The Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI-25)

Optimal Classification Trees for Continuous Feature Data Using Dynamic Programming with Branch-and-Bound

Catalin E. Brita[1, 2], Jacobus G. M. van der Linden[2*], Emir Demirović[2]

[1] University of Amsterdam, The Netherlands,
[2] Delft University of Technology, The Netherlands
Catalin.Brita@student.uva.nl, {J.G.M.vanderLinden, E.Demirovic}@tudelft.nl

Abstract

Computing an optimal classification tree that provably maximizes training performance within a given size limit, is NP-hard, and in practice, most state-of-the-art methods do not scale beyond computing optimal trees of depth three. Therefore, most methods rely on a coarse binarization of continuous features to maintain scalability. We propose a novel algorithm that optimizes trees directly on the continuous feature data using dynamic programming with branch-and-bound. We develop new pruning techniques that eliminate many sub-optimal splits in the search when similar to previously computed splits and we provide an efficient subroutine for computing optimal depth-two trees. Our experiments demonstrate that these techniques improve runtime by one or more orders of magnitude over state-of-the-art optimal methods and improve test accuracy by 5% over greedy heuristics.

1 Introduction

Decision trees combine human comprehensibility and accuracy and can capture complex non-linear relationships. As such, decision trees are well-suited models for explainable AI (Rudin 2019; Arrieta et al. 2020). Despite their straightforwardness, constructing an optimal decision tree (ODT, a tree with the smallest training error) of limited size is NP-hard (Hyafil and Rivest 1976). Therefore, greedy heuristics, such as CART (Breiman et al. 1984) and C4.5 (Quinlan 1993), have been widely used. These methods attain scalability by locally optimizing an information gain metric at each decision node, but yield on average less accurate and larger trees than optimal (Van der Linden et al. 2024).

To compute ODTs, some employ general-purpose solvers such as mixed-integer programming (MIP) (Bertsimas and Dunn 2017), constraint programming (Verhaeghe et al. 2020), or Boolean satisfiability (SAT) solvers (Narodytska et al. 2018). However, these approaches struggle to scale with the number of observations and features.

Better scalability is obtained by specialized algorithms using dynamic programming (DP) and branch and bound (BnB) (Aglin, Nijssen, and Schaus 2020a; Demirović et al. 2022). However, most of these algorithms cannot directly deal with numeric data which are frequently present in real-world datasets. Therefore, these algorithms either use a coarse binarization resulting in loss of optimality; or require a binary feature for every possible threshold on the numeric data, which drastically hurts scalability, because their runtime scales exponentially with the number of such features.

To the best of our knowledge, Quant-BnB (Mazumder, Meng, and Wang 2022) is the only specialized optimal algorithm that processes continuous features directly. This BnB algorithm considers splits at certain quantiles of the feature distribution. Obtained solutions are used as bounds for pruning, whereas other parts are further explored with splits on quantiles of subregions of the feature distribution. Though Quant-BnB outperforms other algorithms by a large margin on numeric data, scalability is still an issue: it requires several hours to find trees of depth three for some datasets and the authors recommend against using it beyond depth three.

In this work, we present ConTree, which combines existing DP and BnB techniques with new bounding techniques to improve the scalability of computing optimal classification trees for numeric data. Our new lower-bounding techniques prune large parts of the search space while adding negligible computational overhead. Furthermore, for depth-two trees, we propose a specialized subroutine that exploits the fact that we can sort numeric data.

Our experiments show that these algorithmic improvements boost scalability by one or more orders of magnitude over Quant-BnB and three previous MIP and SAT approaches by an ever larger margin. This makes ConTree the first approach to compute depth-four ODTs beyond small or binarized datasets within a reasonable time. When trained with the same size limit, ConTree's test accuracy is on average 5% higher than CART and 0.7% higher than ODTs trained with a coarse binarization.

2 Related Work

Heuristics Traditionally, decision trees are trained using top-down induction heuristics such as CART (Breiman et al. 1984) and C4.5 (Quinlan 1993) because of their scalability. These heuristics recursively divide the data based on local criteria such as information gain or Gini impurity. On average this yields trees that are larger than the optimal tree (Murthy and Salzberg 1995) or, if constrained by a fixed depth, have lower out-of-sample accuracy than optimal trees under the same size limit (Van der Linden et al. 2024).

*Corresponding author
Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Optimal Optimal decision trees globally optimize an objective (e.g., minimize the number of misclassifications) within a given size limit on the training data. Computing ODTs, however, is NP-hard (Hyafil and Rivest 1976), and thus scalability is challenging. To address this challenge, many different approaches have been proposed.

General-purpose solvers The first MIP approaches were proposed by Bertsimas and Dunn (2017) and Verwer and Zhang (2017). Many other formulations followed, typically using binarization of the continuous feature data to improve the scalability (Verwer and Zhang 2019; Günlük et al. 2021; Aghaei, Gómez, and Vayanos 2024; Liu et al. 2024). Hua, Ren, and Cao (2022) instead use a multi-stage MIP model with novel lower bounds. Alès, Huré, and Lambert (2024) obtain stronger linear relaxations from a novel quadratic formulation. Narodytska et al. (2018), Janota and Morgado (2020), Avellaneda (2020), Hu et al. (2020) and Alòs, Ansótegui, and Torres (2023) propose SAT models that also require binarization of continuous feature data. As far as we know, Shati, Cohen, and McIlraith (2021, 2023) propose the only SAT-based algorithm that can directly process continuous and categorical features. Finally, Verhaeghe et al. (2020) propose a constraint programming approach that also requires binary data. However, despite improvements, these MIP, SAT, and CP methods still face problems scaling beyond a few thousand data instances and trees of depth three.

Specialized algorithms Nijssen and Fromont (2007, 2010) introduced DL8, an early DP approach. Aglin, Nijssen, and Schaus (2020a,b) improved it to DL8.5 with branch-and-bound and extended caching techniques. Hu, Rudin, and Seltzer (2019) and Lin et al. (2020) contribute new lower bounds including a subproblem similarity bound. Demirović et al. (2022) introduce a specialized subroutine for trees of depth two and constraints for limiting the number of branching nodes. Kiossou et al. (2022) and Demirović, Hebrard, and Jean (2023) improve the anytime performance of the search. Van der Linden, De Weerdt, and Demirović (2023) generalize previous DP approaches beyond classification. Because of all these algorithmic advances, a recent survey considers the DP approach currently the best in terms of scalability (Costa and Pedreira 2023). However, all of these methods require binarized input data.

