Generalized and Scalable Optimal Sparse Decision Trees

Jimmy Lin [*1] **Chudi Zhong** [*2] **Diane Hu** [2] **Cynthia Rudin** [2] **Margo Seltzer** [1]

Abstract

Decision tree optimization is notoriously difficult from a computational perspective but essential for the field of interpretable machine learning. Despite efforts over the past 40 years, only recently have optimization breakthroughs been made that have allowed practical algorithms to find *optimal* decision trees. These new techniques have the potential to trigger a paradigm shift where it is possible to construct sparse decision trees to efficiently optimize a variety of objective functions without relying on greedy splitting and pruning heuristics that often lead to suboptimal solutions. The contribution in this work is to provide a general framework for decision tree optimization that addresses the two significant open problems in the area: treatment of imbalanced data and fully optimizing over continuous variables. We present techniques that produce optimal decision trees over a variety of objectives including F-score, AUC, and partial area under the ROC convex hull. We also introduce a scalable algorithm that produces provably optimal results in the presence of continuous variables and speeds up decision tree construction by several orders of magnitude relative to the state-of-the art.

1. Introduction

Decision tree learning has served as a foundation for interpretable artificial intelligence and machine learning for over half a century (Morgan & Sonquist, 1963; Payne & Meisel, 1977; Loh, 2014). The major approach since the 1980's has been decision tree induction, where heuristic splitting and pruning procedures grow a tree from the top down and prune it back afterwards (Quinlan, 1993; Breiman et al., 1984). The problem with these methods is that they tend to produce suboptimal trees with no way of knowing how suboptimal the solution is. This leaves a gap between the performance that a decision tree *might* obtain and the performance that one actually attains, with no way to check on (or remedy) the size of this gap–and sometimes, the gap can be large.

Full decision tree optimization is NP-hard, with no polynomial-time approximation (Laurent & Rivest, 1976), leading to challenges in proving optimality or bounding the optimality gap in a reasonable amount of time, even for small datasets. It is possible to create assumptions that reduce hard decision tree optimization to cases where greedy algorithms suffice, such as independence between features (Klivans & Servedio, 2006), but these assumptions do not hold in reality. If the data can be perfectly separated with zero error, SAT solvers can be used to find optimal decision trees rapidly (Narodytska et al., 2018); however, real data is generally not separable, leaving us with no choice other than to actually solve the problem.

Decision tree optimization is amenable to branch-and-bound methods, implemented either via generic mathematical programming solvers or by customized algorithms. Solvers have been used from the 1990's (Bennett, 1992; Bennett & Blue, 1996) to the present (Verwer & Zhang, 2019; Blanquero et al., 2018; Nijssen et al., 2020; Bertsimas & Dunn, 2017; Rudin & Ertekin, 2018; Menickelly et al., 2018; Vilas Boas et al., 2019), but these generic solvers tend to be slow. A common way to speed them up is to make approximations in preprocessing to reduce the size of the search space. For instance, "bucketization" preprocessing is used in both generic solvers (Verwer & Zhang, 2019) and customized algorithms (Nijssen et al., 2020) to handle continuous variables. Continuous variables pose challenges to optimality; even one continuous variable increases the number of possible splits by the number of possible values of that variable in the entire database, and each additional split leads to an exponential increase in the size of the optimization problem. Bucketization is tempting and seems innocuous, but we prove in Section 3 that it sacrifices optimality.

Dynamic programming methods have been used for various decision tree optimization problems since as far back as the early 1970's (Garey, 1972; Meisel & Michalopou-

[*]Equal contribution [1]University of British Columbia, Vancouver, Canada [2]Duke University, Durham, North Carolina, USA. Correspondence to: Cynthia Rudin <cynthia@cs.duke.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.

los, 1973). Of the more recent attempts at this challenging problem, Garofalakis et al. (2003) use a dynamic programming method for finding an optimal subtree within a predefined larger decision tree grown using standard greedy induction. Their trees inherit suboptimality from the induction procedure used to create the larger tree. The DL8 algorithm (Nijssen & Fromont, 2007) performs dynamic programming on the space of decision trees. However, without mechanisms to reduce the size of the space and to reduce computation, the method cannot be practical. A more practical extension is the DL8.5 method (Nijssen et al., 2020), which uses a hierarchical upper bound theorem to reduce the size of the search space. However, it also uses bucketization preprocessing, which sacrifices optimality; without this preprocessing or other mechanisms to reduce computation, the method suffers in computational speed.

The CORELS algorithm (Angelino et al., 2017; 2018; Larus-Stone, 2017), which is an associative classification method rather than an optimal decision tree method, breaks away from the previous literature in that it is a custom branch-and-bound method with custom bounding theorems, its own bit-vector library, specialized data structures, and an implementation that leverages computational reuse. CORELS is able to solve problems within a minute that, using any other prior approach, might have taken weeks, or even months or years. Hu et al. (Hu et al., 2019) adapted the CORELS philosophy to produce an Optimal Sparse Decision Tree (OSDT) algorithm that leverages some of CORELS' libraries and its computational reuse paradigm, as well as many of its theorems, which dramatically reduce the size of the search space. However, OSDT solves an exponentially harder problem than that of CORELS' rule list optimization, producing scalability challenges, as we might expect.

This work addresses two fundamental limitations in existing work: unsatisfying results for imbalanced data and and scalability challenges when trying to fully optimize over continuous variables. Thus, the first contribution of this work is *to massively generalize sparse decision tree optimization to handle a wide variety of objective functions, including weighted accuracy (including multi-class), balanced accuracy, F-score, AUC and partial area under the ROC convex hull.* Both CORELS and OSDT were designed to maximize accuracy, regularized by sparsity, and neither were designed to handle other objectives. CORELS has been generalized (Aïvodji et al., 2019; Chen & Rudin, 2018) to handle some constraints, but not to the wide variety of different objectives one might want to handle in practice. Generalization to some objectives is straightforward (e.g., weighted accuracy) but non-trivial in cases of optimizing rank statistics (e.g., AUC), which typically require quadratic computation in the number of observations in the dataset. However, for *sparse* decision trees, this time is much less than quadratic, because all observations within a leaf of a tree are tied in score, and there are a sparse number of leaves in the tree. Taking advantage of this permits us to rapidly calculate rank statistics and thus optimize over them. The second contribution is *to present a new representation of the dynamic programming search space that exposes a high degree of computational reuse when modelling continuous features*. The new search space representation provides a useful solution to a problem identified in the CORELS paper, which is how to use "similar support" bounds in practice. A similar support bound states that if two features in the dataset are similar, but not identical, to each other, then bounds obtained using the first feature for a split in a tree can be leveraged to obtain bounds for the same tree, were the second feature to replace the first feature. However, if the algorithm checks the similar support bound too frequently, the bound slows the algorithm down, despite reducing the search space. Our method uses hash trees that represent similar trees using shared subtrees, which naturally accelerates the evaluation of similar trees. The implementation, coupled with a new type of incremental similar support bound, is efficient enough to handle a few continuous features by creating dummy variables for all unique split points along a feature. This permits us to obtain smaller optimality gaps and certificates of optimality for mixed binary and continuous data when optimizing additive loss functions several orders of magnitude more quickly than any other method that currently exists.

