

Assessing Optimal Forests of Decision Trees

Gaël Aglin

ICTEAM

UCLouvain

Louvain-la-Neuve, Belgium

gael.aglin@uclouvain.be

Siegfried Nijssen

ICTEAM

UCLouvain

Louvain-la-Neuve, Belgium

siegfried.nijssen@uclouvain.be

Pierre Schaus

ICTEAM

UCLouvain

Louvain-la-Neuve, Belgium

pierre.schaus@uclouvain.be

Abstract—The interest in algorithms for learning optimal decision trees (ODTs) has increased significantly in recent years. These algorithms use combinatorial search to find a predictive machine learning model in the form of a tree. It was shown that ODTs can obtain better predictive performance than trees found using traditional, heuristic algorithms. In many applications where accuracy is important, however, in practice machine learning developers use forests of decision trees instead of single trees. The most popular approaches for learning forests, such as Adaboost, are heuristic in nature, and it is not clear where their good performance comes from. In an attempt to study this, a number of earlier papers developed approaches that rely on the definition of a global optimization criterion for learning forests; however, unfortunately, these papers did not present algorithms for optimizing these criteria exactly, hence leaving the value of the optimization criteria unclear. In this work, we show that by using recent techniques for learning exact ODTs, we can now also optimize forests exactly. We show the optimization gap with existing approaches and evaluate the performance of optimal decision forests using different optimization criteria compared to Adaboost. This reveals that optimal decision forests can be a good approach depending on the optimization criterion and in some cases can outperform Adaboost.

Index Terms—Optimal Forests, Decision Trees, Boosting, Column generation, Optimization

I. INTRODUCTION

With the popularity of Explainable AI (XAI), a great interest has emerged into learning interpretable predictive models. One of the most popular existing interpretable models are decision trees (DTs). A decision tree model splits data recursively using successive conditions on features in order to group similar instances into leaves. Traditional algorithms for learning decision trees operate in a heuristic top-down fashion; these algorithms do not optimize a well-defined global optimization criterion.

Recent studies have however shown that on many datasets optimal depth-limited decision trees perform well [3]. These new algorithms often rely on well-known optimization techniques such as Mixed Integer Programming (MIP) [3] or Constraint Programming (CP) [25] to find a decision tree that optimizes a scoring function optimally.

However, it is well-known in the machine learning literature that compared to other machine learning models, the predictive performance of decision trees is limited. One popular way to build stronger classifiers based on DTs, is the use of *ensemble*

learning. The principle of *ensemble learning* is to combine many “weak” classifiers to build a stronger one. Ensemble learning techniques have proven for many years to be able to improve the generalization power of “weak” learners. Even though the ensemble learning concept may apply to any base classifier supporting sample weights, the most well-known use of ensemble learning is in combination with decision trees. In the remainder of this paper, we will only consider ensemble learning models based on decision trees and call the resulting models *forests* or *decision forests*.

One of the most popular techniques for ensemble learning is boosting. Boosting algorithms learn successively a set of classifiers, each having a weight; the weighted prediction of the trees is the prediction of the overall model. One of the most well-known boosting algorithms is the Adaboost algorithm [10]. In Adaboost, heuristics are iteratively used to calculate weights for training instances; for each set of weights, a heuristic decision tree learning algorithm is used to learn a decision tree that takes these weights into account. In practice, this combination of heuristics works well.

The heuristic nature of Adaboost makes it very different from many other popular approaches in machine learning, such as deep learning, in which a global optimization criterion (or loss function) is defined, and subsequently an optimization algorithm is used to find a model that optimizes this criterion.

To bridge this gap, and to get a better understanding in boosting in general, researchers have studied the definition of global optimization criteria for boosting. Accuracy by itself is not a good criterion: it could lead to very complex models that overfit training data. Hence, these studies focused on optimisation criteria that also reduce the risk of overfitting. This led to the development of new boosting techniques, of which LPBoost [8] and MDBoost [24] are the prime examples that we will consider in this paper. However, even though both techniques rely on a definition of a global optimization criterion, to optimize this criterion the authors still propose to use heuristic techniques; these techniques do not possess optimality guarantees, as the authors considered solving these criteria optimally infeasible. As a result, until now the question remains unanswered whether the perceived performance of these newer boosting algorithms derives from the global optimization criterion or the heuristic algorithm used.

In this work, we will first show that by a simple combination of the algorithm used in LPBoost and MDBoost, and recent

algorithms for finding optimal decision trees (ODTs), we can solve the boosting problems proposed in LPBoost and MDBoost till optimality. Hence, ODT techniques make it possible to evaluate the usefulness of the global optimization criteria without interference from the heuristics used in the underlying algorithms till now.

This development makes it possible to answer the following experimental questions:

- 1) what is the optimization gap between LPBoost and MDBoost using an optimal decision tree algorithm and a traditional heuristic algorithm?
- 2) does it make a difference in practice on generalization error to find an optimal ensemble instead of a heuristic one?

The answers to these questions are not obvious. Heuristic decision tree learning algorithms are often used as they are known to have good performance; it is not obvious what ODT algorithms could add. Moreover, Reyzin and Schapire stated in [21] that it is important that the classification models used in boosting are not strong; otherwise the ability to generalize towards new data is compromised. It is not clear that ODTs or Optimal Decision Forests meet this requirement. Our experiments will help to answer these questions.