Continuous features Quant-BnB (Mazumder, Meng, and Wang 2022) is the only specialized algorithm for ODTs that directly optimizes datasets with continuous features. It employs branch-and-bound by splitting on quantiles of the feature distribution. Although Quant-BnB can handle much larger datasets than the MIP and SAT approaches, it also struggles to scale beyond trees of depth three.

Summary Scalability advances for ODT search mostly require binarization. When operating directly on the numeric data, scalability is still challenging.

3 Preliminaries

In this section, we introduce notation, formally define the problem, and describe the lower-bounding technique that provides the basis for ConTree's pruning.

Notation Let $\mathcal{D}$ describe a dataset with $n = |\mathcal{D}|$ observations (x, y) with $x \in \mathbb{R}^p$ and $y \in \mathcal{Y}$ describing respectively the feature vector and label of an observation. Here p is the number of features and $\mathcal{Y}$ the set of class labels. The set of all features is denoted as $\mathcal{F} = \{f_1, \dots, f_p\}$. Each observation x contains the values of these p features such that x_f is the value of feature f in observation x. Let $\mathcal{D}^f$ denote the sorted values x_f for $(x, y) \in \mathcal{D}$ and let U^f describe all unique sorted values in $\mathcal{D}^f$ (with $\mathcal{D}$ determined by context) and similarly

$$S^f = \left\{ \frac{U_1^f + U_2^f}{2}, ..., \frac{U_{|U^f|-1}^f + U_{|U^f|}^f}{2} \right\} \quad (1)$$

the set of possible thresholds on feature f for $\mathcal{D}$: the midpoints between the unique feature values. Let $m = |S^f|$ be the number of possible thresholds. Given a threshold $\tau \in S^f$, let $z(\tau)$ represent the index of the observation in $\mathcal{D}^f$ with the largest value for f smaller than τ. Similarly, let $u(\tau)$ represent the index of τ in U^f. Finally, let $\mathcal{D}(f \leq \tau)$ describe the subset of observations $(x, y) \in \mathcal{D}$ where $x_f \leq \tau$.

Problem definition A decision tree is a function $t : \mathbb{R}^p \rightarrow \mathcal{Y}$ that recursively splits the feature space $\mathbb{R}^p$ into sub-regions and predicts the label of each sub-region. Let $\mathcal{T}(\mathcal{D}, d)$ describe the set of all decision trees for the dataset $\mathcal{D}$ with a maximum depth of d. Then the ODT t_{opt} is the tree within that set that minimizes the misclassification score:

$$t_{\text{opt}} = \underset{t \in \mathcal{T}(\mathcal{D}, d)}{\operatorname{argmin}} \sum_{(x,y) \in \mathcal{D}} \mathbb{1}(t(x) \neq y) \,. \quad (2)$$

This work is limited to binary axis-aligned trees: every branching node splits on precisely one feature $f \in \mathcal{F}$ based on a threshold τ such that every observation with $x_f \leq \tau$ goes left in the tree while the rest goes to the right.

Similarity lower bounding Hu, Rudin, and Seltzer (2019), Lin et al. (2020), and Demirović et al. (2022) propose a similarity-based lower-bounding (SLB) technique that compares a new dataset $\mathcal{D}_{new}$ with a previously analyzed dataset $\mathcal{D}_{old}$ to derive a lower bound on the misclassification score of the new dataset. SLB assumes that all new observations in the new dataset will be classified correctly and all removed observations from the old dataset were classified incorrectly, yielding the following lower bound:

$$\theta_{\mathcal{D}_{new}} \geq \theta_{\mathcal{D}_{old}} - |\mathcal{D}_{old} \setminus \mathcal{D}_{new}| \,, \quad (3)$$

where θ is the misclassification score a decision tree with the same depth limit can achieve on the dataset. From this property, we derive three novel pruning techniques below.

4 The ConTree Algorithm

We present the ConTree algorithm (CT) which constructs an ODT by recursively performing splits on every branching node within a full tree of pre-defined depth. Subproblems are identified by the dataset $\mathcal{D}$ and the remaining depth limit d. This results in the following recursive DP formulation:

$$\text{CT}(\mathcal{D}, d) = \begin{cases} \min\limits_{\hat{y} \in \mathcal{Y}} \sum\limits_{(x,y) \in \mathcal{D}} \mathbb{1}(\hat{y} \neq y) & \text{if } d = 0 \\ \min\limits_{f \in \mathcal{F}} \text{Branch}(\mathcal{D}, d, f) & \text{if } d > 0 \end{cases} \quad (4)$$

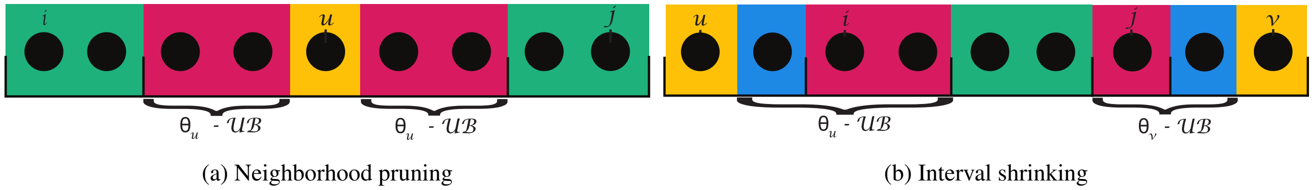

Figure 1: The split points u and v for which the score θ_u and θ_v are calculated are yellow. The newly pruned values are shown in red. Green indicates the remaining split points for further search. Blue indicates unaffected values outside of $[i..j]$.

Leaf nodes assign the label with the least misclassifications. Branching nodes find the feature f with the best misclassification score from the subtrees by calling the subprocedure Branch which iterates over all possible split thresholds τ:

$$\mathrm{Branch}(\mathcal{D}, d, f) = \min_{\tau \in S^f} \mathrm{Split}(\mathcal{D}, d, f, \tau)\,. \tag{5}$$

Every split results in two subproblems:

$$\mathrm{Split}(\mathcal{D}, d, f, \tau) = \mathrm{CT}\left(\mathcal{D}(f \leq \tau), d-1\right) + \mathrm{CT}\left(\mathcal{D}(f > \tau), d-1\right)\,. \tag{6}$$

Given a splitting feature f, computing the misclassification score θ_τ for all possible split points $\tau \in S^f$ is computationally expensive since each split point considered requires solving two (potentially large) subproblems. Therefore, we provide the following runtime improvements (each of which preserves optimality):

Lower-bound pruning three novel pruning techniques specifically designed for continuous features to speed up the computation without losing optimality;

Depth-two subroutine a subroutine for depth-two trees that iterates over sorted feature data to update class occurrences and efficiently solves the two depth-one subproblems in Eq. (6) simultaneously;

Caching the same dataset caching technique as Demirović et al. (2022): ConTree reuses cached solutions to subproblems (defined by $\mathcal{D}$ and d).

