we look at how many calls the regular and the decision based FPT had to make before a solution was found. The results of the experiment can be found in Figure 11. For each instance the regular algorithm and the machine learning based algorithm use the same parameter k. The results are very underwhelming and not as

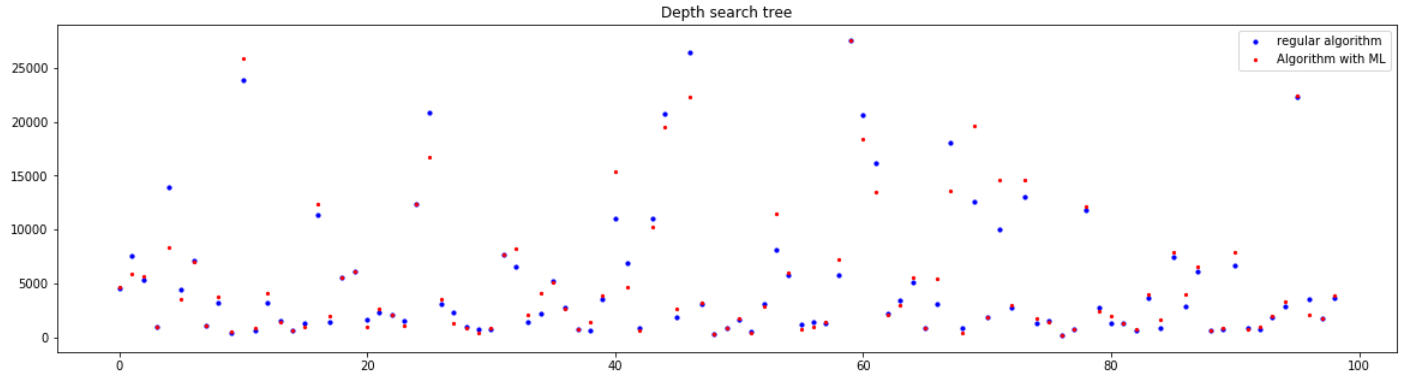

FIGURE 11. The results from the experiment.

good as we had hoped. The figure clearly shows that the performance of the decision based algorithm is very random and very randomly beats the current FPT algorithm. Therefore we believe that the current machine learning model is not yet good enough to improve the performance of the FPT algorithm. There are a few reasons why this is the case.

(1) The accuracy of the tree is not yet good enough.
   (a) We might be missing key features.
   (b) We only fix the first move and not consecutive moves. This means that the first move is determined by prediction and the rest of the moves are performed by the regular algorithm.
   (c) It can be that a decision tree is not sufficient enough to find structure in this problem.

(2) It can be that the solution for each tree pair differs too much from other tree pairs, while the features do not differ that much. In this case we cannot use the current machine learning model to solve this problem, because the decision tree tries to find a pattern in the solutions that depend on features and if the solutions do not depend on the current features the decision tree will not be able to make correct predictions.

A few of these issues could be overcome with some hard work. If the issue lies in the decision tree then these could be solved and if this is the case we do believe that this problem can benefit from machine learning. However, if the problem lies in the difference of solutions then we do not believe that the problem can benefit from machine learning. We do not yet know which exactly, but all in all we do believe that machine learning can help in this problem if we can overcome these hurdles.

4. The hybridization number problem

The hybridization number problem is one of the most fundamental problems within the area of reconstructing phylogenetic networks. As we know, a phylogenetic tree shows how a set of species or more generally a set of taxa has evolved over time by splitting events, also called speciation events. Phylogenetic networks, however, can display additional evolutionary events where different lineages of species merge together. Such events can be hybridization or lateral gene transfer, which are often referred to as reticulation events. The hybridization number problem is to construct a phylogenetic network with the minimum number of reticulation events among all possible networks that contain a given set of phylogenetic trees. This gives us the simplest possible representation of the evolution of a set of taxa consistent with the collection of phylogenetic trees. Reticulation events can lead to difference between phylogenetic trees. Roughly speaking, the requirement of the network to display every tree ensures that we create a path for each species to follow which is consistent with the corresponding tree. Not all the differences between the phylogenetic trees can be traced back to reticulation events. This means that the constructed network will only provide an estimate of the number of reticulation events.

Research showed, however, that computing networks with the minimum number of reticulations is quite challenging. First the case of having just two trees was researched. In this case the problem could be characterised mathematically using maximum agreement forests [3]. This has lead to good fixed parameter algorithms with the fastest algorithm being *Hybridization Number* which runs in $O(3.18^k n)$ time [11]. The problem becomes much more complicated if we consider more than two trees. Many of the created algorithms, such as Hybroscale [1, 2], TREETISTIC [7] and PIRN [12], can only handle a very small number of input trees and reticulation events. However, recently a good performing algorithm for tree-child networks was created by Leo van Iersel, Remie Janssen, Mark Jones, Yukihiro Murakami and Norbert Zeh [5] with a runtime of $O((8k)^k \cdot poly(n,t))$. This algorithm constructs something called a *Tree-child sequence*. We start by introducing some key concepts and afterwards we will introduce the fixed parameter tractable algorithm. After that we will introduce machine learning to the algorithm and explain how the data was generated, the features were selected and how the decision tree was created. Finally, we will evaluate the performance of the decision tree and of the algorithm with the decision tree implementation.

4.1. Theory for the hybridization number problem.

We will start by introducing some key definitions that are required to grasp the algorithm and the hybridization number problem.

Definition 4.1. A node v in a phylogenetic network is called a *tree node* if it has in-degree 1 and out-degree 2. If v has in-degree 2 and out-degree 1 it is called a *reticulation*.

Before we define a tree-child network we will take a look at tree edges and reticulation edges and what it means for a network to represent trees..

Definition 4.2. Given a directed edge uv in a phylogenetic network N, uv is a *reticulation edge* if v is a reticulation. Otherwise uv is called a *tree edge*.

Definition 4.3. A network N is said to *represent* a tree T, if T can be obtained by deleting edges from N.

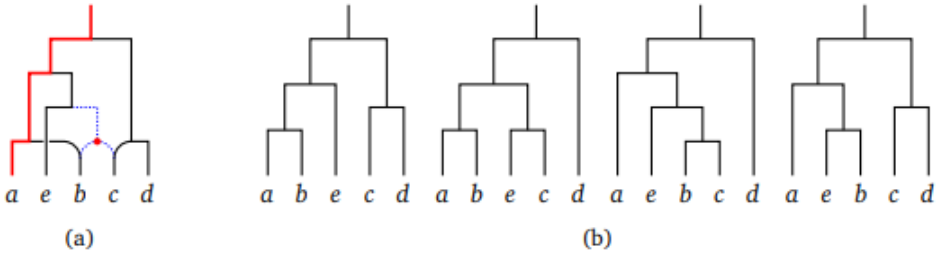

FIGURE 12. The network (a) represents the four given trees. For example, the first tree can be obtained by deleting the dotted edges. Adapted from [5].

An example of a network that represents trees can be found in Figure 12.

Definition 4.4. A network N is *tree-child* if every non-leaf node of N has at least one child that is a tree node.

As mentioned before, the aim of the problem is to construct a phylogenetic network that represents all the given trees and has the smallest possible number of reticulations. Constructing the tree-child sequences required to create such a network, requires knowledge about cherry picking sequences. Informally, a cherry picking sequence is a sequence of operations that are performed on the given set of trees. The sequence S consists of two kinds of pairs. On one hand the sequence contains pairs of the form (x, y), which denotes the operation of removing leaf x in all the trees that have $\{x, y\}$ as a cherry. On the other hand we have pairs of the form $(x, -)$, which is used when at least one of the trees in the set has been reduced to only the single leaf x. Formally, a cherry picking sequence is a sequence

$$S = \langle (x_1, y_1), (x_2, y_2), \ldots, (x_r, y_r), (x_{r+1}, -), (x_{r+2}, -), \ldots, (x_s, -) \rangle$$

with $\{x_1, x_2, \ldots, x_s, y_1, y_2, \ldots, y_r\} \subset X$ as introduced in [5]. We call the sequence *full* if $s > r$ and $\{x_1, x_2, \ldots, x_s\} = X$.

Given a cherry picking sequence S and a tree T, with T/S we denote the tree that we obtain after applying the operations of the sequence S to the tree T. However we also need to define when we have a cherry picking sequence for a set of trees. This is defined in the following way.

Definition 4.5. A full cherry picking sequence S is a cherry picking sequence for a set of trees if every tree T/S has a single leaf and that leaf is in $\{x_1, \ldots, x_s\}$.

We can also define the weight of a cherry picking sequence.

Definition 4.6. The *weight* of a cherry picking sequence is $w(S) = |S| - |X|$.

It was mentioned before that we would only consider tree-child networks, because of this the cherry picking sequence also has to be tree-child.

Definition 4.7. A cherry picking sequence S is *tree-child* if $s \leq r + 1$ and $y_j \neq x_i$ for all $1 \leq i \leq j \leq s$.

This shows that if a cherry picking sequence S for a set of trees is tree-child, then T/S consists of one leaf x_s for every tree T in the set of trees. We will also define when a leaf is forbidden with respect to the tree-child sequence S.

Definition 4.8. A leaf z is *forbidden* if it has already appeared as a first element in prior pairs in the sequence S.

We are now ready to introduce the hybridization numbers.

Definition 4.9. Let $\mathcal{T}$ be a set of trees. Then $h(\mathcal{T})$ is defined as the hybridization number of $\mathcal{T}$ and we define $h_{tc}(\mathcal{T})$ to be the tree-child hybridization number, meaning the smallest number of reticulations among all the tree-child networks representing $\mathcal{T}$.

By $s_{tc}(\mathcal{T})$ we denote the minimum weight of all possible tree-child cherry picking sequences for $\mathcal{T}$.
Linz and Semple [8] showed a connection between $s_{tc}(\mathcal{T})$ and $h_{tc}(\mathcal{T})$ in the following theorem.

Theorem 4.1. *Let X be a set of taxa and $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$ a collection of phylogenetic X-trees. then*

$$s_{tc}(\mathcal{T}) = h_{tc}(\mathcal{T})$$

4.2. **Introducing the Tree-Child Sequence problem.** In this problem we focus on finding a tree-child cherry picking sequence for a set of trees. Linz and Semple [8] showed that there exists a linear time algorithm that can compute a tree-child network given a tree-child cherry picking sequence and the corresponding set of trees.

4.2.1. *Introduction of the algorithm.* Now that we have discussed all the tools required to understand the algorithm we can finally present the pseudocode from [5].

Algorithm 1: TCS($\mathcal{T}, S, k$)

Input : A Collection of phylogenetic trees $\mathcal{T}$, a partial tree-child sequence S and an integer k

Output: An optimal solution of ($\mathcal{T}$,S) if ($\mathcal{T}$,S) has a solution of weight at most k; None otherwise

while *there exists a trivial cherry* $\{x, y\}$ *of* $\mathcal{T}/S$ *with* y *not forbidden with respect to* S **do**
  $S \leftarrow S \circ \langle (x, y) \rangle$
end
$\mathcal{T}' \leftarrow \mathcal{T}/S$
if $\mathcal{T}'$ *contains a cherry* $\{x, y\}$ *with* x, y *both forbidden with respect to* S **then**
  return *None*
else
  $n' \leftarrow |\{x \in X : x \text{ is a leaf of a tree in } \mathcal{T}'\}|$
  $k' \leftarrow |S| - |X| + n'$
  $C \leftarrow \{(x, y) | \{x, y\} \text{ is a cherry of some tree in } \mathcal{T}'\}$
  if $|C| = 0$ **then**
    return $S \circ \langle (x, y) \rangle$, *where* x *is the last remaining leaf in all trees*
  else if $|C| > 8k$ *or* $k' \geq k$ **then**
    return *None*
  else
    $S_{opt} \leftarrow$ None
    foreach $(x, y) \in C$ *with* y *not forbidden with respect to* S **do**
      $S_{temp} \leftarrow \text{TCS}(\mathcal{T}, S \circ \langle (x, y) \rangle, k)$
      if $w(S_{temp}) < w(S_{opt})$ **then**
        $S_{opt} \leftarrow S_{temp}$
      end
    end
    return S_{opt}
  end
end

This algorithm cannot benefit from machine learning as it is constructed in such a way that after selecting the trivial cherries it then exhaustively tries every possible cherry picking sequence before it returns the optimal sequence or it concludes that there exist no sequence with weight at most k. However, we can adjust the algorithm so that it will terminate whenever it has found a full cherry picking sequence of size at most k. The idea behind this is the following. Given a set of trees we want to create a phylogenetic network with the fewest number of reticulations. It is not a good idea to guess a k to start with, because if k is too large the algorithm can run incredibly long and if k is too small it will not find a sequence at all. A better technique is to increase k each time the algorithm was not able to find a sequence of size k, starting with $k = 1$. If we use this technique we can then safely adjust the algorithm to stop when a sequence of size at most k has been found. This is because if there would have been an optimal solution of size smaller than k, say $k - 1$, then that sequence would have been found when the algorithm was run with $k - 1$. This will give

us an algorithm than can benefit from machine learning and that can still return an optimal cherry picking sequence if we would gradually increase k.