Our algorithm is called Generalized and Scalable Optimal Sparse Decision Trees (GOSDT, pronounced "ghost"). A chart detailing a qualitative comparison of GOSDT to previous decision tree approaches is in Appendix A.

2. Notation and Objectives

We denote the training dataset as $\{(x_i, y_i)\}_{i=1}^N$, where $x_i \in \{0,1\}^M$ are binary features. Our notation uses $y_i \in \{0,1\}$, though our code is implemented for multiclass classification as well. For each real-valued feature, we create a split point at the mean value between every ordered pair of unique values present in the training data. Following notation of Hu et al. (2019), we represent a tree as a set of leaves; this is important because it allows us *not* to store the splits of the tree, only the conditions leading to each leaf. A leaf set $d = (l_1, l_2, ..., l_{H_d})$ contains H_d distinct leaves, where l_i is the classification rule of the leaf i, that is, the set of conditions along the branches that lead to the leaf, and $\hat{y}_i^{\text{leaf}}$ is the label prediction for all data in leaf i. For a tree d, we define the objective function as a combination of the loss and a penalty on the number of leaves, with regularization parameter λ:

$$R(d, \mathbf{x}, \mathbf{y}) = \ell(d, \mathbf{x}, \mathbf{y}) + \lambda H_d. \quad (1)$$

Let us first consider *monotonic losses* $\ell(d, \mathbf{x}, \mathbf{y})$, which are monotonically increasing in the number of false positives (FP) and the number of false negatives (FN), and thus can be expressed alternatively as $\tilde{l}(FP, FN)$. We will specifically consider the following objectives in our implementation. (These are negated to become losses.)

- Accuracy = $1 - \frac{FP+FN}{N}$: fraction of correct predictions.
- Balanced accuracy = $1 - \frac{1}{2}(\frac{FN}{N^+} + \frac{FP}{N^-})$: the average of true positive rate and true negative rate. Let N^+ be the number of positive samples in the training set and N^- be the number of negatives.
- Weighted accuracy = $1 - \frac{FP+\omega FN}{\omega N^+ + N^-}$ for a predetermined threshold ω: the cost-sensitive accuracy that penalizes more on predicting positive samples as negative.
- F-score = $1 - \frac{FP+FN}{2N^+ + FP - FN}$: the harmonic mean of precision and recall.

Optimizing F-score directly is difficult even for linear modeling, because it is non-convex (Nan et al., 2012). In optimizing F-score for decision trees, the problem is worse – a conundrum is possible where two leaves exist, the first leaf containing a higher proportion of positives than the other leaf, yet the first is classified as negative and the second classified as positive. We discuss how this can happen in Appendix D and how we address it, which is to force monotonicity by sweeping across leaves from highest to lowest predictions to calculate the F-score (see Appendix D).

We consider two objectives that are rank statistics:

- Area under the ROC convex hull (AUC_{ch}): the fraction of correctly ranked positive/negative pairs.
- Partial area under the ROC convex hull (pAUC_{ch}) for predetermined threshold θ: the area under the leftmost part of the ROC curve.

Some of the bounds from OSDT (Hu et al., 2019) have straightforward extensions to the objectives listed above, namely the **Upper Bound on Number of Leaves** and **Leaf Permutation Bound**. The remainder of OSDT's bounds do not adapt. Our new bounds are the **Hierarchical Objective Lower Bound**, **Incremental Progress Bound to Determine Splitting**, **Lower Bound on Incremental Progress**, **Equivalent Points Bound**, **Similar Support Bound**, **Incremental Similar Support Bound**, and a **Subset Bound**. To focus our exposition, derivations and bounds for balanced classification loss, weighted classification loss, and F-score loss are in Appendix B, and derivations for AUC loss and partial AUC loss are in Appendix C, with the exception of the hierarchical lower bound for AUC_{ch}, which appears in Section 2.1 to demonstrate how these bounds work.

2.1. Hierarchical Bound for AUC Optimization

Let us discuss objectives that are rank statistics. If a classifier creates binary (as opposed to real-valued) predictions, its ROC curve consists of only three points (0,0), (FPR, TPR), and (1,1). The AUC of a labeled tree is the same as the balanced accuracy, because $\text{AUC} = \frac{1}{2}(\frac{TP}{N^+} \times \frac{FP}{N^-}) + (1-\frac{FP}{N^-}) \times \frac{TP}{N^+} + \frac{1}{2}((1-\frac{TP}{N^+}) \times (1-\frac{FP}{N^-}))$, and since $TP = N^+ - FN$, we have $\text{AUC} = \frac{1}{2}(\frac{N^+-FN}{N^+} \times \frac{FP}{N^-}) + (1 - \frac{FP}{N^-}) \times \frac{N^+-FN}{N^+} + \frac{1}{2}(\frac{FN}{N^+} \times \frac{N^--FP}{N^-}) = 1 - \frac{1}{2}(\frac{FP}{N^-} + \frac{FN}{N^+})$. The more interesting case is when we have real-valued predictions for each leaf and use the ROC convex hull (ROCCH), defined shortly, as the objective.