To control the trade-off between training time and accuracy, we also provide a $\mathrm{max\text{-}gap}$ parameter to set the maximum permissible gap to the optimal solution.

Pruning Techniques

Based on the similarity-based lower bound presented in Eq. (3), we present three novel pruning techniques to reduce the number of split points that need to be considered in Eq. (5) without losing optimality. The key idea is that if the feature data is sorted, the solution of any Split call with threshold τ provides a lower bound for all next calls with a different threshold τ' since we can easily count how many observations shifted from the left to the right subtree by subtracting their indices in the sorted data: $|z(\tau) - z(\tau')|$.

Theorem 1. *Let $\mathcal{UB}$ be the best solution so far or the score needed to obtain a better solution. Let $\theta_\tau = \mathrm{Split}(\mathcal{D}, d, f, \tau)$ be the optimal misclassification score for the subtree when branching on f with threshold τ. Then any other threshold τ' with $|z(\tau) - z(\tau')| \leq \theta_\tau - \mathcal{UB}$ cannot yield an improving solution.*

Proof. This follows directly from Eq. (3). If $\tau' > \tau$, then $|\mathcal{D}(f \leq \tau') \setminus \mathcal{D}(f \leq \tau)| = z(\tau') - z(\tau)$ and if $\tau' < \tau$, then $|\mathcal{D}(f > \tau') \setminus \mathcal{D}(f > \tau)| = z(\tau) - z(\tau')$. Therefore, the SLB for a split at τ' is: $\theta_{\tau'} \geq \theta_\tau - |z(\tau) - z(\tau')| \geq \mathcal{UB}$. □

Corollary 1. *Let u be a split index with its corresponding solution value θ_u and index $z(u)$ within $\mathcal{D}^f$. Let Δ be the difference between θ_u and $\mathcal{UB}$ and at least one: $\Delta = \max(1, \theta_u - \mathcal{UB})$. Any improving split must have a threshold smaller than the value in $\mathcal{D}^f$ at index $z(u) - \Delta$ or larger than the value at index $z(u) + \Delta$.*

Per Branch call, ConTree keeps track of the set of threshold indices that may yield a split that improves the current best tree. This set is represented as a set of index intervals Q. Initially, Q contains one interval of all indices: $Q = \{[1..m]\}$. After each Split call, ConTree updates the set Q by using three pruning functions $\mathcal{P}$ that return a list of pruned intervals from the current interval $[i..j]$, the current $\mathcal{UB}$, and one or more subproblem solutions θ. Next, we explain these three pruning functions: neighborhood pruning, interval shrinking, and sub-interval pruning.

Neighborhood pruning (NB) After the misclassification score θ_u for a split point $u \in [i..j]$ is computed, neighborhood pruning uses the SLB to remove similar split points from consideration. A simplified illustration of this pruning technique can be seen in Fig. 1a. Using Cor. 1, we define two functions $\underline{A}$ and $\bar{A}$ that return the closest thresholds from u that could still improve on $\mathcal{UB}$.

$$\underline{A}(u, \Delta) = \max\left\{u' \in [m] \mid U^f_{u'} < \mathcal{D}^f_{z(u)-\Delta}\right\} \tag{7}$$

$$\bar{A}(u, \Delta) = \min\left\{u' \in [m] \mid U^f_{u'+1} > \mathcal{D}^f_{z(u)+\Delta}\right\} \tag{8}$$

The functions $\underline{A}$ and $\bar{A}$ can be implemented using binary search with time complexity $\mathcal{O}(\log(m))$. Using these, we define $\mathcal{P}_{\mathrm{NB}}$ which yields two new intervals:

$$\mathcal{P}_{\mathrm{NB}}([i..j], u, \Delta) = \{[i..\underline{A}(u, \Delta)], [\bar{A}(u, \Delta)..j]\} \tag{9}$$

Example. Consider a continuous feature vector with values $[0.4, 0.5, 0.5, 0.7, 0.8, 0.10]$ with split points $[0.45, 0.6, 0.75, 0.9]$, where we have computed an optimal tree for the split on $\tau = 0.75$ with two more misclassifications than the current best solution $\mathcal{UB}$, that is, $\Delta = 2$. Therefore, any possibly improving split needs to move at least two instances from the left to the right (or vice versa), which means that only the split point $\tau = 0.45$ can yield an improving solution.

Interval shrinking (IS) Interval shrinking is an extension of neighborhood pruning and acts whenever $\mathcal{UB}$ is updated. Given an interval $[i..j] \in Q$, IS searches for the largest threshold index $u \in \mathcal{V}$ smaller than i and the smallest threshold index $v \in \mathcal{V}$ larger than j, with $\mathcal{V}$ the set of split indices for which the misclassification score θ_u and θ_v are already computed. Using Cor. 1, IS then prunes the interval $[i..j]$ as illustrated in Fig. 1b. To search for these indices u and v, we define the function B that uses binary search over $\mathcal{V}$ with time complexity $\mathcal{O}(\log(m))$:

$$B([i..j], \mathcal{V}) = \left\langle \max_{u \in \mathcal{V}: u < i} u, \min_{v \in \mathcal{V}: v > j} v \right\rangle. \quad (10)$$

Additionally, IS uses the following theorem:

Theorem 2. *Let w by any split point with a solution θ_w with a left subtree misclassification score $\theta_{w,L}$ of zero. Then any split point $u < w$ will yield $\theta_u \geq \theta_w$. Similarly, if the right subtree misclassification score $\theta_{w,R}$ is zero, then for all $v > w$ also $\theta_v \geq \theta_w$.*

Proof. W.l.o.g., consider the left case. Since $u < w$, it holds that $\mathcal{D}(f \leq S^f_u) \subset \mathcal{D}(f \leq S^f_w)$ and $\mathcal{D}(f > S^f_u) \supset \mathcal{D}(f > S^f_w)$. Eq. (3) then implies that $\theta_{u,L} \leq \theta_{w,L} = 0$ and $\theta_{u,R} \geq \theta_{w,R}$. Thus $\theta_u = \theta_{u,R}$, $\theta_w = \theta_{w,R}$, and $\theta_u \geq \theta_w$. □

Let M_L and M_R represent the right-most (left-most) index with a zero left (right) misclassification score found so far, incremented (decremented) by one. By combining Cor. 1 and Theorem 2, we define:

$$\mathcal{P}_{\text{IS}}([i..j], u, v, \Delta_u, \Delta_v, M_L, M_R) = \\ [\max(i, M_L, \bar{A}(u, \Delta_u)), \min(j, M_R, \underline{A}(v, \Delta_v))], \quad (11)$$

where $\Delta_u = \theta_u - \mathcal{UB}$ and $\Delta_v = \theta_v - \mathcal{UB}$.

