UCLouvain

École polytechnique de Louvain

Constructing Optimal Forests of Decision Trees

Author: **Sami Bosch**
Supervisors: **Hélène Verhaeghe, Siegfried Nijssen**
Readers: **Pierre Schaus, Gaël Aglin**
Academic year 2019–2020
Master [120] in Computer Science

Contents

Abstract

In this master thesis we aim to show a new method of building forests of decision trees using methods borrowed from optimisations. We first show that a naive method based on smarter decision trees doesn't work. The reasons for this are then explored in detail and some alternative improvements are given and analysed.

We then focus on showing how a more in depth algorithm based on column generation performs in comparison to state of the art methods that work similarly, and show cases where it works better than those. We also give further improvements that can be added in the future, and see which bottlenecks still need to be solved for this method to be applicable in practice.

All the code used for this thesis can be accessed at `https://github.com/samibosch/Master-Thesis-Random-Forests/`.

Chapter 1

Introduction

In the recent history of computer science, the field of machine learning has seen spectacular amounts of attention directed at it. Part of the reasons for this field's spur of growth can be linked to the advent of decision trees and linked methods, popularised due to their ease of interpretation as well as their solid predictive capabilities.

Due to the NP-completeness of finding optimal trees (optimal in the sense of minimising the number of tests required to classify new elements) [1], the most methods attempt to build up decision trees while avoiding overfitting. Notably, greedy algorithms [2–4] are quite widespread in use. These work by trying to maximise the quality of decisions locally, and potentially also using some post processing, such as pruning, to improve the quality of the tree on new data. In practice, they offer consistent and good prediction accuracy, but don't offer any guarantees. They may produce trees of higher than necessary complexity and lower accuracy than what could be possible, and can't work with additional constraints [5] on their output's structure.

In order to find a fix to these issues, more recent work has been aimed at trying to work with the NP-completeness of building optimal trees using inspiration from various other fields, notably optimisation and data mining. Constraint programming is one method that seems to show promise [6], as well as mining from itemset lattices [7].

Other work that aims to build upon the foundation of decision trees can be seen in bagging and similar methods [8] [9]. These methods work with multiple decision trees (or in the case of bagging, any other machine learning algorithm) and use a majority vote between the various instances to discriminate the input data. Boosting algorithms also exist, that iteratively improve upon a given classifier, and try compensating for its weaknesses by building new classifiers that fill the weak spots. Both bagging and boosting methods have shown to give great results, better than what single decision trees can achieve on most data. However, what they gain

in predictive information they lose in ease of readability. Whereas it's easy for a human to interpret a decision tree, bagging methods remain on the more opaque side.

The most notable of these might be random forests [9], which work specifically with decision trees, building the trees by partitioning the data randomly among them. Another important algorithm in this domain can be found in gradient boosting [10], which aims to iteratively improve trees by giving higher weight to previously wrongly classified data of the training set.

Another method that will be tested and adapted, which fits into this idea of building optimal forests is the LPBoost algorithm [11]. Another adaptation has already been done by gBoost [12]. The main idea behind both is to use column generation in order to get optimally constructed classifiers.

In this thesis, we outline a new method of creating bagging classifiers in multiple steps. In the first chapter, we will review all the work that has led up to constructing this new algorithm. Then, in the second chapter, we show a naive version of the final algorithm, and explain why this algorithm fails to provide sufficient results. We also give further details on improvements that were attempted, and how the results can be interpreted. In the third chapter, we create the final version of the algorithm, explore its strengths and weaknesses, and show how it can be adapted to fit different data. Finally, we will also review further improvements that could be added to this method if the time were given.

Chapter 2

Related Work

2.1 Decision Trees

Decision trees form the root of every method that will be analysed here and need to be discussed in depth. A decision tree is a structure, where each node represents a decision. For each node, a certain attribute of the data is assigned. There is one child per possible value for the attribute. Leaves are also given an associated class.

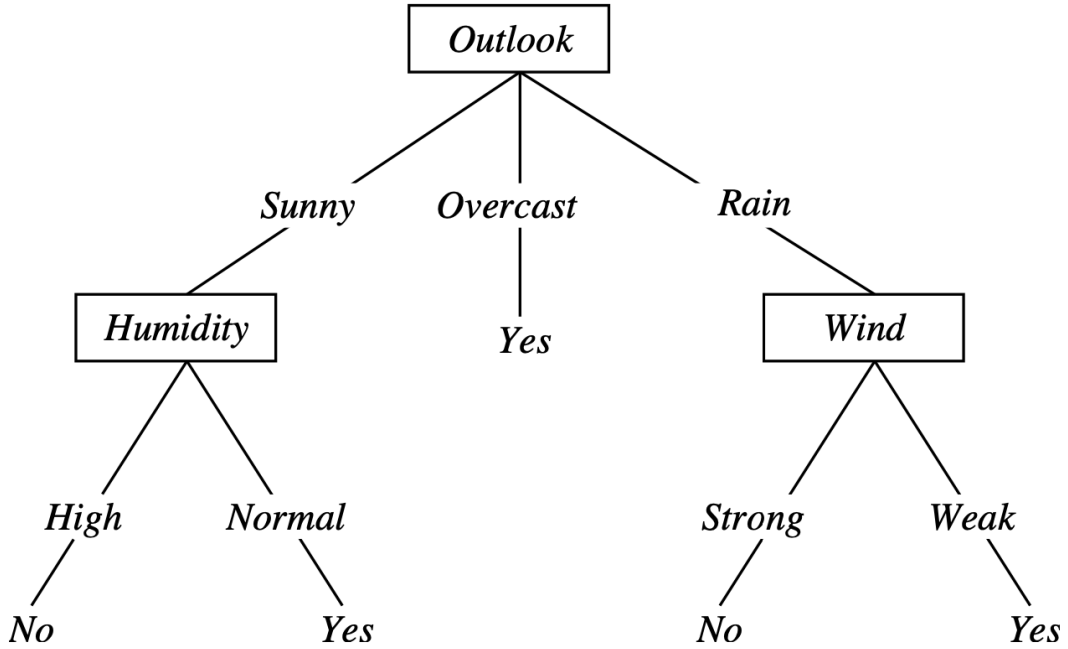

Figure 2.1: Example of a simple decision tree. Illustration from *Machine Learning*, T. Mitchell, McGraw Hill, 1997.

A lot of research has been spent trying to achieve algorithms that consistently find optimal decision trees. These trees are usually optimised on depth and error rates, with several constraints helping prevent overfitting, such as a minimum support (do not split a node if it has less than the support elements) in leaf nodes, a maximal depth or other constraints. One such example is the research done on building trees using SAT [13]. However, it has been proven that finding such trees is NP-complete [1]. In fact, we can show that the exact set cover problem can be reduced to the problem of finding an optimal tree. Since the exact set cover is a known NP-complete problem, we can conclude that building an optimal decision tree follows. We can however also optimise on accuracy directly, and this is also highly suspected to be NP-complete, though no formal proof exists yet.

The decision function $d(T, s)$ for tree T and sample s is calculated by iterating over the nodes and always going down the branch fitting the currently tested attribute. Given for subtree T_i, attribute $\mathbf{A_i}$ and child $T_i^{s_A}$ corresponding to value s_A for the attribute, and for all leaves a class c_{Ti} the function can be defined as:

$$d(T, s) = \begin{cases} c_T & \text{if leaf}(T) \\ d(T_i^{s_A}, s) & \text{otherwise} \end{cases}$$

Overall, decision trees are extremely easy and fast to use for both construction and classification, however one important drawback is that adding new data forces a full rebuild. Another extremely big advantage for decision trees generally is that they're extremely easy for humans to interpret unlike other types of classifiers.

Multiple algorithms exist that can be used to build Decision Trees. In this paper, we will focus on 3 of them, detailed in the next sections.

2.1.1 ID3

There are multiple ways of building decision trees, but let's first take a look at one of the most popular, ID3 [2]. ID3 is a greedy algorithm that seeks to find the best attribute possible for each node locally. In order to do so, it aims to maximise a value known as Information Gain. Let us assume that, for each class c, the node in question contains p_c elements. Let us further assume that the node contains n elements in total, and that there are C classes. We can calculate the information needed to encode the node with:

$$\mathrm{I}(\mathbf{p}) = \sum_{c=1}^{C} \frac{p_c}{n} \log_2 \frac{p_c}{n}$$

Let $\{A_1, A_2, ..., A_v\}$ be the values of attribute $\mathbf{A}$. Suppose it will partition our node into multiple nodes with $\{\mathbf{p^1}, ..., \mathbf{p^v}\}$ as their respective elements, each

corresponding to one A_i. The expected information required for the tree then becomes:

$$\mathrm{E}(\mathbf{A}) = \sum_{i=1}^{v} \frac{\mathbf{p^i}}{\mathbf{p}} \mathrm{I}(\mathbf{p^i})$$

We can then calculate the information gain associated to attribute $\mathbf{A}$:

$$\mathrm{gain}(\mathbf{A}) = \mathrm{I}(\mathbf{p}) - \mathrm{E}(\mathbf{A})$$

On each node, ID3 will find $\mathbf{A}* = \mathrm{argmax}_A \left[\mathrm{gain}(A)\right]$ such that it maximises $\mathrm{gain}(\mathbf{A})$. This algorithm has been shown to have good results in practice, and is also quite fast. On top, multiple variations have been given that prevent overfitting. The most common are to not split a node if it reaches below a given purity of $\frac{\max_c(p_c)}{n}$ given or once there's less than m minority class elements left in the node, or to prune nodes after building the tree. There are multiple methods to prune the nodes, using a validation set on top of the training set, notably:

- **Reduced error pruning:** Starting at the bottom, replace nodes with their majority class, ignore further attribute splits. If this doesn't worsen the classification accuracy, keep the new node. If not, keep the old node and stop pruning parents.
- **Cost complexity pruning:** This pruning method aims to optimise the tree by iteratively replacing the subtree which has the worst accuracy improvement (or in some cases, degradation) on the tree with a majority class node, then picking the resulting tree with the best classification accuracy. On step $i+1$, build tree:

  $$T_{i+1} = T_i - \underset{t \in T_i}{\mathrm{argmax}} \left[\frac{\mathrm{err}(T_i - t) - \mathrm{err}(T_i)}{|\mathrm{leaves}(T_i - t)| - |\mathrm{leaves}(T_i)|} \right]$$

  where $T_i - t$ is the tree T_i with subtree t replaced, $\mathrm{err}(T_i)$ is the validation error of T_i and $\mathrm{leaves}(T_i)$ is the set of leaves in T_i.