Algorithm 2: TCS2($\mathcal{T}, S, k$)

Input : A Collection of phylogenetic trees $\mathcal{T}$, a partial tree-child sequence S and an integer k

Output: An optimal solution of ($\mathcal{T}$,S) if ($\mathcal{T}$,S) has a solution of weight at most k; None otherwise

```
while there exists a trivial cherry {x, y} of T/S with y not forbidden with respect to S do
    S ← S ∘ ⟨(x, y)⟩
end
T' ← T/S
if T' contains a cherry {x, y} with x, y both forbidden with respect to S then
    return None
else
    n' ← |{x ∈ X : x is a leaf of a tree in T'}|
    k' ← |S| − |X| + n'
    C ← {(x, y)|{x, y} is a cherry of some tree in T'}
    if |C| = 0 then
        return S ∘ ⟨(x, y)⟩, where x is the last remaining leaf in all trees
    else if |C| > 8k or k' ≥ k then
        return None
    else
        S_opt ← None
        foreach (x, y) ∈ C with y not forbidden with respect to S do
            S_temp ← TCS2(T, S ∘ ⟨(x, y)⟩, k)
            if w(S_temp) < w(S_opt) then
                S_opt ← S_temp
            end
            if w(S_opt) <= k then
                break
            end
        end
        return S_opt
    end
end
```

The question still remains where and how we can apply machine learning. This will be done in the next section.

4.2.2. *Introducing machine learning to the algorithm.* The first algorithm that was introduced had no way to benefit from machine learning, but with the adjustments introduced in the previous section the second algorithm, TCS2, can benefit from machine learning. This is because TCS2 no longer exhaustively tries every possible cherry picking sequence, but stops when a good enough sequence has been found. We can introduce machine learning into the

algorithm if we have to make a choice. In line 17 of the algorithm we will keep checking each element $(x, y) \in C$ and we stop checking if we have found a sequence of size $\leq k$. This means that if we correctly choose which elements from C to check first, then we will find a cherry picking sequence sooner. This is something a machine learning classifier can help us with. If the classifier can determine which pairs (x, y) we should use each time, then we can speed up the algorithm considerably. The classifier that will be used is again a decision tree that has been trained on generated data. We will now introduce the algorithm with the decision tree implementation and afterwards we will explain how we constructed the decision tree, training and test data.

The algorithm is almost completely the same as $TCS2$. The only difference is a newly introduced line between line 16 and 17 in $TCS2$, namely $C \leftarrow Predictor(C, \mathcal{T})$.
The new Predictor function that is introduced is the decision tree classifier and it works as follows:

(1) For each $(x, y) \in C$ the decision tree will predict either 1 or 0.
(2) C gets sorted according to the prediction. Pairs corresponding to 1 will be sorted in front of pairs that correspond to 0.
(3) The function returns the new sorted list C_{sorted}

This means that the new algorithm will start to check cherries that are deemed to be 'good' cherries according to the decision tree. We will now discuss how we generated the data and how we created the decision tree.

4.3. **Creating the decision tree classifier.** In this section we will discuss the creation of the decision tree. Firstly, we will start by introducing the features that will be used. Secondly, we will discuss the procedure that was used to generate the training and test data.

4.3.1. *Determining the features.* We need strong features to be able to grow a decision tree that performs well. It can be very difficult to determine features that help create the best model. In most cases it is not clear which features will perform well, therefore different features will be introduced to the model and during evaluation and testing we will see which features perform well and which features are redundant. We will first introduce the features and afterwards we will explain why we chose to use those specific features.

We have used the following features for growing the decision tree.

(1) In how many trees does the cherry appear, averaged over the total number of trees.
(2) In how many trees does x from the pair (x, y) appear in a different pair, averaged over the total number of trees.
(3) Average distance from x to y among all the trees, averaged over the total number of trees.
(4) Average size of the subtree beneath the lowest common ancestor among all the trees.

We will now discuss each feature individually. This will show our earliest thought process and later evaluation of the model will give a clear idea of which features are stronger than the others.

The first feature was a very clear choice for us. The operations of the form (x, y) remove leaf x in any tree that has cherry (x, y). We therefore had a feeling that the number of trees that contain the cherry that would be added to the cherry picking sequence determined whether

or not it was a good cherry to pick. We have averaged over the number of trees to avoid influencing the data because of a different number of trees. However, this should not be necessary when creating a decision tree, because a decision tree also finds structure in data that has not been normalised.

The second feature should not be mistaken as the complement of the first feature, because it is not required for the leaves of a cherry to also be contained in a cherry in all the other trees. Given a cherry (x, y), we believed that the number of cherries in which x appears and y does not is an indication for (x, y) being a good or bad cherry to add to the cherry picking sequence. We did not know if appearing in a few other cherries was a good indication or appearing in many other cherries.

As mentioned before, determining features is very difficult, because it is a game of trial and error. This means that some features that we tried had no real thought process other than that it is something that appears in all trees and can be checked quite easily. One of those features is the third feature. This feature looks at the distance between two leaves in all the trees. The only reason that we added this feature was the possibility that maybe to closer the leaves were in all trees the more likely it would be to add the corresponding cherry to the sequence. Testing will show whether or not this was a good choice.

The fourth feature is somewhat the same as the third feature, however, this features only focuses on the subtrees in which both x and y are present and looks at the size of the subtree and not the distance between x and y. We believed that the average size of the subtrees was a feature that gave structure to the data and could help determine whether or not a cherry should be added to the cherry picking sequence.

4.3.2. *Creating the data.* For the creation of the data we have used the techniques from [5]. Firstly, the techniques of generating networks and trees will be discussed. Secondly, the generating of the training and test data will be explained in detail.

The aim of the problem is to find a tree-child network with as few reticulations as possible, given a set of phylogenetic trees. To generate a set of trees that can be used to find a network we reverse this process. We start by constructing a network with a set number of reticulations and then retrieve the number of trees that we require from that network. We will start by explaining the network generation that is used in [5].

We start with a network N that is a tree with two leaves. A network N that has n leaves and k reticulations can then be obtained by adding $s = n + k - 2$ tree nodes and $k^{'} = k$ reticulations to N. Adding a tree node is done by selecting a leaf node u and adding two children v and w to u. This makes u a tree node and v and w are two new leafs. This means that adding a tree node results in increasing the number of leaves by 1. Adding a reticulation is done by selecting two leaves u and v and merging them together into the same node u and adding a child node w to parent u. Now u has become a reticulation and w has become a new leaf. This means that adding a reticulation results in decreasing the total number of leaves by 1. This shows that we indeed obtain a network N with n leaves and k reticulations if we add $n + k - 2$ tree-nodes and k reticulations to the network.

The total number of nodes that will be added that are not leaf nodes is $s + k^{'}$. As long as $s > 0$ and $k^{'} > 0$ we will add tree nodes and reticulations to the network. If we add a reticulation to the network we saw that we merged two leaves into one and therefore did not affect any other reticulations or tree nodes. This means that if we add one reticulation to the network, $k^{'}$ decreases by one and s stays the same. If we add a tree node to the network we saw that we used one leaf and introduced two children to the leaf, making it a tree node. this does not affect any other tree nodes or reticulations, thus s decreases by one and $k^{'}$ remains unchanged.

We need to ensure that the network that we create remains tree-child. To do this we make sure that the nodes that we merge when adding a reticulation are chosen from a set M of leaves whose parents are tree nodes. We also make sure to pick leaves that do not have the same parent. If it is not possible to add a reticulation then we add a tree node. If is is possible to add a reticulation, then we add a reticulation with probability $\frac{k^{'}}{k^{'}+s}$ and we add a tree node with probability $\frac{s}{s+k^{'}}$. If a tree node gets added to the network, the candidate leaf node gets selected uniformly at random. If a reticulation gets added to the network, the leaves get selected uniformly at random from set M.
The addition of tree nodes and reticulations continues until $s = 0$ or $k^{'} = 0$. If $k^{'} = 0$ and $s > 0$, tree nodes get added until $s = 0$. If $s = 0$ and $k^{'} > 0$, we add reticulations until $k^{'} = 0$ or until we can no longer add reticulations to the network. This can result in a network with fewer reticulations than k.

After we have generated the network N we can generate the trees we are going to use to generate the data. We generate t trees that are displayed by N in the following way.

(1) Delete one of the parent edges of each reticulation in N uniformly at random.
(2) Update the network by suppressing all the nodes with only one child node.
(3) If the tree already exists in the list, then we do not add the tree to the list.
(4) Maintain a counter that tracks the total number of time a tree is generated that is already present in the list.
(5) If the counter reaches 100 or if the list contains t trees, the process is terminated and the list will contain the trees.

Note that this procedure does not guarantee a list of t trees, but only a list of at most t trees. For example, if we have a network with 2 reticulations we cannot generate a list of ten trees without having duplicates. Another thing that should be mentioned is the fact that the list of trees that we generate can have a reticulation number that is smaller than than of the original network. This is because of two things. There is a possibility that we could no longer add a reticulation to the network and $k^{'} > 0$, this would result in a network N with a reticulation number that is smaller than k. On the other hand, we only return a subset of the trees that are displayed by N. This means that there can exist another network different from N, with fewer reticulations, that also display all the trees in the list.

Now that we know how to generate the test networks and trees, we can take a look at how we will use these to train the decision tree. We will now explain how we generate the data for one set of trees.

Cherry	Feature 1: Average #Trees with cherry	Feature 2: Average #Trees with different cherry	Feature 3: Average distance between leaves of cherry	Feature 4: Average size of LCA tree	Class:
(20,17)	0.18	0	4.82	6.1	1
(19,15)	0.55	0	3.73	4.27	1
(17,12)	0.82	0.18	2.18	2.18	1
(12,17)	0.81	0	2.18	2.18	0

TABLE 7. Samples of the generated training data.

(1) We start by generating the partial cherry picking sequence S by adding all the trivial cherries to S.
(2) We apply S to the set of trees $\mathcal{T}$ to obtain $\mathcal{T}/S$.
(3) We determine all pairs (x, y) that are present in $\mathcal{T}/S$.
(4) For each pair (x, y) we create $S' = S+\langle(x, y)\rangle$ and we run the algorithm $TCS2(\mathcal{T}, S', k)$ and keeping track of the size of the search tree in the algorithm.
(5) The top 50% that have both the smallest size search tree and a solution will be considered a good pair and will be labeled as 1. The other pairs will be labeled as 0.

This means that we will obtain data that shows how long the program runs if we select the first pair and afterwards let the algorithm run naively. We figured that it was not necessary to add data where we tracked the running time after the first choice as made, because the structure of the problem does not change after we make one choice. Therefore we believe that the model can learn to distinguish between 0 and 1 if we only use this data. This makes the problem less complicated and makes sure that we do not get an overflow of data.

If we now generate multiple networks we can generate more data to train the model. In many cases the algorithm takes very long to find a solution, therefore in generating the data we gave the algorithm a maximum time of five minutes to find a solution. If the algorithm failed to find a solution within five minutes we classified the pair as a 0.

The data that we have generated to train the model had the following structure:

- We generated 200 networks with 2 to 7 reticulations and 25 to 35 leaves.
- We extracted 2 to 20 trees from the networks to create the tree lists.

Afterwards, for each tree list we repeated the procedure explained above to classify each pair (x, y) as 0 or 1. Finally we added the features to each data sample and saved it into an array to use as training data. Table 7 shows a few samples of the data. In some cases the instances were to difficult to solve, resulting in many None instances. This issue was resolved by upsampling the data, which we will discuss next.

The decision tree was created in the same way as for the Maximum Agreement Forest problem, using SkLearn and the DecisionTreeClassifier module. Evaluating the data showed that we had a total of 1536 usable samples of which 988 were classified as 0 and 548 were classified as 1. Before preparing the training and test set we made sure that the data had an equal distribution of 0's and 1's. As mentioned before, to achieve this we upsampled the 1's to create 988 samples with the 1 classification. This gave us a total of 1976 samples. The data set is split into a training and a test set. The training set holds 70 percent of the

Instance	1	2	3	4	5	6	7	8	9	10	mean
Score	0.67	0.70	0.66	0.72	0.65	0.70	0.67	0.72	0.70	0.69	0.69

TABLE 8. Results from the 10-fold cross validation.