Sub-interval pruning (SP) Sub-interval pruning can prune an entire interval $[i..j]$ using the following theorem:

Theorem 3. *Let $[i..j]$ be any threshold index interval. Let θ_u and θ_v be optimal solutions for previously computed split points $u < i$ and $v > j$, with the corresponding left and right misclassification scores $\theta_{u,L}$, $\theta_{u,R}$, $\theta_{v,L}$, and $\theta_{v,R}$. Then, if $\theta_{u,L} + \theta_{v,R} > \mathcal{UB}$, any split point $w \in [i..j]$ cannot improve on $\mathcal{UB}$.*

Proof. $w \geq i > u$ and $w \leq j < v$ and thus $\theta_{w,L} \geq \theta_{u,L}$ and $\theta_{w,R} \geq \theta_{v,R}$. Therefore, $\theta_w = \theta_{w,L} + \theta_{w,R} > \mathcal{UB}$. □

From Theorem 3, we define:

$$\mathcal{P}_{SP}([i..j], \mathcal{UB}, \theta_{u,L}, \theta_{v,R}) = \\ \begin{cases} \emptyset, & \text{if } \theta_{u,L} + \theta_{v,R} > \mathcal{UB} \land u < i \land v > j \\ [i..j], & \text{otherwise}. \end{cases} \quad (12)$$

General Recursive Case

To eliminate the exploration of unnecessary splits in the recursion of Eq. (4), we use the pruning mechanisms presented above and keep track of upper bounds.

Algorithm 1: Branch$(\mathcal{D}, d, f, \mathcal{UB})$ uses the pruning techniques to find the optimal threshold for feature f.

$\theta_{\text{opt}} \leftarrow \min_{\hat{y} \in \mathcal{Y}} \sum_{(x,y) \in \mathcal{D}} \mathbb{1}(\hat{y} \neq y)$
$M_L \leftarrow 0, M_R \leftarrow m + 1$
$Q \leftarrow \{[1..m]\}, \mathcal{V} \leftarrow \emptyset$
while $|Q| > 0$ **do**
  $[i..j] \leftarrow Q.\text{pop}()$
  $u, v \leftarrow B([i..j], \mathcal{V})$
  $\Delta_u \leftarrow \theta_u - \mathcal{UB}, \Delta_v \leftarrow \theta_v - \mathcal{UB}$
  $[i..j] \leftarrow \mathcal{P}_{\text{IS}}([i..j], u, v, \Delta_u, \Delta_v, M_L, M_R)$
  $[i..j] \leftarrow \mathcal{P}_{\text{SP}}([i..j], \mathcal{UB}, \theta_{u,L}, \theta_{v,R})$
  if $|[i..j]| = 0$ **then continue**
  $w \leftarrow \lfloor \frac{i+j}{2} \rfloor$
  if $d = 2$ **then** $\theta_{w,L}, \theta_{w,R} \leftarrow \text{D2Split}(\mathcal{D}, f, w)$
  else
    $\mathcal{D}_L \leftarrow \mathcal{D}(f \leq S^f_w), \mathcal{D}_R \leftarrow \mathcal{D}(f > S^f_w)$
    $\theta_{w,L} \leftarrow \text{CT}(\mathcal{D}_L, d-1, \mathcal{UB})$
    $\eta \leftarrow \min(z(w) - z(i), z(j) - z(w))$
    $\mathcal{UB}_R \leftarrow \max(\mathcal{UB} - \theta_{w,L}, \eta)$
    if $\mathcal{UB}_R \leq 0$ **then** $\theta_{w,R} \leftarrow \theta_{\text{opt}} - \theta_{w,L}$
    else $\theta_{w,R} \leftarrow \text{CT}(\mathcal{D}_R, d-1, \mathcal{UB}_R)$
  $\theta_w \leftarrow \theta_{w,L} + \theta_{w,R}$
  if $\theta_{w,L} = 0$ **then** $M_L \leftarrow w + 1$
  if $\theta_{w,R} = 0 \land \mathcal{UB}_R > 0$ **then** $M_R \leftarrow w - 1$
  if $\theta_w < \theta_{\text{opt}}$ **then**
    $\mathcal{UB} \leftarrow \min(\mathcal{UB}, \theta_w), \theta_{\text{opt}} \leftarrow \theta_w$
    if $\theta_{\text{opt}} = 0$ **then break**
  $Q.\text{push}(\mathcal{P}_{\text{NB}}([i..j], w, \max(1, \theta_w - \mathcal{UB})))$
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{w\}$
return θ_{opt}

Interval pruning In each Branch call, described in Eq. (5), for each feature, we keep track of a set of intervals Q that may still contain the optimal split. We then choose a split point from one of these intervals, compute an optimal solution for this split, and then prune the search space according to the techniques mentioned above. When Q is empty, we have arrived at the optimal solution.

The pseudo-code of this procedure is shown in Alg. 1. It loops over the set of intervals $[i..j] \in Q$. Using Eq. (10), it finds $u, v \in \mathcal{V}$, the closest indices outside of $[i..j]$ for which the misclassification score is already computed. First, it attempts to reduce the interval using SP and IS. Then we need to select any point $w \in [i..j]$. If w is close to previously computed splits, only a little new information is gained. Therefore, we choose the midpoint $w = \lfloor \frac{i+j}{2} \rfloor$.

Then, if the remaining tree depth budget is two, we use a special subroutine which is explained in the subsection below. Otherwise, the dataset is split and a recursive CT call is made to get the optimal left subtree. This left solution is used to compute an upper bound for the right subtree. Only if the right upper bound indicates that an improving solution can still be found, is another recursive call made to get the right optimal subtree. If either the left or right subtree has zero misclassifications, the M_L or M_R indices are updated accordingly. If a better solution is found, the upper bound is

Algorithm 2: $\text{D2Split}(\mathcal{D}, f_1, w)$ finds splits of depth two more efficiently than the normal recursion.