Overall, ID3 built decision trees have been shown to have good accuracy compared to other types of classifiers, but their speed is where they excel.

2.1.2 DL8

DL8 [7] is an algorithm that aims to find optimal decision trees using data mining techniques. This method is based on an itemset lattice (cf figure 2.2) representation of decision trees. The lattice is built such that at each layer n, it contains a set of

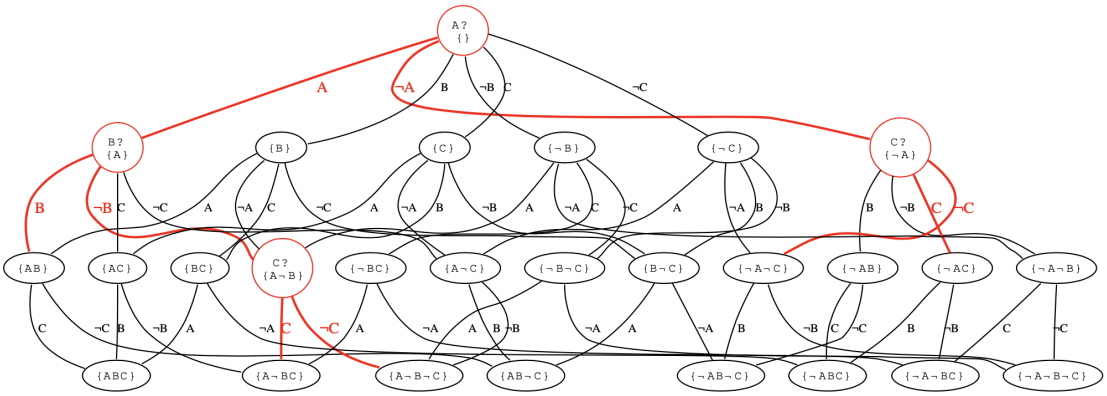

Figure 2.2: An example lattice for attribute set $\{A, B, C\}$. [7]

nodes which each represent one permutation of n attributes of the dataset. We can then execute queries on the lattice in order to limit its size and make its exploration quicker. Queries such as a minimum number of elements in leaves or a minimum correlation in leaves allow trimming down the lattice considerably. The possible queries given in the paper follow this pattern:

$$\mathcal{T} := \{T | T \in DecisionTrees, \forall I \in \text{leaf}(T), p(I)\}$$

where $\mathcal{T}$ is the resulting set of trees T, for which all the leaves respect a certain condition $p(I)$. This condition can be composed of the following parts:

- $freq_i(I) \geq minfreq$: checking for a minimum of $minfreq$ elements of class i in leaf I.
- $freq(I) \geq minfreq$: checking for a minimum of $minfreq$ elements in leaf I.
- $\chi^2(I) \geq mincorr$: checking for a minimum of $mincorr$ correlation between elements of I.
- $diff(I) \geq mindiff$: where $diff(I) = freq_m(I) - \max_{c \neq m} freq_c(I)$, ensuring that no class is majority by too wide a margin in any leaf.

We then can explore the lattice recursively. This allows us to use both dynamic programming and backwards maximisation to find the best trees we want without having to explore the entirety of the problem space. Using this, DL8 can usually find the best tree under constraints within a reasonable amount of time. However, the itemset lattice does still grow exponentially according to the number of parameters and the maximum depth to explore. Since at least partially building the lattice is

important to solving the problem, we can still run into execution time issues on bigger datasets.

This algorithm will remain central for this thesis, as both methods showcased in the coming chapters are based on DL8. DL8 can in theory be swapped with CP trees, however an existing implementation of the algorithm made DL8 much easier to work with in practice.

2.1.3 CP trees

CP trees [6] use Constraint Programming in order to find optimal trees. Constraint Programming is a sub-domain of optimisation, which relies on setting constraints on variables and combining with search algorithms in order to explore the entire feasible set of values for the given variables. In order to adapt this to decision trees, we need to define specific constraints.

The paper generalises an existing constraint, CoverSize, which is used to help with the filtering of the trees in the branch and bound algorithm. Particularly, a sparse version of this constraint is used. This sparse version is also adapted to take into account both the absence and presence of an attribute in a transaction. The constraint is defined as follows:

$$\texttt{CoverSizeSR}(\{K_1, \ldots, K_a\}, \{L_1, \ldots, L_b\}, D, c) \Leftrightarrow$$
$$c = \left| (\cap_{i=1}^{a} \{t \in \{1, \ldots, n\} \,|D_{t,K_i} = 1\}) \bigcup (\cap_{i=1}^{b} \{t \in \{1, \ldots, n\} \,|D_{t,L_i} = 0\}) \right|$$

The set $\{K_1, \ldots, K_a\}$ represents the set of selected features for the tree and $\{L_1, \ldots, L_b\}$ the set of dropped features. D is our binary dataset, of size $n \times m$. D_{t,K_i} represents the state of attribute K_i for transaction t. In essence, this constraint ensures that the number of overlaps between a specific selection of attributes among all transactions remains equal to c.

Using this constraint, and other more general CP constraints, we can then define a model that allows us to run a Branch and Bound algorithm on the constraints, progressively solving for the optimal tree. Similarly to DL8, the algorithm tries maximising a given objective function, usually defined by the error rate of the classifier on the training set.

2.2 Ensemble learning

Ensemble learning methods are a type of classifier that builds multiple instances of a simple one, then uses majority vote for classification. Bagging methods are a type of ensemble learning, based on creating multiple instances T of a given

classifier method, then using majority vote to decide the predicted class. The decision function for ensemble learning, given a set of classifiers T and a sample s, for a set of classes C, is given by:

$$d(T, s) = \underset{c \in C}{\operatorname{argmax}} \left[\# \left\{ t | t \in T \wedge d(t, s) = c \right\} \right]$$

Bagging classifiers are often used with decision trees as building blocks. A few notable examples of specific bagging algorithms using trees will be discussed here.

2.2.1 Random Forests

Random Forests are the most important building block to discuss for this paper, as it's the algorithm which we will aim to improve. Random forests rely on generating bags of trees based on randomly sampled data. Samples are taken with replacement. On top of that, trees are also build on the data samples semi randomly, enforcing limited selection between attributes on each tree. This method relies mostly on randomness to prevent overfitting, however, pruning and the classical methods traditionally used for decision trees are still perfectly working here.

Random forests take a considerable amount more time to construct compared to single decision trees, and run a decent amount slower as well, however, on most data set they perform much better than standalone trees for accuracy. It is also doable, though not very practical to extract the importance of the different attributes, so this advantage is also partly kept from the decision trees.

2.2.2 Gradient Boosting

Gradient Boosting take a vastly different approach to building ensemble classifiers than randomness. Instead the general idea is to build classifiers sequentially, and increase the importance of previously misclassified instances so that new ones adapt to previous mishaps.

In our case, we will only discuss gradient boosting used on decision trees. In order for this to work, we need to define behaviour for trees with weighted samples. The only difference with a standard tree is that, $\forall c \in C, p_c = \sum_{i:c_i=c} w_i$, where w_i is the weight of sample i and c_i its class.

This general principle is entirely akin to a discrete gradient descent. First of all, a loss function $L_y(f, x)$ must be defined for a classifier f, and a data set (x, y), where x is the set of samples and y their set of associated classes. The general step wise method from here is to:

- Initialise a tree T_0 with the chosen algorithm.
- For each step i:

– Obtain all previously misclassified samples s_i by way of $L_y(f, x)$ where f is the ensemble classifier $\{T_0, ...T_{i-1}\}$.
– Increase the weight of each sample in s_i by a factor α.
– Compute T_i based on the new weighted sample set.
– Add T_i to f.

2.3 Other algorithms

2.3.1 gBoost

gBoost [12] is a very essential algorithm in the context of this thesis. The context here isn't machine learning, but rather graph mining. The idea is to extract frequent and most notably informative patterns from graphs, allowing for subsequent classification on these graphs.

However, since the potential set of informative patterns is infinite, it is important to find a workaround to this. The way the authors approached this is by using a column generation based approach, building patterns gradually and updating weights. In order to solve this, the following parts were defined:

First of all, the primal problem, finding the necessary weight of each pattern to combine and reconstruct the original graph. The primal is defined as follows:

$$
\begin{aligned}
&\min_{\boldsymbol{\alpha},\boldsymbol{\xi},\rho} -\rho + D\sum_{n=1}^{l}\xi_n \\
\text{s.t.} \quad &\sum_{(t,\omega)\in\mathcal{T}\times\Omega} y_n\alpha_{t,\omega}h(\mathbf{x}_n;t,\omega) + \xi_n \geq \rho && \forall n \in \{1,\ldots,l\} \\
&\sum_{(t,\omega)\in\mathcal{T}\times\Omega} \alpha_{t,\omega} = 1 \\
&\xi_n \geq 0 && \forall n \in \{1,\ldots,l\} \\
&\alpha_{t,\omega} \geq 0 && \forall t,\omega \in \mathcal{T}\times\Omega
\end{aligned}
$$

ρ is the soft-margin, which is used to separate negative and positive examples, given an error margin ξ_n for each sample. $\mathbf{x}_n$ is a graph, to which class y_n is associated. $\boldsymbol{\alpha}$ is a set of weights associated to each pattern. $\mathcal{T}$ is the set of all such patterns that we consider at the current time. $\omega \in \Omega = \{-1, 1\}$ is used to check for both the presence and absence of patterns in a specific graph. h is a function defined as:

$$h(\mathbf{x}; t, \omega) = \omega(2x_t - 1)$$

and is used to check whether pattern t is present resp. absent in graph $\mathbf{x}$. $D \in \{0, 1\}$ is a parameter that is defined ahead of time.