This paper is organized as follows. Section II presents the state-of-the-art of boosting techniques as well as different intuitions about why Adaboost performs well. Section III presents the technical background related to learning optimal ensembles while Section IV introduces our methodology to boost ODTs. Finally, section V presents a comparison of optimal forests, heuristic approaches and Adaboost.

II. RELATED WORK

In this section, we discuss previous work related to boosting. Adaboost [10] is one of the most popular boosting algorithms. It produces a weighted ensemble of several “weak” DTs to build a strong model. The different DTs are generated successively, each new tree focusing on the errors of previous trees in the ensemble. For this, the weight of each instance is re-evaluated after adding a new tree, in such a way that the next tree focuses on misclassified instances. The resulting approach is known to work very well in practice. Forests found by Adaboost show great performance on training set as well as on test set data. The resulting model is insensitive to overfitting on many datasets [6].

Even though Adaboost works well in practice, it is not clear why this is. Many studies have been done on Adaboost to find out the reason for its success. Breiman argued that Adaboost can be viewed as minimizing a loss function: *exponential loss* [5]. Moreover, Mason et al. [15] suggested that Adaboost performs a sort of gradient descent. This perspective on Adaboost has been shared by other researchers [11], [15]. However, experiments conducted by Schapire and Freund [22] rejected the hypothesis that the success of Adaboost is only due to exponential loss minimization. Indeed, they implemented a gradient descent algorithm which minimizes

the exponential loss, but it generalizes poorly compared to Adaboost.

Schapire et al. associated the success of Adaboost to a concept called *margin* [23]. This is a confidence score of an instance given an ensemble of classifiers. While a positive margin indicates a correct classification contrary to a negative one, a higher margin indicates a higher confidence in the prediction than a lower one. To support the margin argument, they showed the cumulative distribution plot of training instance margins obtained by Adaboost given different numbers of iterations. This experiment showed that as the number of iterations increases, Adaboost tends to maximize the *margin* of training instances: this explains why Adaboost continues adding new trees even when it reaches a perfect accuracy on a training set. Breiman [5] and Rätsch et al. [20] proposed respectively the Arc-gv and Adaboost* algorithms in order to maximize the minimum margin of training instances in a more fundamental manner than Adaboost. After Breiman [4] proved that the minimum margin maximization can be formulated as a linear program, Grove and Schuurmans [13] proposed two algorithms to maximize the minimum margin: LP-Adaboost and DualLPboost. LP-Adaboost is an Adaboost adaptation in which the tree weighting part is replaced by a linear program maximizing the minimum margin. DualLPboost takes this a step further, and is the first algorithm relying on a global optimization criterion based on margins; for the optimization of this criterion, this approach was the first to propose the use of *column generation* optimisation technique in combination with a heuristic DT learning algorithm.

Despite an elegant mathematical formulation with a clear optimization criterion, these algorithms maximize the minimum margin slightly worse than Adaboost in practice. Reyzin and Schapire [21] looked at the Arc-gv algorithm to understand why it does not perform as well as Adaboost. They concluded first that the minimum margin maximization leads to the construction of too complex classifiers in the ensemble. This leads overfitting the training data. On the other hand, they found that the minimum margin maximization sacrifices the margin of the other instances. In their analysis of DualLPboost, Grove and Schuurmans [13] reached the conclusion that the hard maximisation of the minimum margin in DualLPboost could only work if data are linearly separable.

These observations led to the development of two boosting techniques based on column generation that are the basis of our work [8], [24]. Demiriz et al. [8] proposed LPBoost, in which the minimum margin is maximized while also allowing for mistakes by means of a slack variable for each instance. The minimum margin maximization objective is then regularized by the minimization of the sum of misclassifications. When running their algorithm for a limited number of iterations, it was found to outperform DualLPboost, but it did not outperform Adaboost.

To reduce even more the impact of the minimum margin maximization on the other margins, Shen and Li [24] proposed MDBoost, a new column generation approach to maximize the margin distribution. Concretely, this approach maximizes the

average of margins and minimizes their variance at the same time. The model includes a regularization parameter to decide whether the algorithm focuses more on the margin distribution maximization or on their variance minimization.

So far, within LPBoost and MDBoost heuristic DT algorithms were used; the resulting ensemble models are thus not guaranteed to be optimal given the optimization criterion proposed by the authors. Our work addresses this weakness.

IPboost [19] is a branch-and-price approach to learn boosted trees with respect to a misclassification rate optimisation criterion. Also within this technique only heuristic DT learning methods are used. Another approach in the literature is [16]. This work proposes a MIP model that for given a predefined ensemble structure, fills in the features used in the ensemble, taking misclassification rate as optimisation criterion. Contrary to our work, this approach limits the size of the ensemble and the structures of the trees in the ensemble.

The different algorithms described above all use heuristic-based DT learning algorithm as base learner. The heuristic-based DT learning algorithms split recursively data in a top-down fashion by selecting at each level the feature reducing at most an error. However, the trees learned using such a heuristic approach are often not *optimal*: for a given limit on the depth of trees, these algorithms do not optimize a criterion such as training set error. An early algorithm that showed the feasibility of finding optimal trees that optimize training set accuracy under constraints was DL8 [17]. Subsequently, Bertsimas and Dunn [3] showed experimentally that depth-limited ODTs outperform heuristic-based DTs on a significant number of datasets in terms of predictive performance. Since then, many researchers have proposed different techniques to learn ODTs in a short period of time. OCT [3] is a MIP approach. Verhaeghe et al. [25] proposed a constraint programming approach. DL8.5 [1] is a recent, more efficient version of DL8 that makes it possible to learn optimal decision trees on many datasets.