$\theta_L \leftarrow |\mathcal{D}|, \theta_R \leftarrow |\mathcal{D}|$
$FQ^y_L \leftarrow 0 \quad \forall y \in \mathcal{Y}$
for $(x, y) \in \mathcal{D}(f \leq S^{f_1}_w)$ **do** $FQ^y_L \leftarrow FQ^y_L + 1$
$FQ^y_R \leftarrow FQ^y - FQ^y_L \quad \forall y \in \mathcal{Y}$
for $f_2 \in \mathcal{F}$ **do**
  $C^y_L \leftarrow 0, C^y_R \leftarrow 0 \quad \forall y \in \mathcal{Y}$
  for $(x, y) \in \mathcal{D}$ *sorted by* f_2 **do**
    if $x_{f_1} \leq S^{f_1}_w$ **then**
      $\theta_{LL} \leftarrow \min_{\hat{y}}(C^{\hat{y}}_L)$
      $\theta_{LR} \leftarrow \min_{\hat{y}}(FQ^{\hat{y}}_L - C^{\hat{y}}_L)$
      if $\theta_{LL} + \theta_{LR} \leq \theta_L$ **then**
        $\theta_L \leftarrow \theta_{LL} + \theta_{LR}$
      $C^y_L \leftarrow C^y_L + 1$
    else
      Same logic for instances going right
    if $\theta_L + \theta_R = 0$ **then break**
return θ_L, θ_R

updated. If a tree with zero misclassifications is found, the search is done. Finally, the remaining interval is divided into two using NB, both of which are added to Q, and the search continues with the next interval in Q.

Upper bounds If we set the upper bound for the right subtree tightly to the bound of what right solution can improve the current solution ($\mathcal{UB}_R \leftarrow \mathcal{UB} - \theta_{w,L}$), the right subproblem can be terminated early if no such solution exists. However, when doing so, we do not gain any information for pruning and we observed that this therefore typically decreases performance. On the other hand, no upper-bound-based pruning at all results in many unnecessary recursive calls. In this trade-off, we settled on a hybrid approach that in practice works well: the right upper bound $\mathcal{UB}_R$ is set to the maximum of $\mathcal{UB} - \theta_{w,L}$ and the length η of the longest side of the interval edges $z(i)$ and $z(j)$ to the midpoint $z(w)$.

Depth-Two Subroutine

To improve runtime, Demirović et al. (2022) introduced a specialized subroutine for trees of depth-two that is more efficient than doing recursive calls. However, it requires a quadratic amount of memory in terms of the number of binary features, which is problematic if we consider a binary feature for every possible threshold on continuous data.

Instead, we provide a specialized subroutine that simultaneously finds an optimal left and right subtree of a depth-two split and does not have this quadratic memory consumption by exploiting the fact that we can sort the observations by their feature values. Since splitting the data preserves the order, we sort the dataset once in the beginning. Then for the sorted data, Alg. 2 shows how an optimal depth-two tree can be found in $\mathcal{O}(|\mathcal{D}||\mathcal{F}|)$ for a given split point w on feature f_1 for the root node of the subtree. The core idea is that we first count with the variables FQ^y_L and FQ^y_R how many observations of class y go to the left and right by splitting at point w. Then, when deciding on the second splitting feature f_2 (of either the left or right subtree), we traverse all observations sorted by f_2 and incrementally keep track of the current counts per label C^y_L and C^y_R. Based on these two label counts, the label counts for all splitting thresholds on f_2 for all four leaf nodes of a depth-two tree can be determined as the dataset is traversed:

$$\begin{aligned} C^y_{LL} &= C^y_L\,, & C^y_{LR} &= FQ^y_L - C^y_L\,, \\ C^y_{RL} &= C^y_R\,, & C^y_{RR} &= FQ^y_R - C^y_R\,. \end{aligned} \tag{13}$$

Alg. 2 shows how the minimal misclassifications in the subtrees θ_{LL} and θ_{RR} can be computed based on these values.

Optimality Gap

We add a max-gap parameter that determines how far from optimal the final solution is allowed to be. This increases the pruning strength at the expense of optimality. For example, for NB, the distance Δ to a possibly improving split is now computed as $\Delta = \max(1, \theta_u - (\mathcal{UB} - \text{max-gap}))$. To use the gap parameter across multiple depths of the search, we set the max-gap for the current depth to half of the total. The other half is distributed evenly over the two subproblems. This allows a trade-off between training time and accuracy.

Comparison to Previous Bounding Methods

Mazumder, Meng, and Wang (2022) propose three lower bounds that require splitting the data into quantiles for a given feature. Their first lower bound is similar to our sub-interval pruning, but our definition is independent of the remaining depth budget. Their other lower bounds are tighter, but also more expensive to compute. Future work could investigate using such or similar lower bounds in ConTree.

The similarity-based lower bound (SLB) was proposed in previous work (Hu, Rudin, and Seltzer 2019; Lin et al. 2020; Demirović et al. 2022), but our application of the bound is more efficient. Lin et al. (2020) point out that the *similar support* bound in OSDT (Hu, Rudin, and Seltzer 2019) is too expensive to compute frequently. Therefore, they propose to use *hash trees* to identify similar subtrees but provide no further details. The implementation of their method, GOSDT, computes the difference between two subproblems by computing the xor of bit-vectors that represent the dataset corresponding to the subproblems. Demirović et al. (2022) loop once over the two sorted lists of identifiers of the datasets to count the differences. Both of these approaches require $\mathcal{O}(n)$ operations to compute the difference. ConTree, on the other hand, exploits the properties of sorted numeric feature data and computes the bound in $\mathcal{O}(1)$ by computing the difference between the two split indices. It applies the bound using the novel pruning techniques in $\mathcal{O}(\log(m))$.