This primal problem then gives us a dual problem that can be solved, whose optimum is equivalent to the primal. The dual is formulated as follows:

$$\begin{aligned}
&\min_{\boldsymbol{\lambda}, \gamma} \gamma \\
\text{s.t.} &\sum_{n=1}^{l} \lambda_n y_n h(\mathbf{x}_n; t, \omega) \leq \gamma && \forall t, \omega \in \mathcal{T} \times \Omega \\
&\sum_{n=1}^{l} \lambda_n = 1 \\
&0 \leq \lambda_n \leq D && \forall n \in \{1, \ldots, l\}
\end{aligned}$$

where $\boldsymbol{\lambda}$ is the set of weights set on the graphs instead of the patterns. γ is a margin, similar to ρ, but without any soft-margin. This dual problem can then be used to solve the column generation step, which will generate the next best pattern not yet included in $\mathcal{T}$. In order to do so, we find the pattern $(t*, \omega*)$ which violates the dual constraints maximally:

$$(t*, \omega*) = \underset{t \in \mathcal{T}, \omega \in \Omega}{\operatorname{argmax}} \sum_{n=1}^{l} \lambda_n y_n h(\mathbf{x}_n; t, \omega)$$

The reason gBoost is critical to our optimal algorithm will be explained in chapter 4.

Now let's view the first method we implemented and tested.

Chapter 3

First step: A naive algorithm

The first algorithm that was tested was based on a very simple idea: replacing ID3 in random forests with DL8.

3.1 Method

In order to implement this DL8-forest, we used a simple bagging method, which creates a given amount n trees using DL8 and randomly selected subsets of the data of a given size s. We then use a majority vote solution exactly like standard random forests in order to predict on new data.

These variations were tested for both sample and attribute selection including:

- Random sample selection: For samples we mainly used a method that selects a certain percentage of the dataset to train each tree. For each new tree, an independently selected subset was taken, without checking for overlaps with previous tree training sets, then used for building the new classifier.
- All attributes: The simple baseline was to select all attributes. This was only used in conjunction with random sample selection.

3.2 Preliminary Results

The first results were done using random sample selection and all attributes.

As we can see in figure 3.1, this method does not provide any respectable results. In the best case, it beats DL8 and the Random Forest by 1% on splice-1, but onall other datasets it either fails to surpass the others, or trails behind by up to 17% on lymph. In fact, in most cases, it has worse prediction accuracies than the basic DL8 tree.

In order to understand these results, we've done an internal analysis of the forest's structure, checking two very important factors:

- How frequently does each feature appear? If an element appears too often, or if few elements are used at all, we will know that the forest lacks in diversity. Since DL8 is a non greedy method, often times small variations in the data won't affect it sufficiently to cause major differences in the tree's parameter structure.

- How often the trees remain unanimous in a forest. If the trees are mostly unanimous most of the time, that means they are too redundant and fail to cover cases that the other trees might no classify properly. Essentially, if the classifiers are too similar, the forest ends up no better than the original algorithm, which is the case here.

Data for these two metrics are given in figures 3.2 and 3.3. In the unanimity graph, we can clearly notice that over 90% of the time, all trees predict the same class. Looking into the tree structure in the attribute spread graph, we can note that parameter 58 notably appears in a total of 72.5% of trees. On top of this, the parameters 16 and 0 appear notably often as well, close to twice as often as any other parameter. It is also interesting to note that the root only has 4 parameters that appear more than 10% of the time. As such, both metrics explain the weak results that are observed. Trees are too uniform, remaining mostly unanimous across the entire dataset.

Now that we have figured out the root cause of this, what can we do to improve the results?

3.3 Further improvements

Multiple other methods were tested to inject more randomness or prevent frequent reuse of the same parameters. These methods are the following:

- All samples: The second method that was tested included all samples for each tree. This method was only tested in conjunction with random or progressive attribute selection methods.

- Progressive attribute selection: In this type of attribute selection, we always eliminate the attribute that was considered the best and train the subsequent trees without said attribute.

- Progressive random attribute selection: Here the principle is similar to progressive selection, but we use randomness to decide which attribute to

exclude next. The randomness is done such that at any subtree of the previously trained tree, we have a 1 in 3 chance of excluding the root, and a 1 in 3 chance to continue on to the left respectively right subtree. This effectively means that the less important an attribute is, the lower its chance of being excluded.

- Random attribute selection: Similarly to the random sample selection, a method was tested in which each tree worked from a randomly sampled subset of attributes. This was also done as a percentage of the total available attributes, similarly to samples.

Let's see how these new methods work/interact with each other.

3.3.1 Results and discussion

First, let's look at results for the progressive, progressive random and random attribute sampling methods. As one would expect, both methods should decrease the redundancy of trees, albeit at different degrees each.

As we can see in figure 3.4, these methods barely have any impact compared to selecting all attributes (differences are within 1%), except random attribute sampling, which causes clear degradation of the results by almost 5%. We can check the unanimity graph of the progressive method to see if the redundancy of the trees has been altered at all.

We can notice in figure 3.5 that the unanimity spread has barely improved at all. Still vastly more than 90% of transactions have all trees remaining unanimous. It remains difficult to prevent the forest from building trees with similar internal structure without degrading the results drastically (like in the random attribute selection method).

On smaller datasets, such as vote, the variation in trees is bigger. Let's check how the progressive sample selection method compares on vote.

As we can see in figure 3.6, results do improve by roughly 5%, but still fail to surpass those that the standard random forest provides. While they come closer, the margin still remains. Let's check how the unanimity graph stacks up on this dataset.

We can clearly notice in figure 3.7 that redundancy is much less of an issue here. Here, thre trees are aligned only roughly 80% of the time, a clear improvement compared to the larger sets. However, since even the best results here fail to compare to the random forest, this method shows no particular use in concrete use cases.

Let's give this one last try by comparing progressive, progressive-random and random attribute sampling combined with all sample selection. We can see in

figure 3.8 that results on hypothyroid are minimally impacted, only seeing an improvement of 0.5% at best. On vote however accuracy does drop by roughly 1%. As such, we can see that this is not a consistent improvement over the initial methods. An analysis of tree unanimity shows that the forest retains a very similar internal structure as with random sample selection.

Interesting to note however is that random attribute selection has worse results with all samples than with a restrained subset. This is most likely due to all trees having equal voices, and random subsets of attributes creates trees which end up having extremely low predictive power. As we can also see, smaller datasets end up with worse accuracies on random attribute subsets as well.

3.4 Conclusion

At this point, we can clearly notice that this naive method is not really a promising method. So we can ask ourselves, what should our approach be instead? In the next chapter, we review the second method that this paper outlines.

Dataset	Size	DL8	DL8-forest	Random Forest
zoo-1	25	96.58%	94.74%	99.34%
hepatitis	34	76.60%	71.36%	82.72%
lymph	37	73.03%	63.87%	80.45%
audiology	54	93.21%	88.02%	89.01%
heart-cleveland	74	70.68%	71.22%	78.11%
primary-tumor	84	77.74%	74.68%	77.46%
ionosphere	87	80.27%	84.43%	87.80%
vote	108	92.32%	90.40%	94.28%
soybean	157	92.39%	89.22%	85.94%
australian-credit	163	81.78%	83.61%	84.12%
breast-wisconsin	170	93.45%	94.42%	96.49%
diabetes	192	70.64%	72.00%	75.14%
anneal	203	81.71%	79.89%	81.94%
vehicle	211	91.94%	92.50%	92.13%
tic-tac-toe	239	74.45%	76.12%	72.99%
german-credit	250	68.55%	70.71%	70.76%
yeast	371	67.46%	67.91%	69.83%
segment	577	99.44%	98.33%	99.79%
splice-1	797	92.02%	93.05%	91.02%
kr-vs-kp	799	93.88%	93.89%	92.44%
hypothyroid	811	97.68%	97.48%	95.90%
pendigits	1873	99.09%	99.33%	98.65%
mushroom	2031	99.90%	99.63%	97.06%
letter	5000	98.11%	98.17%	95.98%

Figure 3.1: Results for DL8 vs Random Forest vs the naive DL8 forest. Results done with a depth limit of 3 (no other restrictions). These results are taken for classifiers trained on 25% of the data sets, and tested on the remaining 75%. Accuracy is given as average on 10 runs.

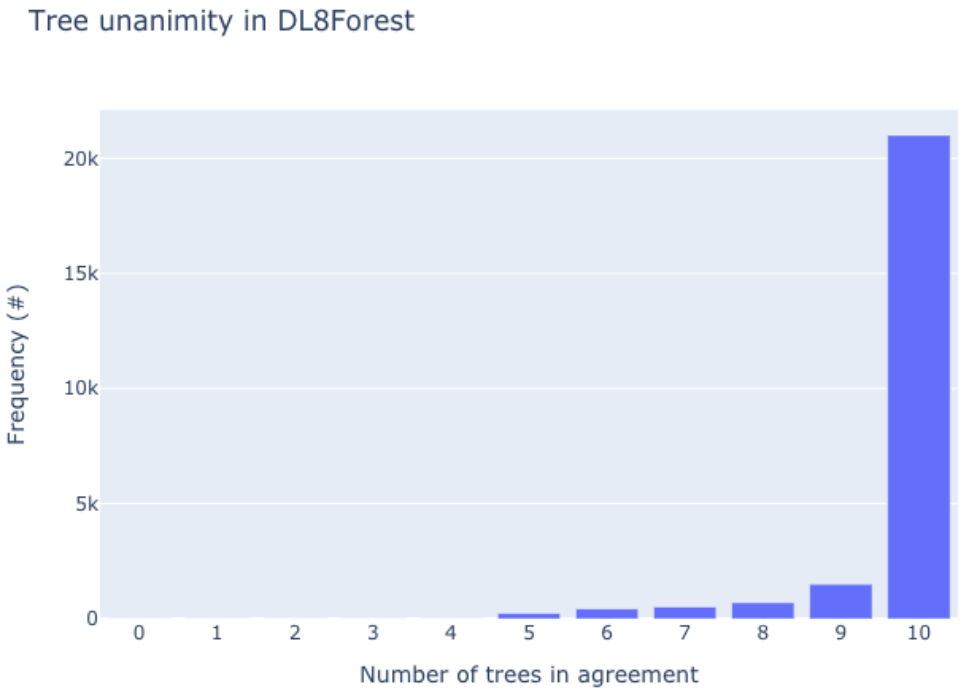

Figure 3.2: The tree unanimity for the dataset "hypothyroid". This is for a forest of 10 trees.

