Bachelor thesis
Computer Science

Radboud University

Improving decision tree learning by looking ahead

Author:
Mees Neijenhuis
S4353501

First supervisor/assessor:
ir. Jesse Krijthe
J.Krijthe@cs.ru.nl

Second assessor:
Prof. Tom Heskes
t.heskes@science.ru.nl

April 18, 2018

1 Abstract

Decision trees are used to make predictions about data. Many different algorithms exist to make such trees, but none of them are perfect. An algorithm that makes a decision tree has to somehow find the best split to make at each point. In this thesis I will discuss an idea for an algorithm to make better decision trees. This algorithm will not only take into account the current split, but also possible splits after making a certain split and take that into account when deciding on the best split to make. To test this algorithm I compared it to other existing algorithms. In it's current form, the algorithm is not yet better than the other available algorithms, but improvements can still be made to maybe change this.

2 Introduction

Decision trees are used to make predictions about certain data. These predictions can be about almost anything. They can range from recognizing spam e-mails to classifying types of plants. For an e-mail, the input data can things like: who sent it, when was it sent, does it contain a link to a website, etc. The output will, in this case, be a simple yes or no answer to indicate if the e-mail was spam or not.

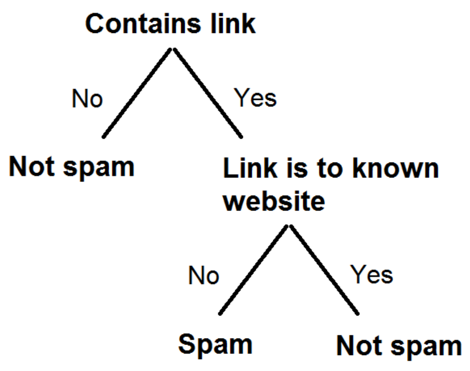
When making a tree you already need to have some data to use as examples. This data contains both the input and output values. The goal of a decision tree algorithm is to try to make a tree that represents this data as well as possible. This is done by choosing a feature of the input and splitting the data into subsets. I will explain how this feature is chosen in section 2.3. In which subset each example goes, depends on the value of the feature. If for example the feature is "the e-mail contains a link", then all the e-mails that do contain a link will be placed in one subset and all the e-mails that do not contain a link will be placed in the other subset.

These splits are made until a certain stopping criterion is met. Each subset that is now left if called a leaf. Each leaf belongs to a prediction about the output. If for example a leaf contains 90% spam e-mails, this leaf will have the prediction that the e-mail is spam linked to it.

Below in figure 1 an example of what a tree that classifies spam would look like. Of course this is a simplified example and in real life, there would be many more splits, but this should give a basic idea of what a tree looks like.

Of course we would want these predictions to be accurate 100% of the time and when training a tree on the given data, it's often possible to make a tree that predicts all this data correctly. An easy example of how to do this would be to make a leaf for each of the examples, so they will always be classified correctly. The problem, however, is that new data that has not

Figure 1: An example of a decision tree



been seen before, can be (and often will be) different from the data used to train the tree, but making predictions about this new data is the actual goal of the decision tree, so we want the predictions on the new data to be as good as possible even if this means that the predictions on the training data won't be 100% accurate. Making your tree to specific to the training data is called overfitting.

There are already many different approaches to try to make as good of a decision tree as possible. In this thesis I will first discuss some of these approaches and then I will make a suggestion for another possible way to make decision trees by looking ahead multiple features when deciding which split to make and evaluate its performance.

2.1 Basic steps to make a decision tree[6]

The steps needed to make a decision tree are as follows.

1. Decide which split to make. Here you just try every split and choose which one seems best according to the splitting criteria used by the algorithm.

2. Make this split and choose which class to give to both parts of the split. The class chosen is the class that most of the observations in this part belong to.

3. For each subset make a new split in the same way. Order in which to split these subsets is irrelevant, as they will all have to be looked at.

4. Repeat this until some stopping criteria is met. This can be the depth of the tree or the amount of observations left in a node for example.

5. Depending on the algorithm, pruning is used at the end. For pruning you delete a part of the tree and see if this make the tree better. The way to measure how good the tree is, differs per algorithm.

2.2 Existing algorithms

Now that the general idea of how decision trees work is clear, I will explain the most popular algorithms to make decision trees in more detail and talk about the choices made in those algorithms. The existing splitting criteria and pruning methods for the algorithms will first be explained and then we will look at the different algorithms.

2.3 Splitting criteria explained

Splitting criteria are used to measure how good a split is. The general idea is that you look at the data before the split and after the split and compare how good the prediction of your tree is. When making a split it is desirable to get as many instances of one class together in one subset, because this means a big percentage of the data belongs to one class there, so predicting that all the data in that leaf belongs to that class will usually be the right prediction.