Let n_i^+ be the number of positive samples in leaf i (n_i^- is the number of negatives) and let r_i be the fraction of positives in leaf i. Let us define the area under the ROC convex hull (ROCCH) (Ferri et al., 2002) for a tree. For a tree d consisting of H_d distinct leaves, $d = (l_1, ..., l_{H_d})$, we reorder leaves according to the fraction of positives, $r_1 \geq r_2 \geq ... \geq r_{H_d}$. For any $i = 0, ..., H_d$, define a labeling S_i for the leaves that labels the first i leaves as positive and remaining $H_d - i$ as negative. The collection of these labelings is $\Gamma = S_0, S_1, ..., S_{H_d}$, where each S_i defines one of the $H_d + 1$ points on the ROCCH (see e.g., Ferri et al., 2002). The associated misranking loss is then 1-AUC_{ch}:

$$\ell(d, \boldsymbol{x}, \boldsymbol{y}) = 1 - \frac{1}{2N^+N^-} \sum_{i=1}^{H} n_i^- \left[\left(\sum_{j=1}^{i-1} 2n_j^+ \right) + n_i^+ \right]. \quad (2)$$

Now let us derive a lower bound on the loss for trees that are incomplete, meaning that some parts of the tree are not yet fully grown. For a partially-grown tree d, the leaf set can be rewritten as $d = (d_{\text{fix}}, r_{\text{fix}}, d_{\text{split}}, r_{\text{split}}, K, H_d)$, where d_{fix} is a set of K fixed leaves that we choose not to split further and d_{split} is the set of $H_d - K$ leaves that can be further split; this notation reflects how the algorithm works, where there are multiple copies of a tree, with some nodes allowed to be split and some that are not. r_{fix} and r_{split} are fractions of positives in the leaves. If we have a new fixed d'_{fix}, which is a superset of d_{fix}, then we say d'_{fix} is a child of d_{fix}. We define $\sigma(d)$ to be all such child trees:

$$\sigma(d) = \{(d'_{\text{fix}}, r'_{\text{fix}}, d'_{\text{split}}, r'_{\text{split}}, K', H'_d) : d_{\text{fix}} \subseteq d'_{\text{fix}}\}. \quad (3)$$

Denote N^+_{split} and N^-_{split} as the number of positive and negative samples captured by d_{split} respectively. Through additional splits, in the best case, d_{split} can give rise to pure leaves, where positive ratios of generated leaves are either 1 or 0. Then the top-ranked leaf could contain up to N^+_{split} positive samples (and 0 negative samples), and the lowest-ranked leaf could capture as few as 0 positive samples and up to N^-_{split} samples. Working now with just the leaves in d_{fix}, we reorder the leaves in d_{fix} by the positive ratios (r_{fix}),

such that $\forall i \in \{1, ..., K\}, r_1 \geqslant r_2 \geqslant ... \geqslant r_K$. Combining these fixed leaves with the bounds for the split leaves, we can define a lower bound on the loss as follows.

Theorem 2.1. *(Lower bound for negative AUC convex hull) For a tree* $d = (d_{\text{fix}}, r_{\text{fix}}, d_{\text{split}}, r_{\text{split}}, K, H_d)$ *using* AUC_{ch} *as the objective, a lower bound on the loss is* $b(d_{\text{fix}}, \mathbf{x}, \mathbf{y}) \leqslant R(d, \mathbf{x}, \mathbf{y})$*, where:*

$$b(d_{\text{fix}}, \mathbf{x}, \mathbf{y}) = 1 - \frac{1}{2N^+N^-}\left(\sum_{i=1}^{K} n_i^- \left[2N^+_{\text{split}} + \left(\sum_{j=1}^{i-1} 2n_j^+\right) + n_i^+\right] + 2N^+N^-_{\text{split}}\right) + \lambda H_d. \quad (4)$$

This leads directly to a hierarchical lower bound for the negative of the AUC convex hull.

Theorem 2.2. *(Hierarchical objective lower bound for negative AUC convex hull) Let* $d = (d_{\text{fix}}, r_{\text{fix}}, d_{\text{split}}, r_{\text{split}}, K, H_d)$ *be a tree with fixed leaves* d_{fix} *and* $d' = (d'_{\text{fix}}, r'_{\text{fix}}, d'_{\text{split}}, r'_{\text{split}}, K', H'_d) \in \sigma(d)$ *be any child tree such that its fixed leaves* d'_{fix} *contain* d_{fix}*, and* $H'_d > H_d$*, then* $b(d_{\text{fix}}, \mathbf{x}, \mathbf{y}) \leqslant R(d', \mathbf{x}, \mathbf{y})$.

This type of bound is the fundamental tool that we use to reduce the size of the search space: if we compare the lower bound $b(d_{\text{fix}}, \mathbf{x}, \mathbf{y})$ for partially constructed tree d to the best current objective R^c, and find that $b(d_{\text{fix}}, \mathbf{x}, \mathbf{y}) \geq R^c$, then there is no need to consider d or any subtree of d, as it is provably non-optimal. The hierarchical lower bound dramatically reduces the size of the search space. However, we also have a collection of tighter bounds at our disposal, as summarized in the next subsection.

We leave the description of partial AUC to Appendix C. Given a parameter θ, the partial AUC of the ROCCH focuses only on the left area of the curve, consisting of the top ranked leaves, whose FPR is smaller than or equal to θ. This metric is used in information retrieval and healthcare.

2.2. Summary of Bounds

Appendix B presents our bounds, which are crucial for reducing the search space. Appendix B presents the Hierarchical Lower Bound (Theorem B.1) for any objective (Equation 1) with an arbitrary monotonic loss function. This theorem is analogous to the Hierarchical Lower Bound for AUC optimization above. Appendix B also contains the Objective Bound with One-Step Lookahead (Theorem B.2), Objective Bound for Sub-Trees (Theorem B.3), Upper Bound on the Number of Leaves (Theorem B.4), Parent-Specific Upper Bound on the Number of Leaves (Theorem B.5), Incremental Progress Bound to Determine Splitting (Theorem B.6), Lower Bound on Incremental Progress (Theorem B.7), Leaf Permutation Bound (Theorem B.8), Equivalent Points Bound (Theorem B.9), and General Similar Support Bound (Theorem B.10). As discussed, no similar support bounds have been used successfully in prior work. In Section 4.2, we show how a new *Incremental Similar Support Bound* can be implemented within our specialized DPB (*dynamic programming with bounds*) algorithm to make decision tree optimization for additive loss functions (e.g., weighted classification error) much more efficient. Bounds for AUC_{ch} and pAUC_{ch} are in Appendix C, including the powerful Equivalent Points Bound (Theorem C.3) for AUC_{ch} and pAUC_{ch} and proofs for Theorem 2.1 and Theorem 2.2. In Appendix G, we provide a new *Subset Bound* implemented within DPB algorithm to effectively remove thresholds introduced by continuous variables.

Note that we do not use convex proxies for rank statistics, as is typically done in supervised ranking (learning-to-rank). Optimizing a convex proxy for a rank statistic can yield results that are far from optimal (see Rudin & Wang, 2018). Instead, we optimize the original (exact) rank statistics directly on the training set, regularized by sparsity.