Occurrence	True positive	True negative	False positive	False negative	Correct	% Correct	%False positives	%False negatives
	205	198	98	92	403	68	16.5	15.5

TABLE 9. Results from the accuracy score evaluation.

samples and the test set holds the remaining samples. Both the training and test set have an even representation of 1's and 0's. The training set is then used to grow a decision tree and the test set is later used as a final evaluation set before we introduce unseen data.

4.4. **Evaluation of the model.** In this section we will firstly evaluate the machine learning model on how well it performs. We do this by using k-fold cross validation and looking at two performance measures, namely the accuracy score and the AUROC score of the model. When we compute the accuracy score we will also take a look at the percentage of false positives and false negatives. It should be noted that there are many more ways to evaluate the model, however, we found that these scores gave the best representation of the model. The accuracy score simply gives us an idea on how well the model finds true positives and true negatives and the AUROC score gives us an idea on how well the model can distinguish between 0 and 1. This is different from the accuracy score as the AUROC score also takes the rate of false positives into account, which the accuracy score does not.

4.4.1. *10-fold cross validation.* In this section we will focus on the machine learning model. Here we do not introduce new data to the model, but we evaluate the model using the training and the test data. We start by performing k-fold cross validation. We take $k = 10$ and we use the accuracy performance score. Table 8 shows all the individual scores and the final mean score. The cross validation shows that the model performs reasonably well on "unseen" data from the training set with an average accuracy score of around 70 percent. This shows that the model does not overfit the data and that we should not be worried about a biased model. To further evaluate the model we will now take a look at how well the model performs on the test set by computing the accuracy score and the AUROC score respectively. With the 10-fold cross validation we grew a decision tree using a part of the training data. To determine the accuracy score and the AUROC score we grow a tree using the complete training data.

4.4.2. *The accuracy score.* We will now evaluate how many instances from the test data are predicted correctly by the model and how many false positives and false negatives are produced. The test data had a total of 593 samples. The results from the evaluation can be found in Table 9. We see that the model performs as well as the 10-fold cross validation showed. We achieve an accuracy of roughly 70 percent with the current model and we see that we have an even spread of false positives and false negatives. After we have discussed the AUROC score of the model we will briefly discuss the results.

True Class	Probability 0	Probability 1
0	1	0
0	1	0
1	0.35	0.65
0	0.29	0.71
0	0.61	0.39
0	0	1

TABLE 10. Part of the calculated probabilities from the test data.

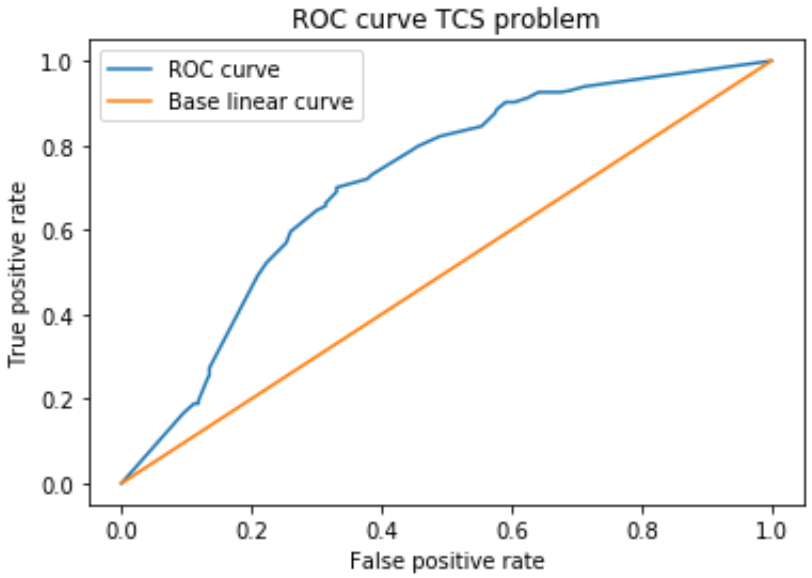

FIGURE 13. The ROC curve created from the test data.

4.4.3. *The AUROC score.* The AUROC score is a very difficult concept to understand, but it is one of the most used evaluation methods. This is why we will also compute the AUROC score for our model to get and idea how well it can distinguish between a 0 and a 1. We again use the test data to determine the score, but instead of predicting a 0 or 1, we predict the probability that a sample in the test data gets classified as a 0 or a 1. This is done by looking at the fraction of 0's and 1's in the leaf that a sample ends up in. Table 10 shows a small part of the predicted probabilities for the data samples.
We can then use the probability predictions of the 1 class and the real test data classes to compute the true positive rates and the false positive rates for a different number of thresholds and thus create the ROC curve. Figure 13 shows the ROC curve. The linear curve shows the curve for which every sample would randomly be classified as either 1 or 0. The AUROC score is the area under the ROC curve, which we calculate using a SkLearn module from python. This gives us an AUROC score of 0.7136 or 71.36 percent. This means that the model can distinguish between 0 and 1 rather well.

The first tests show very promising results for the model. It shows that the model has predictive capability and truly can distinguish between the different classes. In the next section we will evaluate the performance of the machine learning algorithm. There was also the issue of false positives and false negatives. This will also be addressed in the next section, because they have an influence on the performance of the algorithm.

4.5. **Evaluating the algorithm.** In this section we will discuss the performance of the tree-child sequence algorithm with and without the machine learning implementation. We start by explaining how we created the data used for the experiments and how the experimental runs were performed. Afterwards we will discuss the results and evaluate whether or not the algorithm can benefit from a machine learning implementation.

4.5.1. *The experiment.* We will now discuss what data we used to perform the experiments with the algorithms. For the test we created multiple data sets each with a different amount of reticulations. We used 2 to 7 reticulations and 2 to 11 trees. This means that for each reticulation number we got 10 different test instances. The procedure for reticulation number k went as follows:

(1) For each $i \in \{2, 3, \ldots, 10, 11\}$ generate a network with 20 to 25 leaves and k reticulations.
(2) Extract i trees from the network.

The networks were created according to the same procedure that was used to generate the data that trained the machine learning model. This will result in a total of 60 instances that will be used to test the algorithms. We will test both algorithms on two things, namely the run time in seconds and the size of the search tree, which can also be interpreted as the number of recursion calls that are made by the algorithms. For each instance we give both algorithms a maximum of ten minutes to solve the instance. The pseudo code introduced in this chapter does not contain the implementation that tracks the number of recursion calls that the algorithms make and it also does not contain the implementation that makes sure that the algorithms do not run longer than ten minutes. This are two very simple implementations and are not significant for the algorithms and are therefore omitted from the pseudo code.

Now that we know how the data is generated to test the algorithms and how we run the algorithms we can take a look at the results.

4.5.2. *Results.* We will now discuss the results from the experiments that were introduced in the previous subsection. We will first start by evaluating how many cases could be solved by both algorithms, how many cases could only be solved by the regular algorithm or the machine learning algorithm and finally how many cases could not be solved by both algorithms. The results can be found in Table 11.
This does not tell us anything about the true performance of both algorithms. This does, however, show that the machine learning algorithm is very promising, because it could solve instances that the regular algorithm could not solve. We will now take a look at the individual run times and the sizes of the search trees. The results of the run time are captured in Figures 14 and 15 and the results of the size of the search trees are captured in Figures 16 and 17. Some of the instances seem to be unsolved by both algorithms, but there are some instances that were solved by the machine learning algorithm just before hitting the 10 minute mark.

Reticulation	Both Solved	Regular Solved	Machine learning Solved	Neither Solved
2	10	-	-	-
3	8	-	1	1
4	6	-	1	3
5	4	-	-	6
6	3	-	1	6
7	-	-	3	7

TABLE 11. This shows the number of solved and unsolved instances.

This occurred very few times and therefore we still believe that the figures represent the data quite well. The figures show that in most cases the machine learning algorithm performs very well. To get a better idea of how the algorithms perform we will also take a look how they perform on average over all the instances for each reticulation. We will capture these results in a table to get a clear overview of the performance.

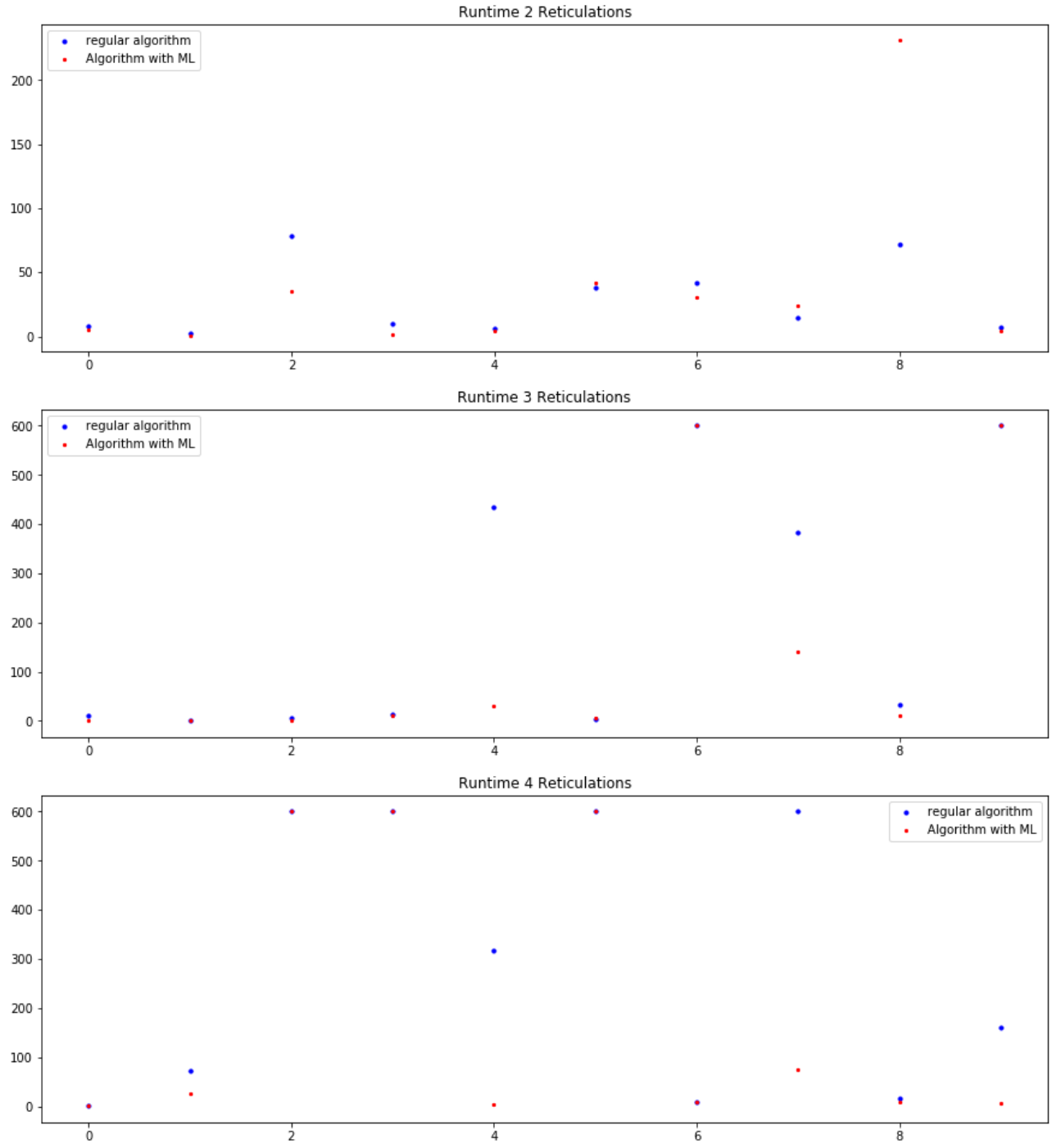

FIGURE 14. Runtimes for reticulations 1 to 4.

