Efficient Lookahead Decision Trees

Harold Kiossou[0000−0001−6972−9885], Pierre Schaus[0000−0002−3153−8941], Siegfried Nijssen[0000−0003−2678−1266], and Gaël Aglin[0000−0002−6760−4752]

UCLouvain, Louvain-la-Neuve, Belgium
first.last@uclouvain.be

Abstract. Conventionally, decision trees are learned using a greedy approach, beginning at the root and moving toward the leaves. At each internal node, the feature that yields the best data split is chosen based on a metric like information gain. This process can be regarded as evaluating the quality of the best depth-one subtree. To address the shortsightedness of this method, one can generalize it to greater depths. Lookahead trees have demonstrated strong performance in situations with high feature interaction or low signal-to-noise ratios. They constitute a good trade-off between optimal decision trees and purely greedy decision trees. Currently, there are no readily available tools for constructing these lookahead trees, and their computational cost can be significantly higher than that of purely greedy ones. In this study, we introduce an efficient implementation of lookahead decision trees, specifically LGDT, by adapting a recently introduced algorithmic concept from the MurTree approach to find optimal decision trees of depth two. Additionally, we utilize an efficient reversible sparse bitset data structure to store the filtered examples while expanding the tree nodes in a depth-first-search manner. Experiments on state-of-the-art datasets demonstrate that our implementation offers remarkable computation-time performance.

Keywords: Decision Trees · Lookahead · Optimization.

1 Introduction

Decision tree-based learning algorithms are among the most widely used methods in machine learning, both for their predictive performance and because humans can relatively easily interpret these models. Finding an optimal tree (that minimizes the learning error) on a data set is an NP-hard problem. This is why learning a decision tree is usually done greedily, for example, using algorithms such as CART [5] and C4.5 [18]. A greedy algorithm selects an attribute in a node locally, starting from the root, down to the leaf nodes. The decision to select an attribute is made based on a single split and is never reconsidered later. In recent years, thanks to optimization solvers and new algorithmic ideas, approaches to infer exact decision trees have been introduced [1–3, 8, 14, 16, 20, 21].

These approaches have sparked excitement in the scientific community due to empirical evidence showing improved classification rates on unseen data [3].

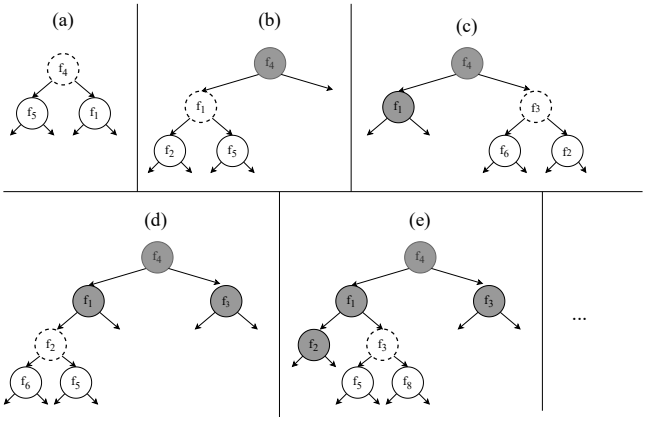


Fig. 1: Illustration of iterations to discover the feature in each node using level-two optimal decision trees. The level-two decision trees are rooted at dashed white nodes, while the fixed features are represented in dark-gray nodes. The tree is grown level-wise in this example. However, building it in a depth-first search order would result in the same final decision tree since each decision splits the data into two separate subproblems.

Despite recent algorithmic enhancements, they struggle with inferring trees on medium-sized datasets within reasonable computational times (e.g., German-credit dataset [2]). To address the limitations of both greedy and optimal decision trees, lookahead decision trees present a potential solution. These trees aim to overcome greedy decision tree shortcomings by considering future decisions during construction. A lookahead algorithm determines the next feature's decision in a less myopic, yet still greedy, manner based on sliding subtrees, as illustrated in Figure 1 with a depth of two. Standard approaches like C4.5 use heuristics to decide each node's best one-level subtree or stump [13]. This concept can be extended to more than one level in lookahead decision trees [11]. Here, the attribute selected at a node corresponds to the root of the optimal or suboptimal two-level decision tree before moving to the next level. This allows the algorithm to anticipate the next step and make a more informed choice regarding the subsequent attribute. Setting the lookahead level to the maximum depth enables the identification of the optimal decision tree.