3. Data Preprocessing Using Bucketization Sacrifices Optimality

As discussed, a preprocessing step common to DL8.5 (Nijssen et al., 2020) and BinOct (Verwer & Zhang, 2019) reduces the search space, but as we will prove, also sacrifices accuracy; we refer to this preprocessing as *bucketization*.

Definition: The *bucketization* preprocessing step proceeds as follows. Order the observations according to any feature j. For any two neighboring positive observations, no split can be considered between them. For any two neighboring negative observations, no split can be considered between them. All other splits are permitted. While bucketization may appear innocuous, we prove it sacrifices optimality.

Theorem 3.1. *The maximum training accuracy for a decision tree on a dataset preprocessed with bucketization can be lower (worse) than the maximum accuracy for the same dataset without bucketization.*

The proof is by construction. In Figure 1 we present a data set such that optimal training with bucketization cannot produce an optimal value of the objective. In particular, the optimal accuracy without bucketization is 93.5%, whereas the accuracy of training with bucketization is 92.2%. These numbers were obtained using BinOCT, DL8.5, and GOSDT. Remember, the algorithms provide a proof of optimality; these accuracy values are optimal, and the same values were found by all three algorithms.

Our dataset is two-dimensional. We expect the sacrifice to become worse with higher dimensions.

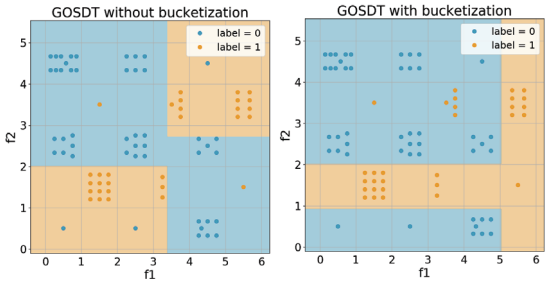

Figure 1. Proof by construction. Bucketization leads to suboptimality. The left plot's vertical split is not allowed in bucketization.

4. GOSDT's DPB Algorithm

For optimizing non-additive loss function, we use PyGOSDT: a variant of GOSDT that is closer to OSDT (Hu et al., 2019). For optimizing additive loss functions we use GOSDT which uses *dynamic programming with bounds* (DPB) to provides a dramatic run time improvement. Like other dynamic programming approaches, we decompose a problem into smaller child problems that can be solved either recursively through a function call or in parallel by delegating work to a separate thread. For decision tree optimization, these subproblems are the possible left and right branches of a decision node.

GOSDT maintains two primary data structures: a *priority queue* to schedule problems to solve and a *dependency graph* to store problems and their dependency relationships. We define a dependency relationship $dep(p_\pi, p_c)$ between problems p_π and p_c if and only if the solution of p_π depends on the solution of p_c. Each p_c is further specified as p_l^j or p_r^j indicating that it is the left or right branch produced by splitting on feature j.

Sections 4.1, 4.2, and 4.3 highlight two key differences between GOSDT and DL8.5 (since DL 8.5 also uses a form of dynamic programming): (1) DL8.5 uniquely identifies a problem p by the *Boolean assertion* that is a conjunctive clause of all splitting conditions in its ancestry, while GOSDT represents a problem p by the *samples that satisfy the Boolean assertion*. This is described in Section 4.1. (2) DL8.5 uses blocking recursive invocations to execute problems, while GOSDT uses a *priority queue* to schedule problems for later. This is described in Section 4.3. Section 4.4 presents additional performance optimizations. Section 4.5 presents a high-level summary of GOSDT. Note that GOSDT's DPB algorithm operates on weighted, additive, non-negative loss functions. For ease of notation, we use classification error as the loss in our exposition.

4.1. Support Set Identification of Nodes

GOSDT leverages the Equivalent Points Bound in Theorem B.9 to save space and computational time. To use this bound, we find the unique values of $\{x_1, ..., x_N\}$, denoted by $\{z_1, ..., z_U\}$, so that each x_i equals one of the z_u's. We store fractions z_u^+ and z_u^-, which are the fraction of positive and negative samples corresponding to each z_u.

$$\begin{aligned} Z &= \{z_u : z_u \in \text{unique}(X), 1 \leq u \leq U \leq N\} \\ z_u^- &= \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}[y_i = 0 \wedge x_i = z_u] \\ z_u^+ &= \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}[y_i = 1 \wedge x_i = z_u]. \end{aligned}$$

Define a as a Boolean assertion that is a conjunctive clause of conditions on the features of z_u (e.g., a is true if and only if the first feature of z_u is 1 and the second feature of z_u is 0). We define *support set* s_a as the set of z_u that satisfy assertion a:

$$s_a = \{z_u : a(z_u) = \text{True}, 1 \leq u \leq U\}. \tag{5}$$

We implement each s_a as a bit-vector of length U and use it to *uniquely identify* a *problem* p in the dependency graph. The bits $\{s_{au}\}_u$ of s_a are defined as follows.

$$s_{au} = 1 \iff z_u \in s_a. \tag{6}$$

With each p, we track a set of values including its *lower bound* (lb) and *upper bound* (ub) of the optimal objective classifying its support set s_a.

In contrast, DL8.5 identifies each problem p using Boolean assertion a rather than s_a: this is an important difference between GOSDT and DL8.5 because many assertions a could correspond to the same support set s_a. However, given the same objective and algorithm, an optimal decision tree for any problem p depends only on the support set s_a. It does not depend on a if one already knows s_a. That is, if two different trees both contain a leaf capturing the same set of samples s, the set of possible child trees for that leaf is the same for the two trees. GOSDT solves problems identified by s. This way, it simultaneously solves all problems identified by any a that produces s.

4.2. Connection to Similar Support Bound

No previous approach has been able to implement a similar support bound effectively. We provide GOSDT's new form of similar support bound, called the *incremental similar support bound*. There are two reasons why this bound works where other attempts have failed: (1) This bound works on *partial trees*. One does not need to have constructed a full tree to use it. (2) The bound takes into account the hierarchical objective lower bound, and hence

leverages the way we search the space within the DBP algorithm. In brief, the bound effectively removes many similar trees from the search region by looking at only one of them (which does not need to be a full tree). We consider weighted, additive, non-negative loss functions: $\ell(d, \mathbf{x}, \mathbf{y}) = \sum_i \text{weight}_i \text{loss}(x_i, y_i)$. Define the maximum weighted loss: $\ell^{\max} = \max_{x,y}[\text{weight}(x, y) \times \text{loss}(x, y)]$.