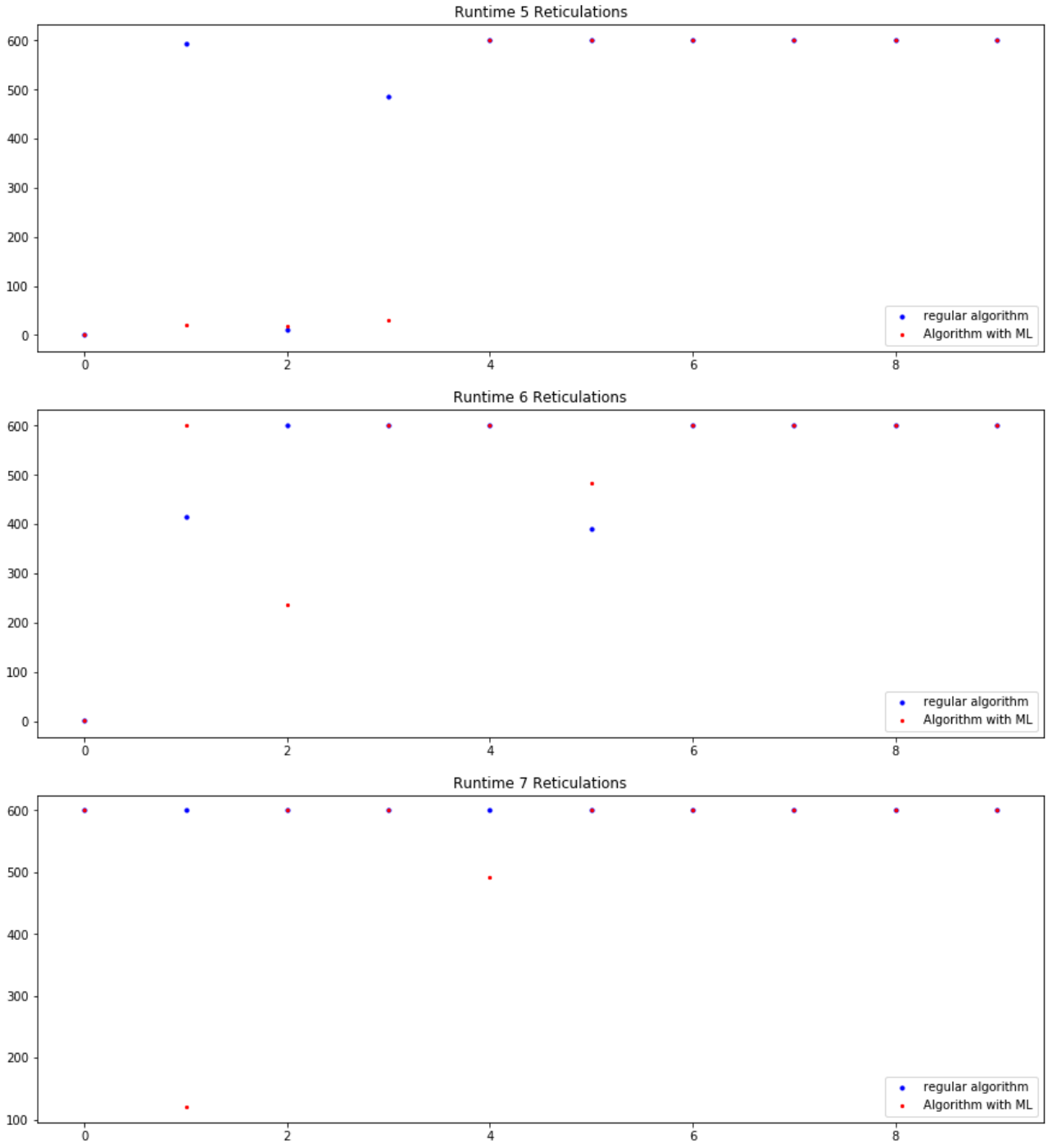

FIGURE 15. Run times for reticulations 5 to 7.

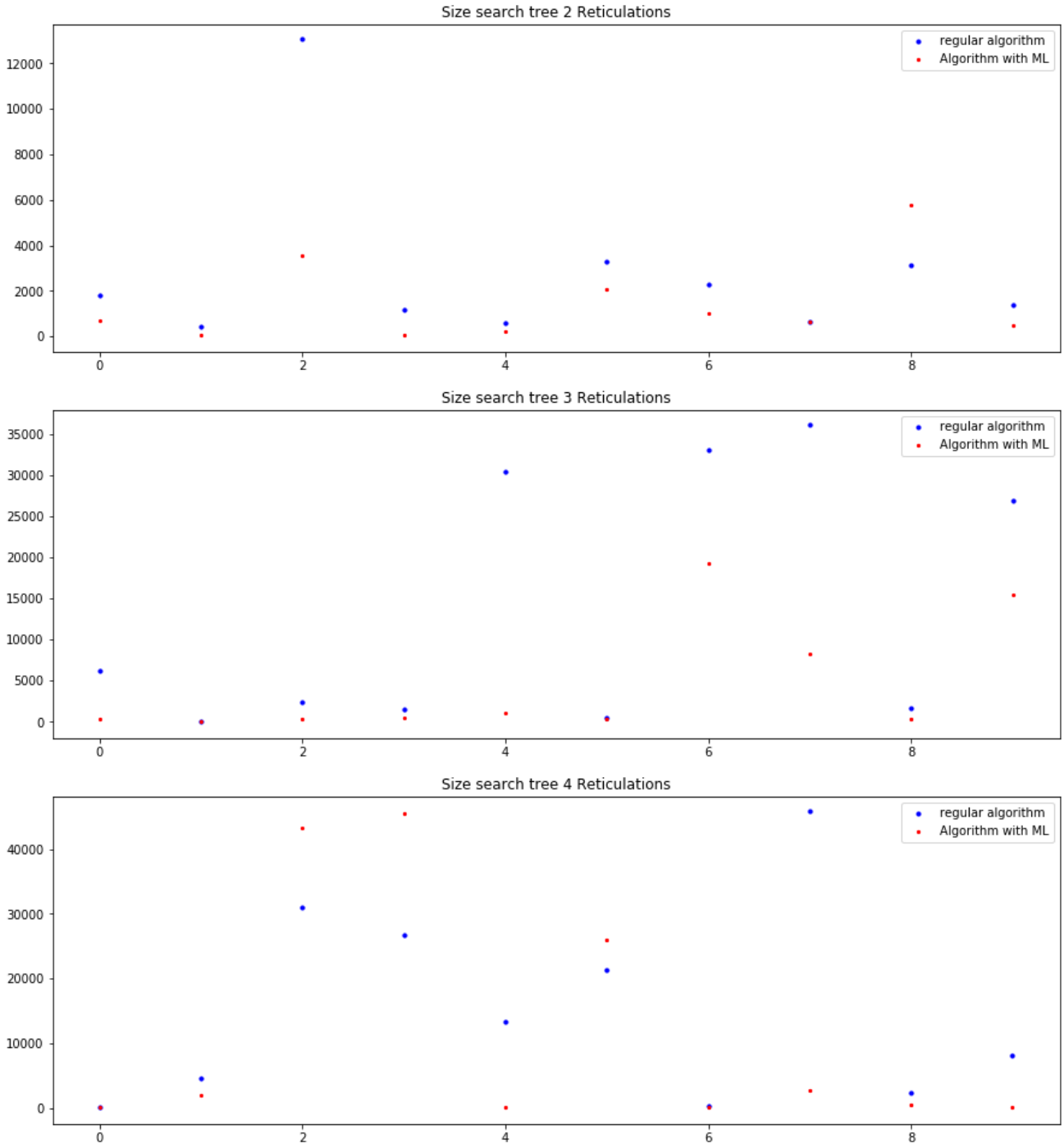

FIGURE 16. Size of the search tree for reticulations 1 to 4.

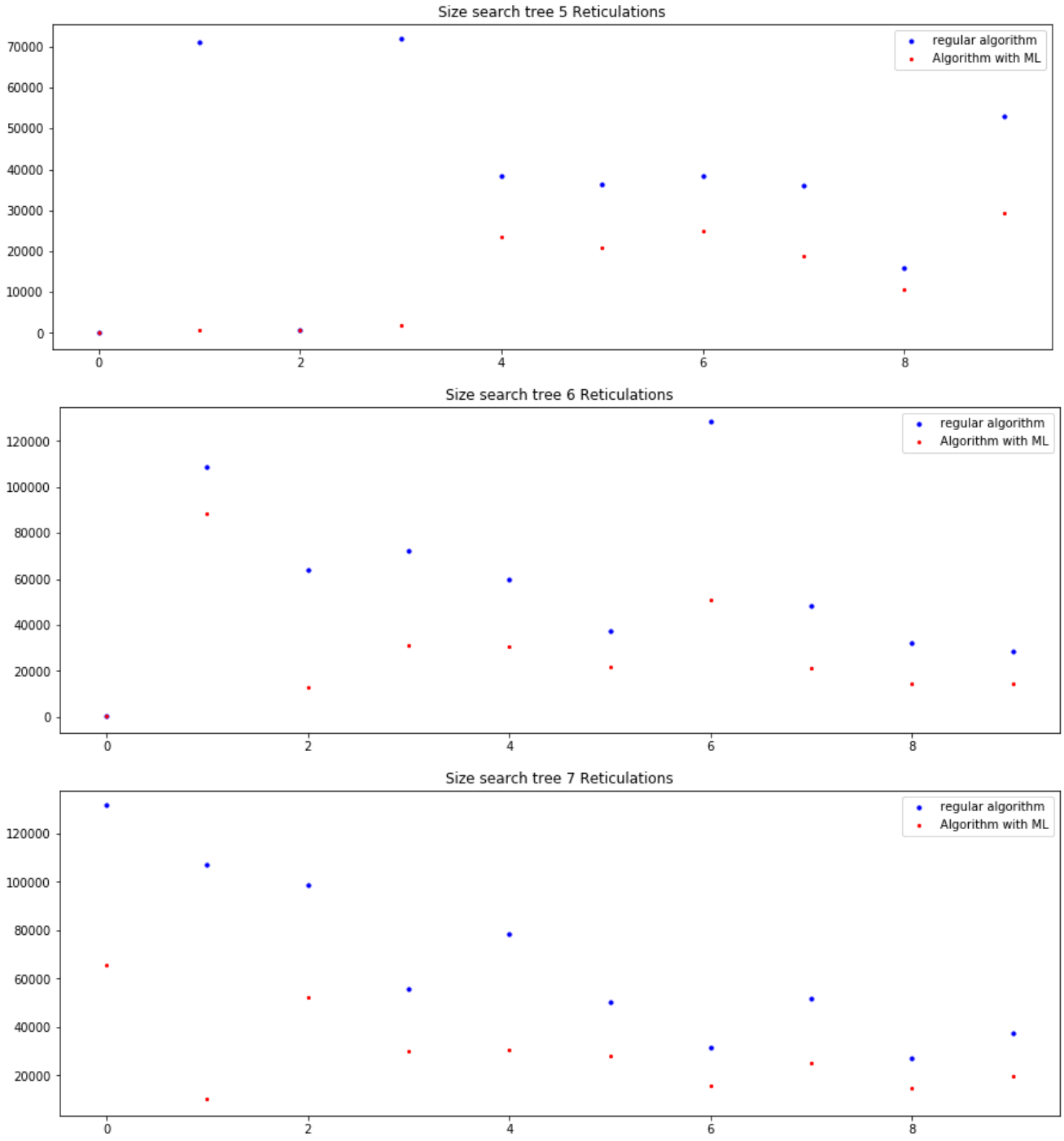

FIGURE 17. Size of the search tree for reticulations 5 to 7.

The table will hold the following data for each reticulation number:

(1) The number of instances solved by the regular algorithm.
(2) The number of instances solved by the machine learning algorithm.
(3) The average run time of the regular algorithm in seconds.
(4) The average run time of the machine learning algorithm in seconds.
(5) The average size of the search tree for instances solved by the regular algorithm.
(6) The average size of the search tree for instances solved by the machine learning algorithm.
(7) The average size of the search tree for unsolved instances for the regular algorithm.
(8) The average size of the search tree for unsolved instances for the machine learning algorithm.

The results can be found in Table 12. We see some interesting things when looking at the results in the table. In many cases we see that the machine learning algorithm performs much faster than the regular algorithm and we see that the machine learning algorithm requires far less recursions to find a correct solution. This shows that the machine learning program is very promising and also shows that it does not perform worse than the regular algorithm. Therefore we believe the that the tree-child cherry picking sequence algorithm can benefit from a machine learning implementation and we believe that the results support this claim.

One thing should be mentioned regarding the possibilities of having false positives and false negatives. We believe that false negatives will not affect the performance of the algorithm in a major way, because the way that the algorithm is constructed makes sure that if we missclassify a cherry as a 0 that it will still be used in the algorithm if a correct solution is not yet found. We do believe, however, that false positives can hurt the performance of the algorithm more severely, because picking a bad cherry can result in many useless recursion calls that could have been avoided if we had classified the cherry correctly. Therefore, we believe that the current algorithm performs well, but will perform better if we can reduce the false positive rate of the model.

In the following section we introduce the tail move problem and a possible application of machine learning to aid in improving the current solutions.

Reticulations	S I Reg alg	S I ML alg	Av Time Reg alg (sec)	Av Time ML alg (sec)	Av size ST Reg alg Solved	Av size ST ML alg Solved	Av size ST Reg alg Unsolved	Av size ST ML alg Unsolved
2	10	10	**28**	38	2784	**1464**	-	-
3	8	9	111	**89**	9844	**3381**	29994	15436
4	6	7	96	**19**	4863	**869**	31251	38226
5	4	4	272	**18**	35981	**854**	36320	21348
6	3	4	**269**	330	48954	**30819**	61998	27193
7	0	3	-	**404**	-	**35285**	66980	26484

TABLE 12. The average results for each reticulation number where the best performances are given in bold.