III. TECHNICAL BACKGROUND

In this section, we present the technical background on which our work relies.

A. Boosting through margin optimization problem

As mentioned previously, Adaboost performs well in practice and many studies have been performed to find the reason for its success. Among the different research axes explored, the one on which we build in this paper is the concept of *margin* maximization. Let $\{(\mathbf{x}_i, y_i)\}, i = 1, \dots, M$ be the training data, where $\mathbf{x}_i$ is the feature vector of an instance, associated to a class $y_i \in \{-1, +1\}$. Let $h(\mathbf{x}) \in \mathcal{H}$ be a function representing a base classifier that associates to each instance $\mathbf{x}$ a class $y \in \{-1, +1\}$; in our case $\mathcal{H}$ represents the space of all possible decision trees for a given dataset, possibly under constraints, such as a constraint on the depth of the trees allowed. The boosting problem consists in finding a weighted set $\mathcal{H}' = \{h_1(\mathbf{x}), \dots, h_T(\mathbf{x})\} \subseteq \mathcal{H}$ which obtains good classification accuracy on *future* data. However, given

that we do not have these instances at training time, we need to find a criterion that can be applied on training data to avoid overfitting.

There are some common ideas in how heuristic-based boosting approaches tackle this problem. Based on the prediction and the weight of each classifier in an ensemble, a prediction score is computed for each training instance. For a tree t predicting a class $h_t(\mathbf{x}) \in \{-1, +1\}$ and associated to a weight w_t , the score of a training instance $\mathbf{x}_i$ is equal to $\varrho_{ti} = y_i h_t(\mathbf{x}_i) w_t$, where y_i is the true label according to the training data. ϱ_{ti} takes values in $\{-w_t, w_t\}$. The sign of the score denotes whether the tree classifies correctly the instance or not while the value w_t denotes the confidence of the decision. For a set of T trees, let $H \in \mathbb{R}^{M \times T}$ be a matrix where the entry (i, t) of H is $H_{it} = h_t(\mathbf{x}_i)$. H_{it} is the output of the classifier t given the training instance $\mathbf{x}_i$. $H_{i:}$ is the vector of predictions for the instance $\mathbf{x}_i$ and $H_{:t}^\top$ is the vector of predictions for all of the training data given a tree t . $\mathbf{w} = (w_1, \dots, w_T)$ is the vector of weights of different trees. The prediction score for an instance $\mathbf{x}_i$ given an ensemble of trees is computed by summing its score for each tree, $\varrho_{Ti} = \sum_{t=1}^T y_i h_t(\mathbf{x}_i) w_t$. This score is the unnormalized *margin* of an instance given an ensemble of classifiers. Its vector expression is $\varrho_{Ti} = y_i H_{i:} \mathbf{w}$. The normalized *margin* ensuring that the weights of all classifiers sum to 1 is $\rho_{Ti} = y_i H_{i:} \mathbf{w} / \|\mathbf{w}\|_1$. The *margin* score of an ensemble is a kind of balance between the incorrect decisions and the correct ones. Each decision having a weight, the final decision will be correct if trees predicting correctly have a higher weight than the other and vice versa. Concretely, an instance is predicted as 1 when its margin is positive and -1 when it is negative. As for a single tree, the final score expresses how confident the model is about a prediction. A model with higher margins on data is more confident than one with lower margins. Schapire et al. [23] stated that Adaboost performs a kind of margin maximization, arguing that it continues to increase the test score by adding new trees even when the training score reaches 100% of accuracy.

Many studies [12], [23] concluded that the success of Adaboost is due to an optimization of the margin distribution on the training data without concretely stating what a good distribution is. In the following, we present LPBoost and MDBoost, two approaches optimizing the margins in training data but using different criteria.

B. Existing techniques to optimize margins

The key idea underlying the boosting techniques on which we build, is to formalize the boosting problem as a mathematical programming problem in which we associate a Boolean variable to each possible decision tree that is allowed to be put in the ensemble; the optimization problem is to assign truth values to these Boolean variables.

Given the high number of possible trees, it is not feasible to materialize the optimization problem with all possible variables. However, given that most variables will receive the value false, this may also not be necessary; intuitively, if we

would be able to instantiate the optimization problem only for those trees that are part of the optimal solution, and would solve the resulting optimization problem, the resulting solution would still be optimal; it is not possible to add a variable and obtain a better solution.

Column generation [9] is a well-known approach to efficiently solve large-scale optimization problems based on this idea. The column generation process starts the solving with one variable and will lazily and iteratively add the most promising variables capable of improving the objective value, until no variable can be found whose addition to the problem still leads to better solutions. It splits the problem into 2 problems: *the primal* problem allowing solving the global problem using only selected variables and *the pricing* problem allowing generating the new variable with negative reduced cost¹ to add to the primal in order to improve its objective value.

DualLPboost [13] was the first approach to show that it is possible to solve boosting problems using a column generation approach. In it at each iteration the *dual* of the optimization problem over selected variables is solved. The solution of the dual provides new weights for the samples; these are used to find new trees. The primal is used to compute the weights of the different trees. The process is repeated for a fixed number of iterations.