Despite the potential benefits of using lookahead approaches their adoption has been limited due to various factors. One of the primary concerns is the additional computation cost associated with lookahead, which requires substantial processing power and memory resources. Furthermore, while some studies have reported better performance using lookahead compared to greedy methods, there is no consistent evidence that lookahead consistently outperforms greedy counterparts across a range of applications. [11] argued that lookahead is more advantageous when there is high attribute interaction which was later confirmed and reinforced by [10], whose work demonstrated that the superior performance

of lookahead decision trees is more pronounced when nonlinear relationships between feature pairs exist and when the signal-to-noise ratio is particularly low.

The enhanced capability of lookahead decision trees in handling feature interdependencies can result in more accurate predictive models. Despite their potential, there's a lack of widely available tools for researchers and practitioners to experiment with these methods. To address this, we introduce an efficient lookahead algorithm with remarkable computation runtimes. Our two-level lookahead approach achieves competitive error rates comparable to optimal trees, scaling to larger depths similar to standard greedy decision tree algorithms like CART or C4.5. We leverage an algorithmic concept from the MurTree algorithm [8] to identify optimal level-two decisions, adapting it for both information gain and misclassification error rate. Experiments on standard benchmarks show that this hybrid, less greedy approach strikes an excellent balance between tree learning speed and error rate. The paper structure includes a discussion of related work in Section 2, technical background in Section 3, an exploration of our methods in Section 4, and presentation of benchmark results in Section 5.

2 Related Work

Tree-based algorithms like CART and C4.5 often rely on heuristics for creating greedy decision trees, where attributes are chosen based on metrics like information gain or the Gini index. Despite their scalability, these trees may lack accuracy due to their myopic nature. To address this, lookahead searches, aim at optimizing upcoming iterations rather than just the next one. Norton [17] and Ragavan and Rendell [19] demonstrated successful results with lookahead, with the latter excelling in high attribute interaction scenarios at the cost of increased computational complexity. Esmeir and Markovitch [11] introduced ID3-k and LSID3 as lookahead strategies. ID3-k calculates k-level information gain (gain-k) at each node, selecting the attribute maximizing it for further splits. LSID3, using dynamic lookahead and a shallower tree-favoring criterion, surpasses greedy algorithms, especially with more available time. Relying on this, Donick et al.[10] introduced a random forest approach that is a stepwise lookahead variation. It considers three split nodes simultaneously in tiers of depth two, enhancing the identification of feature interdependencies. The lookahead algorithm outperforms the greedy algorithm in cases involving non-linear relationships between feature pairs and a low signal-to-noise ratio.

Thanks to hardware advancements and optimization solvers, various approaches for learning exact decision trees have emerged. These fall into categories like mixed integer programming [1, 3, 4, 21], constraint programming [20], SAT solvers [15], and dynamic programming [2, 8, 16]. Among these, dynamic programming methods, like DL8 [16], are considered the fastest and most accurate in a depth-constrained setting. DL8 uses a caching technique to save obtained subtrees, optimizing performance. DL8.5 [2] enhances DL8 with an upper-bound strategy and a lower-bound technique to efficiently explore sub search spaces. Further advancements by MurTree [8] include limiting tree nodes, an efficient

depth-two tree computation, and a novel similarity-based lower bounding approach.