5. The tail move problem

Within the area of phylogenetics we would like to be able to explore the space of rooted trees and networks. Methods that are often used that explore these spaces make use of a tool that we have encountered before, namely rearrangement moves. One of the possible rearrangement moves is rooted subtree prune and regraft move, which we introduced in the first section. However there are more rearrangement moves, for example, the rooted nearest neighbour interchange (rNNI). For any two rooted network with the same number of reticulations it has been shown that they are connected by a sequence of rSPR and also by a sequence of rNNI moves. Janssen et al. [6] showed that it is also possible to find a connection that only uses tail moves and have also given an upper bound on the number of moves required.

The proof of this upper bound gives us a heuristic that can be used to obtain a sequence of tail moves that transforms one network into the other. In this section we will start by introducing some theoretic concepts before we take a look at the heuristic. Afterwards we will introduce the heuristic and discuss possible ways to introduce machine learning to the heuristic and how to grow a decision tree for this problem and finally we will compare the performance of the two variants of the heuristic to that of the heuristic that uses machine learning.

5.1. **Theory for the tail move problem.**

In this section we will introduce the theory that is needed to get an understanding of the tail move problem and the heuristic that can be used to get a solution for this problem. We will introduce the theory used in [6].
We want to know with what kind of networks we are working when we are studying the tail move problem. For this we introduce the tier of a network.

Definition 5.1. A network N is of the *kth tier* if the network has k reticulations.

The heuristic that we will introduce uses a certain type of set, namely the downward closed set. We will now define this set.

Definition 5.2. Given a network N and a set Y containing nodes of N. We say that Y is *downward closed* if for every $u \in Y$ the children of u with respect to N are also in Y.

Not every edge in the networks can be moved around freely. Therefore, we need to define when an edge is movable. We will first take a look at general movability before we restrict ourselves to tail moves.
To avoid confusion we will also quickly introduce what parallel edges are.

Definition 5.3. Two edges e and f are *parallel* if they have the same tail and head node.

Definition 5.4. Given a network N and an edge $e = (u, v)$ in N. We say that e is non-movable if u is the root of N, if u is a reticulation or if removal of the e, i.e. moving the edge, and suppression of u results in parallel edges.

Luckily, there is only one way that a tail move can result in parallel edges. Say that we want to move edge (u, v), then the only way to obtain parallel edges is if there is an edge from the parent of u to a child of u other than v. It is easy to see that we indeed obtain a parallel edge. Let (x, u), (x, z), (u, v) and (u, z) be edges in a network N. Note that (x, u), (x, z) and (u, z) form a triangle in N. If we were to move the tail of (u, v), the following happens:

(1) We move edge (u, v).
(2) Node u now has in degree and out degree equal to 1 and is contained in triangle xuz
(3) Suppressing u now results in parallel edges from x to z.

This phenomenon is introduced in the paper [6] in the following way:

Definition 5.5. Given a network N and nodes u, v, w of N. We say that u, v and w form a **triangle** in N if there exist edges (u, v), (u, w) and (v, w) in N. The edge (u, w) is called the **long** edge and the edge (v, w) is called the **bottom** edge of the triangle.

So far only the possibility to move an edge has been discussed, however, the movability of an edge does not guarantee that the tail of an edge can be moved to another edge. The following criteria guarantee that we can move the tail of an edge to another edge.

Definition 5.6. Given an edge $e = (u, v)$. We say that we can move the tail of e to another edge $f = (w, x)$ if and only if the following conditions hold:

(1) The edge e is movable;
(2) The edge f is not below v, meaning that w and x do not have v as a common ancestor;
(3) $x \neq v$.

It is clear why we need the first criterion. The second one makes sure that we do not create cycles within the graph. To see this we take a look at the simple directed path $P = \{(a, u), (u, v), (v, w), (w, x)\}$. If we now move the tail of (u, v) to (w, x) the following happens:

(1) We move the tail of (u, v) to (w, x), splitting edge (w, x) into (w, u') and $(u'x)$.
(2) We also have edge $(u'v)$ and after suppressing u we have edge (a, v).
(3) We now have a cycle $(v, w), (w, u'), (u', v)$.

If f is below v we will always have such a simple directed path. The third criterion assures that no parallel edges will be created.
In the first chapter we have already introduced what a tail move is and we now have all the tools to introduce the tail move problem.

5.2. **Introducing the tail move problem.** We will now formally introduce the tail move problem. Given two phylogenetic networks on X of the kth tier we want to find a sequence of tail moves that transforms one network into the other. As of this moment, there is no known algorithm that finds the shortest possible sequence of tail moves. However, as mentioned in the introduction, there exists a heuristic that gives us a sequence of tail moves. In this section we will take a look at the heuristic that was found by Remie Janssen, Mark Jones, Péter L. Erdős, Leo van Iersel and Celine Scornavacca [6]. We will not go into detail regarding the proof presented in the paper, however, because the proof is constructive we can obtain a heuristic from it that we will be evaluating.

5.2.1. *Informal idea of the heuristic.* We will use the same name for the heuristic as the paper [6], namely the green line heuristic. The name comes from the idea of creating a green line in the graphs and constructing an isomorphism between the parts of the graphs below the green line. We do this in the following way. Say that we have two networks N and N'

- Create an isomorphism between the leaf sets of both networks. This is possible, because both networks have the same number of leaves. This is our green line.
- Find a reticulation or tree node that is just above this green line in, for example, N.

- Find a corresponding reticulation or tree node in the other network. Here we will keep adding tail moves to the tail move sequence until we can extend the isomorphism.
- Extend the isomorphism.
- Repeat until we have an isomorphism between N and N'.

This is a very compact and informal introduction to the heuristic, but this is exactly the core that is required to find a sequence of tail moves. In the next subsection we will introduce the heuristic in a more detailed way. However, this will not explain how to specifically find the move that is added to the tail move sequence. The program that is used to generate the sequences has a clear structure and shows exactly what edges are moved to which location. The aim of the detailed introduction is to show where the heuristic can benefit from machine learning and how we want to implement that.

5.2.2. *Detailed introduction of the heuristic.* We will first introduce the heuristic as it is presented in the paper [6]. Afterwards some changes will be introduced that in many instances improves the performance.
The heuristic can be split into four cases, however, a few cases are symmetrical and therefore we will only introduce two cases. We will consider two networks. N is the network we move from and N' is the network we want to move to. The heuristic also uses downward closed sets. Y will be the downwards closed set corresponding to N and Y' is the downwards closed set corresponding to N'.

We call u in $N \setminus Y$ a lowest node if all descendants of u are contained in Y. The same holds for u' in N'. This will be used in the heuristic to select the nodes that will extend the isomorphism. The notation $N[Y]$ means the subgraph of N obtained by removing the nodes from N not in Y and the corresponding edges.

We start by creating the sets Y and Y'. The sets consist of the leaves of N and N' respectively. Now the heuristic divides in four main cases.

(1) **There exists a lowest node u' in $N' \setminus Y'$, such that u' is a reticulation**. This means that u' has a single child x' in Y' and therefore there must exist an x in Y such that x' is mapped to x by the isomorphism constructed thus far. By construction, this node x will have the same number of parents in N as x' has in N' and the same holds for the number of parents in Y and Y' respectively. This means that x must have at leas one parent, say z, that is not in the downward closed set Y. Now there are a few possible subcases that can occur.
   (a) z **is a reticulation**. This is a very quick case, because if z is also a reticulation we can extend the isomorphism very easily. If we let $Y_1 = Y \cup z$ and $Y'_1 = Y' \cup u'$ we extend the isomorphism between $N[Y]$ and $N'[Y']$ to an isomorphism between $N[Y_1]$ and $N'[Y'_1]$. This means we have to perform no tail moves in this case.
   (b) z **is not a reticulation**. This must mean that z is a tree node. In this case we can have two subcases. Either the edge (z, x) is movable or (z, x) is not movable, because doing so would result in parallel edges.
      (i) (z, x) **is movable**. In this case we search for a reticulation u in $N \setminus Y$. We know that $N \setminus Y$ must contain a reticulation because u' is a reticulation in $N' \setminus Y'$ and N and N' have the same number of reticulations. We let v

be the child of u. If $v = x$ we are in case 1a if we substitute z by v. Now we will consider what happens if $v \neq x$. If $z = v$, we let v be the child of z other than x and we let w be a parent of u other than z. This gives us then two moves to perform before we can extend the isomorphism, namely $((z, x), z, (u, v))$ and $((z, v), z, (w, u))$. So case 1bi leads to two internal tail moves. We extend the isomorphism by adding u to Y and u' to Y'.

(ii) (z, x) **is not movable**. Because moving the edge would result in parallel edges, this must mean that there exists two nodes c, d other than x that form a triangle together with z, where (c, d) is the long edge. Note that c has outdegree 2, meaning that it cannot be the root of N and therefore c has a parent node b. Now we have two subcases, namely b is not the root of the network and b is the root of the network.

(A) b **is not the root of** N. This means that b has a parent in N and edge (c, d) is movable. If we now perform the move $((c, d), c, (a, b))$ we end up in case 1bi and we end up with a total of 3 internal tail moves. We extend the isomorphism by adding u to Y and u' to Y'.

(B) b **is the root of** N. This is a more complicated case. One should note that d is not yet in the isomorphism. Let e be the child of d in N. If x is a tree node, let s and t be its children. Then performing the moves $((x, s), x, (d, e))$,$((x, e), x, (z, t))$ and $((x, t), x, (d, s))$ gives us a new network N such that we can extend the isomorphism by adding d to Y and u' to Y'. If e is a tree node, let s and t be the children of e. Then preforming the moves $((e, s), e, (z, x))$,$((e, x), e, (d, t))$ and $((e, t), e, (z, s))$ results in a network N such that we can extend the isomorphism by adding d to Y and u' to Y'.

(2) **None of the lowest nodes in both** $N \setminus Y$ **and** $N' \setminus Y'$ **is a reticulation**. This must mean that all the lowest nodes in both cases are tree nodes. We let u' be an arbitrary lowest node of $N' \setminus Y'$. u' is a tree node and therefore has two children in Y'. Let x' and y' be these children. As x' is in Y' that means that there exist an x in Y. Again x has the same number of parents in Y as x' has in Y' and x also has a parent that is not in Y, the same goes for y. Now there are a few subcases that we must consider.

(a) x **and** y **have a common parent that is not yet in** Y. Let u be this common parent. This is a very simple case. If we let $Y_1 = Y \cup u$ and $Y_1' = Y' \cup u'$ we get an extension of the isomorphism from $N[Y_1]$ to $N[Y_1']$.

(b) x **and** y **do not a have common parent**. This means that x and y have different parents that are both not in Y, let these parents be denoted by z_x and z_y. Now we have three subcases to consider, namely (z_x, x) is movable, (z_y, y) is movable or neither are movable.

(i) **The edge** (z_x, x) **is movable**. If this is the case then we can move z_x to the edge (z_y, y). If we perform this move then x and y have a common parent that is not in Y and this returns us to the first case, giving us one extra internal tail move before we can extend the isomorphism by adding z_x to Y and u' to Y'.

(ii) **The edge** (z_y, y) **is movable**. This case is symmetric to the previous case and also results in one internal tail move before we can extend the isomorphism by adding z_y to Y and u' to Y'.

(iii) **Both** (z_x, x) **and** (z_y, y) **are not movable**. This means that both of the edges hang to the side of a triangle. We want to find the top node of the triangle for both x and y. For x, let c_x be the parent of z_x in N and let b_x be the parent of c_x in N. For y we obtain c_y and b_y in the same way. If c_x is the child of the root node we need to swap the role of x and y, meaning that we get $(x, y = y, x)$, $(z_x, z_y = z_y, z_x)$, $(b_x, b_y = b_y, b_x)$ and $(c_x, c_y = c_y, c_x)$. Finally we find the parent of b_x denoted by a_x and the child of c_x other than x denoted by d_x. Performing the moves $((c_x, d_x), c_x, (a_x, b_x))$ and $((z_x, x), z_x, (z_y, y))$ will result in a network N such that we can extend the isomorphism by adding z_x to Y and u' to Y'.

There are two other cases that are entirely symmetric to the two cases that are discussed above. Instead of looking at a lowest node u' in $N' \setminus Y'$ that is a reticulation, we look at a lowest node u in $N \setminus Y$ that is a reticulation and instead of picking an arbitrary lowest tree node u' of $N' \setminus Y'$, we pick an arbitrary lowest tree node u of $N \setminus Y$.