Theorem 4.1. *(Incremental Similar Support Bound) Consider two trees $d = (d_{\text{fix}}, d_{\text{split}}, K, H)$ and $D = (D_{\text{fix}}, D_{\text{split}}, K, H)$ that differ only by their root node (hence they share the same K and H values). Further, the root nodes between the two trees are similar enough that the support going to the left and right branches differ by at most ω fraction of the observations. (That is, there are ωN observations that are captured either by the left branch of d and right branch of D or vice versa.) Define $S_{\text{uncertain}}$ as the maximum of the support within d_{split} and D_{split}: $S_{\text{uncertain}} = \max(\text{supp}(d_{\text{split}}), \text{supp}(D_{\text{split}}))$. For any child tree $d' \in \sigma(d)$ grown from d (grown from the nodes in d_{split}, that would not be excluded by the hierarchical objective lower bound) and for any child tree $D' \in \sigma(D)$ grown from D (grown from nodes in D_{split}, not excluded by the hierarchical objective lower bound), we have:*

$$|R(d', \mathbf{x}, \mathbf{y}) - R(D', \mathbf{x}, \mathbf{y})| \leq (\omega + 2S_{\text{uncertain}})\ell^{\max}.$$

The proof is in Appendix F. Unlike the similar support bounds in CORELS and OSDT, which require pairwise comparisons of problems, this incremental similar support bound emerges from the support set representation. The descendants $\sigma(d)$ and $\sigma(D)$ share many of the same support sets. Because of this, the shared components of their upper and lower bounds are updated simultaneously (to similar values). This bound is helpful when our data contains continuous variables: if a split at value v was already visited, then splits at values close to v can reuse prior computation to immediately produce tight upper and lower bounds.

4.3. Asynchronous Bound Updates

GOSDT computes the objective values hierarchically by defining the minimum objective $R^*(p)$ of problem p as an aggregation of minimum objectives over the child problems p_l^j and p_r^j of p for $1 \leq j \leq M$.

$$R^*(p) = \min_j (R^*(p_l^j) + R^*(p_r^j)). \tag{7}$$

Since DL8.5 computes each $R^*(p_l^j)$ and $R^*(p_r^j)$ in a blocking call to the the child problems p_l^j and p_r^j, it necessarily computes $R^*(p_l^j) + R^*(p_r^j)$ after solving p_l^j and p_r^j, which is a disadvantage. In contrast, GOSDT computes bounds over $R^*(p_l^j) + R^*(p_r^j)$ that are available before knowing the exact values of $R^*(p_l^j)$ and $R^*(p_r^j)$. The bounds over $R^*(p_l^j)$ and $R^*(p_r^j)$ are solved asynchronously and possibly in parallel. If for some j and j' the bounds imply that $R^*(p_l^j) + R^*(p_r^j) > R^*(p_l^{j'}) + R^*(p_r^{j'})$, then we can conclude that p's solution no longer depends on p_l^j and p_r^j. Since GOSDT executes asynchronously, it can draw this conclusion and focus on $R^*(p_l^{j'}) + R^*(p_r^{j'})$ without fully solving $R^*(p_l^j) + R^*(p_r^j)$.

To encourage this type of bound update, GOSDT uses the priority queue to send high-priority signals to each parent p_π of p when an update is available, prompting a recalculation of $R^*(p_\pi)$ using Equation 7.

4.4. Fast Selective Vector Sums

New problems (i.e., p_l and p_r) require initial upper and lower bounds on the optimal objective. We define the initial lower bound lb and upper bound ub for a problem p identified by support set s as follows. For $1 \leq u \leq U$, define:

$$z_u^{\min} = \min(z_u^-, z_u^+). \tag{8}$$

This is the fraction of minority class samples in equivalence class u. Then,

$$lb = \lambda + \sum_u s_u z_u^{\min}. \tag{9}$$

This is a basic equivalence points bound, which predicts all minority class equivalence points incorrectly. Also,

$$ub = \lambda + \min\left(\sum_u s_u z_u^-, \sum_u s_u z_u^+\right). \tag{10}$$

This upper bound comes from a baseline algorithm of predicting all one class.

We use the *prefix sum trick* in order to speed up computations of sums of a subset of elements of a vector. That is, for any vector z^{vec} (e.g., $z^{\min}$, z^-, z^+), we want to compute a sum of a subsequence of z^{vec}: we precompute (during preprocessing) a prefix sum vector $z^{\text{cumulative}}$ of the vector z^{vec} defined by the cumulative sum $z_u^{\text{cumulative}} = \sum_{j=1}^u z_j^{\text{vec}}$. During the algorithm, to sum over ranges of contiguous values in z^{vec}, over some indices a through b, we now need only take $z^{\text{cumulative}}[b] - z^{\text{cumulative}}[a-1]$. This reduces a linear time calculation to constant time. This fast sum is leveraged over calculations with the support sets of input features–for example, quickly determining the difference in support sets between two features.

4.5. The GOSDT Algorithm

Algorithm 1 constructs and optimizes problems in the *dependency graph* such that, upon completion, we can extract the optimal tree by traversing the dependency graph by greedily choosing the split with the lowest objective value.

This extraction algorithm and a more detailed version of the GOSDT algorithm is provided in Appendix J. We present the key components of this algorithm, highlighting the differences between GOSDT and DL8.5. Note that all features have been binarized prior to executing the algorithm.

Lines 8 to 11: Remove an item from the queue. If its bounds are equal, no further optimization is possible and we can proceed to the next item on the queue.

Lines 12 to 18: Construct new subproblems, p_l, p_r by splitting on feature j. Use lower and upper bounds of p_l and p_r to compute new bounds for p. *We key the problems by the bit vector corresponding to their support set* s. Keying problems in this way avoids processing the same problem twice; other dynamic programming implementations, such as DL8.5, will process the same problem multiple times.

Lines 19 to 22: Update p with lb' and ub' computed using Equation 7 and propagate that update to all ancestors of p by enqueueing them with high priority (p will have multiple parents if two different conjunctions produce the same support set). This triggers the ancestor problems to recompute their bounds using Equation 7. *The high priority ensures that ancestral updates occur before processing* p_l *and* p_r. This scheduling is one of the key differences between GOSDT and DL8.5; by eagerly propagating bounds up the dependency tree, GOSDT prunes the search space more aggressively.