Other approaches aim to correct the earlier discussed weaknesses of DualLPboost.

a) *LPBoost*: While LPBoost essentially chooses to maximize the lowest margin value similar to DualLPboost, it uses a soft margin maximization of the minimum margin instead of a hard margin maximization:

$$\begin{aligned} \max_{\mathbf{w}, \xi, \rho} \quad & \rho - D \sum_{i=1}^M \xi_i \\ \text{s.t.} \quad & y_i H_i; \mathbf{w} + \xi_i \geq \rho, \quad i = 1, \dots, M \\ & \sum_{t=1}^T w_t = 1 \\ & \xi_i \geq 0, \quad i = 1, \dots, M \\ & w_t \geq 0, \quad t = 1, \dots, T \end{aligned} \quad (1)$$

In this model ρ represents the margin of the solution. The model includes a slack variable ξ_i for each instance x_i . This value represents a tolerance to misclassification for each instance; if $\xi_i > 0$, an instance is located on the wrong side of the chosen margin. The sum of the slack variables is minimized in the objective function, while maximizing the margin. This minimization is regularized by a parameter D in order to control the trade-off between the size of the margin and the extent to which instances are wrongly positioned w.r.t. this margin. The good value of D is determined by hyperparameter tuning.

Note that in this model the sum over variables t is over *all possible trees* that can be constructed for a given dataset.

¹It also corresponds to discover one violated constraint of the dual

LPBoost addresses this using a column generation process. The corresponding dual model allows finding at each iteration of the column generation process, the new sample weights $\mathbf{u} = \{u_1, \dots, u_M\}$ to learn a new tree. The formulation of the dual is then:

$$\begin{aligned} \min_{r, \mathbf{u}} \quad & r \\ \text{s.t.} \quad & \sum_{i=1}^M u_i y_i H_{it} \leq r, \quad \forall t = 1, \dots, T \\ & \sum_{i=1}^M u_i = 1 \\ & 0 \leq u_i \leq D, \quad i = 1, \dots, M \end{aligned} \quad (2)$$

A traditional decision tree learning algorithm is used in each iteration to find a DT that takes into account the weights of the instances. However, given that these algorithms are heuristic, LPBoost cannot decide at which moment an optimal ensemble has indeed been found.

b) *MDBoost*: Reyzin and Schapire [21] stated that the minimum margin maximization is performed at the expense of the other margins. Therefore MDBoost maximizes the unnormalized average margin and simultaneously minimizes the variance of the margin distribution under the assumption that the margin distribution follows a Gaussian distribution [24]. The primal objective function of MDBoost for T trees is $\max_{\mathbf{w}} \bar{\rho} - \frac{1}{2} \sigma^2$, s.t. $\mathbf{w} \geq \mathbf{0}, \mathbf{1}^\top \mathbf{w} = D$, where $\bar{\rho}$ is the mean of the unnormalized margins over the training data, σ^2 is the variance and D is a regularization parameter which indicates whether the model mostly concentrates on the average margin maximization or the variance minimization. After some transformations, the primal can be rewritten as follows:

$$\begin{aligned} \max_{\mathbf{w}, \boldsymbol{\rho}} \quad & \mathbf{1}^\top \boldsymbol{\rho} - \frac{1}{2} \boldsymbol{\rho}^\top A \boldsymbol{\rho} \\ \text{s.t.} \quad & \boldsymbol{\rho}_i = y_i H_i; \mathbf{w}, \quad \forall i = 1, \dots, M \\ & \mathbf{1}^\top \mathbf{w} = D \\ & \mathbf{w} \geq \mathbf{0} \end{aligned} \quad (3)$$

A is a symmetric matrix with all entries on the main diagonal to 1, and the rest $\frac{-1}{M-1}$. This matrix allows the variance calculation and is positive semidefinite. The model is hence a convex quadratic problem in $\boldsymbol{\rho}$. The corresponding dual model to reweigh the training samples is then:

$$\begin{aligned} \min_{r, \mathbf{u}} \quad & r + \frac{1}{2D} (\mathbf{u} - \mathbf{1})^\top A^{-1} (\mathbf{u} - \mathbf{1}) \\ \text{s.t.} \quad & \sum_{i=1}^M u_i y_i H_{it} \leq r, \quad \forall t = 1, \dots, T \end{aligned} \quad (4)$$

Both LPBoost and MDBoost require solving a same pricing problem. Indeed one can see in their dual formulation (2) and (4) that identifying a violated constraint corresponds to finding a decision tree that minimizes the weight (given by the dual variables considered as fixed) of correctly classified

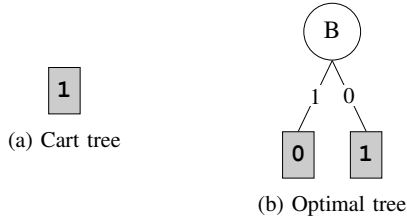


Fig. 1: Tree found for a max depth = 1, given the database of Table I and the sample weight $\mathbf{u}$.

instances minus the weights of incorrectly classified ones. The column generation alternates the two steps of 1) solving the primal problem and collecting the dual values and 2) solving the pricing problem which amounts of finding an optimal weighted decision tree that is then added to the set of trees considered by the primal.

IV. LEARNING OPTIMAL FORESTS

This section describes our approach to learn provably optimal decision forests.