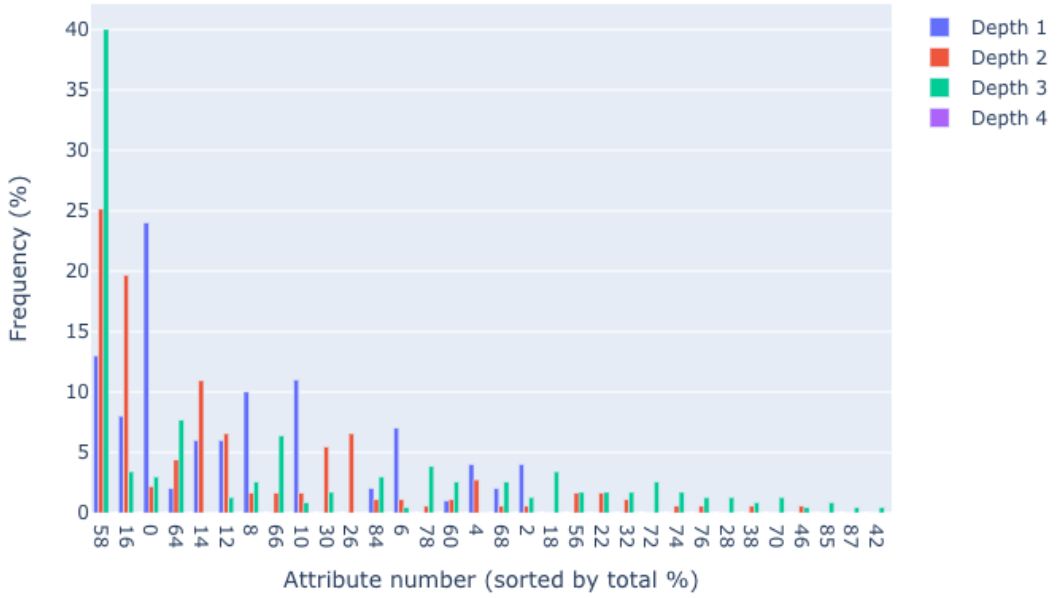

Figure 3.3: The feature selection for the dataset "hypothyroid". The parameters are ordered by total frequency.

Dataset	Size	DL8	All	Progressive	Progressive random	Random
hypothyroid	811	97.76%	97.48%	97.62%	97.15%	92.88%

Figure 3.4: Results on hypothyroid for all, progressive, progressive random and random attribute selection in DL8-forests. Accuracy is given as average on 10 runs.

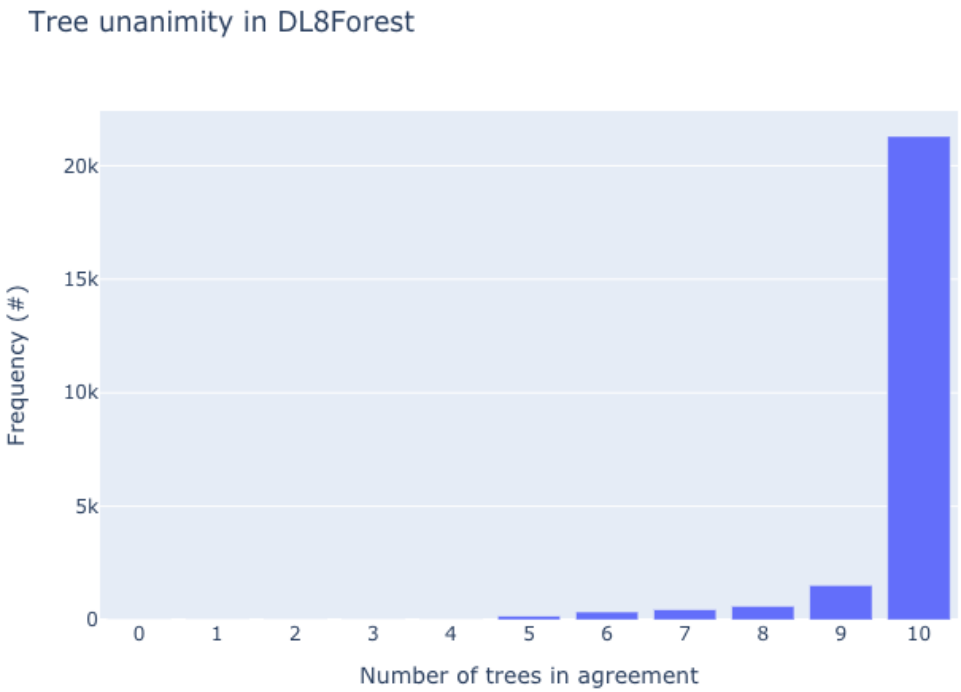

Figure 3.5: The tree unanimity for random sample selection combined with progressive attribute selection. This is for a forest of 10 trees.

Dataset	Size	DL8	DL8-forest	R-forest
vote	108	91.07%	94.74%	95.02%

Figure 3.6: Results with progressive sample selection on vote.

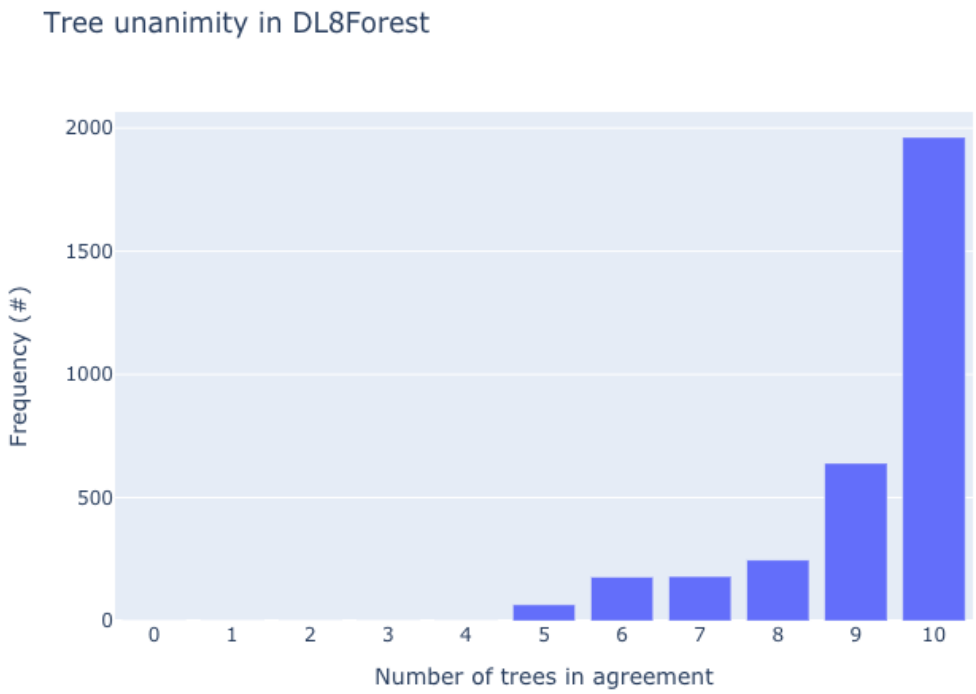

Figure 3.7: Unanimity with progressive sample selection on vote.

Dataset	Size	DL8	progressive	progressive-random	random
vote	108	92.32%	93.70%	91.52%	63.58%
hypothyroid	811	97.43%	95.47%	97.83%	91.62%

Figure 3.8: Results for all samples on vote and hypothyroid. Accuracy is given as average on 10 runs.

Chapter 4

Second step: An optimal forest

In order to fix the previous forest, we introduce a new key idea. Instead of accumulating a bunch of forest, some of them often the same or very similar, and then having them vote unanimously, it could be a great idea to apply a weight to each tree's vote. Let's see how this would look in theory, and how we applied this in code.

4.1 Method

4.1.1 Mathematical principles

Weighted Forests

In order to define the mathematical basis for this method, we first need to define a few concepts. First of all, the decision function of the new forest. Given a forest F made up from a set of trees T and a weight vector $\mathbf{w}$, such that w_t is the weight associated to tree $t \in T$, trying to discriminate into a set of classes C, the decision function of F on a sample s is:

$$d(F, s) = \underset{c \in C}{\operatorname{argmax}} \left[\sum_{t \in T \wedge d(t,s)=c} w_t \right]$$

For simplicity, we can consider only the two class case, though it is not difficult to extend to multi-class later. Now that we have the forest's decision function, the next steps are to define T and $\mathbf{w}$.

The set T of trees, optimally should be the infinite set of all possible trees given the current set of attributes. However, in practice, optimising on an infinite set is impossible. Thus, we can do a first filtering on various parameters:

- We can limit the depth of trees, such that $\forall t \in T : \text{depth}(t) \leq \delta$.

- We can limit the tree's minimum support, s.t. a node cannot be split if it contains less than m samples.

This does mean that the set of trees is not optimal in theory, but we can optimise under constraints. However this is still not sufficient. With only 10 attributes, with a max depth of exactly 3, we still have to explore over 3 million different trees. This number grows exponentially the more attributes there are in the dataset. Thus we still have to find another way to reduce the size of T.

Multiple methods can be used here, however the very important note to keep in mind is that we can never had 2 identical trees in T'. In this master thesis, we focus on comparing two methods to generate T'. The first is to generate only exactly a single tree as a starting point over the whole dataset. Since we use DL8, we know that we will always obtain the best tree possible under the given constraints and data. We then let the column generation handle the remaining trees. The second option is to borrow from our naive method, and generate a set of trees using either the progressive or progressive-random methods. We will later compare the two and see how they impact results.

Optimisation principles

For this, we will borrow the idea of the primal-dual maximisation problem used by gBoost [12], and adapt it to our needs. The idea behind here is to apply a column generation algorithm. The entire method works as follows:

1. Start with a set of trees T', small subset of T under constraints.
2. Solve the weight optimisation problem over T', such that the weights optimise a certain objective function. We will talk about the choice of this function later.
3. Solve the dual weight optimisation problem over T', such that the weights are set on all the samples of the sample set.
4. Find the best tree t that gives the best improvement to the objective function. If t improves the primal objective, update $T' \leftarrow T' \cup t$ and repeat from step 2. Otherwise, return.

Now let's explore the different steps of the method. The first step, building an initial set T', was explained in the previous section, so let's review the remaining steps of the algorithm.