Lines 25 to 31: Enqueue p_l and p_r only if the interval between their lower and upper bounds overlaps with the interval between p's lower and upper bounds. This ensures that eventually the lower and upper bounds of p converge to a single value.

Lines 33 to 41: Define **FIND_OR_CREATE_NODE**, which constructs (or finds an existing) problem p corresponding to support set s, initializing the lower and upper bound and checking if p is eligible to be split, using the Incremental Progress Bound to Determine Splitting (Theorem B.6) and the Lower Bound on Incremental Progress (Theorem B.7) (these bounds are checked in subroutine *fails_bounds*). *get_lower_bound* returns lb using Equation 9. *get_upper_bound* returns ub using Equation 10.

An implementation of the algorithm is available at: https://github.com/Jimmy-Lin/GeneralizedOptimalSparseDecisionTrees

Algorithm 1 GOSDT(R, x, y, λ)

1: **input:** R, Z, z^-, z^+, λ *// risk, samples, regularizer*
2: $Q = \emptyset$ *// priority queue*
3: $G = \emptyset$ *// dependency graph*
4: $s_0 \leftarrow \{1, ..., 1\}$ *// bit-vector of 1's of length U*
5: $p_0 \leftarrow$ FIND_OR_CREATE_NODE(G, s_0) *// root*
6: Q.push(s_0) *// add to priority queue*
7: **while** $p_0.lb \neq p_0.ub$ **do**
8: $s \leftarrow Q$.pop() *// index of problem to work on*
9: $p \leftarrow G$.find(s) *// find problem to work on*
10: **if** $p.lb = p.ub$ **then**
11: **continue** *// problem already solved*
12: $(lb', ub') \leftarrow (\infty, \infty)$ *// loose starting bounds*
13: **for** each feature $j \in [1, M]$ **do**
14: $s_l, s_r \leftarrow$ split(s, j, Z) *// create children*
15: $p_l^j \leftarrow$FIND_OR_CREATE_NODE(G, s_l)
16: $p_r^j \leftarrow$FIND_OR_CREATE_NODE(G, s_r)
// create bounds as if j were chosen for splitting
17: $lb' \leftarrow \min(lb', p_l^j.lb + p_r^j.lb)$
18: $ub' \leftarrow \min(ub', p_l^j.ub + p_r^j.ub)$
// signal the parents if an update occurred
19: **if** $p.lb \neq lb'$ **or** $p.ub \neq ub'$ **then**
20: $(p.lb, p.ub) \leftarrow (lb', ub')$
21: **for** $p_\pi \in G$.parent(p) **do**
// propagate information upwards
22: Q.push$(p_\pi$.id, priority $= 1)$
23: **if** $p.lb = p.ub$ **then**
24: **continue** *// problem solved just now*
// loop, enqueue all children
25: **for** each feature $j \in [1, M]$ **do**
// fetch p_l^j and p_r^j in case of update
26: repeat line 14-16
27: $lb' \leftarrow p_l^j.lb + p_r^j.lb$
28: $ub' \leftarrow p_l^j.ub + p_r^j.ub$
29: **if** $lb' < ub'$ **and** $lb' \leq p.ub$ **then**
30: Q.push$(s_l$, priority $= 0)$
31: Q.push$(s_r$, priority $= 0)$
32: **return**

33: **subroutine** FIND_OR_CREATE_NODE(G,s)
34: **if** G.find$(s) =$ NULL *// p not yet in graph*
35: $p.id \leftarrow s$ *// identify p by s*
36: $p.lb \leftarrow$ get_lower_bound(s, Z, z^-, z^+)
37: $p.ub \leftarrow$ get_upper_bound(s, Z, z^-, z^+)
38: **if** fails_bounds(p) **then**
39: $p.lb = p.ub$ *// no more splitting allowed*
40: G.insert(p) *// put p in dependency graph*
41: **return** G.find(s)

5. Experiments

We present details of our experimental setup and datasets in Appendix I. GOSDT's novelty lies in its ability to optimize a large class of objective functions and its ability to efficiently handle continuous variables without sacrificing optimality. Thus, our evaluation results: 1) Demonstrate our ability to optimize over a large class of objectives (AUC in particular), 2) Show that GOSDT outperforms other approaches in producing models that are both accurate and sparse, and 3) Show how GOSDT scales in its handling of continuous variables relative to other methods. In Appendix I we show time-to-optimality results.

Optimizing Many Different Objectives: We use the FourClass dataset (Chang & Lin, 2011) to show optimal decision trees corresponding to different objectives. Figures 2 and 3 show the training ROC of decision trees generated for six different objectives and optimal trees for accuracy, AUC and partial area under ROC convex hull. Optimizing different objectives produces different trees with different FP and FN. (No other decision tree method is designed to fully optimize any objective except accuracy, so there is no comparison to other methods.)

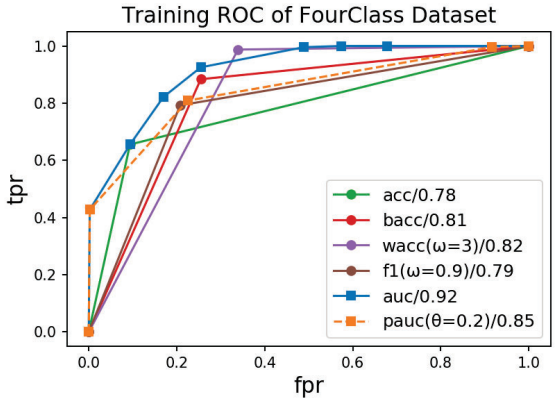

Figure 2. Training ROC of FourClass dataset ($\lambda = 0.01$). A/B in the legend at the bottom right shows the objective and its parameters/area under the ROC.

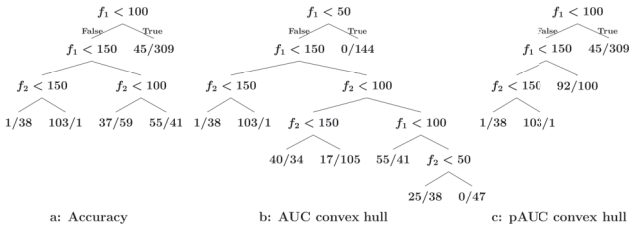

Figure 3. Trees for different objectives. A/B in the leaf node represents the number of positive/negative samples amongst the total samples captured in that leaf.