In many parts of the algorithm an arbitrary choice is made when selecting nodes. For example, in case 1 when z is not a reticulation we search for an arbitrary reticulation in $N \setminus Y$. However, we have coded this in Python and there does not exist something called "arbitrary" in Python or other programming languages. The list of possible reticulations is always generated in the same way and therefore the same node is always selected. A positive point of this method is that we always get the same result when running the algorithm. A negative of this point should be crystal clear, however. In no way do we know that the first encountered reticulation results in the best possible choice. This means that we want to get rid of the arbitrary choice by replacing it firstly by a random choice and secondly by a machine learning decision. The decision based learner will decide on randomly selected nodes whether or not it is a good one. To refine the choice of nodes as best as possible we will work with two different decision trees. Before we can take a look at the results we will take a look at the process of creating the decision trees used to make the decisions.

5.3. **Creating the decision trees.** In this subsection we will take a look at the process of creating the decision trees. Firstly we will discuss how the data used for training data is generated and after that we will discuss the features that were used to grow the final decision trees. Finally the performance of the trees is presented based on the accuracy, 10-fold cross validation and the AUROC score.

5.3.1. *Generating the data.* The data generation is done is a similar way as with the hybridization number problem. Again we generate a network with k reticulations in the same way as described in section 4.3.2 and [5], however we do not require the network to be tree child and therefore we can have a much more consistent network creation that guarantees the number of reticulations that we want. After we have created the desired network we create the target network by performing a number of tail moves. We were able to use a very good implementation of the heuristic and tail moves thanks to Remie Janssen. This saved incredible amounts of time and gave us the opportunity to try out many different data sets.

After generating the networks we needed features that we were going to use to create the data. The aim is to improve the practical performance of the heuristic. To this end we decided to create data that did not look at what nodes needed to be moved, but looked at pairing of nodes to decide if certain moves were correct. Take for example the first case of the heuristic. In this case we obtain u' which is a reticulation node in $N' \setminus Y'$. Afterwards we obtain a parent z in $N \setminus Y$. If z was a reticulation then we were done. This pairing of nodes is then denoted by (u', z, None). The next case will make it clear why the tuple contains a None instance. In case 1b we find that the parent z is not a reticulation and this means that we need to find a node u in $N \setminus Y$ that is a reticulation. This would give the pairing (u', z, u). To more easily handle the data we would also track which case and networks the tuple belonged to. This resulted in final tuples that looked like $(u', z, u, 1bi, N, N')$, where $1bi$ indicates the case we are in. This means the data generation worked as follows after the networks were created:

(1) Obtain all lowest nodes in both N' and N, where N' was our target network.
(2) For each lowest node determine all possible cases that can occur. For example, case 1bi could occur multiple times with multiple different reticulations u.
(3) Each of the pairings functioned as the first move of the heuristic. To make this clear, take the pairing $(u', z, u, 1bi)$. If we were to start with this pairing we would have to follow the heuristic to case 1bi and thus we had to perform two internal tail moves before we could extend the isomorphism. This means that each of the pairings fixes the first extension of the isomorphism.
(4) After the first fixed extension is performed we run the algorithm using a random selection, meaning that from the possible lowest nodes a random node is selected.
(5) Finally the length of the tail move sequence is recorded and used to give an indication on how good the selected pairing was.

To take the randomness of the heuristic into account we came up with a clever work around to guarantee we got a good indication of each pairing. We created small networks with 8 to 12 leaves and with only 3 to 5 reticulations and ran the heuristic 500 times and recorded all the lengths of the tail move sequences. Finally we selected the smallest value of the set of all tail move sequence lengths. This way we made sure that the random element from the heuristic had a very small impact on the resulting test data. We had to do it this way, because we had no other way to find tail move sequences in a timely manner.

The final data set that was used to grow the best decision trees used 500 networks with the aforementioned number of leaves and reticulations. The target network were created by performing 10 tail moves on each network. This gave us a total of 500 network pairs. For each of the network pairs we could generate the data in the way that was previously discussed. This resulted in a number of pairings and after running the heuristic for each of these pairings we knew exactly which of the pairings seemed good and which seemed worse. However, growing a decision tree still requires features, which we will discuss right now.

It was quite difficult to come up with good features that captured enough information for the decision trees to make accurate decisions. In the end we came up with six features that were primarily focused on selecting the correct reticulations. We will first introduce the features and afterwards discuss each individually. After we have discussed all the features we will address an issue regarding tree node selection.

(1) The number of internal tail moves that the heuristic needs to make before we can extend the isomorphism.
(2) The distance between the second and third element of a pairing. For example, the distance between z and u in the pairing $(u', z, u, 1bi)$. The distance is 0 if the third element is None. This was normalised by the number of nodes that were not yet in the isomorphism.
(3) The difference in depth of u' and z, if z is a reticulation. If z is not a reticulation we take the difference in depth of u' and u.
(4) Indication of case 1.
(5) Indication of case 2.
(6) Indication of case 3.

We came up with these features, because we believed that these features were very important for the problem and were also quite easy to compute.
The first feature says something about the number of tail moves that we are making. We hoped that this had a clear connection with the final number of tail moves in a sense that many internal tail moves would be considered bad. This meant that we hoped that the decision tree would firstly regard reticulations before it would move on to tree nodes.
The second features says something about the distance of the moves that need to be made. We suspected that the shorter distance moves would be regarded as better and therefore believed that this would be a good feature.
The third feature also says something about how the nodes are structured in both networks. We hoped that nodes that were around the same depths in both networks would result in good pairings and therefore we regarded as good nodes to select in our decision trees.
The final three features were simply to help the decision tree to distinguish between the three cases. Using these features the decision trees would learn what type of cases were better to select first before other cases. We chose for this split to obtain clear data, but we know that case 1 and case 2 are symmetrical and therefore it should not matter if we would consider case 1 before case 2 or the other way around. We believe that these are good features to train the decision trees to pick the right reticulations and hopefully the right tree nodes.

We are still figuring out a way to select tree nodes. Currently we are using the same features for the tree nodes and the reticulations. However, the case for tree nodes is a little bit different than the case that uses reticulations. Luckily, case 3 does result in the same number of nodes in the pairing and therefore we are able to use the same features and in that sense we trick the decision trees to also learn how to classify tree nodes. Currently we have not solved this issue and thus we train the decision trees in this way to get some preliminary results that can show the benefit of machine learning.

After adding all the features to the generated pairings we were left with labeling the data. The data will be labelled two times, because two different trees will be used. In the first data set a threshold of 5% is used to label the data, this means that the 5% best pairs get a 1 and the rest is labelled as 0. In the second data set a threshold of 5% is used. Table 13 shows a small portion of the final data that was used to grow the decision trees.
For growing the decision trees we split the data into 70 percent training data and 30 percent test data. Now that we have an idea of how the data is prepared we can take a look at the performance of the grown decision trees.

Instance	Datapoint	length sequence	feat 1	feat 2	feat 3	feat 4	feat 5	feat 6	Class
1	(11, 18, 10, '1bi', N, N')	7	2	0.14	4	1	0	0	1
2	(11, 18, 11, '1bi', N, N')	7	2	0.07	3	1	0	0	1
3	(20, 20, None, '1a', N, N')	7	0	0	2	1	0	0	0
4	(20, 20, None, '2a', N, N')	7	0	0	2	0	1	0	0
5	(11, 18, 20, '1bi', N, N')	10	2	0.07	1	1	0	0	0

TABLE 13. A small part of the generated data.

Instance	1	2	3	4	5	6	7	8	9	10	mean
Score	0.82	0.83	0.80	0.79	0.84	0.80	0.8	0.84	0.78	0.83	0.81

TABLE 14. Results from 10-fold cross validation for the 5%-model.

Instance	1	2	3	4	5	6	7	8	9	10	mean
Score	0.82	0.73	0.83	0.79	0.83	0.76	0.81	0.77	0.81	0.84	0.81

TABLE 15. Results from 10-fold cross validation for the 10%-model.

5.4. **Evaluation of the models.** In this section we will firstly evaluate the performance of the machine learning model. We start by performing a 10-fold cross validation and we will again look at two performance measures, namely the accuracy score and the AUROC score of the model. As with the hybridization number, when we take a look at the accuracy score we will also take a look at the percentage of false positives and false negatives.

5.4.1. *10-fold cross validations.* To perform the 10-fold cross validation we only need to use the training data, which was 70 percent of the total data. We use the accuracy performance measure to determine how well each generated tree performs. Table 14 shows that the 5%-model performs quite well with an average score of around 81 percent and Table 15 shows that the 10%-model also performs quite well with an average score of around 80 percent. This shows that the models do not overfit the data and that we are not working with biased models. To make sure that the models perform well we will also take a look at the accuracy scores when we introduce the test data and lastly we will take a look at the AUROC scores and corresponding curves.

5.4.2. *The accuracy scores.* We will now evaluate how many instances of the test data are classified correctly by the models and how many false positives and false negatives are created by the models. The test data had a total of 1203 samples for the 5%-model and 1191 samples for the 10%-model. The results can be found in Table **??** and 17. We see that the models performs as well as we did expect from the 10-fold cross validation. We achieve an accuracy of around 81 percent and 80 percent respectively. We see that we have a small percentage of false positives and the rest is classified as a false negative. After we have evaluated the AUROC scores and curves we will discuss the results.

5.4.3. *The AUROC scores.* We again need to consider this difficult performance measure to get an idea on how well the models can distinguish between a 0 and a 1. We will again use the test data sets to calculate the AUROC scores. For this we firstly need to calculate the

Occurrence	True positive	True negative	False positive	False negative	Correct	%Correct	%False positives	%False negatives
	516	463	146	78	979	81.4	12.1	6.5

TABLE 16. Results from the accuracy score evaluation for the 5%-model.

Occurrence	True positive	True negative	False positive	False negative	Correct	%Correct	%False positives	%False negatives
	494	454	143	100	948	79.6	12.0	8.4

TABLE 17. Results from the accuracy score evaluation for the 10%-model.

True Class	Probability 0	Probability 1
0	1	0
1	0.19	0.81
1	0.19	0.81
1	0.84	0.16
0	0.23	0.77
1	0.18	0.82

TABLE 18. Part of the calculated probabilities from the test data of the 5%-model.

True Class	Probability 0	Probability 1
0	0	1
1	0	1
0	0.75	0.25
1	0.18	0.82
1	1	0
1	0.1	0.9

TABLE 19. Part of the calculated probabilities from the test data of the 10%-model.

probabilities that certain samples get classified as a 0 or 1. This is again done by determining the percentage of 0's and 1's we have in the leaf the samples end up in. Table 18 shows a small part of the predicted probabilities for the test data of the 5%-model and Table 19 shows a small part of the predicted probabilities for the test data of the 10%-model. We can now use the probabilities from class 1 and the real test data classes to compute the true positive rates and the false positive rates for a different number of thresholds. Using this we can then create the ROC curve which can be found in Figure 19. The straight line again shows the curve for which every sample would be randomly classified as a 0 or 1. Using SkLearn from Python, we can again calculate the area under the curve and doing so gives a AUROC score of 0.84 or 84 percent. This means that the model is very good in distinguishing between a 0 and 1.

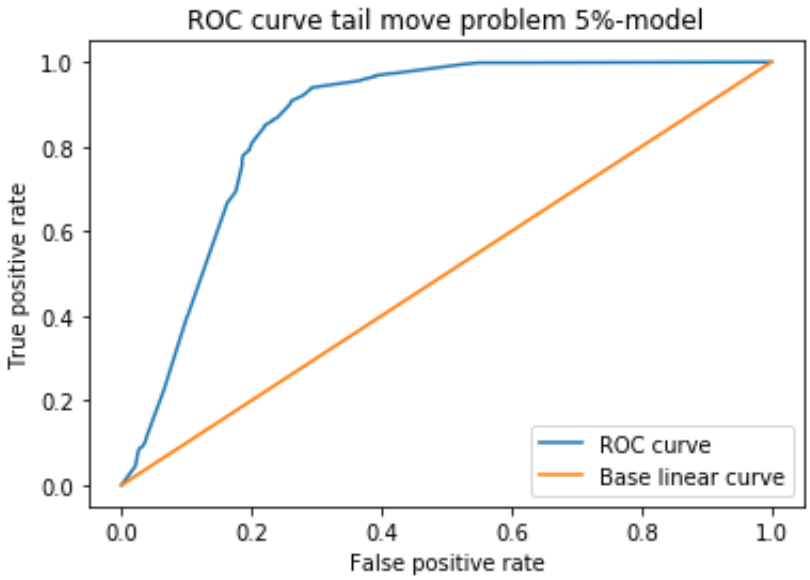

FIGURE 18. The ROC curve created from the test data of the 5%-model.