The second step is the primal. The objective function we will optimise on will not be given by a direct optimisation of the training accuracy. We will explain in

the results section why this does not work well. Instead, we will try to maximise over a margin function. This margin, noted ρ, is defined such that it always takes the value of the least well classified sample in the set (e.g. -1 if no tree in T correctly classifies it, +1 if all trees in T correctly classify it) up to an error term ξ_s for each sample s. Since we consider only two classes in the following scenario, we will consider the classes as -1 and +1, which simplifies the expression to check for a misclassified object on tree t to $c_s d(t, s)$ with c_s being the correct class of sample s. We can extend to the multiple class case simply by replacing that term by a term checking that is -1 if the sample is misclassified, and 1 otherwise. In order to ensure the solution is unique, we will set the sum of the weights to be equal to 1. We also constrain the weights and error terms to be positive, as negative values would not be logical in this context. Lastly, knowing this error term, we want to minimise over it, at a ratio D compared to the maximisation of ρ. Given all those definitions, here is the mathematical expression the primal problem, ran over a sample set S:

$$
\begin{aligned}
& \max_{\mathbf{w},\rho} \rho - D \sum_{s \in S} \xi_s \\
\text{s.t.} \quad & \rho \leq \xi_s + \sum_{t \in T} c_s w_t d(t, s) && \forall s \in S \\
& \sum_{t \in T} w_t = 1 \\
& w_t \geq 0 && \forall t \in T \\
& \xi_i \geq 0 && \forall i \in s
\end{aligned}
$$

As for the dual step, we will simply convert the primal directly. Considering an objective γ, and a set of sample weights $\boldsymbol{\lambda}$ such that λ_s is the weight associated to sample s, we have:

$$
\begin{aligned}
& \min_{\boldsymbol{\lambda},\gamma} \gamma \\
\text{s.t.} \quad & \gamma \geq \sum_{s \in S} c_s \lambda_s d(t, s) && \forall t \in T \\
& \sum_{s \in S} \lambda_s = 1 \\
& 0 \leq \lambda_s \leq D && \forall s \in S
\end{aligned}
$$

From here, we can run DL8 with a new error function using the given sample weights $\boldsymbol{\lambda}$. This gives us the column generation step. The error function we want DL8 to minimise is the following:

$$error(T', S, \boldsymbol{\lambda}) =_{s \in S} c_s d(T', s) \lambda_s$$

Thus, DL8 will construct the tree t minimising this error function. In order to check if the tree will improve the primal objective, we simply have to check if its error γ_t is greater than γ. Thus we evaluate:

$$\gamma_t = \sum_{s \in S} c_s \lambda_s d(t, s)$$

If the tree is fit, and if $t \notin T'$, we can add it to the set $T' \leftarrow T' \cup t$. If not, we stop and return the current T' and $\mathbf{w}$.

In order to implement this, we used the gurobipy optimisation toolkit. This toolkit allowed us to model and solve the primal and dual problems easily. Using this toolkit, and then defining the error function for DL8, the code is relatively simple to write.

Let's see then how well this iteration of the algorithm works in practice.

4.2 Important notes

In order to properly view how well the algorithm works, let's first get some cases in which it doesn't work out of the way, starting with building an initial set T' bigger than 1 tree. In all tests that have been done, this has led to drastic drops in accuracy on all datasets. This method will thus not be viewed in detail, and we will instead focus on building a forest starting with $|T'| = 1$.

Another interesting point to note is that a second primal objective was tested, which aimed to optimise on the training accuracy directly. This however resulted in extremely weak forests, because of 2 main factors: overfitting and the fact that on all datasets, 2 trees were always enough to reach 100% training accuracy, meaning no further improvements could be done. The testing accuracy on the testing sets hovered below 70% on all datasets, sometimes even reaching below 50%. This method will thus also be disregarded.

The most interesting point to note however is one related to the run time of the algorithm. Due to the algorithm's excessive run time on large datasets, a lot of the results shown for depth 3 in the next sections used a timeout on the construction of trees with DL8. We tested multiple different timeout values, but the variation in results was extremely minimal, thus we kept a timeout of 60 seconds. However, on the larger datasets still, the algorithm could sometimes take entire days to build a forest. We thus took to analysing how the accuracy of the forest evolved as we added more trees, and noted that it plateaus quickly after the objective ρ reaches above 0.

We can see in figures 4.1 and 4.2 that accuracy tends to stagnate quickly. In terms of execution time, the german-credit dataset took over a full week of runtime to generate the 10 forests graphed here, while vote took a few mere seconds. Let's see what information we can extract from this.

We can first notice that, at the point where $\rho > 0$, the average accuracy on german-credit reaches 69.01%. With 10, 20 and 50 more trees, the accuracy varies by roughly 0.5% up or down, without any noticeable improvement. By the end, with roughly 5 hours of execution time per forest, we reach an accuracy of 70.12%, barely any improvement given the time. On vote, we notice the same kind of stagnation, albeit with far less trees. We have thus decided to stop improving forests after set amounts of trees one the point where $\rho > 0$ has been passed. In practice, 10 trees seems to work well.

A stopping value that seems to work well in practice is to stop 20 trees after ρ reaches above 0. We will thus use this as the defacto for the results in the next sections.

4.3 Results

Let's first look at results for depths 1, 2 and 3 (cf figures 4.3, 4.4 and 4.5), and compare those to base DL8, Random Forests and Gradient Boosting on equal depth. All these results were done on 25% training size and 75% testing size.

Multiple interesting things to note here are that, first of all, a 1 depth optimal forest is the same as what is known as a decision stump, and performance accordingly is identical (not shown in the tables). It is also interesting to note that on some datasets (primary-tumor and letter), no feasible solution is ever found. Thus, this version of the optimal forest is not universally applicable, and can easily run into issues. It is however an interesting benchmark to compare performance to.

When comparing depth 2 and 3, we can notice that, while depth 2 outperforms the other algorithms consistently on the larger datasets, sometimes even by relatively large margins (>1% on mushroom, kr-vs-kp and even up to 3% difference on soybean), on depth 3, results overall plummet a bit (mushroom only outperforms by less than 0.1%, soybean likewise). Some datasets however show improvement, such as tic-tac-toe with an accuracy 5% superior to any other, and kr-vs-kp which doubles its lead. We can attribute the drop in performance to two factors.

First of all, it is not infrequent for the DL8 base tree used to start building T' to be sufficiently strong to get 100% classification accuracy on the training set. Whenever this happens, the optimal forest cannot build any tree that would improve the forest, and thus stops at a single tree. Therefore it occasionally happens that the forest doesn't provide results that improve upon the base DL8 ones. This happens more frequently on smaller datasets or higher depths. E.g. on zoo-1 we

can notice that both depth 2 and 3's accuracy is exactly equal to that of DL8. This is due to this phenomenon.

The other reason comes down to overfitting. If we take a look at accuracies for letter for example, we can notice that depth 3 actually performs worse than both depth 2 and base DL8. Looking at training accuracies, in both cases the forest reaches 100%. We can thus determine that despite the optimisation problem's formulation, overfitting can easily happen on bigger depths.

Now let's review the cases where the algorithm shows promise.

4.3.1 When does this algorithm work well?

First of all we can note that in larger datasets, the optimal forest tends to perform comparatively better. This is particularly true for a depth of 2, as on some of the more complex ones, a depth of 1 fails to solve, while a depth of 3 causes aforementioned issues. We can see for example that in figure 4.4, on mushroom, the optimal forest beats the other algorithms by over 1%. On the opposite, on lympth, one of the smaller datasets, the optimal forest trails roughly 5.5% behind the random forest. This behaviour remains rather consistent with some outliers.

We can also notice that on some smaller datasets, the forest still outperforms other algorithms by a respectable margin. In fact, if we look at lymph and audiology, we can notice that the forest works well on datasets with high amounts of attributes. Audiology has 147, while lymph has 67. Compared to zoo-1 which only has 35 attributes. If we check all datasets, we can consistently notice that sets with more parameters are classified better by the algorithm. Interestingly, splice-1 has the largest amount of parameters in the tested sets, yet figures among those where performance is worst. It is hard to conclude on this, as the structure of the data definitely influences results, but this remains relatively consistent on all datasets otherwise.

4.3.2 Execution times

However, the algorithm suffers greatly in terms of run time. We can see in figure 4.6 that runtimes reach up to half hours on the larger datasets (letter most notable, however splice-1 and pendigits aren't spared the long wait times either). We compare here a depth 3 random forest with a depth 2 optimal forest, as their results are very close to each other. In most cases, it remains in reasonable margins (only a handful of seconds per tree). However, running cross-validation on the larger datasets in order to fine tune parameters takes easily upwards of full days depending on when we set the cutoff point for the forest.

4.3.3 Parameters and their effects

Minimum support in leaves

One parameter we tried tweaking was the amount of minimum support in each tree. This should in theory accelerate DL8 in finding trees without hurting performance in any major way. However, in practice, the results suffered a fair amount (cf figure 4.8 and 4.7). We can note that on a minimum support of 5, even the datasets where normally optimal forests outperform the others, such as kr-vs-kp, audiology and soybean, results fall behind by upwards of 12%. It is also interesting to note that the algorithm on depth 1 suffers from this especially as it occurs more frequently that no solution is found (e.g. diabetes which is normally solved without a minimum support). We suspect that a rather large amount of sufficiently good trees were discarded, leading in a rapid deterioration in the overall prediction quality.

Another change that was attempted was to remove the error coefficient ξ_i from the formulas (and, consequently, remove the factor D and its implication on the dual), and using a minimum support to emulate the role of the error coefficient, however the dual becomes unsolvable in most cases with this change. Thus we discarded this idea.

We can thus deduce that using minimum supports is more harmful than it is helpful in the context of optimal forests.

Tweaking the parameter D

The error multiplier D in the primal can be tweaked for differing results depending on the dataset. Various values were tested on all the datasets, testing 0.2, 0.5 and 1 (cf Figures 4.9 and 4.10). However, these values only caused minimal change in the results. Most datasets showed the best results at $D = 0.2$ (example, depth = 2 on australian-credit gives an accuracy of 81.06% instead of 80.96%), while some worked best with $D = 0.5$ (example, depth = 1 on australian-credit gives an accuracy of 80.65% instead of 80.61%). The results in the previous sections are all given with $D = 1$, so small improvements in the result quality can be done with slight tweaking here. However, these variations are extremely negligible. They are never considerable enough to push the optimal forest above any of the other algorithms on a dataset on which it is usually worse.

The best way to work with this would be to use cross-validation to test and get the best value of D possible for each dataset. However, as touched upon in section 4.3.2, cross-validation is sometimes not a realistic option for the optimal forests.