Binary Datasets, Accuracy vs Sparsity: We compare

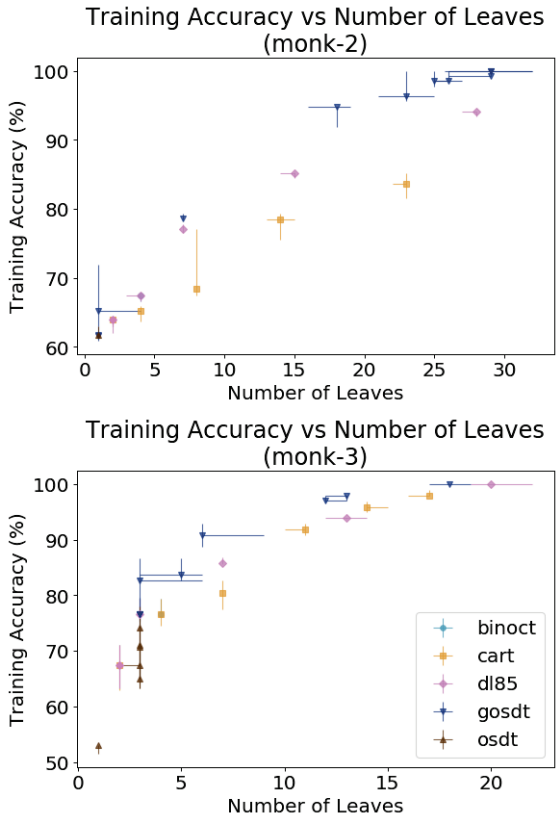

Figure 4. Training accuracy achieved by BinOCT, CART, DL8.5, GOSDT, and OSDT as a function of the number of leaves.

models produced from BinOCT (Verwer & Zhang, 2019), CART (Breiman et al., 1984), DL8.5 (Nijssen et al., 2020), OSDT (Hu et al., 2019), and GOSDT. For each method, we select hyperparameters to produce trees of varying numbers of leaves and plot training accuracy against sparsity (number of leaves). GOSDT directly optimizes the tradeoff between training accuracy and number of leaves, producing points on the efficient frontier. Figure 4 and 5 show (1) that GOSDT typically produces excellent training and test accuracy with a reasonable number of leaves, and (2) that we can quantify how close to optimal the other methods are. Learning theory provides guarantees that training and test accuracy are close for sparse trees; more results are in Appendix I.

Continuous Datasets, Slowdown vs Thresholds: We preprocessed by encoding continuous variables as a set of binary variables, using all possible thresholds. We then reduced the number of binary variables by combining sets of k variables for increasing values of k. Binary variables were ordered, firstly, in order of the indices of the continuous variable to which they correspond, and secondly, in order of their thresholds. We measure the slow-down introduced by increasing the number of binary variables relative

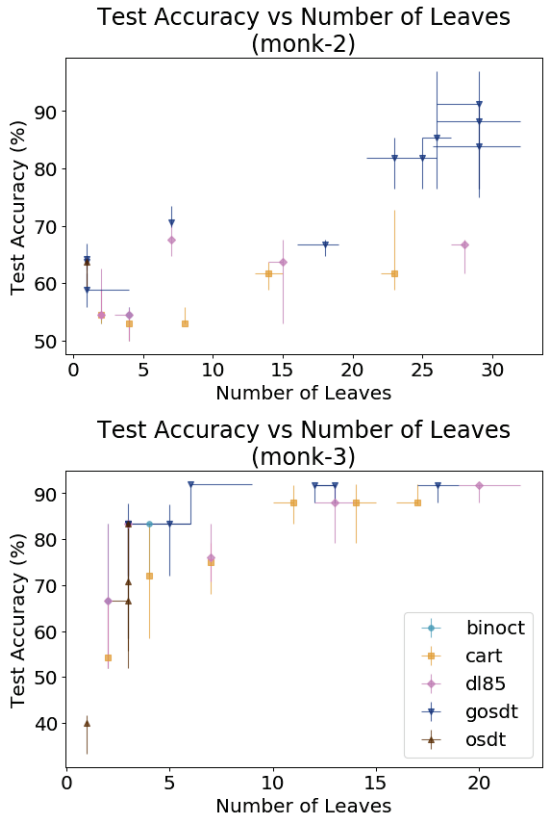

Figure 5. Test accuracy achieved by BinOCT, CART, DL8.5, GOSDT, and OSDT as a function of the number of leaves.

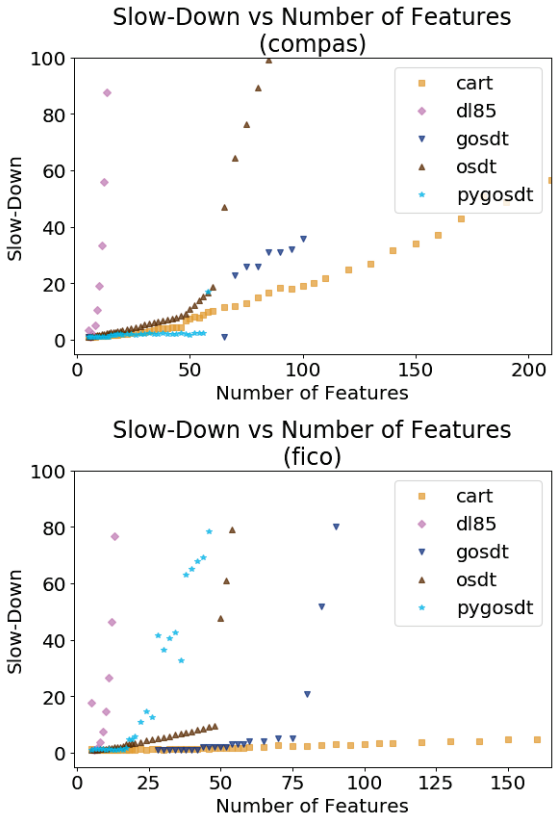

Figure 6. Training time of BinOCT, CART, DL8.5, GOSDT, and OSDT as a function of the number of binary features used to encode the continuous dataset ($\lambda = 0.3125$ or max depth = 5).

to each algorithm's own best time. Figure 6 shows these results for CART, DL8.5, GOSDT, and OSDT. As the number of features increases (i.e., k approaches 1), GOSDT typically slows down less than DL8.5 and OSDT. This relatively smaller slowdown allows GOSDT to handle more thresholds introduced by continuous variables.

Appendix I presents more results including training times several orders of magnitude better than the state-of-the-art.