A. Problems in the existing approaches

The different approaches we described in the previous sections were all proposed to achieve better performances than Adaboost by optimizing a global criterion. Crucially, although the column generation process used by these approaches allows solving a problem until optimality, two elements in the solving process annihilate the optimality of solutions found.

a) Heuristic-based learners: The algorithms used by LPBoost and MDBoost for the pricing problem at each iteration of the column generation process are heuristic-based, that is, the algorithm does not provide any guarantee that it finds a tree to improve the objective, even when one exists. Consider the example dataset in Table I and a sample weight distribution $\mathbf{u} = (0.05, 0.06, 0.33, 0.02, 0.09, 0.02, 0.22, 0.04, 0.02, 0.08, 0.07)$. The tree found by the Cart implementation of Scikit-learn for a maximum depth equal to 1 is represented in Figure 1a. It always predicts the class 1 and thus misclassifies 5 instances. The tree minimizing the misclassification rate for these weights the most is represented in Figure 1b and misclassifies 4 instances. Therefore, using a heuristic-based tree in a column generation process does not guarantee optimality. In this case, if the next tree to be added to the ensemble must have at least the quality of the optimal tree, the Cart algorithm cannot find it and will stop the process without reaching optimality. To solve this problem, we have to use an algorithm to learn an optimal decision tree and taking into account the sample weights.

b) The number of iterations: In the different approaches, a maximum number of iterations is set after which the column generation process is stopped and the result at this stage is used. At that moment we have no guarantee that the result found is optimal. To deal with this problem, we need to let the algorithm run until no tree can be added, i.e. no tree was found violating a dual constraint.

TABLE I: Example database

A	B	C	class
0	1	1	0
1	0	1	1
0	0	1	1
0	1	0	0
1	0	0	1
0	0	0	0
0	0	1	0
1	1	0	1
0	0	0	1
0	0	1	0
0	0	0	1

B. Optimal weighted decision trees (OWDT)

We build on the recent techniques for finding ODTs. However, to the best of our knowledge, there does not exist any efficient algorithm to learn ODTs in which instances are weighted. Therefore we adapted the DL8.5 [1] algorithm and its open-source implementation PyDL8.5 [2] as it is the most efficient up-to-date algorithm to learn ODTs. DL8.5 is a dynamic programming approach integrating a branch-and-bound mechanism to improve the search. PyDL8.5 is a scikit-learn compatible wrapper for DL8.5 and it provides an easy interface to define user-specific error function. The only important thing to handle in order to integrate the sample weights in DL8.5 is the way the score is computed in each leaf. Basically, the formula used in DL8.5 to compute the error in a leaf covering a set ℓ of instances is:

$$error(\ell) = |\ell| - \max_{c \in \mathcal{C}} |\{i; i \in \ell \wedge y_i = c\}| \quad (5)$$

To handle sample weights, the error function must integrate the weight of each instance falling in a leaf. The new error is the difference between the sum of weights of instances in a leaf and the maximum sum of weights of instances grouped by class.

$$error(\ell) = \sum_{i \in \ell} w_i - \max_{c \in \mathcal{C}} \sum_{i \in \ell \wedge y_i = c} w_i \quad (6)$$

The error in each leaf of DL8.5 is computed based on counting. This is efficiently implemented in PyDL8.5 using bit-wise operations and a data structure called the *Reversible Sparse bitset* [7]. This needed to be adapted in our case to iterate over the true bits of the bit-sets and collect the weight of the corresponding instance. Instead of checking each instance, we implement this efficiently using bit-wise operation to retrieve the index of the rightmost non-zero bit. This operation coupled to *shift* operation allow to iterate only over the covered instances.

C. Our approach to learn optimal forests

We summarize our generic algorithm for learning optimal decision forests, *Optiboost*, in Algorithm 1. This algorithm can be used for various optimization criteria, such as the ones proposed in LPBoost and MDBoost. We call it *OptiLPBoost* or *OptiMDBoost* when it is used respectively with the LPBoost or MDBoost model. The distinguishing feature of *Optiboost* is that we use an optimal algorithm for finding the new trees to add in the column generation process that respects the

customized sample weights, while we continue the search till no new tree can be found.

Algorithm 1: Optiboost($\{(\mathbf{x}_i, y_i)\}, \text{depth}, D$)

```

1  $\mathbf{w}, \mathcal{H}', r \leftarrow \mathbf{0}, \{\}, -\infty$ 
2  $u_i \leftarrow 1, i = 1, \dots, M$ 
3  $h \leftarrow \text{OWDT}(\{(\mathbf{x}_i, y_i)\}, \text{depth}, \mathbf{u})$ 
4 do
5    $\mathcal{H}' \leftarrow \mathcal{H}' \cup h$ 
6   solve the primal (1) for LPBoost and (3) for
     MDBoost
7   retrieve and update the dual variables  $r$  and  $\mathbf{u}$ 
8    $h \leftarrow \text{OWDT}(\{(\mathbf{x}_i, y_i)\}, \text{depth}, \mathbf{u})$ 
9 while  $\sum_{i=1}^M u_i y_i h_{\mathbf{x}_i} > r$ ;
10 return  $\mathcal{H}', \mathbf{w}$ 
```

A second distinguishing feature of the algorithm is that we do not fix the size of the ensemble in advance. Instead, we repeat the column generation process until the OWDT learning algorithm concludes that no tree can be added to the ensemble that violates a constraint of the dual. At that moment, the optimality of the ensemble has been proven and the weight of each decision tree is returned.

We start the process with an OWDT with all equal sample weights $\mathbf{u}$ set to 1 (line 2) which means that the first tree added to the ensemble optimizes its accuracy. The primal is then solved to update the variables r and the vector $\mathbf{u}$ (a by-product provided by the solver). The next iteration restarts with the new weights. If the new tree added does not violate any dual constraint, the trees and their weights are returned.