4.3.4 Resistance to noise

Another aspect that we tested was how well the algorithm performs when there's noise in the dataset. We used random perturbation in the dataset to emulate natural noise, randomly flipping the presence or absence of a parameter with a given percent chance. However, as shown in figures 4.11 and 4.12, the algorithms perform extremely similarly with or without noise. The datasets where the optimal forest is normally the best remain the best (such as letter or pendigits), while the datasets where it usually is beaten by others, it remains worse (such as splice-1 or german-credit). There are some notable differences however in that some datasets see the optimal forest perform far worse in depth 3, such as mushroom, where the accuracy plummets by a staggering 4%, or vehicle where accuracy takes only a slight dip of 0.4%, while the others drop enough to fall behind. Due to a lack of any distinguishable pattern in this, we can thus deduce that this algorithm doesn't resist noise any better than the other algorithms that we used as comparison.

4.4 Future improvements

Multiple improvements can still be done on this algorithm. First of all, it is very important to note that the library that was used for DL8 did not allow for defining a custom heuristic method. This means that the DL8 trees were build in manners that might be sub-optimal given the cap on execution time. If we had a proper heuristic, it would in theory drastically improve execution times. This leads us to multiple potential improvements.

First of all, combining tests with a real cross-validation to determine both the depth and D for each dataset would be extremely useful and clearly boost the results of the forest. It is hard to say exactly how impactful this change would be, but it is definitely a factor to consider.

Second of all, with lower execution time, DL8 will be able to return better trees much faster, meaning that on the bigger datasets where the time limit often came into account, accuracy might improve drastically. This means that, given a better implementation, this algorithm could be preferable to gradient boost or random forests in large datasets. The question remains of the trade-off between execution time and accuracy, as despite any improvement in accuracy we can bring like this, the optimal forest remains a rather time consuming algorithm to run.

Another very important note here, is that all the tests were ran on binarised datasets, meaning that they had vastly more attributes than a normal dataset would have. Many of the attributes were also mutually exclusive, but the algorithms weren't ran with this taken into account. Extending to a multi-class case with non-binary data could show promise, as this would heavily reduce execution time

of DL8 again.

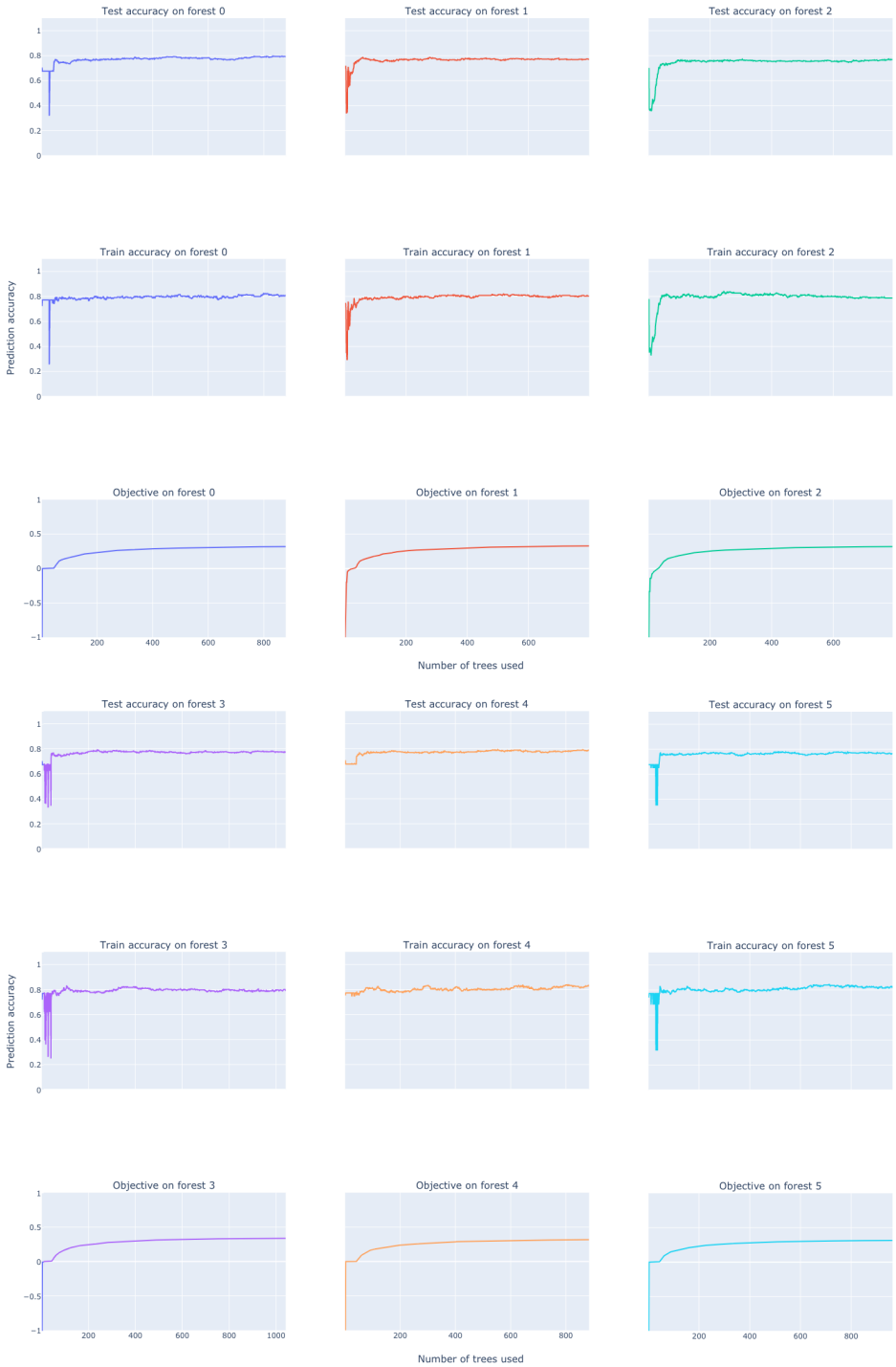

Figure 4.1: Evolution of the training and testing accuracy as well as the optimum ρ for the dataset german-credit on depth 3. Training was done with 25% of the datasets respectively. $D = 1$, no cutoff point for the maximum amount of trees.

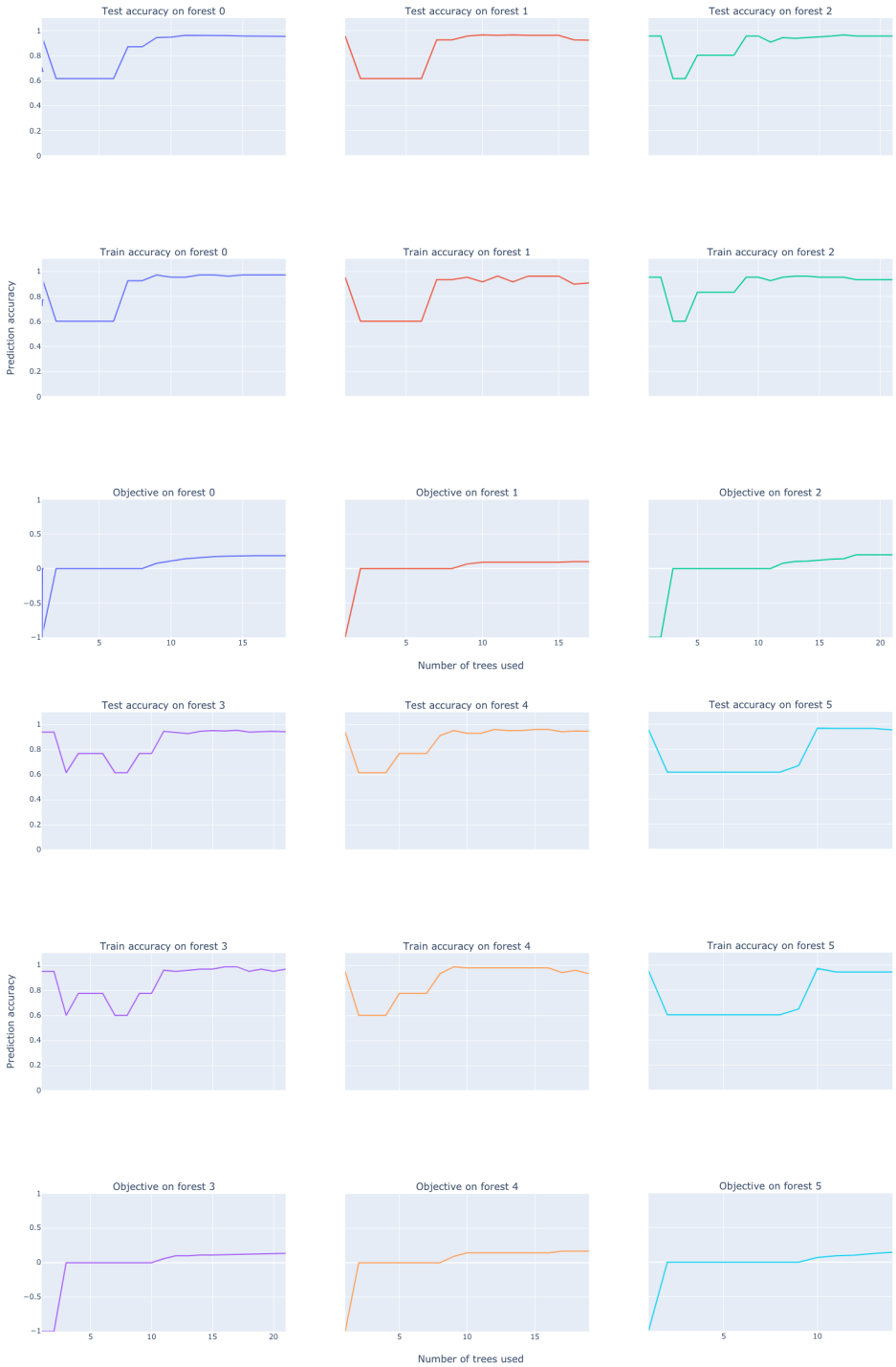

Figure 4.2: Evolution of the training and testing accuracy as well as the optimum ρ for the dataset vote on depth 1. Training was done with 25% of the datasets respectively. $D = 1$, no cutoff point for the maximum amount of trees.