2.3.1 Information gain

Information gain is the difference in entropy. So the information gain is equal to the current entropy minus the entropy after making the split[6]:

$$IG(T,a) = H(T) - H(T|a)$$

Here $IG(T,a)$ is the information gain on tree T if split a is made. $H(T)$ is the current entropy and $H(T|a)$ is the entropy after split a. The entropy is calculated by looking at the percentage of instances classified correctly if you predict everything in the leaf as the majority class (the most common class in this leaf). When calculating the entropy after a split, you also need to take into account how big each subset is. If one subset is twice as big as the other one, you need to count this one twice and then take the average. This is called the wieghted entropy. The higher the information gain, the better the split is.

In some cases, the information gain is a bit too simple and values certain splits the same way even though one is actually better. Going into detail about when this is the case, is not important for this research, but the fact that it can happen still needs to be taken into account when choosing a splitting criterion for my own algorithm.

2.3.2 Gini index

Consider the following formula:[6]

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k) \tag{1}$$

In this formula $\hat{p}_{mk}$ is the proportion of class k observations in node m, N_m is the amount of observations in node m, $x_i \in R_m$ means every observation in region m and $I(y_i = k)$ checks if that observation belongs to class k and returns a 1 if it does and a 0 otherwise.

Now the gini index can be described using the following formula:

$$\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'}$$

The lower the gini index is, the better. So when using this to calculate how good a split is, you look at the gini before the split and subtract from that the weighted gini after the split. The higher this value is, the better the split is.

2.3.3 Cross entropy

For cross entropy, we again need to use $\hat{p}_{mk}(1)$, which is described above at gini index. The formula for cross entropy is:[6]

$$-\sum_{k=1}^{K} \hat{p}_{mk} log(\hat{p}_{mk})$$

Also how good a split is, is calculated by taking the cross entropy before the split and subtracting from that the weighted cross entropy after the split. Again a higher value for this is better.

2.4 Pruning methods explained

Pruning is done to counter overfitting. If the tree you make is too specific on the training data, this will usually result in worse results on new data. Pruning is done after a tree is made. For each leaf of the tree, the pruning algorithm will check if it is better to keep this leaf or to remove it, based on some kind of evaluation. It then removes leaves accordingly and also checks new leaves that are created this way.

2.4.1 Error based pruning[4]

Starting at the leaves, each split that was made is removed, so the node above it becomes the new leaf. This leaf will have the prediction of the most popular class. If this change doesn't affect the prediction accuracy (usually measured with information gain), then the change is kept. Otherwise it is reverted to how it was. This is done one by one for each leaf and each new leaf created. Until everything has been tested. This method is simple, but fast.

2.4.2 Cost-complexity pruning[4]

This method keeps removing part of the tree and checks if this improves the tree. The part that is removed, is the part that gave the smallest amount of gain. The formula used for this also takes the amount of nodes in the tree into account, so a much smaller tree with a bit less accuracy will be considered better. The accuracy of the tree is measured using misclassification error:

$$1 - \hat{p}_{mk(m)}$$

$\hat{p}_{mk}(1)$ is described at the gini index section. This formula is just 1 minus the proportion of correctly classified observations.

2.5 Handling missing values[6]

Sometimes values are missing and we need a way to deal with this. For missing values that are categorical, we can make a new category we name "missing". This way we can classify them like every other observation and we may even find out that most observations with a missing value are classified as a certain class while other observations without that value missing are usually classified as a different class.
Another method that can be used for both categorical and numerical values is as follows: For each split we keep a list of the best splits in order of how good they are. Then if a value for a certain split is missing, you make that decision based on the second best split or if that is also missing, on the third best split etc. This method works best if there is high correlation between the featuress.
A last and less common method is to take the average of the features where the value isn't missing and to use that instead of the missing value.

2.6 The algorithms

There are many different algorithms out there, each with their own pros and cons. First we have ID3, C4.5 and C5.0. ID3 is the first version of the algorithm that has since been upgraded into C4.5 and then even into C5.0.

ID3 is really basic and nowadays not very useful any more, but since it is the first version, I thought I'd list it here anyway.[13]

C4.5 is where is gets interesting. It splits based on information gain and keeps splitting until everything is of the same class or until there is no more information gain. This algorithm accepts both categorical and numerical values as input and uses error based pruning. Unfortunately it is susceptible to outliers.[3]

C5.0 is almost the same as C4.5, but it was optimized a lot. It is faster and uses less memory.[3]

CART is another popular algorithm. It makes splits based on either the gini index or cross entropy and uses cost-complexity pruning, which is done using the misclassification rate. This algorithm can handle outliers.[6]