V. EXPERIMENTS

In this section, we answer the two questions:

- Q1** what is the impact on the objective value when embedding an ODT algorithm rather than a greedy one such as Cart for solving the pricing problem in LPBoost and MDBoost?
- Q2** does it make a difference in practice on the generalization error to use an ODT learning algorithm?

To solve the linear and quadratic models of LPBoost and MDBoost, we use Gurobi [14] Optimizer 9.1.0 as solver. For the different base learners, we used the implementation of Cart available in Scikit-learn [18] as heuristic DT algorithm. For the optimal DT algorithm, we used our customized DL8.5 [1] algorithm. Experiments were run on a Red Hat 4.8.5-16 server with an Intel Xeon E5-2640 CPU and 128GB of memory.

To obtain the results in this section, we performed a 5-fold cross validation. As the different models use a regularization parameter, for each fold, we used grid search coupled to a 4-fold cross validation as hyperparameter tuning technique to find the best parameter. The different regularization parameters tested for LPBoost and MDBoost are respectively $\{0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1\}$ and $\{0.5, 1, 3, 6, 12, 25, 50, 100\}$. The complexity of MDBoost increases as the number of instances increases. Indeed, the size of matrix

A depends on the number M of instances. Considering the number of iterations and the fact that the model must be solved at each iteration and also considering that the complexity of the OWDT algorithm increases depending on the size of the dataset, we performed the experiments on 10 well-known UCI² datasets with reasonable size. The datasets were already preprocessed and available on the CP4IM³ project website. The different base learners used for the experiments have a maximum depth equal to 3.

To answer question **Q1**, we present in Table II the optimization gap between ODT (OptiLPBoost, OptiMDBoost) and heuristic (LPBoost and MDBoost) based algorithms. The first columns of the table describe the different datasets. $nFeat$ stands for the number of features and $nTrans$ represents the number of instances in the dataset. To determine the best value for the hyperparameter D , we performed hyperparameter tuning. For bold columns hyperparameter tuning has been performed for the indicated approach. The same value of the hyperparameter is then used for the other approach, shown in the adjacent column. We notice clearly in the table II that the optimal value reached by our approach is better than classic LPBoost and MDBoost. The optimization value reached is on average improved by an order of 2.

We can also notice in Figure 2 and Figure 3 how the objective value evolves as the iterations increase in the boosting process on different folds. We show these plots for the dataset *tic-tac-toe* as it has the highest number of instances in our datasets and the resulting graphs are representative for the results on other datasets as well. For space reasons, we only show the first three folds of our 5 folds. The same regularization parameter is used for the different approaches. On these plots, the parameters used have been tuned for LPBoost and MDBoost. Notice that the number of iterations reached by our optimal approach is always greater than the classic approaches. This indicates that the heuristic approaches stop without proving the inexistence of trees. Note that already from the first iterations, the objective value of our approach is better than the classic LPBoost and MDBoost. Moreover, we can notice that, similar to Adaboost, the optimization criterion of LPBoost and MDBoost does not focus on accuracy. The number of trees increases even when the accuracy on the training set is 100%, trying to increase the margin of instances.

To answer question **Q2**, we show in Table III a performance comparison of classic LPBoost and MDBoost against our optimal forest approaches, where the regularization value is tuned for each approach. For each approach, we present the number of trees found, the accuracy on training set and test set. The best accuracy on the test set is in bold. The column Adaboost will be discussed later. Not surprisingly, and as explained above, the number of trees found with OptiLPBoost and OptiMDBoost are higher than LPBoost and MDBoost. However, this is not always the case because this number also depends on the value of the regularization parameter.

²<https://archive.ics.uci.edu/ml/datasets.php>

³<https://dtai.cs.kuleuven.be/CP4IM/datasets/>

TABLE II: Comparison of the objective reached when using the same hyperparameters

Dataset	nFeat	nTrans	LPBoost	OptiLPBoost	OptiLPBoost	LPBoost	MDBoost	OptiMDBoost	OptiMDBoost	MDBoost
anneal	93	812	0.0	0.0	0.0	0.0	44.57	82.15	165.43	103.4
audiology	148	216	0.4	0.62	0.62	0.4	103.57	135.94	142.73	118.32
heart-cleveland	95	296	0.11	0.34	0.34	0.11	89.93	122.06	133.29	104.31
hepatitis	68	137	0.29	0.58	0.58	0.29	38.12	69.07	71.11	42.01
lymph	68	148	0.28	0.55	0.55	0.28	46.28	77.57	77.08	47.54
primary-tumor	31	336	0.0	0.0	0.0	0.0	59.08	79.75	90.1	67.37
soybean	50	630	0.01	0.13	0.13	0.01	104.06	189.68	274.37	171.06
tic-tac-toe	27	958	0.11	0.28	0.28	0.1	105.39	196.59	190.45	104.1
vote	48	435	0.23	0.46	0.46	0.23	154.89	223.69	239.1	179.92
zoo-1	36	101	1.0	1.0	1.0	1.0	80.8	80.8	80.8	80.8