3 Technical Background

We consider binary datasets in which each feature is a value in the set $\{0, 1\}$. Let $\mathcal{F}$ be the set of n features and $\mathcal{C} = \{+, -\}$ the set of two classes [1]. A binary dataset is defined by: $\mathcal{D} = \{(\boldsymbol{f}_i, c_i), \forall i \in \{1, 2, \ldots |D|\}\}$, where $\boldsymbol{f}_i = (f_{(i,1)}, f_{(i,2)}, \ldots, f_{(i,n)}) \in \{0, 1\}^n$ is a feature vector of length n and $c_i \in \mathcal{C}$ is the class label for the i-th instance. Thus, each instance in $\mathcal{D}$ is represented by a feature vector and a corresponding class label, and the feature vectors are composed of elements from the set of features $\mathcal{F}$. Given a feature vector $\boldsymbol{f}$, $f_k = 1$, indicating the presence of f_k, is denoted as $f_k \in \boldsymbol{f}$ and $\bar{f}_k \notin \boldsymbol{f}$ otherwise for its absence (we ignore i for simplicity). The binary dataset $\mathcal{D}$ can be partitioned into the positive class of instances $\mathcal{D}^+$ and the negative class $\mathcal{D}^-$ such that $\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^-$. The MurTree algorithm [8] for finding optimal decision trees of arbitrary depth uses a specialized algorithm for level-two decision trees whenever it reaches the one level before the last layer (the final trees have a fixed known limit on depth). This specialized level-two exact algorithm is presented next for the sake of completeness.

3.1 Level-two specialized algorithm

MurTree uses a dynamic programming approach with the upper-bound notion of DL8.5 to determine the optimal decision trees. It works in two phases. In the first phase, it computes the frequency for each pair of features (f_i, f_j). Let $FQ^+(f_i)$ and $FQ^+(f_i, f_j)$ be the frequency counts in positive instances for a single feature f_i and a pair of features (f_i, f_j), respectively. $FQ^-(f_i)$ and $FQ^-(f_i, f_j)$ are defined analogously for negative instances. Note that, based on $FQ(f_i)$ and $FQ(f_i, f_j)$, it is possible to compute $FQ(\bar{f}_i)$, $FQ(f_i, \bar{f}_j)$, $FQ(\bar{f}_i, f_j)$, and $FQ(\bar{f}_i, \bar{f}_j)$ without explicit counting. The frequency counts FQ^+ can be computed in $\mathcal{O}(|\mathcal{D}+| \cdot m_+^2)$ time with m_+ the maximum number of features in a single positive instance [8]. In the second phase, the tree is computed using the missclassification score $MS(f_i, f_j) = \min\{FQ^+(f_i, f_j), FQ^-(f_i, f_j)\}$ for a couple of features (f_i, f_j). Given a depth two decision tree, let MS_{left} and MS_{right} denote the misclassification scores of the left and right subtrees:

$$MS_{left}(f_{root}, f_{left}) = MS(\bar{f}_{root}, \bar{f}_{left}) + MS(\bar{f}_{root}, f_{left}) \tag{1}$$

$$MS_{right}(f_{root}, f_{right}) = MS(f_{root}, \bar{f}_{right}) + MS(f_{root}, f_{right}), \tag{2}$$

where f_{root} is the feature of the root node, f_{left} and f_{right} are the features of the left and right-child nodes.

[1] All the formula of this section can easily be adapted for multi-class contexts.

The procedure then iterates over all pairs of (f_i, f_j) of $\mathcal{F}$ to find the triplet $(f_{root}, f_{left}, f_{right})$ independently optimized for the left and right branches using the following equation:

$$\begin{aligned} &\min_{f_{root}, f_{left}, f_{right} \in \mathcal{F}} MS(f_{root}, f_{left}, f_{right}) \\ &= \min_{f_{root}} [\min_{f_{left} \in \mathcal{F}} MS_{left}(f_{root}, f_{left}) \\ &+ \min_{f_{right} \in \mathcal{F}} MS_{right}(f_{root}, f_{right})]. \end{aligned} \tag{3}$$

The algorithm iterates through each pair (f_{root}, f_{child}), computes the misclassification score of the left subtree using Equation (1), updates the best left child for the feature f_{root}, and performs the same operation for the right child. Each subtree can be computed in $\mathcal{O}(|\mathcal{F}^2|)$ time. The global complexity of the algorithm is then $\mathcal{O}(|\mathcal{D}| \cdot m^2 + |\mathcal{F}^2|)$ with m the upper limit on the number of features in any single positive and negative instance.