CHAID is quite different from the above algorithms. The most interesting difference being that it can make a multi-way split. This means it doesn't just split the data in two like the other algorithms, but can split it in any number of subsets. It also uses a completely different way of deciding the best split, but for this research, that's not important, since we won't be making splits in such a way.[1]

2.6.1 Algorithms used in popular programming languages

Python uses CART with some optimizations. Python doesn't support pruning yet and the algorithm doesn't work with categorical values. [8]
Matlab also uses CART. [7]

2.6.2 Random Forest

For a random forest, you make a lot of decision trees and then combine their results to classify an object. Because you have to make a lot of trees, each individual tree will be simpler than a normal decision tree, but because you can later combine the results of all trees, you still get a good classification. Individual trees can often contain some noise, so making many trees and taking the average, should counteract this. Usually to make these trees, bootstrapping is used and each tree can only look at a part of all the features at each split. This way you get many different trees that looked at different parts of the training set and that looked at different features at different splits. A good value for the amount of features looked at seems to be $\sqrt{p}$, where p is the total amount of features, but this can be different for different problems. If you don't look at only a part of the features, then the trees will mostly be the same and the random forest has almost no effect. Pruning is usually not needed, since taking the majority vote from many trees already works against overfitting.

For making a random forest, any of the algorithms above can be used.

The reason a random forest is interesting to look at, is because random forests can be made with any number of trees, so it's easy to make a forest that has about the same running time as my new algorithm, so it's fair to compare them.

2.6.3 Basic steps to make a random forest[5]

These are the steps needed to make a random forest.

1. Use bootstrapping[2] to create a data set from the original data set.
2. At each split take a part of the features at random.
3. Make a decision tree as described in the section above, but only use the newly created data set and look at only the features chosen in the step before.
4. Repeat this to make many different trees that all looked at different parts of the data and at different features.
5. To classify an object, you now classify it with all the different decision trees and you use some way to combine these results. Usually this will be a majority vote, but you can also have trees that seem better have a bigger weight when making the votes for example.

2.7 The problem with the current algorithms

Usually when a decision tree is made, a splitting criterion is used to select the best split to make at that moment in a greedy way, but this is only a prediction, so further down the tree it may actually turn out that this split wasn't the best split to make at that moment. If you could "look in the future" and see if this is the case, then you could make the better split instead.

3 Method

If a decision tree algorithm can look ahead when deciding which split to make, it will have more information about which split is best to make at that moment. The basic idea is to try all splits and then from each of these splits look what the best next split would be. This way you look two steps ahead instead of the usual one step.

To make an algorithm that looks ahead, I couldn't use the standard implementation of the algorithm, because looking ahead requires a change in how the splits are made and you don't have the option to make that change in the existing algorithm. So I decided to program the whole decision tree making algorithm from scratch, in Python. I chose Python, because it is open source, free and easy to use.

3.1 Details about the algorithm

The standard implementation for decision trees in Python doesn't accept categorical values, but having this option can be really useful, so I decided to include this in my implementation, so it works with both categorical and continuous values.

For the splitting criterion using information gain should be avoided, so that left me with two options: gini index or cross entropy. Any of those can be used, so I chose to use the gini index.

If a split is made on a categorical value, a split is made for each category, so if there are five different categories, the data will be split into five. This seemed like the most simple way to make this split. By keeping it simple the run time of the algorithm will be shorter, which is important. Why this is important is discussed a bit later on. For continuous values this method of splitting doesn't work, because each different value would then result in a different split, so here the data is split into three parts: higher than some value, lower than some value and "missing". To choose the optimal value, each value in the data set is looked at and the best is chosen. For categorical splits, "missing" is just another category and for continuous, it is specifically made. This is how missing data is handled when making the splits.

I decided to keep the stopping criteria simple: Stop when a leaf has just one class in it or when a leaf has 10 or fewer examples in it. This last criterion is to stop overfitting to some extend. Because these stopping criteria already take overfitting into account, no pruning is used.

Looking ahead can be computationally expensive, because for each split you have to look at each next split etc. This is why it's important to keep the rest of the algorithm simple. For this reason there is also an option to change how far you want to look ahead and also another option. This option is to not look ahead for every feature, but just for a few of the best ones found. For example you can see which ten features are the best to split on right now, based on the gini index and then just look ahead for those ten features, instead of all the features. To look ahead, every split is made and from every split you then look what the best split would be based on the gini index, and choose the best one found and make the first split to get there. Then you repeat this until you have your tree.

3.2 How random forests are made

For this I decided to just do it the standard way as described above2.6.2, with one adjustment. At the beginning of making a tree, a random subset of the features is made and the algorithm only splits on these features when making a tree. In the normal algorithm, these features are chosen again for each split, but I decided to only choose them once at the start. This way it is less computationally expensive to make the trees for the forest, so it was easier to make more trees. The trees for the forest are made by my own algorithm without looking ahead. This is done to keep the comparison fair.