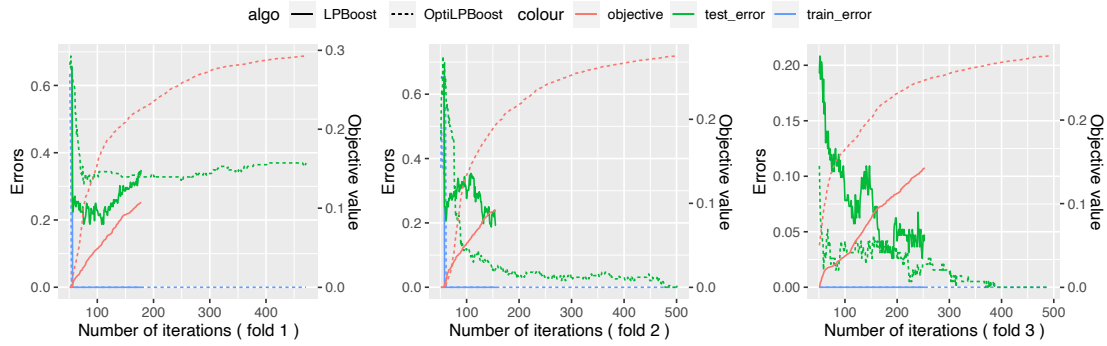
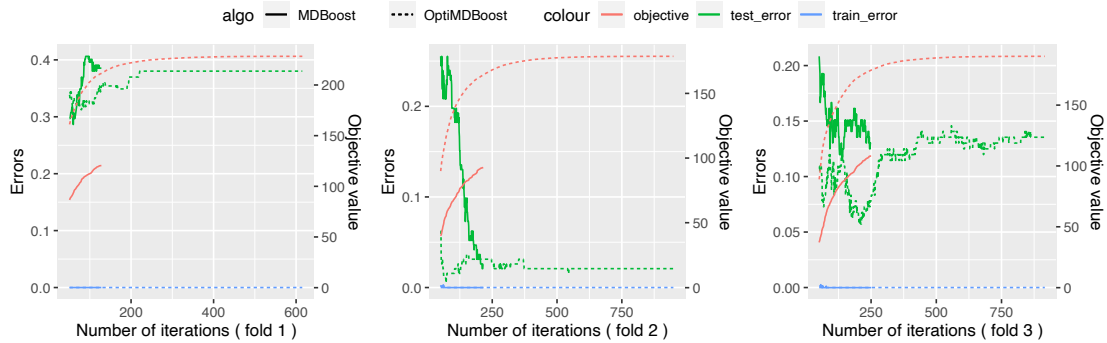
Fig. 2: Evolution of objective value with LPBoost on *tic-tac-toe* datasetFig. 3: Evolution of objective value with MDBoost on *tic-tac-toe* dataset

TABLE III: Comparison of generalization performance

Dataset	OptiLPBoost			LPBoost			Adaboost	OptiMDBoost			MDBoost			Adaboost
	n_trees	train_acc	test_acc	n_trees	train_acc	test_acc		n_trees	train_acc	test_acc	n_trees	train_acc	test_acc	
anneal	14.4	61.43	61.43	2.0	27.15	27.22	88.79	205.0	92.98	88.92	75.8	91.62	87.93	88.79
audiology	31.6	100.0	95.82	21.2	100.0	93.96	93.53	43.2	99.77	93.5	25.2	99.08	94.89	93.53
heart-cleveland	119.2	100.0	78.02	55.8	100.0	78.36	<u>79.39</u>	89.4	96.03	78.36	23.2	93.41	81.41	79.39
hepatitis	55.2	100.0	82.49	29.2	100.0	80.32	82.46	188.6	99.63	83.23	30.2	99.45	81.83	82.46
lymph	64.2	100.0	89.15	31.6	100.0	85.82	<u>89.79</u>	114.2	99.66	89.17	21.8	97.47	85.13	<u>89.79</u>
primary-tumor	29.0	88.02	71.46	29.4	78.12	69.33	<u>77.37</u>	110.2	92.04	82.74	27.8	90.48	82.14	77.37
soybean	58.6	98.97	93.81	26.4	99.17	94.13	<u>95.71</u>	129.4	98.13	95.71	53.4	97.86	96.03	95.71
tic-tac-toe	153.2	100.0	89.47	98.8	100.0	82.89	84.99	875.8	100.0	87.49	204.0	100.0	87.28	84.99
vote	58.4	100.0	94.25	28.8	100.0	94.25	<u>95.63</u>	163.6	99.48	96.09	40.6	98.85	95.63	95.63
zoo-1	1.0	100.0	97.0	1.0	100.0	97.0	<u>100.0</u>	1.0	100.0	97.0	1.0	100.0	97.0	<u>100.0</u>

According to the authors of LPBoost and MDBoost, increasing the margin of instances should also reduce the generalization error of the model on unseen instances. The results in Table III show that this is not always the case. Although the generalization error of LPBoost and MDBoost decreases on most datasets with OptiLPBoost and OptiMDBoost, there are still some datasets on which boosting ODTs perform worse. This is confirmed by the Figures 2 and 3, where the test error obtained by OptiLPBoost and OptiMDBoost on some folds is higher than errors obtained by classic approaches, although the same regularization values are used and the objective reached by the optimal approaches are higher. In conclusion, there is a difference between boosting a traditional heuristic algorithm compared to ODTs. Concretely, boosting ODTs can increase performance, but not on all datasets. The result depends also on the optimization criterion. Notice that the results of MDBoost are better than LPBoost. This implies that it is more interesting to maximize all the margins than maximizing the lowest one.