3.2 Level-two lookahead Information Gain.

In our algorithm, we can also use a level-two lookahead on information gain. This is based on the work of [11], where they developed ID3-k, which uses a level-k lookahead for information gain at each node to evaluate each feature. To fully take advantage of the level-two specialized algorithm, we only do a lookahead of two as it is easy to compute the different frequencies using $FQ^+(f_i)$ and $FQ^+(f_i, f_j)$ and $FQ^-(f_i)$ and $FQ^-(f_i, f_j)$. The information gain is then computed using the following equation (4):

$$\begin{aligned} &\max_{f_{root}, f_{left}, f_{right} \in \mathcal{F}} IG(f_{root}, f_{left}, f_{right}) \\ &= \max_{f_{root}} [\max_{f_{left} \in \mathcal{F}} IG_{left}(f_{root}, f_{left}) \\ &+ \max_{f_{right} \in \mathcal{F}} IG_{right}(f_{root}, f_{right})], \end{aligned} \tag{4}$$

with:

$$IG(f_{root}, f_{left}) = H(f_{root}) - \sum_{i=1}^{2} \frac{|leaf(f_{root}, f_{left})_i|}{|root|} H(leaf(f_{root}, f_{left})_i), \tag{5}$$

$$IG(f_{root}, f_{right}) = H(f_{root}) - \sum_{i=1}^{2} \frac{|leaf(f_{root}, f_{right})_i|}{|root|} H(leaf(f_{root}, f_{right})_i), \tag{6}$$

and

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i), \tag{7}$$

where $IG(f_{root}, f_{child})$ is the depth two information gain of a branch from f_{root} to f_{child} and $child \in \{left, right\}$. $leaf(f_{root}, f_{child})_i$ corresponds to the leaves of branches (f_{root}, f_{child}), $H(S)$ the entropy of a node or leaf S with p_i the probability that an element belongs to the class i. We independently maximize the information gain of each branch of the tree to obtain a tree with the highest information gain.

4 Less Greedy Decision Trees

Algorithm 1: LGDT(D, *minsup*, *maxdepth*)

```
1  if maxdepth ≤ 2 then return sliding_window(D, maxdepth)
2  tree ← Tree()
3  sub_tree ← sliding_window(D, 2)
4  tree.root ← make_tree(sub_tree.root)
5  Recursion(tree.root, maxdepth − 1, D)
6  return solution
7  Procedure Recursion(node, depth, D)
8      if depth > 0 and node.error > 0 then
9          left ← node.left
10         right ← node.right
11         if FQ(left) ≥ minsup and FQ(right) ≥ minsup then
12             ws ← min(2, depth)
13             D.save()
14             D.project(left)
15             sub_tree ← sliding_window(D, ws)
16             node_tree ← make_tree(sub_tree.root)
17             child ← tree.set_node(left, node_tree)
18             Recursion(child, depth − 1, D)
19             D.restore()
20             D.save()
21             D.project(right)
22             sub_tree ← sliding_window(D, ws)
23             node_tree ← make_tree(sub_tree.root)
24             child ← tree.set_node(right, node_tree)
25             Recursion(child, depth − 1, D)
26             D.restore()
```

This section presents Less Greedy Decision Trees (LGDT), a more informed decision tree algorithm than classical greedy algorithms. The pseudocode of LGDT is described in Algorithm 1.

It fixes the feature of each node of the depth-limited tree in a depth-first way. In each node, it relies on the use of a generated depth-two decision tree

based on the current data as a sliding window (lines 1, 3, 15 and 22). The sliding window uses D the current data subset in a node, then computes FQ^+ and FQ^- and uses both to build a level-two decision tree based on one of the following approaches:

- MurTree level-two specialized algorithm which returns an optimal decision tree minimizing the misclassification rate;
- a depth-two information gain tree taking advantage of the level-two specialized algorithm.

When the maximum depth is 1 the sliding window returns the depth-one tree with the feature optimizing the evaluated metric. At first, when building up to a depth-two tree, it is enough to return the tree generated by the sliding window (line 1). On the other hand, for deeper trees, a subtree is generated using D and the sliding window (line 3). The root node of the solution tree is set to the root of the subtree (line 4). The `make_tree` function generates for the feature and the data its two leaves and computes the misclassification error. `Recursion` is then called in a depth-first search fashion starting from the root node to build the decision tree. An internal node is only refined if the depth constraint is respected and the node is not pure (line 8). Otherwise, the recursion is stopped, and the node is considered as a leaf node. Moreover, before expanding a node the search ensured the number of data falling in its left and right branches (9-10) respects the minimum support constraint at line 11.