Dataset	Size	DL8	R-forest	G-boosting	OptDL8-forest
zoo-1	25	99.21%	99.61%	99.21%	99.21%
hepatitis	34	80.29%	82.91%	82.91%	76.31%
lymph	37	71.08%	75.59%	74.41%	77.30%
audiology	54	86.23%	76.54%	89.38%	90.43%
heart-cleveland	74	74.55%	74.59%	80.05%	73.24%
primary-tumor	84	76.87%	75.83%	77.82%	-
ionosphere	87	82.01%	81.02%	87.65%	84.66%
vote	108	94.74%	91.96%	94.98%	93.70%
soybean	157	84.88%	85.24%	85.43%	78.14%
australian-credit	163	86.24%	79.24%	86.24%	77.78%
breast-wisconsin	170	91.35%	95.01%	95.24%	94.64%
diabetes	192	73.18%	71.79%	74.36%	72.03%
anneal	203	81.15%	79.21%	81.64%	78.85%
vehicle	211	75.81%	74.13%	83.29%	92.90%
tic-tac-toe	239	68.62%	66.11%	68.80%	72.61%
german-credit	250	68.35%	70.68%	70.89%	67.85%
yeast	371	69.61%	68.72%	68.79%	67.82%
segment	577	98.17%	98.15%	98.32%	99.75%
splice-1	797	82.09%	83.78%	88.57%	89.71%
kr-vs-kp	799	68.00%	80.19%	90.73%	85.87%
hypothyroid	811	96.47%	91.40%	96.53%	92.46%
pendigits	1873	93.21%	89.50%	96.96%	98.36%
mushroom	2031	88.74%	91.31%	95.53%	95.14%
letter	5000	95.96%	95.96%	95.96%	-

Figure 4.3: Comparative results for the different algorithms on depth 1. Dashes represent datasets where no forest was found with positive objective. Cutoff set at 10 trees after the point where $\rho > 0$. $D = 1$. Accuracy is given as average on 10 runs.

Dataset	Size	DL8	R-forest	G-boosting	OptDL8-forest
zoo-1	25	95.79%	99.74%	99.47%	95.79%
hepatitis	34	79.42%	83.40%	81.26%	80.29%
lymph	37	74.14%	79.28%	79.10%	73.60%
audiology	54	93.40%	81.30%	94.14%	94.69%
heart-cleveland	74	73.87%	78.24%	78.33%	73.56%
primary-tumor	84	80.83%	77.98%	79.80%	73.77%
ionosphere	87	82.99%	87.31%	88.52%	85.42%
vote	108	93.06%	93.73%	94.50%	93.09%
soybean	157	88.54%	85.24%	89.30%	92.33%
australian-credit	163	84.06%	82.22%	85.55%	80.80%
breast-wisconsin	170	94.48%	96.45%	95.69%	95.07%
diabetes	192	71.39%	74.18%	73.70%	68.26%
anneal	203	81.79%	81.69%	82.97%	75.14%
vehicle	211	88.93%	85.95%	93.40%	93.92%
tic-tac-toe	239	68.14%	68.26%	69.76%	80.49%
german-credit	250	70.25%	69.55%	71.40%	66.87%
yeast	371	68.29%	68.79%	69.29%	66.97%
segment	577	99.17%	98.46%	98.88%	99.38%
splice-1	797	83.05%	89.12%	94.22%	89.59%
kr-vs-kp	799	87.01%	91.46%	94.17%	95.86%
hypothyroid	811	97.60%	92.77%	97.90%	96.68%
pendigits	1873	97.82%	96.93%	98.67%	99.39%
mushroom	2031	96.90%	94.64%	98.87%	99.94%
letter	5000	96.97%	95.92%	97.39%	98.77%

Figure 4.4: Comparative results for the different algorithms on depth 2. Cutoff set at 10 trees after the point where $\rho > 0$. $D = 1$. Accuracy is given as average on 10 runs.

Dataset	Size	DL8	R-forest	G-boosting	OptDL8-forest
zoo-1	25	95.79%	99.74%	99.08%	95.79%
hepatitis	34	76.80%	82.82%	81.26%	76.80%
lymph	37	71.44%	78.74%	76.04%	71.71%
audiology	54	91.48%	92.10%	92.84%	91.98%
heart-cleveland	74	69.28%	78.60%	76.58%	71.04%
primary-tumor	84	78.17%	79.88%	79.80%	47.42%
ionosphere	87	83.52%	89.24%	88.22%	85.76%
vote	108	91.07%	94.43%	93.55%	91.10%
soybean	157	92.39%	86.34%	90.57%	92.39%
australian-credit	163	82.12%	83.22%	85.22%	79.45%
breast-wisconsin	170	93.14%	96.49%	94.64%	93.76%
diabetes	192	71.04%	75.09%	74.18%	68.26%
anneal	203	83.55%	81.72%	83.60%	74.58%
vehicle	211	92.93%	93.06%	94.02%	93.39%
tic-tac-toe	239	73.84%	72.80%	79.94%	84.05%
german-credit	250	68.35%	70.65%	71.49%	66.49%
yeast	371	67.36%	69.72%	70.93%	61.07%
segment	577	99.57%	99.07%	99.62%	99.57%
splice-1	797	91.97%	90.14%	94.89%	90.65%
kr-vs-kp	799	93.73%	93.45%	94.06%	96.31%
hypothyroid	811	97.74%	95.53%	97.85%	93.09%
pendigits	1873	99.14%	98.76%	99.43%	99.31%
mushroom	2031	99.89%	96.94%	99.84%	99.93%
letter	5000	98.10%	96.27%	98.40%	96.41%

Figure 4.5: Comparative results for the different algorithms on depth 3. Cutoff set at 10 trees after the point where $\rho > 0$. Timeout of 60s set for DL8. $D = 1$. Accuracy is given as average on 10 runs.

Dataset	Size	R-forest	OptDL8-forest
zoo-1	25	0.031	0.008
hepatitis	34	0.034	0.142
lymph	37	0.044	0.181
audiology	54	0.034	0.350
heart-cleveland	74	0.038	1.569
primary-tumor	84	0.035	0.511
ionosphere	87	0.037	4.996
vote	108	0.029	0.233
soybean	157	0.037	1.497
australian-credit	163	0.037	6.907
breast-wisconsin	170	0.037	1.336
diabetes	192	0.035	12.719
anneal	203	0.036	6.663
vehicle	211	0.043	19.898
tic-tac-toe	239	0.035	4.764
german-credit	250	0.037	18.201
yeast	371	0.042	52.149
segment	577	0.049	1.786
splice-1	797	0.062	185.775
kr-vs-kp	799	0.056	31.202
hypothyroid	811	0.100	29.044
pendigits	1873	0.098	147.971
mushroom	2031	0.065	43.75
letter	5000	0.203	1654.676

Figure 4.6: Execution time in seconds compared between a depth 3 random forest and a depth 2 optimal forest. Cutoff set at 10 trees after the point where $\rho > 0$. $D = 1$. Accuracy is given as average on 10 runs.

Dataset	Size	DL8	R-forest	G-boosting	d = 1	d = 2	d = 3
zoo-1	25	95.26%	99.21%	98.42%	96.84%	96.84%	96.84%
hepatitis	34	77.48%	81.84%	80.58%	75.83%	78.64%	75.34%
lymph	37	74.32%	81.71%	78.11%	79.10%	74.95%	76.13%
audiology	54	91.42%	89.81%	94.44%	92.65%	92.96%	92.84%
heart-cleveland	74	72.61%	78.96%	77.21%	75.09%	75.32%	74.23%
primary-tumor	84	78.49%	78.85%	78.33%	-	57.30%	59.76%
ionosphere	87	78.94%	88.45%	85.61%	84.58%	84.81%	80.72%
vote	108	93.09%	95.08%	95.05%	94.59%	93.06%	92.69%
soybean	157	92.49%	86.15%	93.76%	89.81%	89.34%	78.41%
australian-credit	163	82.67%	85.24%	85.39%	79.78%	79.45%	79.45%
breast-wisconsin	170	94.64%	97.35%	96.24%	95.03%	94.66%	94.93%
diabetes	192	72.80%	76.30%	73.42%	-	68.14%	67.80%
anneal	203	82.41%	82.28%	86.11%	77.73%	70.49%	72.76%
vehicle	211	92.02%	94.06%	95.54%	92.31%	93.64%	93.09%
tic-tac-toe	239	72.81%	73.02%	92.78%	88.32%	80.35%	83.84%
german-credit	250	67.72%	70.51%	71.61%	-	67.33%	67.33%
yeast	371	67.67%	69.60%	71.55%	-	62.64%	58.83%
segment	577	99.38%	99.05%	99.67%	99.76%	99.51%	99.64%
splice-1	797	91.65%	93.08%	96.03%	90.33%	90.13%	90.69%
kr-vs-kp	799	93.76%	93.77%	97.37%	-	91.73%	95.97%
hypothyroid	811	97.58%	95.83%	97.58%	91.56%	89.96%	85.76%
pendigits	1873	99.13%	98.83%	99.76%	98.30%	99.49%	99.34%
mushroom	2031	99.86%	97.79%	99.90%	99.92%	99.91%	99.88%
letter	5000	98.13%	96.11%	99.13%	-	98.82%	90.71%

Figure 4.7: Comparative results for the different algorithms on depths 3 and Optimal forests on depths 1, 2 and 3. Minimum support in leaves was set to 5. Cutoff set at 10 trees after the point where $\rho > 0$. Dashes represent datasets where no solution was found. $D = 1$. Accuracy is given as average on 10 runs.