An implementation of our experiments is available at: https://github.com/Jimmy-Lin/TreeBenchmark

6. Discussion and Future Work

GOSDT (and related methods) differs fundamentally from other types of machine learning algorithms. Unlike neural networks and typical decision tree methods, it provides a proof of optimality for highly non-convex problems. Unlike support vector machines, ensemble methods, and neural networks again, it produces sparse interpretable models without using convex proxies–it solves the exact problem of interest in an efficient and provably optimal way. As usual, statistical learning theoretic guarantees on test error are the tightest for simpler models (sparse models) with the lowest empirical error on the training set, hence the choice of accuracy, regularized by sparsity (simplicity). If GOSDT is stopped early, it reports an optimality gap, which allows the user to assess whether the tree is sufficiently close to optimal, retaining learning theoretic guarantees on test performance. GOSDT is most effective for datasets with a small or medium number of features. The algorithm scales well with the number of observations, easily handling tens of thousands.

There are many avenues for future work. Since GOSDT provides a framework to handle objectives that are monotonic in FP and FN, one could create many more objectives than we have enumerated here. Going beyond these objectives to handle other types of monotonicity, fairness, ease-of-use, and cost-related soft and hard constraints are natural extensions. There are many avenues to speed up GOSDT related to exploration of the search space, garbage collection, and further bounds.

References

Aïvodji, U., Ferry, J., Gambs, S., Huguet, M.-J., and Siala, M. Learning Fair Rule Lists. *arXiv e-prints*, pp. arXiv:1909.03977, Sep 2019.

Angelino, E., Larus-Stone, N., Alabi, D., Seltzer, M., and Rudin, C. Learning certifiably optimal rule lists for categorical data. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2017.

Angelino, E., Larus-Stone, N., Alabi, D., Seltzer, M., and Rudin, C. Learning certifiably optimal rule lists for categorical data. *Journal of Machine Learning Research*, 18 (234):1–78, 2018.

Bennett, K. Decision tree construction via linear programming. In *Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society Conference, Utica, Illinois*, 1992.

Bennett, K. P. and Blue, J. A. Optimal decision trees. Technical report, R.P.I. Math Report No. 214, Rensselaer Polytechnic Institute, 1996.

Bertsimas, D. and Dunn, J. Optimal classification trees. *Machine Learning*, 106(7):1039–1082, 2017.

Blanquero, R., Carrizosa, E., Molero-Rıo, C., and Morales, D. R. Optimal randomized classification trees. August 2018.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. *Classification and Regression Trees*. Wadsworth, 1984.

Chang, C.-C. and Lin, C.-J. Libsvm: a library for support vector machines, 2011. software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Chen, C. and Rudin, C. An optimization approach to learning falling rule lists. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2018.

Dheeru, D. and Karra Taniskidou, E. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Ferri, C., Flach, P., and Hernández-Orallo, J. Learning decision trees using the area under the ROC curve. In *International Conference on Machine Learning (ICML)*, volume 2, pp. 139–146, 2002.

FICO, Google, Imperial College London, MIT, University of Oxford, UC Irvine, and UC Berkeley. Explainable Machine Learning Challenge. https://community.fico.com/s/explainable-machine-learning-challenge, 2018.

Garey, M. Optimal binary identification procedures. *SIAM J. Appl. Math.*, 23(2):173—186, September 1972.

Garofalakis, M., Hyun, D., Rastogi, R., and Shim, K. Building decision trees with constraints. *Data Mining and Knowledge Discovery*, 7:187–214, 2003.

Hu, X., Rudin, C., and Seltzer, M. Optimal sparse decision trees. In *Proceedings of Neural Information Processing Systems (NeurIPS)*, 2019.

Klivans, A. R. and Servedio, R. A. Toward attribute efficient learning of decision lists and parities. *Journal of Machine Learning Research*, 7:587–602, 2006.

Larson, J., Mattu, S., Kirchner, L., and Angwin, J. How we analyzed the COMPAS recidivism algorithm. *ProPublica*, 2016.

Larus-Stone, N. L. *Learning Certifiably Optimal Rule Lists: A Case For Discrete Optimization in the 21st Century*. 2017. Undergraduate thesis, Harvard College.

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

Loh, W.-Y. Fifty years of classification and regression trees. *International Statistical Review*, 82(3):329–348, 2014.

Meisel, W. S. and Michalopoulos, D. A partitioning algorithm with application in pattern classification and the optimization of decision tree. *IEEE Trans. Comput., C-22*, pp. 93–103, 1973.

Menickelly, M., Günlük, O., Kalagnanam, J., and Scheinberg, K. Optimal decision trees for categorical data via integer programming. *Preprint at arXiv:1612.03225*, January 2018.

Morgan, J. N. and Sonquist, J. A. Problems in the analysis of survey data, and a proposal. *J. Amer. Statist. Assoc.*, 58:415–434, 1963.

Nan, Y., Chai, K. M., Lee, W. S., and Chieu, H. L. Optimizing F-measure: A tale of two approaches. *arXiv preprint arXiv:1206.4625*, 2012.

Narodytska, N., Ignatiev, A., Pereira, F., and Marques-Silva, J. Learning optimal decision trees with SAT. In *Proc. International Joint Conferences on Artificial Intelligence (IJCAI)*, pp. 1362–1368, 2018.

Nijssen, S. and Fromont, E. Mining optimal decision trees from itemset lattices. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 530–539. ACM, 2007.

Nijssen, S., Schaus, P., et al. Learning optimal decision trees using caching branch-and-bound search. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

Payne, H. J. and Meisel, W. S. An algorithm for constructing optimal binary decision trees. *IEEE Transactions on Computers*, C-26(9):905–916, 1977.

Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

Rudin, C. and Ertekin, S. Learning customized and optimized lists of rules with mathematical programming. *Mathematical Programming C (Computation)*, 10:659–702, 2018.

Rudin, C. and Wang, Y. Direct learning to rank and rerank. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 775–783, 2018.

Verwer, S. and Zhang, Y. Learning optimal classification trees using a binary linear program formulation. In *Proc. Thirty-third AAAI Conference on Artificial Intelligence*, 2019.

Vilas Boas, M. G., Santos, H. G., Merschmann, L. H. d. C., and Vanden Berghe, G. Optimal decision trees for the algorithm selection problem: integer programming based approaches. *International Transactions in Operational Research*, 2019.

Wang, T., Rudin, C., Doshi-Velez, F., Liu, Y., Klampfl, E., and MacNeille, P. A Bayesian framework for learning rule sets for interpretable classification. *Journal of Machine Learning Research*, 18(70):1–37, 2017.