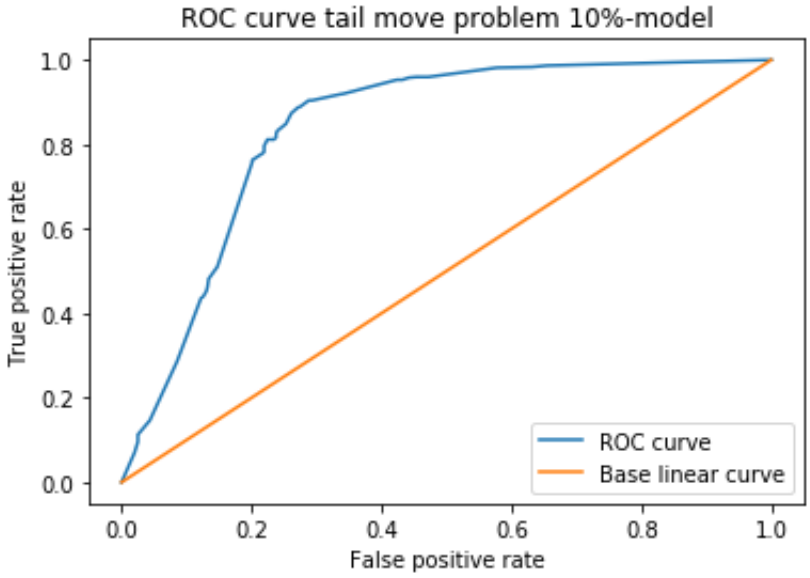

FIGURE 19. The ROC curve created from the test data of the 10%-model.

Both the accuracy score and the AUROC score show very promising results. The model works well in classifying different pairings using their features. This does not yet guarantee that the heuristic will perform better with the implementation of machine learning. Now that we know how well the model performs on the data we are able to evaluate how well the heuristic will perform when we introduce the decision tree into the heuristic.

5.5. **Evaluating the heuristic.** In this section we will discuss the performance of three different iterations of the heuristic. The first version is straight from the paper [6]. This heuristic looks at arbitrary nodes, meaning that it will always return the same result in python, because due to the implementation the first node that fits the criteria will always be the same. The second version of the heuristic swaps out this arbitrary choice for a random choice. The heuristic will no longer return the same value each time we run it, but will explore more of the possible solution space. The final version of the heuristic will have the machine learning implementation. In this version, from the possible nodes a random node is selected and then presented to the decision tree. If the node is predicted as a good node the heuristic will continue, else a new node is selected. This method still holds randomness, meaning that sometimes we can have better solutions and sometimes we will have worse solutions. We will start by explaining the experiment that was used to evaluate the heuristics. Finally we will discuss the results and make some final remarks regarding the heuristics.

5.5.1. *The experiment.* In this section we will discuss the experiment that we used to evaluate the algorithm. We created multiple network pairs which all had a different number of reticulations. All the networks had 10 to 15 leaves and we used 2 to 8 reticulations. Giving us a total of 42 different test cases. In each of the test cases the following was done:

(1) the standard heuristic was run once. The length of the tail move sequence was recorded.
(2) The random heuristic and the decision based heuristic were run once. The length of both tail move sequences was recorded. We want to know how well both heuristics perform when they also can only be run once.
(3) To also take into account the random factor that remains in the random and decision based heuristic we run the heuristics 100 times for each instance and average over all 100 runs. This is to see how well the heuristics performs on average.

The networks were created according to the same procedure that we used to generate the training and test data. We are looking at the length of the tail move sequences and not at the run time of the heuristics. In all our test cases all the heuristics roughly performed at the same speed. The only data that we are interested in is the length of the tail move sequences and for the random heuristic and the decision based heuristic we are also interested in how constant the results are.

5.5.2. *Results.* In this subsection we will discuss the results from the experiment. After the results have been discussed some final remarks will be made about the heuristic and the current decision based learner. The results will be split into 7 tables and 7 figures. Each table and figure will correspond to a reticulation number. We will take a look at the values for both the random and decision based heuristic and compare them and the average values of heuristics to the value of the standard heuristic. We will not discuss every table individually, but we will discuss the results as a whole. We present it in 7 different tables

and figures to improve the legibility. The results can be found in Tables 20 - 26 and in Figures 20 - 26. We have not included the average of the 100 runs for the heuristics in the figures.

There are a few things that we can mention regarding the results of the tail move problem. The decision based heuristic and the random heuristic do not yet consistently outperform the standard heuristic. This is due to the randomness that we still have in the heuristic that we have yet to solve. However, some hopeful comments can be made that show that machine learning can help the tail move problem. In 26 of the 42 cases the decision based heuristic performs as good or even better than the regular heuristic. The random heuristic does this also in 26 of the cases and together they beat the standard heuristic in 34 of the 42 cases. We do also see that the decision based heuristic performs very consistently if we compare the lengths of the single runs to that of the averaged runs. This is one indication that a good choice leads to shorter tail move sequences. We cannot yet say that the current model is a good model that works for this problem. We do, however, see that our decision based learner does find shorter sequences than the standard heuristic in some instances, which is an indication that there are exist good and bad choices. The current heuristic is still plagued by randomness and the issue of classifying the tree nodes in a good manner. We do not believe that this is a feature issue, because using these features shows a very accurate decision tree. We have not figured out yet if the problem is structure-less, meaning that finding a good tail move sequence does not depend on a set of features, but differs completely per pair of networks. However, we believe that there is a certain way to pick the pairings that work for each set of networks and not for only a few individual pairings. We believe that this heuristic can benefit from machine learning if we can solve the following problems:

(1) Lift the random factor from the data generation, meaning that we find a way to classify the data samples without using the version of the heuristic with random choice. This way we get a guarantee that the data we generate is a optimal as possible. If our computer at more capacity we could have performed a far deeper data generation that not only considers the first move, but multiple consecutive moves.
(2) Find a good way to classify tree nodes. We currently believe that our current approach leads to inconsistencies, but we are not completely convinced by this yet.
(3) Find a different implementation of the decision tree. We currently select a random node and let the decision trees decide whether or not this is a good node. We try to refine this choice by implementing two different trees that have different thresholds, one of 5% good nodes and the other of 10% good nodes. If there are different nodes that all lead to a good solution then this is not an issue, but the testing show that this is not always the case. One way of solving this could be to introduce more features that we might have missed or use far bigger networks to be able to create a tree that has even a lower threshold of 1% to refine the choice of a node even more.

To summarize, the current model is not yet adequate enough to improve the heuristic. The experiment does show promise and we believe that the heuristic can benefit from machine learning. However, before we can create such a model we need to solve some remaining problems. After those problems are solved we believe that a model will always be able to perform as good as the current heuristic. So also the tail move problem shows that it can benefit from machine learning, but some hurdles must still be taken before we can see a real practical improvement.

Instance	10 Leaves	11 Leaves	12 Leaves	13 Leaves	14 Leaves	15 Leaves
Prediction	**7**	**11**	9	**13**	15	**12**
Random	10	**11**	9	15	11	13
Average Prediction	9	**11**	**8**	15	13	13
Average Random	10	**11**	**8**	15	12	13
Standard	8	**11**	**8**	16	**10**	13

TABLE 20. The results for 2 reticulations.

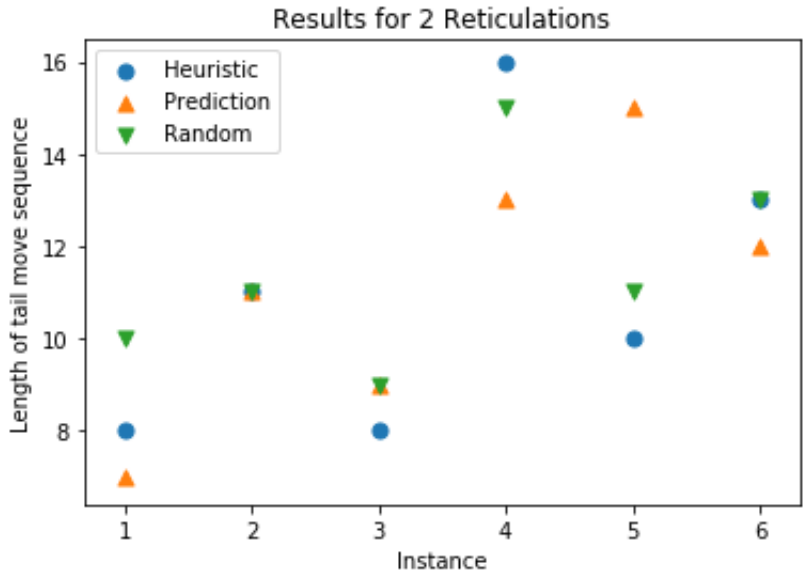

FIGURE 20. The results for 2 reticulations.

Instance	10 Leaves	11 Leaves	12 Leaves	13 Leaves	14 Leaves	15 Leaves
Prediction	13	6	**11**	**15**	**10**	**11**
Random	**12**	**5**	12	**15**	**10**	14
Average Prediction	13	6	12	16	11	15
Average Random	13	7	12	16	11	15
Standard	13	6	13	**15**	11	13

TABLE 21. The results for 3 reticulations.

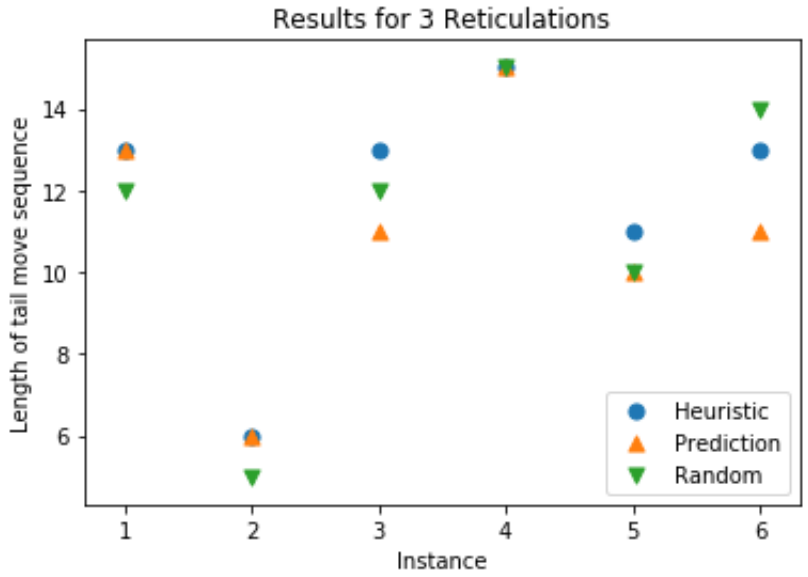

FIGURE 21. The results for 3 reticulations.

Instance	10 Leaves	11 Leaves	12 Leaves	13 Leaves	14 Leaves	15 Leaves
Prediction	16	**10**	19	13	10	15
Random	**15**	12	17	**11**	10	**14**
Average Prediction	16	13	18	13	9	15
Average Random	16	13	**16**	12	9	16
Standard	**15**	13	18	**11**	**8**	16

TABLE 22. The results for 4 reticulations.

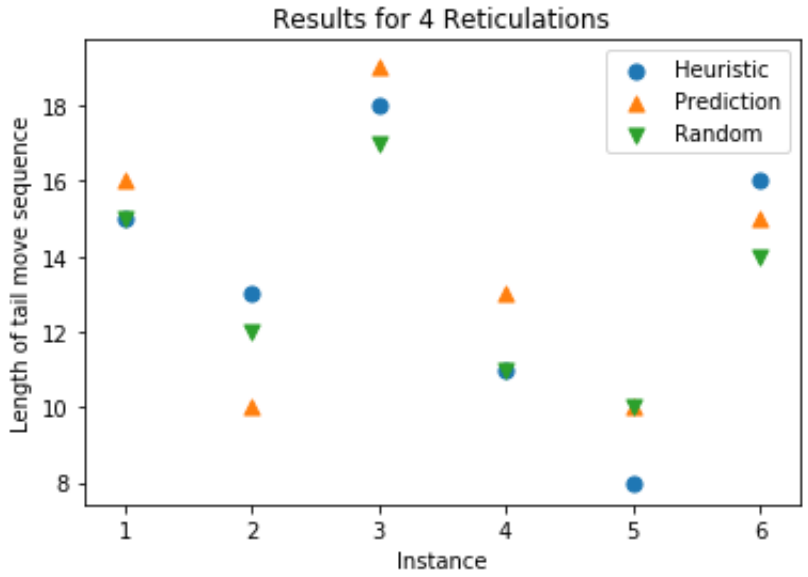

FIGURE 22. The results for 4 reticulations.

Instance	10 Leaves	11 Leaves	12 Leaves	13 Leaves	14 Leaves	15 Leaves
Prediction	13	**14**	24	23	**22**	**19**
Random	**11**	16	**19**	**18**	**22**	20
Average Prediction	17	16	20	21	**22**	20
Average Random	15	16	20	20	**22**	**19**
Standard	16	18	21	19	24	**19**

TABLE 23. The results for 5 reticulations.