The algorithm will then be called recursively on the left and right parts. Lines 14 and 21 will update the data representation D to be able to list all examples falling respectively in $left$ and $right$. Using the updated D the sliding window will build a subtree for the left and right (lines 15 and 22). The algorithm will append the sub-node trees (lines 16 and 23) to the parent node (lines 17 and 24). The recursive method is then called on the child with the current data representation D and an increment to the depth (lines 18 and 25).

In our implementation, we use a special data representation to efficiently store and process the input data. By using this representation, we aim to improve the performance and scalability of the algorithm, while reducing the memory consumption allowing us to efficiently generate decision trees for large datasets.

Data representation. Table 1 summarizes various dataset representations. Each example, identified by a unique *tid*, is linked to a feature set (Feats). The *tid-list* groups tids into example subsets. The boolean representation (Table 1a) uses $0s$ and $1s$ to show feature presence (1) or absence (0) per example. The horizontal representation (Table 1b) records present features. The vertical representation [12] (Table 1c) aligns rows with features, simplifying *tid-list* length computation. Intersection operations on *tid-lists* [12] locate transactions covered by two features, e.g., A, C arises from $t_2, t_3 \cap t_1, t_2, t_3 = t_2, t_3$.

We use intersection operations to significantly reduce data processing, saving time and memory. In this work, using the vertical data representation enhances algorithm efficiency, and the reversible sparse bitset structure reduces unnecessary computation and enables fast bitwise operations.

Table 1: Dataset representations.

(a) Boolean.

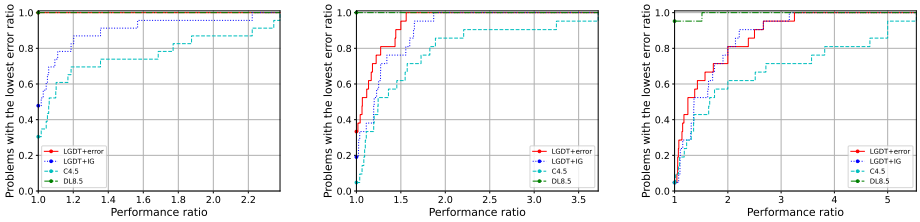
$\mathbf{t_{id}}$	**A**	**B**	**C**
t_1	0	0	1
t_2	1	0	1
t_3	1	1	1

(b) Horizontal.

$\mathbf{t_{id}}$	**Feats**
t_1	{C}
t_2	{A, C}
t_3	{A, B, C}

(c) Vertical.

Feats	**tid-list**
A	t_2, t_3
B	t_3
C	t_1, t_2, t_3



(a) $max\_depth = 2$ (b) $max\_depth = 3$ (c) $max\_depth = 4$

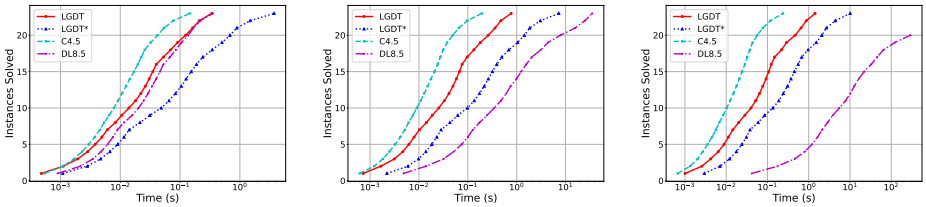
Fig. 2: Performance Profile plots comparing the error rate of the two versions of LGDT against C4.5 and DL8.5.

Bitsets and Bitwise operations. In the vertical representation, the feature *tid-list* can be stored using arrays or bitsets. Arrays group all integer values (*tid*) associated with a feature, while bitsets use bits to represent each possible *tid* value. A bitset is the size of the dataset ($|D|$), with each bit at index i indicating the presence (1) or absence (0) of the example at $tid = i$ in the feature *tid-list*. For a dataset with eight examples ($tid \in \Omega = \{1, 2, \ldots, 8\}$), an array retains valid values post-operations, while a bitset maintains the same size, setting unnecessary values to 0.