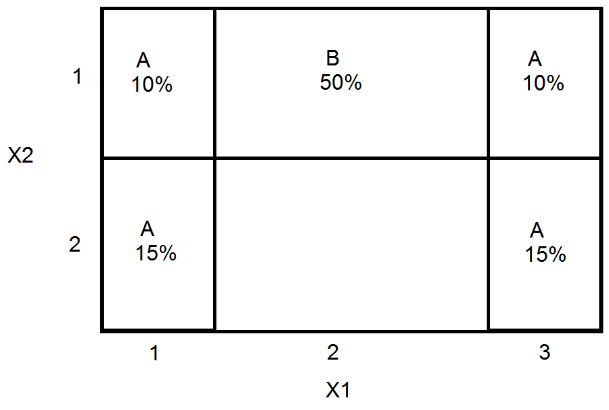
3.3 Classification

To classify a new example, it is put in the decision tree and based on the value of each feature, splits are made until a leaf is reached, in this leaf is a prediction for the class. This will be the prediction.

3.4 Example when the algorithm works

Figure 2 below illustrates a data set where looking ahead can produce a better result.

Figure 2: An exapmle data set



A normal algorithm that doesn't look ahead will see that splitting on X2 will result an error of 20% while splitting on X1 will result in an error of 25%. Having more data classified correctly after a split is better according to splitting criteria like the gini index, so the algorithm will first split on X2. After this, two more split are needed to get all data split in subsets with just one class in each subset. The associated decision tree would look something like figure 3 below.

If you have an algorithm that looks ahead two features, instead of the usual one, it will see that splitting the data on X1 twice in a row, will result in a tree splits the data completely. Figure 4 below illustrates how the tree would look in that case.

In this example, both algorithms get a tree that completely splits the data, but with looking ahead you actually get a smaller tree, which is preferred, since smaller trees are usually less likely to be overfitted.

Figure 3: What a normal tree would look like

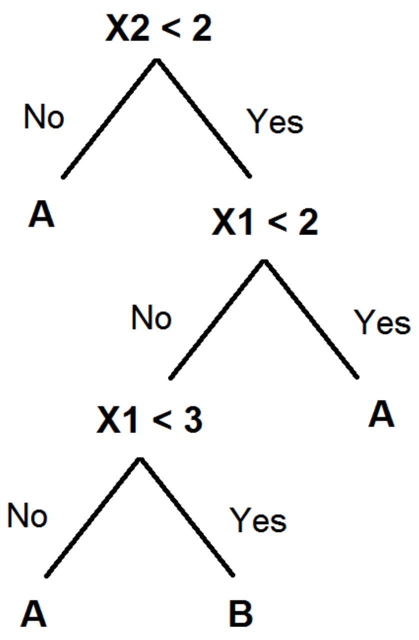
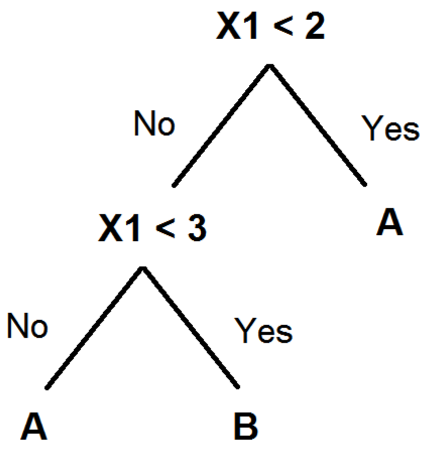


Figure 4: What a tree would look like when looking ahead



4 Results

4.1 The experiment

Once I completed my algorithm, I tested different ways to make the tree on different data sets.

4.1.1 The different algorithms tested

The following algorithms were tested:

1. The standard Python implementation (on the data sets without categorical values).[8]

2. My own algorithm to make a normal decision tree.

3. Looking ahead one feature.

4. Looking ahead two features.

5. Random forest using my decision tree algorithm with 5 trees.

6. Random forest using my decision tree algorithm with 10 trees.

7. Random forest using my decision tree algorithm with 20 trees.

4.1.2 The Datasets used

I didn't want my results to be influenced by the kind of data set used, so I decided to test my algorithm on four different data sets. These are the data sets I used:

1. Iris data, a data set about different types of iris plant. This data set has 150 instances, 4 features and only numerical values as input. The input is about the length and width of the leaves of the plant and the output is the type of plant.[9]

2. Car data, a data set about evaluating cars. This data set has 1728 instances, 6 features and has both categorical and numerical values as input. The input is about the attributes of a car and the output is how good this car is.[12]

3. Balance Scale data, a data set about whether a scale is balanced or not. This data set has 625 instances, 4 features and only numerical values as input. The input is about weights on both sides of the scale and the distance to the centre of the scale and the output is about which side goes down.[11]

4. Tic-Tac-Toe data, a data set about winning and losing positions in tic-tac-toe. This data set has 958 instances, 9 features and only categorical values as input. The input is an example of a game of tic-tac-toe and the output is about how good this position is.[10]

The goal in choosing these data sets, was to find sets that are different from each other in some way. As you can see some data sets are bigger than others and they have more or fewer features than others. Also the kind of input is different. The idea behind this, is that some algorithms can work well on some data sets and not work well on others, so by choosing different kinds of data sets, you can see if this is the case.