5 Experiments

Our experiments aim to answer the following questions: 1) what is the effect of the pruning techniques and the depth-two subroutine; 2) how does ConTree's runtime compare to state-of-the-art ODT algorithms; and 3) what is ConTree's anytime performance? In the appendices B, C, and D, we additionally answer the following questions: 4) how does

Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{Y}\rvert$	OCT	RS-OCT	SAT-Shati	Quant-BnB	ConTree (No D2)	ConTree
Avila	10430	10	12	(100%)	(46%)	> 4h	2	63	**0.1**
Bank	1097	4	2	(31%)	177	62	0.5	0.1	**< 0.1**
Bean	10888	16	7	(100%)	(24%)	> 4h	4	199	**0.2**
Bidding	5056	9	2	(100%)	(67%)	672	0.9	1	**0.1**
Eeg	11984	14	2	OoM	(165%)	> 4h	4	178	**0.2**
Fault	1552	27	7	(100%)	(126%)	> 4h	2	240	**0.1**
Htru	14318	8	2	OoM	(301%)	> 4h	2	929	**0.2**
Magic	15216	10	2	(100%)	(88%)	> 4h	2	42	**0.3**
Occupancy	8143	5	2	(100%)	(9%)	355	0.9	2	**< 0.1**
Page	4378	10	5	(100%)	(136%)	2836	1	3	**< 0.1**
Raisin	720	7	2	(99%)	(15%)	485	0.7	1	**< 0.1**
Rice	3048	7	2	(100%)	(54%)	> 4h	1	24	**0.1**
Room	8103	16	4	(100%)	(88%)	4327	2	10	**< 0.1**
Segment	1848	18	7	(100%)	2896	442	2	54	**0.1**
Skin	196045	3	3	-	(43%)	> 4h	3	63	**0.1**
Wilt	4339	5	5	(100%)	(35%)	68	0.9	2	**< 0.1**

Table 1: Runtime (s) for optimizing depth-two trees of OCT, RS-OCT, SAT-Shati, Quant-BnB, and ConTree (with and without the depth-two subroutine). Runtimes are averaged over twenty runs. Values in parentheses show the optimality gap at time out. OoM means out of memory. '-' means the linear relaxation was unsolved at the time limit. Best results are in bold.

ConTree's memory usage compare to previous methods; 5) how well does ConTree scale to larger depth limits; and 6) how does ConTree's out-of-sample performance compare to CART and ODTs on binarized data?

The results show that our pruning techniques and depth-two subroutine make ConTree one or more orders of magnitude faster than the state-of-the-art optimal methods while using significantly less memory. Additionally, its out-of-sample performance is better than both CART and ODTs on binarized data. Finally, good solutions are often found early but much time is spent proving optimality.

Setup We have implemented ConTree in C++ and provide it as a Python package.[1] All computations were performed on an Intel Xeon E5-6448Y 32C 2.1GHz processor with 25GB RAM running Linux Red Hat Enterprise 8.10. We set a timeout of four hours. We evaluate on 16 datasets from the UCI repository (Dua and Graff 2017, see Appendix A). Unless specified otherwise, we run ConTree with its max-gap set to zero, thus yielding optimal solutions.

Pruning techniques Fig. 2 shows the impact of the neighborhood pruning (NB), interval shrinking (IS), and subinterval pruning (SP) techniques in comparison to the baseline with no pruning for trees of depth two. On average, all pruning techniques significantly prune the search space, with NB, IS, and SP respectively pruning on average 91.1%, 97.5%, and 99.6% of all D2Split calls, while retaining the optimal solution. SP generally provides the best results since it efficiently prunes entire intervals. Combining all three methods (NB+IS+SP) on average yields a 10% reduction of D2Split calls in comparison to only using SP.

Depth-two subroutine The last two columns of Table 1 show the performance of ConTree with the depth-two subroutine D2Split versus without (ConTree with "No D2") for computing depth-two trees. Both methods use all the pruning techniques. Averaged over twenty runs, the depth-two solver improves the computation time by a factor of 320 compared to the baseline (geometric mean).

[1]https://github.com/ConSol-Lab/contree.

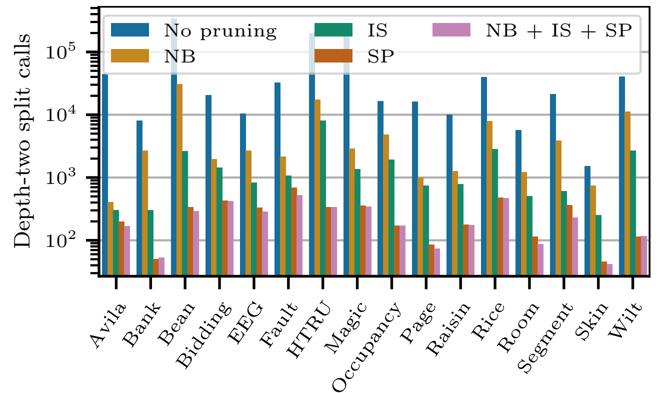

Figure 2: The number of D2Split calls for no pruning, the three pruning techniques, and all three combined.

Runtime Mazumder, Meng, and Wang (2022) compared Quant-BnB with BinOCT (Verwer and Zhang 2019), DL8.5 (Aglin, Nijssen, and Schaus 2020a), and MurTree (Demirović et al. 2022), each of which requires explicit binarization. Quant-BnB scales one or more orders of magnitude better than all of these methods when those are trained with binary features for every possible threshold. Additionally, they report that the optimal methods OCT (Bertsimas and Dunn 2017), GOSDT (Lin et al. 2020), FlowOCT, and BendersOCT (Aghaei, Gómez, and Vayanos 2024) cannot solve any of their datasets within a four-hour time limit.

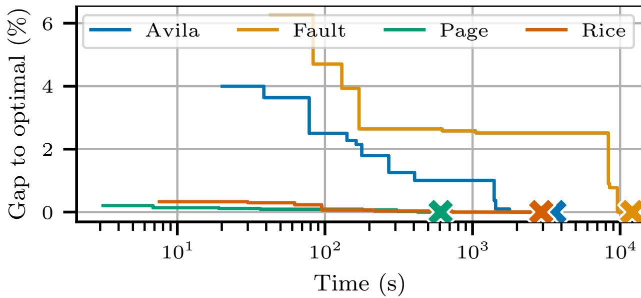

Figure 3: The distance to the optimal solution for ConTree's best depth-four solution over time for three datasets. The optimal solution is typically found significantly earlier than the end of the search (the final cross).

We compare ConTree to the MIP methods OCT and RS-OCT (Hua, Ren, and Cao 2022), the SAT method by Shati, Cohen, and McIlraith (2023), and Quant-BnB (Mazumder, Meng, and Wang 2022).[2] Each of these methods optimizes ODTs directly on the numeric feature data. We initialize both OCT and RS-OCT with a warm start from CART and set the node cost to zero. Both OCT and RS-OCT could use up to eight threads. OCT is solved with Gurobi 9.5.2. Each method is run twenty times on each dataset, except when it exceeds the four-hour time-out. For the MIP methods, we report the optimality gap at time-out.