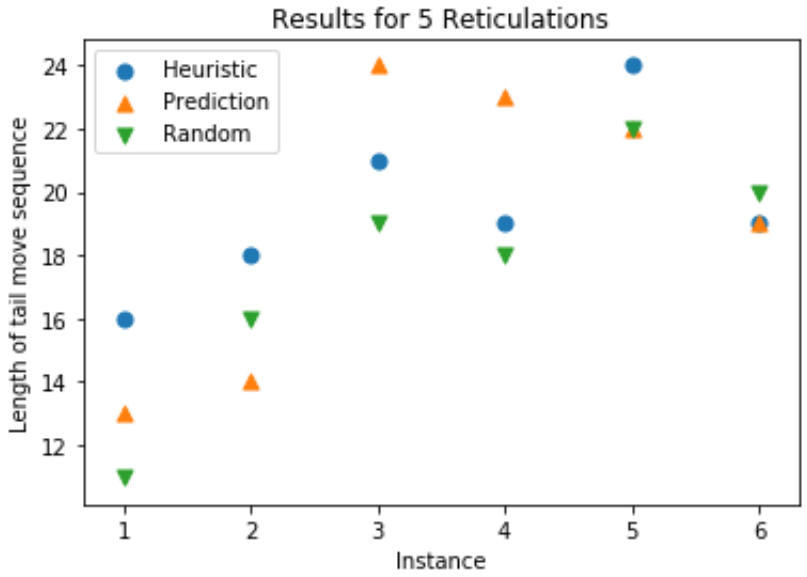

FIGURE 23. The results for 5 reticulations.

Instance	10 Leaves	11 Leaves	12 Leaves	13 Leaves	14 Leaves	15 Leaves
Prediction	19	**17**	22	**14**	23	21
Random	21	19	**12**	19	**16**	21
Average Prediction	17	19	18	19	23	22
Average Random	18	20	18	18	21	21
Standard	**16**	21	15	19	**16**	**20**

TABLE 24. The results for 6 reticulations.

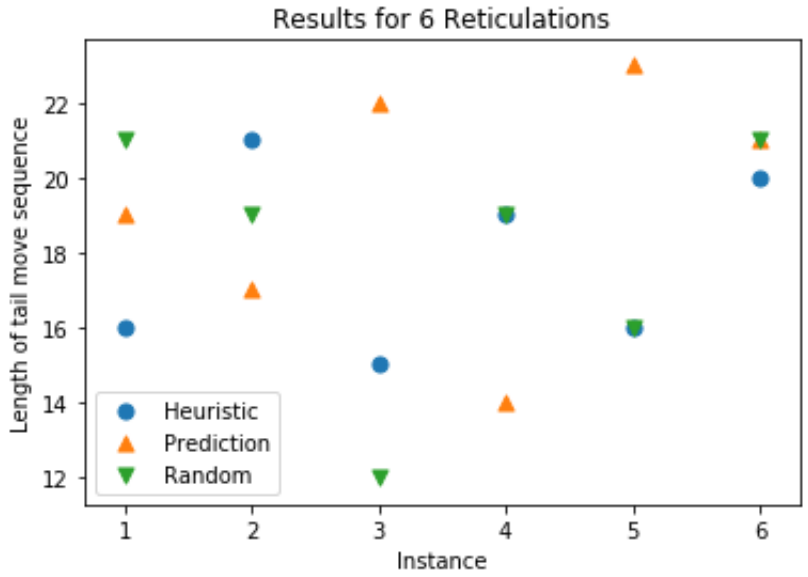

FIGURE 24. The results for 6 reticulations.

Instance	10 Leaves	11 Leaves	12 Leaves	13 Leaves	14 Leaves	15 Leaves
Prediction	18	**16**	29	**22**	21	**17**
Random	**12**	18	**28**	27	**20**	23
Average Prediction	20	17	**28**	24	22	20
Average Random	17	17	**28**	**22**	22	20
Standard	19	**16**	29	26	21	**17**

TABLE 25. The results for 7 reticulations.

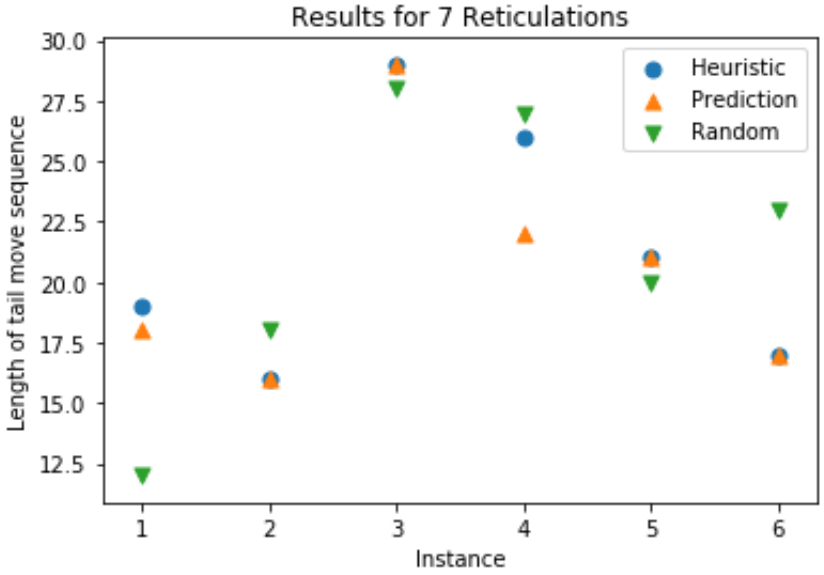

FIGURE 25. The results for 7 reticulations.

Instance	10 Leaves	11 Leaves	12 Leaves	13 Leaves	14 Leaves	15 Leaves
Prediction	24	24	28	**17**	**28**	23
Random	23	22	28	26	32	21
Average Prediction	24	23	27	23	**28**	21
Average Random	22	**21**	27	20	**28**	20
Standard	**20**	23	**25**	18	30	**15**

TABLE 26. The results for 8 reticulations.

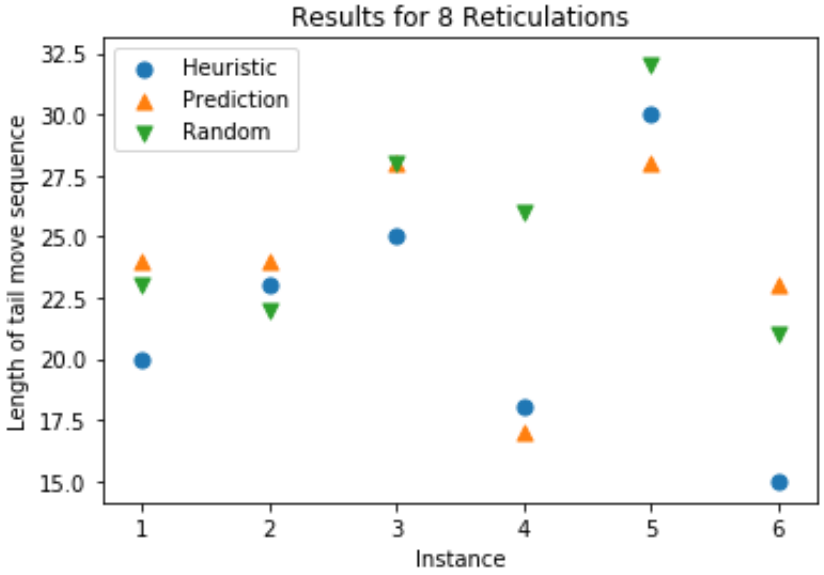

FIGURE 26. The results for 8 reticulations.

6. Conclusion

We have seen three different problems within phylogenetics for which we hoped all could benefit from machine learning. The first problem that we encountered was the maximum agreement forest problem. The FPT algorithm held much randomness and this led us to believe that it could benefit from machine learning. However, we saw in the testing that our model just could not get a grip on the data and could not clearly decide on a good picking strategy. The accuracy of the model was decent, but it seemed that the actual maximum agreement forest problem had no real structure in the sense that each pair of trees benefited of different choices. That is why we believed that the maximum agreement forest problem did not benefit from our current approach. We do highly encourage people with a deeper understanding of machine learning to pick up this problem and create a more complex model that can find the structure within this problem, but for us it was out of the scope of possible models that we could try. One idea is to give a prediction on a sequence of moves rather than a single move, because one move might not be good alone, but together with a few followup moves it could be the best possible one, which is often used in chess.

The second problem showed far better results. We created a model for the tree child hybridization number problem that performed better than the current FPT algorithm used to solve this problem. Though it was a mere prove of concept it showed very promising results for machine learning. We believe strongly that the hybridization number problem in general can benefit from machine learning and we encourage anyone with an interest in both topics to further search towards a model that can even benefit the general search of hybridization numbers, because we believe that there is still ground to be covered and run time to be won. Currently we order the possible cherries to be picked, but it can be possible that we need to also consider cherries that can be picked after. As with the Maximum agreement forest problem we think that it is interesting to see what happens if we do not limit ourselves to only one move, but try to implement a prediction on a sequence of moves. Maybe this would result in creating an even smaller search space.

The final problem that we discussed was about the tail move problem. The problem clearly presented itself as a method that could benefit from machine learning, because of the arbitrary choice that was present in the heuristic. Though we have seen that there are currently some issues that still need to be worked out before we can guarantee that the tail move problem can benefit from machine learning. The tests, however, showed much promise regarding the performance of the model, but because of the randomness still present we see that the implementation of the machine learner into the heuristic does not yet lead to consistent results. All the data shows that the tail move problem can benefit from machine learning if a few hurdles can be overcome. We again encourage people to try to overcome these hurdles or come up with a better and more robust model that will lead to an improvement in heuristic performance. It could be interesting to consider multivariate decision trees using combinations of features to split instead of univariate decision trees. Another area that could be explored is again to look at a sequence of pairs and not at a single pair each time. More generally, for solving the tail move problem one could also consider to look at finding a connection between a good tail to move and to location to where to move to. This could maybe be achieved by first splitting the problem into two decision problems, namely deciding which tail to move and then deciding on the location to move to and afterwards

combine this into a model that decides on good moves. This way we could limit the large search space we would otherwise have if we would take every possible tail move into account.

All in all we do believe that some parts of phylogenetics can certainly benefit from machine learning. However, more complicated models should be considered to create a model that consistently can outperform current algorithms. As a proof of concept the decision tree based learner does show promise and therefore we believe that a lot can be won if we were to embrace machine learning into phylogenetics.

References

[1] Albrecht, B. (2014). Computing hybridization networks for multiple rooted binary phylogenetic trees by maximum acyclic agreement forests. arXiv:1408.3044

[2] Albrecht, B. (2015). Computing all hybridization networks for multiple binary phylogenetic input trees. *BMC Bioinformatics*, 16(1), 236. https://doi.org/10.1186/s12859-015-0660-7

[3] Baroni, M., Grünewald, S., Moulton, V., Semple, C. (2005). Bounding the Number of Hybridisation Events for a Consistent Evolutionary History. *Journal of Mathematical Biology*, 51(2), 171–182. https://doi.org/10.1007/s00285-005-0315-9

[4] Felsenstein, J. (n.d.). The Newick tree format. Felsenstein / Kuhner Lab. Retrieved August 24, 2020, from https://evolution.genetics.washington.edu/phylip/newicktree.html

[5] van Iersel, L., Janssen, R., Jones, M., Murakami, Y., Zeh, N. (2019). A practical fixed-parameter algorithm for constructing tree-child networks from multiple binary trees. arXiv:1907.08474v1

[6] Janssen, R., Jones, M., Erdős, P. L., van Iersel, L., Scornavacca, C. (2018). Exploring the Tiers of Rooted Phylogenetic Network Space Using Tail Moves. *Bulletin of Mathematical Biology*, 80(8), 2177–2208. https://doi.org/10.1007/s11538-018-0452-0

[7] Kelk, S. (2012). TREETISTIC. https://skelk.sdf-eu.org/clustistic/

[8] Linz, S., Semple, C. (2019). Attaching leaves and picking cherries to characterise the hybridisation number for a set of phylogenies. *Advances in Applied Mathematics*, 105, 102–129. https://doi.org/10.1016/j.aam.2019.01.004

[9] McKinney, W. (2018). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (2nd ed.). O'Reilly Media, Inc.

[10] Raschka, S. (2015). Python machine learning : unlock deeper insights into machine learning with this vital guide to cutting-edge predeictive analytics. Packt Publishing Ltd.

[11] Whidden, C., Beiko, R. G., Zeh, N. (2013). Fixed-Parameter Algorithms for Maximum Agreement Forests. *SIAM Journal on Computing*, 42(4), 1431–1466. https://doi.org/10.1137/110845045

[12] Wu, Y. (2010). Close lower and upper bounds for the minimum reticulate network of multiple phylogenetic trees. *Bioinformatics*, 26(12), i140–i148. https://doi.org/10.1093/bioinformatics/btq198