Dataset	Size	DL8	R-forest	G-boosting	d = 1	d = 2	d = 3
zoo-1	25	95.39%	99.74%	98.68%	97.24%	95.39%	95.39%
hepatitis	34	76.80%	83.50%	80.97%	81.46%	76.70%	76.80%
lymph	37	73.96%	78.11%	76.13%	78.83%	77.48%	73.96%
audiology	54	92.96%	87.84%	93.70%	92.22%	93.70%	92.96%
heart-cleveland	74	73.20%	77.03%	77.48%	76.53%	75.09%	75.41%
primary-tumor	84	78.97%	78.06%	79.44%	-	66.55%	64.05%
ionosphere	87	81.40%	89.66%	87.84%	84.73%	85.45%	83.64%
vote	108	92.60%	94.37%	93.67%	93.27%	93.91%	92.51%
soybean	157	92.90%	87.36%	92.94%	82.60%	71.84%	71.82%
australian-credit	163	81.53%	83.47%	85.41%	77.41%	80.20%	78.59%
breast-wisconsin	170	93.80%	96.69%	94.91%	94.81%	94.81%	93.80%
diabetes	192	70.92%	74.81%	74.72%	-	68.28%	68.18%
anneal	203	81.33%	82.45%	83.68%	78.82%	78.33%	69.43%
vehicle	211	92.14%	92.61%	94.03%	91.42%	92.83%	90.41%
tic-tac-toe	239	73.70%	72.70%	81.21%	89.18%	79.43%	85.01%
german-credit	250	67.77%	70.85%	71.55%	-	66.79%	67.00%
yeast	371	67.51%	70.18%	70.94%	-	54.44%	52.64%
segment	577	99.32%	99.49%	99.68%	99.68%	99.58%	99.32%
splice-1	797	91.83%	91.12%	94.76%	89.31%	89.91%	90.38%
kr-vs-kp	799	93.65%	92.92%	93.91%	-	96.18%	91.95%
hypothyroid	811	97.79%	96.23%	97.87%	83.63%	87.00%	95.92%
pendigits	1873	99.10%	98.58%	99.37%	98.58%	99.39%	99.22%
mushroom	2031	99.85%	95.72%	99.76%	99.97%	99.92%	99.90%
letter	5000	98.12%	95.98%	98.37%	-	98.85%	99.41%

Figure 4.8: Comparative results for the different algorithms on depths 3 and Optimal forests on depths 1, 2 and 3. Minimum support in leaves was set to 3. Cutoff set at 10 trees after the point where $\rho > 0$. Dashes represent datasets where no solution was found. $D = 1$. Accuracy is given as average on 10 runs.

Dataset	Size	$D = 1$	$D = 0.5$	$D = 0.2$
zoo-1	25	95.13%	95.13%	95.13%
hepatitis	34	78.54%	78.64%	78.16%
lymph	37	73.60%	73.69%	71.53%
audiology	54	92.59%	94.01%	93.33%
heart-cleveland	74	75.95%	75.09%	75.59%
primary-tumor	84	73.77%	73.51%	73.21%
ionosphere	87	86.52%	85.11%	85.30%
vote	108	93.36%	93.03%	92.35%
soybean	157	93.32%	93.00%	92.56%
australian-credit	163	80.96%	80.12%	81.06%
breast-wisconsin	170	93.37%	92.61%	92.46%
diabetes	192	68.44%	66.89%	68.58%
anneal	203	74.45%	78.60%	82.10%
vehicle	211	94.02%	93.98%	94.38%
tic-tac-toe	239	79.22%	79.74%	78.57%
german-credit	250	68.20%	68.11%	66.63%
yeast	371	58.11%	52.70%	66.86%
segment	577	99.45%	99.50%	99.34%
splice-1	797	89.72%	89.32%	88.99%
kr-vs-kp	799	95.97%	96.41%	96.07%
hypothyroid	811	94.11%	94.49%	96.46%
pendigits	1873	99.48%	99.47%	99.49%
mushroom	2031	99.95%	99.93%	99.93%
letter	5000	98.88%	98.88%	98.76%

Figure 4.9: Different values of D tested on a depth 2 optimal forest with no minimum support. Cutoff set at 10 trees after the point where $\rho > 0$. Accuracy is given as average on 10 runs.

Dataset	Size	$D = 1$	$D = 0.5$	$D = 0.23$
zoo-1	25	96.05%	96.05%	96.05%
hepatitis	34	80.87%	80.87%	80.87%
lymph	37	71.17%	70.45%	69.64%
audiology	54	91.85%	92.04%	91.91%
heart-cleveland	74	72.25%	72.03%	72.30%
ionosphere	87	84.13%	84.32%	82.61%
vote	108	91.93%	91.96%	91.77%
soybean	157	91.25%	90.89%	92.66%
australian-credit	163	80.35%	80.65%	80.61%
breast-wisconsin	170	93.24%	93.22%	93.39%
diabetes	192	69.27%	68.26%	68.73%
anneal	203	77.65%	80.10%	80.51%
vehicle	211	93.10%	93.07%	92.54%
tic-tac-toe	239	83.50%	84.03%	84.02%
german-credit	250	67.24%	66.65%	67.31%
yeast	371	55.82%	62.73%	66.44%
segment	577	99.55%	99.55%	99.55%
splice-1	797	87.66%	87.14%	90.97%
kr-vs-kp	799	96.25%	95.98%	96.50%
hypothyroid	811	93.60%	94.32%	96.98%
pendigits	1873	99.24%	99.27%	99.25%
mushroom	2031	99.94%	99.93%	99.91%

Figure 4.10: Different values of D tested on a depth 1 optimal forest with no minimum support. Cutoff set at 10 trees after the point where $\rho > 0$. Accuracy is given as average on 10 runs.

Dataset	Size	DL8	R-forest	G-boosting	OptDL8-forest
zoo-1	25	95.66%	98.82%	98.16%	95.66%
hepatitis	34	81.55%	82.52%	83.11%	81.07%
lymph	37	75.59%	77.39%	76.31%	77.48%
audiology	54	91.48%	78.52%	91.42%	92.47%
heart-cleveland	74	71.62%	78.60%	79.05%	74.95%
primary-tumor	84	77.10%	77.18%	79.40%	56.55%
ionosphere	87	85.57%	86.52%	89.36%	87.61%
vote	108	92.97%	93.09%	94.89%	92.97%
soybean	157	89.87%	85.62%	90.27%	79.37%
australian-credit	163	82.14%	82.37%	86.12%	79.37%
breast-wisconsin	170	93.61%	96.02%	94.95%	93.98%
diabetes	192	72.50%	74.37%	75.14%	68.58%
anneal	203	81.99%	81.07%	82.35%	72.22%
vehicle	211	88.28%	85.70%	93.10%	93.54%
tic-tac-toe	239	68.48%	68.73%	70.61%	77.76%
german-credit	250	70.00%	70.32%	71.31%	67.00%
yeast	371	68.24%	68.67%	69.41%	66.82%
segment	577	99.41%	98.29%	99.25%	99.66%
splice-1	797	83.32%	88.37%	94.56%	89.13%
kr-vs-kp	799	86.98%	89.43%	93.62%	94.98%
hypothyroid	811	97.72%	92.98%	97.86%	94.59%
pendigits	1873	97.85%	96.20%	98.73%	99.50%
mushroom	2031	96.93%	93.36%	97.57%	95.16%
letter	5000	96.99%	95.98%	97.46%	98.96%

Figure 4.11: Comparative results for the different algorithms on depth 2 with 0.01% noise added in. Cutoff set at 10 trees after the point where $\rho > 0$. $D = 1$

Dataset	Size	DL8	R-forest	G-boosting	OptDL8-forest
zoo-1	25	95.53%	99.74%	99.87%	95.53%
hepatitis	34	74.95%	81.26%	76.02%	74.95%
lymph	37	69.10%	79.73%	78.56%	69.19%
audiology	54	90.74%	88.02%	93.52%	91.17%
heart-cleveland	74	71.17%	79.86%	76.58%	71.58%
primary-tumor	84	78.97%	78.10%	79.48%	45.00%
ionosphere	87	79.39%	87.73%	87.01%	79.73%
vote	108	91.28%	93.21%	93.88%	91.25%
soybean	157	90.53%	86.66%	91.42%	86.98%
australian-credit	163	82.06%	83.82%	85.18%	80.41%
breast-wisconsin	170	93.00%	96.65%	94.25%	93.06%
diabetes	192	70.90%	75.14%	73.84%	69.13%
anneal	203	82.58%	81.56%	82.51%	76.06%
vehicle	211	92.14%	93.18%	94.36%	92.85%
tic-tac-toe	239	73.34%	71.97%	77.75%	82.77%
german-credit	250	67.91%	71.32%	71.52%	66.96%
yeast	371	67.68%	69.30%	71.44%	67.02%
segment	577	99.56%	99.11%	99.49%	99.56%
splice-1	797	90.96%	91.17%	94.63%	85.75%
kr-vs-kp	799	93.83%	92.07%	94.12%	95.37%
hypothyroid	811	97.51%	95.13%	97.70%	95.42%
pendigits	1873	98.97%	98.35%	99.36%	99.34%
mushroom	2031	99.65%	95.81%	99.74%	95.01%
letter	5000	98.09%	95.92%	98.28%	98.84%

Figure 4.12: Comparative results for the different algorithms on depth 3 with 0.01% noise added in. Cutoff set at 10 trees after the point where $\rho > 0$. $D = 1$

Chapter 5

Conclusion

In this thesis, we have shown how optimisation techniques can be applied to forests of decision trees. We have built two algorithms based on optimal decision trees, showing how the naive version fails to work, as well as showing when the optimal version works best. We have also analysed the remaining issues in this method, and given future paths for improvement.

As it stands, neither method is applicable in practice. The improvements in accuracy are sometimes consequent compared to the state of the art methods, however execution time remains a major issue. Some tweaks, such as those named in section 4.4, might be able to mitigate this, but this remains in the domain of the plausible.

Bibliography

[1] Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[2] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[3] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[4] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[5] Sina Aghaei, Mohammad Javad Azizi, and Phebe Vayanos. Learning optimal and fair decision trees for non-discriminative decision-making. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1418–1426, 2019.

[6] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. In *The 25th International Conference on Principles and Practice of Constraint Programming (CP2019)*, 2019.

[7] Siegfried Nijssen and Elisa Fromont. Mining optimal decision trees from itemset lattices. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 530–539. ACM, 2007.

[8] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

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

[10] Alexey Natekin and Alois Knoll. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, 7:21, 2013.

[11] Ayhan Demiriz, Kristin P Bennett, and John Shawe-Taylor. Linear programming boosting via column generation. *Machine Learning*, 46(1-3):225–254, 2002.

[12] Hiroto Saigo, Sebastian Nowozin, Tadashi Kadowaki, Taku Kudo, and Koji Tsuda. gboost: a mathematical programming approach to graph classification and regression. *Machine Learning*, 75(1):69–89, 2009.

[13] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, Joao Marques-Silva, and IS RAS. Learning optimal decision trees with sat. In *IJCAI*, pages 1362–1368, 2018.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**