Table 1 shows the runtime results for optimizing trees of depth two. Even after four hours, the MIP methods typically have a large optimality gap remaining. OCT ran out of memory twice and once did not solve the linear relaxation before the time out. The SAT approach performs better but also hits the time-out for seven datasets. Quant-BnB and ConTree, on the other hand, run in the (sub-)second range. Therefore, we further compare Quant-BnB and ConTree.

Table 2 shows that for depth three, ConTree outperforms Quant-BnB on average by a factor 63 (geometric mean), ranging from 7 times faster for Bidding up to 266 times faster for Fault. The comparisons of Quant-BnB with BinOCT, DL8.5, MurTree, GOSDT, OCT, FlowOCT, and BendersOCT by Mazumder, Meng, and Wang (2022) combined with our new runtime results show that ConTree outperforms state-of-the-art optimal methods by one or more orders of magnitude.

Quant-BnB's implementation only permits training trees up to depth three, so for depth four, we report only ConTree's performance. For all but four datasets, ConTree finds the optimal depth-four tree within the time limit. Appendix B further shows that for depth five and six, ConTree finds and proves optimal solutions within the time limit for eight and seven datasets respectively.

The last column in Table 2 shows the runtime ConTree requires to find a depth-four tree that is provably within 1% of the optimal solution (with max-gap $= \lfloor 0.01|\mathcal{D}| \rfloor$). This significantly reduces ConTree's runtime and allows a user to make a trade-off between training accuracy and runtime.

[2] https://github.com/LucasBoTang/Optimal_Classification_Trees, https://github.com/YankaiGroup/optimal_decision_tree, https://github.com/HarisRasul12/ESC499-Thesis-SAT-Trees, https://github.com/mengxianglgal/Quant-BnB.

	Depth = 3		Depth = 4	
Dataset	Quant-BnB	ConTree	ConTree	$\leq 1\%$ Gap
Avila	4451	24	4195	3237
Bank	2	< 0.1	< 0.1	< 0.1
Bean	583	61	> 4h	3640
Bidding	15	2	5	< 0.1
Eeg	8535	136	> 4h	> 4h
Fault	> 4h	55	12331	8592
Htru	13147	74	> 4h	191
Magic	1419	60	> 4h	5719
Occupancy	76	0.4	17	0.1
Page	388	2	499	63
Raisin	65	0.5	65	27
Rice	817	12	2215	231
Room	92	1	44	0.1
Segment	64	2	191	75
Skin	218	10	211	5
Wilt	26	0.1	0.3	< 0.1

Table 2: Runtime (s) comparison between Quant-BnB and ConTree. Averaged over twenty runs. For depth three, ConTree is on average one or two orders of magnitude faster than Quant-BnB. ConTree's runtime can be significantly reduced by setting a permissible optimality gap.

Anytime performance Fig. 3 shows the best solution found by ConTree at any time during a depth-four search, expressed as the distance to the optimal solution for the Avila, Fault, Page, and Rice datasets. For the Fault dataset, a final improving solution is found late after the start of the search. In the other three cases, a good solution is found early and most of the time is spent on proving optimality.

6 Conclusion

We introduce ConTree, a dynamic programming and branch-and-bound algorithm that outperforms state-of-the-art algorithms for training optimal classification trees with continuous features by one or more orders of magnitude. This result is obtained through three novel pruning techniques that on average prune 99.6% of recursive calls, and a depth-two subroutine that computes splits 320 times faster than a naive approach. Moreover, ConTree obtains an out-of-sample accuracy for depth-three trees 5% higher than CART and 0.7% higher than optimal decision trees trained with a coarse binarization. These results enable the application of optimal decision trees for real-world scenarios.

Future work could add a node cost or node limit to further encourage sparsity and extend ConTree to regression by using bounds from Zhang et al. (2023) and Van den Bos, Van der Linden, and Demirović (2024). Finally, branching on the power set of categorical variables as done by Shati, Cohen, and McIlraith (2023), could be explored.

References

Aghaei, S.; Gómez, A.; and Vayanos, P. 2024. Strong Optimal Classification Trees. *Operations Research*.

Aglin, G.; Nijssen, S.; and Schaus, P. 2020a. Learning Optimal Decision Trees Using Caching Branch-and-Bound Search. In *Proceedings of AAAI-20*, 3146–3153.

Aglin, G.; Nijssen, S.; and Schaus, P. 2020b. PyDL8.5: a Library for Learning Optimal Decision Trees. In *Proceedings of IJCAI-20*, 5222–5224.

Alès, Z.; Huré, V.; and Lambert, A. 2024. New optimization models for optimal classification trees. *Computers & Operations Research*, 164: 106515.

Alòs, J.; Ansótegui, C.; and Torres, E. 2023. Interpretable decision trees through MaxSAT. *Artificial Intelligence Review*, 56(8): 8303–8323.

Arrieta, A. B.; Díaz-Rodríguez, N.; Del Ser, J.; Bennetot, A.; Tabik, S.; Barbado, A.; García, S.; Gil-López, S.; Molina, D.; Benjamins, R.; Chatila, R.; and Herrera, F. 2020. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58: 82–115.

Avellaneda, F. 2020. Efficient Inference of Optimal Decision Trees. In *Proceedings of AAAI-20*, 3195–3202.

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

Bhatt, R.; and Dhall, A. 2012. Skin Segmentation Dataset. UCI Machine Learning Repository.

Bock, R. 2007. MAGIC Gamma Telescope Dataset. UCI Machine Learning Repository.

Breiman, L.; Friedman, J. H.; Olshen, R. A.; and Stone, C. J. 1984. *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks.

Brodley, C. 1990. Image Segmentation Dataset. UCI Machine Learning Repository.

Buscema, M.; Terzi, S.; and Tastle, W. 2010. Steel Plates Faults Dataset. UCI Machine Learning Repository.

Candanedo, L. M. I.; and Feldheim, V. 2016. Accurate occupancy detection of an office room from light, temperature, humidity and CO2 measurements using statistical learning models. *Energy and Buildings*, 112: 28–39.

Cinar, I.; and Koklu, M. 2019. Classification of rice varieties using artificial intelligence methods. *International Journal of Intelligent Systems and Applications in Engineering*, 7(3): 188–194.

Çınar, İ.; Koklu, M.; and Taşdemir, Ş. 2020. Classification of raisin grains using machine vision and artificial intelligence methods. *Gazi Journal of Engineering Sciences*, 6(3): 200–209.

Costa, V. G.; and Pedreira, C. E. 2023. Recent Advances in Decision Trees: An Updated Survey. *Artificial Intelligence Review*, 56: 4765–4800.