Due to the fact that LPBoost and MDBoost are not optimal, it was not possible to assess their optimization criterion against Adaboost. In Table III, we also compare the different approaches to Adaboost with 100 trees. The accuracy found on test sets are put in the column *Adaboost*. We compare the results according to whether it is LPBoost or MDBoost. In each case, we compare the classic approach, the optimal approach, as well as Adaboost; the best one is underlined. Notice that the performance of LPBoost models is lower than Adaboost even when the optimal forest approach managed to outperform Adaboost on 3 instances. On the other hand, concerning the MDBoost model, the optimal forest approach outperforms the classic approach as well as Adaboost on many datasets. We can conclude that it is a good approach to infer optimal forests by boosting ODTs. However, the performance is mainly ensured by the optimization criterion. A good optimization criterion will result in a good generalization.

VI. CONCLUSIONS

In this paper, we explore the question of learning optimal forests. We present existing techniques to boost DTs and the intuition on which they rely. We demonstrated that the results found by these techniques are not optimal. We then build on recent works on learning ODTs to propose an ODT algorithm supporting sample weights. We build on this OWDT algorithm to propose a way to find optimal forests. For the first time, this allowed us to infer optimal decision forests given a global optimisation criterion. We then performed a number of experiments to (1) show that there is a gap between optimal forests and existing approaches. (2) We compare optimal forests to existing approaches and found that optimal forests increase the generalization character of these models. (3) We found that in addition to optimality, the optimization criterion is important to learn ensembles with a good generalization character. Although we noticed that maximizing all margins is preferable over maximizing the lowest one and can outperform Adaboost, it appears that the existing criteria are not yet very

satisfactory. We hope this work could launch the study of other objective functions. Our source code supports the use of other optimization criteria and is available through PyDL8.5 library at <https://github.com/aia-uclouvain/pydl8.5>.

REFERENCES

- [1] G. Aglin, S. Nijssen, and P. Schaus, "Learning optimal decision trees using caching branch-and-bound search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 3146–3153.
- [2] —, "Pydl8.5: a library for learning optimal decision trees," in *International Joint Conference on Artificial Intelligence*, 2020.
- [3] D. Bertsimas and J. Dunn, "Optimal classification trees," *Machine Learning*, vol. 106, no. 7, pp. 1039–1082, 2017.
- [4] L. Breiman, "Arcing the edge," Technical Report 486, Statistics Department, University of California at ..., Tech. Rep., 1997.
- [5] —, "Prediction games and arcing classifiers," *Technical Report 504, Statistics Department, University of California at Berkeley*, 1997.
- [6] L. Breiman *et al.*, "Arcing classifier (with discussion and a rejoinder by the author)," *The annals of statistics*, vol. 26, no. 3, pp. 801–849, 1998.
- [7] J. Demeulenaere, R. Hartert, C. Lecoutre *et al.*, "Compact-table: efficiently filtering table constraints with reversible sparse bit-sets," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2016, pp. 207–223.
- [8] A. Demiriz, K. P. Bennett, and J. Shawe-Taylor, "Linear programming boosting via column generation," *Machine Learning*, vol. 46, no. 1-3, pp. 225–254, 2002.
- [9] O. Du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen, "Stabilized column generation," *Discrete Mathematics*, vol. 194, no. 1-3, pp. 229–237, 1999.
- [10] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.
- [11] J. Friedman, T. Hastie, R. Tibshirani *et al.*, "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)," *The annals of statistics*, vol. 28, no. 2, pp. 337–407, 2000.
- [12] W. Gao and Z.-H. Zhou, "On the doubt about margin explanation of boosting," *Artificial Intelligence*, vol. 203, pp. 1–18, 2013.
- [13] A. J. Grove and D. Schuurmans, "Boosting in the limit: Maximizing the margin of learned ensembles," in *AAAI/IAAI*, 1998, pp. 692–699.
- [14] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2021. [Online]. Available: <http://www.gurobi.com>
- [15] L. Mason, J. Baxter, P. L. Bartlett, M. Frean *et al.*, "Functional gradient techniques for combining hypotheses," *Advances in Neural Information Processing Systems*, pp. 221–246, 1999.
- [16] V. V. Mišić, "Optimization of tree ensembles," *Operations Research*, vol. 68, no. 5, pp. 1605–1624, 2020.
- [17] S. Nijssen and E. Fromont, "Mining optimal decision trees from itemset lattices," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007, pp. 530–539.
- [18] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [19] M. Pfetsch and S. Pokutta, "Ipboost—non-convex boosting via integer programming," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7663–7672.
- [20] G. Rätsch and M. K. Warmuth, "Efficient margin maximizing with boosting," *Journal of Machine Learning Research*, vol. 6, no. Dec, pp. 2131–2152, 2005.
- [21] L. Reyzin and R. E. Schapire, "How boosting the margin can also boost classifier complexity," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 753–760.
- [22] R. E. Schapire and Y. Freund, "Boosting: Foundations and algorithms," *Kybernetes*, 2013.
- [23] R. E. Schapire, Y. Freund, P. Bartlett, W. S. Lee *et al.*, "Boosting the margin: A new explanation for the effectiveness of voting methods," *The annals of statistics*, vol. 26, no. 5, pp. 1651–1686, 1998.
- [24] C. Shen and H. Li, "Boosting through optimization of margin distributions," *IEEE Transactions on Neural Networks*, vol. 21, no. 4, pp. 659–666, 2010.
- [25] H. Verhaeghe, S. Nijssen, G. Pesant, C.-G. Quimper, and P. Schaus, "Learning optimal decision trees using constraint programming," in *CP2019*, 2019.