Because the standard Python implementation doesn't accept categorical values as input, I couldn't test all the data sets with that algorithm. The reason I still chose to include data sets with categorical values in the tests is because it may produce different results, especially since the way my algorithm splits categorical values is different from how it splits numerical values.

4.1.3 How this all comes together

For each of the above algorithms, a data set was split into a training and a test set. For this I used the "train_test_split" algorithm from Python's scikit-learn[8] with the default values. This means 25% of the data will be used as test set and the remaining 75% is used as training set to train the tree. After training the tree with the training set, I checked the accuracy of the tree on the test set and also on the training set for comparison. This process is repeated 100 times for each algorithm and an average of these accuracies is computed. The reason this is done 100 times instead of just once, is because the randomness of making the train and test set can have an impact on the accuracy and by repeating this many times and taking the average, you get a better idea of what a common result is. This is of course done with each data set.

4.2 Charts

In the charts below are the results I got. The number in brackets behind "Tree", is the number of features the algorithm looks ahead. Here 1 means it's just a normal algorithm to make a tree, because you always look at one next feature. The number in brackets behind "Forest" is the number of trees used to make the forest. Since the standard Python implementation had a run time of close to zero, I decided to not include this in the graphs.

Figure 5: Accuracy of the iris data set

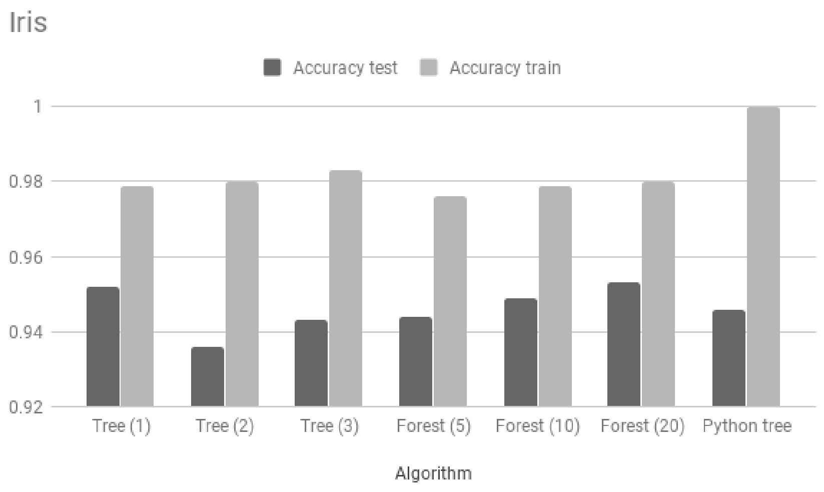


Figure 6: Accuracy of the car data set

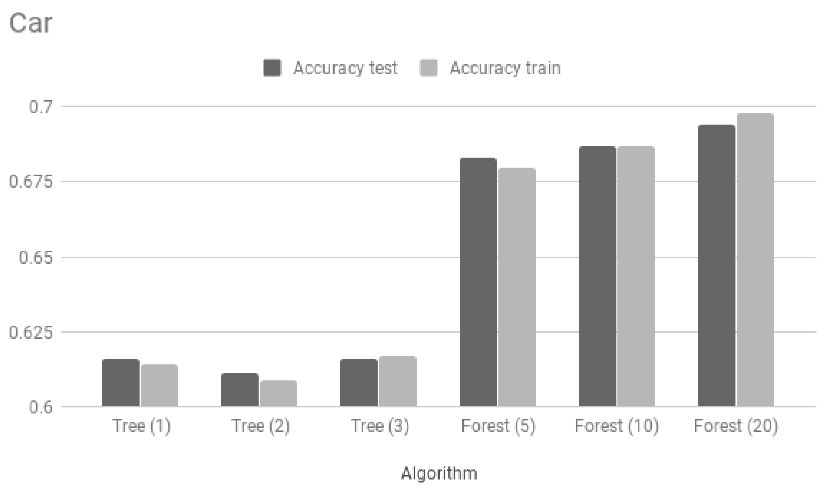


Figure 7: Accuracy of the scale data set

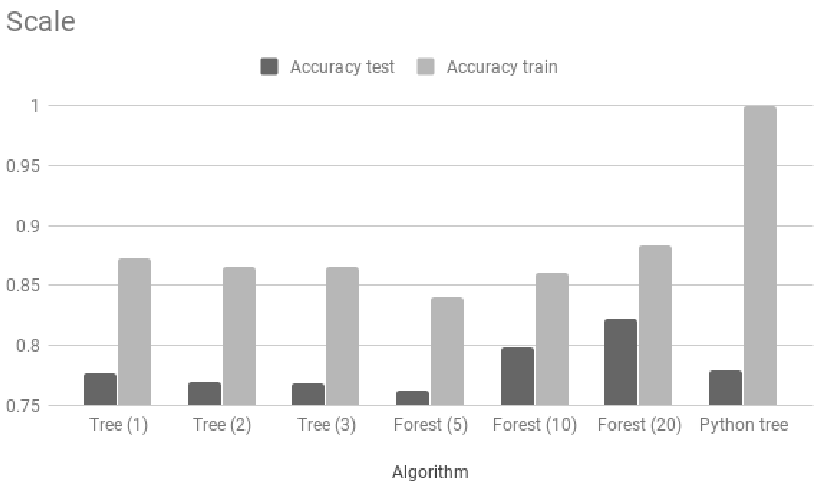


Figure 8: Accuracy of the tic tac toe data set

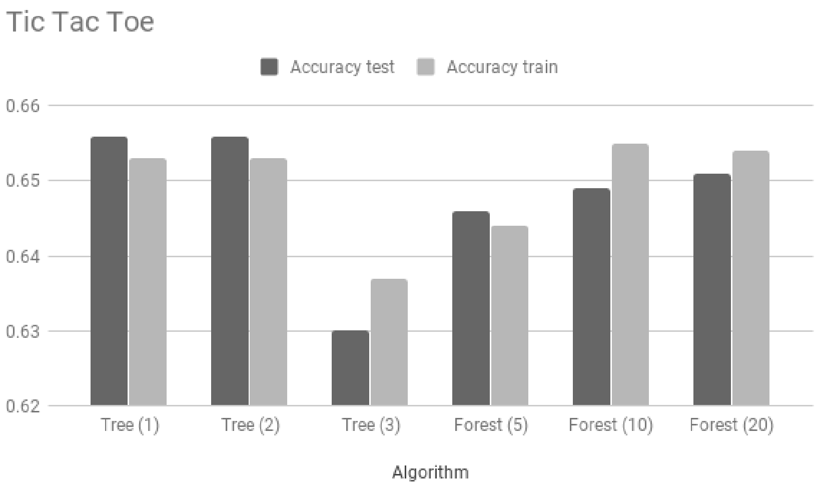


Figure 9: Run time of the iris data set

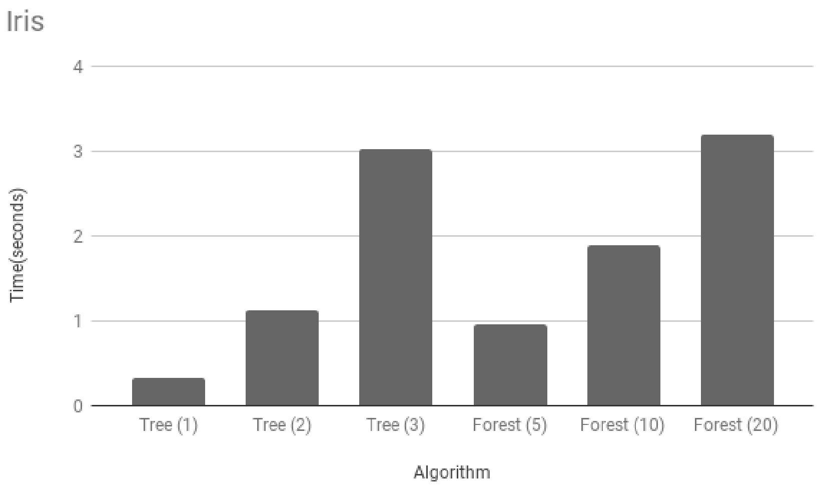


Figure 10: Run time of the car data set

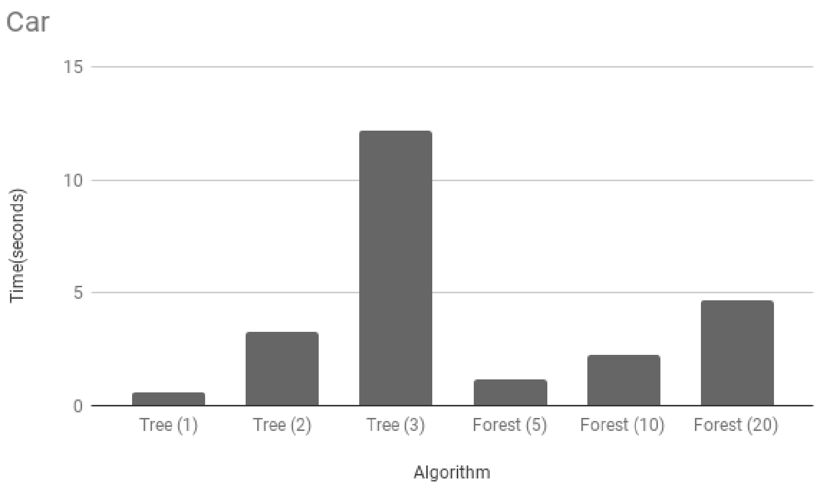


Figure 11: Run time of the scale data set

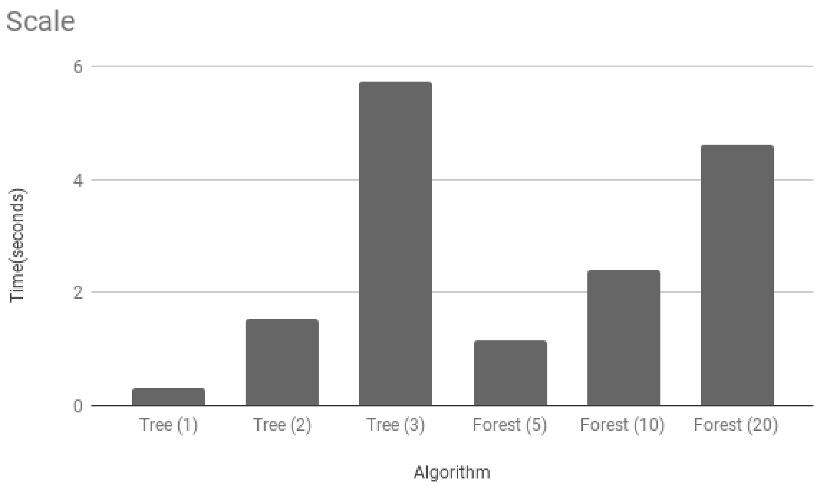
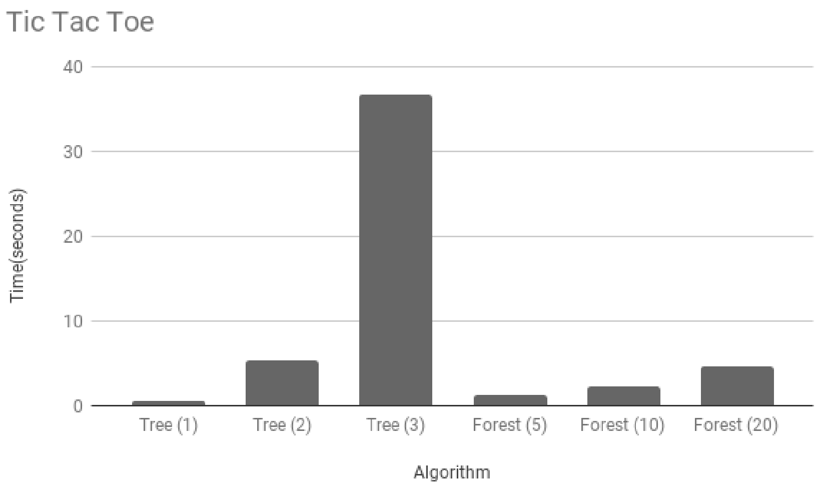


Figure 12: Run time of the tic tac toe data set



5 Discussion

5.1 What we can see from the results

As you can see from the results, the forest algorithm behaves exactly like you would expect: more trees results in better accuracy, but it takes longer, because you have to make more trees. Also the Python algorithm seems pretty straight forward. It always make a tree that perfectly fits with the data used to train it. Because of this, the training accuracy is 100%, but of course this means the tree is overfitted and this means it performs worse on the test set. The looking ahead algorithm on the other hand, produced some unexpected results. First of all, you would expect the accuracy on the training set to increase if you look ahead further like in the iris data, but this isn't always the case in the other data sets. I think the problem here is that when looking ahead, the algorithm doesn't take the stopping criteria into account. So it may find a split that will result in a good accuracy after two more splits, but after making that split the stopping criteria may be met and in that case another split may have been a better choice.

Also in the test set, the looking ahead algorithm doesn't perform well. Not looking ahead always seems to be better. This is probably because of overfitting. Maybe if a pruning algorithm were to be added, then looking ahead may actually be better, but as it is now, it's worse. Also compared to the random forest algorithm, it's not better.