Demirović, E.; Hebrard, E.; and Jean, L. 2023. Blossom: an Anytime Algorithm for Computing Optimal Decision Trees. In *Proceedings of ICML-23*, 7533–7562.

Demirović, E.; Lukina, A.; Hebrard, E.; Chan, J.; Bailey, J.; Leckie, C.; Ramamohanarao, K.; and Stuckey, P. J. 2022. MurTree: Optimal Classification Trees via Dynamic Programming and Search. *Journal of Machine Learning Research*, 23(26): 1–47.

Dua, D.; and Graff, C. 2017. UCI Machine Learning Repository.

eBay. 2020. Shill Bidding Dataset. UCI Machine Learning Repository.

Günlük, O.; Kalagnanam, J.; Li, M.; Menickelly, M.; and Scheinberg, K. 2021. Optimal Decision Trees for Categorical Data via Integer Programming. *Journal of Global Optimization*, 81: 233–260.

Hu, H.; Siala, M.; Hebrard, E.; and Huguet, M.-J. 2020. Learning Optimal Decision Trees with MaxSAT and its Integration in AdaBoost. In *IJCAI-PRICAI 2020*, 1170–1176.

Hu, X.; Rudin, C.; and Seltzer, M. 2019. Optimal Sparse Decision Trees. In *Advances in NeurIPS-19*, 7267–7275.

Hua, K.; Ren, J.; and Cao, Y. 2022. A Scalable Deterministic Global Optimization Algorithm for Training Optimal Decision Tree. In *Advances in NeurIPS-22*, 8347–8359.

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

Janota, M.; and Morgado, A. 2020. SAT-Based Encodings for Optimal Decision Trees with Explicit Paths. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT 2020)*, 501–518.

Johnson, B. 2014. Wilt Dataset. UCI Machine Learning Repository.

Kiossou, H.; Schaus, P.; Nijssen, S.; and Houndji, V. R. 2022. Time constrained DL8.5 using Limited Discrepancy Search. In *Proceedings of ECML-PKDD-22*, 443–459.

Koklu, M.; and Ozkan, I. A. 2020. Multiclass classification of dry beans using computer vision and machine learning techniques. *Computers and Electronics in Agriculture*, 174: 105507.

Lin, J.; Zhong, C.; Hu, D.; Rudin, C.; and Seltzer, M. 2020. Generalized and Scalable Optimal Sparse Decision Trees. In *Proceedings of ICML-20*, 6150–6160.

Liu, E.; Hu, T.; Allen, T. T.; and Hermes, C. 2024. Optimal classification trees with leaf-branch and binary constraints. *Computers & Operations Research*, 166: 106629.

Lohweg, V. 2013. Banknote Authentication Dataset. UCI Machine Learning Repository.

Lyon, R. J.; Stappers, B. W.; Cooper, S.; Brooke, J. M.; and Knowles, J. D. 2016. Fifty years of pulsar candidate selection: from simple filters to a new principled real-time classification approach. *Monthly Notices of the Royal Astronomical Society*, 459(1): 1104–1123.

Malerba, D. 1995. Page Blocks Classification Dataset. UCI Machine Learning Repository.

Mazumder, R.; Meng, X.; and Wang, H. 2022. Quant-BnB: A Scalable Branch-and-Bound Method for Optimal Decision Trees with Continuous Features. In *Proceedings of ICML-22*, 15255–15277.

Murthy, S. K.; and Salzberg, S. 1995. Decision Tree Induction: How Effective Is the Greedy Heuristic? In *Proceedings of KDD-95*, 222–227.

Narodytska, N.; Ignatiev, A.; Pereira, F.; and Marques-Silva, J. 2018. Learning Optimal Decision Trees with SAT. In *Proceedings of IJCAI-18*, 1362–1368.

Nijssen, S.; and Fromont, E. 2007. Mining Optimal Decision Trees from Itemset Lattices. In *Proceedings of SIGKDD-07*, 530–539.

Nijssen, S.; and Fromont, E. 2010. Optimal constraint-based decision tree induction from itemset lattices. *Data Mining and Knowledge Discovery*, 21(1): 9–51.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Roesler, O. 2013. EEG Eye State Dataset. UCI Machine Learning Repository.

Rudin, C. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5): 206–215.

Shati, P.; Cohen, E.; and McIlraith, S. 2021. SAT-Based Approach for Learning Optimal Decision Trees with Non-Binary Features. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP-2021)*, 50:1–50:16.

Shati, P.; Cohen, E.; and McIlraith, S. A. 2023. SAT-based optimal classification trees for non-binary data. *Constraints*, 28(2): 166–202.

Singh, A. P.; Jain, V.; Chaudhari, S.; Kraemer, F. A.; Werner, S.; and Garg, V. 2018. Machine learning-based occupancy estimation using multivariate sensor nodes. In *2018 IEEE Globecom Workshops*, 1–6.

Stefano, C.; Fontanella, F.; Maniaci, M.; and Freca, A. 2018. Avila Dataset. UCI Machine Learning Repository.

Van den Bos, M.; Van der Linden, J. G. M.; and Demirović, E. 2024. Piecewise Constant and Linear Regression Trees: An Optimal Dynamic Programming Approach. In *Proceedings of ICML-24*.

Van der Linden, J. G. M.; De Weerdt, M. M.; and Demirović, E. 2023. Necessary and Sufficient Conditions for Optimal Decision Trees using Dynamic Programming. In *Advances in NeurIPS-23*, 9173–9212.

Van der Linden, J. G. M.; Vos, D.; De Weerdt, M. M.; Verwer, S.; and Demirović, E. 2024. Optimal or Greedy Decision Trees? Revisiting their Objectives, Tuning, and Performance. *arXiv preprint arXiv:2409.12788*.

Verhaeghe, H.; Nijssen, S.; Pesant, G.; Quimper, C.-G.; and Schaus, P. 2020. Learning Optimal Decision Trees using Constraint Programming. *Constraints*, 25(3): 226–250.

Verwer, S.; and Zhang, Y. 2017. Learning decision trees with flexible constraints and objectives using integer optimization. In *Proceedings of CPAIOR-17*, 94–103.

Verwer, S.; and Zhang, Y. 2019. Learning Optimal Classification Trees Using a Binary Linear Program Formulation. In *Proceedings of AAAI-19*, 1625–1632.

Zhang, R.; Xin, R.; Seltzer, M.; and Rudin, C. 2023. Optimal Sparse Regression Trees. In *Proceedings of AAAI-23*, 11270–11279.