When you look at the example given in 3.4, you would think that the algorithm can produce good results as well. The problem with the example however, is that it is pretty specific and in real situations the data will usually be a lot messier. That being said, there may still exist data sets where this algorithm would give good results, but we want an algorithm that is good most of the time and not one that is only good on certain specific data sets, if these data sets even exist in real situations.

Run time wise, looking ahead increases rather quickly when increasing the amount to look ahead. This is of course to be expected, since the work it has to do increases by a lot. On most of the data sets, it has a much longer run time than even making a forest of 20 trees. With the longer run time and the lower accuracy, I can safely say that the algorithm in it's current state is not yet an improvement compared to random forests.

Probably the most important thing we can learn from this, is that overfitting is a real problem and you should always keep that in mind, whether you're making your own algorithm or using existing algorithms overfitting can always be a problem.

5.2 Possible improvements

Like mentioned above, taking the stopping criteria into account and adding a pruning algorithm may help improve the looking ahead algorithm. Another important thing to improve, is the running time of the algorithm. Looking ahead can be computationally expensive and unlike the standard Python implementation, this algorithm doesn't have any optimizations yet. I haven't researched these optimizations, since it was not in the scope of this thesis, but there ways to program things differently to make the program faster. Also some programming languages can produce faster programs. The fact that python checks variables at runtime for example makes it slower than a language that uses static variables. Another way to reduce running time, could be to look ahead for only the two best features with each split, but doing this will also mean the effect of the algorithm will be lower, because it won't be able to check as many features. To compare run times, looking three features ahead sometimes took as long as 36 seconds, while the Python algorithm is done almost instantly. If you have to use the algorithm on a small data set just once, this isn't a huge problem, but because of the large amount of time my algorithm needed, I couldn't test it on bigger data sets, while those may actually be really interesting, since looking ahead may have a bigger effect on a bigger data set. Also to test an algorithm it is best to run it as many times as possible to take a good average. For my tests I only ran each algorithm 100 times. While this may seem like a lot, it can still be influenced by the randomness of making train and test sets quite a bit, because the differences in accuracy between different algorithms can be pretty small, so if there is one outlier in the accuracies, then this can have a significant impact. Testing an algorithm 1000 times or more would probably be desirable, because this would make these outliers less significant. This is also a big reason to make the algorithm faster.

6 Conclusion

While the algorithm in it's current state isn't better than the existing algorithms yet, I think there is still some potential for this to become better. It can definitely become better than it currently is, because there are still improvements that can be added as stated above.5.2 The only question is whether or not it can also become better than the other algorithms out there like the random forest. In theory I think the algorithm would work best on bigger data sets with many features, since there isn't a lot to look ahead to otherwise. Also it would probably be best to have numerical values for features, since this will result in fewer subsets when making splits, which means it will need less time to compute the best split. The problem with bigger data sets however, is that the algorithm will also be much slower. With optimizations this problem may be avoided.

It could also be interesting to see how accurate a random forest would be if the trees in the forest are made by the looking ahead algorithm. Since random forests are a good way of countering overfitting, combining this with the possibly overfitted trees from looking ahead, may give promising results. This does require better running times for looking ahead though, as you need multiple trees to make a forest and if making the trees takes a long time, the forest will take a really long time.

Since there seem to be many obstacles in the way for this algorithm and since there are already so many good algorithms available, I'm afraid that this algorithm won't be good enough any time soon, but I welcome anyone to try and prove me wrong.

References

[1] Wikipedia contributors. Chi-square automatic interaction detection — wikipedia, the free encyclopedia, 2017. [Online; accessed 26-March-2018].

[2] Wikipedia contributors. Bootstrapping (statistics) — wikipedia, the free encyclopedia, 2018. [Online; accessed 26-March-2018].

[3] Wikipedia contributors. C4.5 algorithm — wikipedia, the free encyclopedia, 2018. [Online; accessed 26-March-2018].

[4] Wikipedia contributors. Pruning (decision trees) — wikipedia, the free encyclopedia, 2018. [Online; accessed 26-March-2018].

[5] Wikipedia contributors. Random forest — wikipedia, the free encyclopedia, 2018. [Online; accessed 27-March-2018].

[6] Tibshirani Hastie and Friedman. *The Elements of Statistical Learning (2nd edition).* 2009.

[7] Mathworks. Classregtree, 2017.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] UCI Machine Learning Repository. Iris data set, 1988.

[10] UCI Machine Learning Repository. Tic tac toe data set, 1991.

[11] UCI Machine Learning Repository. Scale data set, 1994.

[12] UCI Machine Learning Repository. Car data set, 1997.

[13] Wikipedia. Id3 algorithm — wikipedia, the free encyclopedia, 2017. [Online; accessed 17-October-2017].