When dealing with large sets, bitsets are more memory-efficient than arrays due to each integer in an array requiring at least 32 or 64 bits, while a bitset needs just 1 bit per integer. Bitsets are especially useful in frequent itemset mining algorithms [6], where bitwise operations like counting and intersection are essential and the number of elements important than the data. These operations are often optimized for 64-bit processing, making bitsets ideal for storing parameters that require such operations. As a result, many data structures use an array of 64-bit bitsets, commonly referred to as *words*, to take advantage of these optimizations.

Reversible Sparse Bitset. The decision tree undergoes expansion via a recursive depth-first search . After data operations such as the `project` in Algorithm 1, the number of elements in the bitset decreases resulting in a sparser bitset. Elements are restored on backtracking from the recursive calls.

The Reversible Sparse Bitset (RSBS) [7] exploits bitset sparsity down the tree and is able to efficiently restore element during backtracking. Operations are per-



(a) $max\_depth = 2$ (b) $max\_depth = 3$ (c) $max\_depth = 4$

Fig. 3: Cumulative number of dataset (y-axis) for which the decision tree algorithm has terminated within the time limit (x-axis).

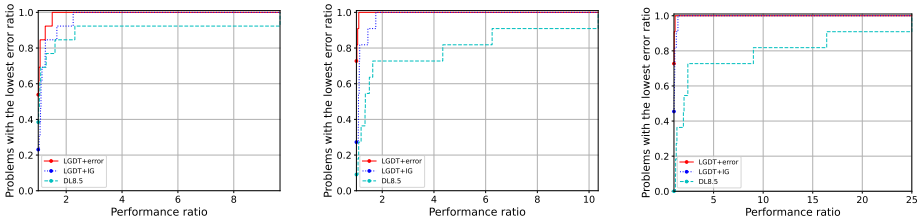
formed per 64-bit word for improved performance. RSBS employs sparse bitsets, separating empty from non-empty words, eliminating unnecessary counting of empty words. This sparsity aids intersection operations, ensuring unnecessary intersections with a 0 bit are avoided. Moreover RSBS employs a reversibility technique from constraint programming solvers, enabling the data structure to recover previous states during searches, thus avoiding the overhead of copying parameters between parent and child nodes.

When backtracking, the RSBS can revert to a previous state using its trail of changes, which is implemented with a stack. In practice, only the size of the number of non-empty words needs to be restored to retrieve the correct partitioning between empty and non-empty words. Throughout the search, the entire algorithm operates with a single instance of the data structure, facilitating incremental changes at each step and ensuring consistency when exploring multiple paths. The stack chronicles successive changes, and during backtracking, the top layers are removed to revert to prior states.

LGDT uses a single instance of the RSBS(D) to maintain the data in the current node. Before going further down, the state is saved (lines 13 and 20) and the state bit vector is projected (using bitwise AND operations) with the next node feature (a fixed precomputed bit vector) (lines 14 and 21). Each projection reduces the dataset by filtering the examples that satisfy the condition of the selected feature. Before exploring a node and proceeding to the second split, the previous state must be restored, as indicated in lines 19 and 26. This save-and-restore mechanism leverages the internal stack state of the RSBS. Every save action corresponds to a push on the stack, and every restore action equates to a pop, allowing for the restoration of the partitioning between empty and non-empty words in constant time.

5 Results

This section presents the results of the experiments we have conducted. The source code and data used for the experiments in the paper are available at https://anonymous.4open.science/r/pytrees-C73F/. First, we conducted a per-



(a) $max\_depth = 4$ (b) $max\_depth = 5$ (c) $max\_depth = 6$

Fig. 4: Performance Profile plots comparing the error rates of LGDT and DL8.5 on large datasets.

formance comparison among three decision tree learning algorithms: C4.5, DL8.5[2], and LGDT. Our primary focus was on LGDT, with a distinction between two implementations based on data representation. The first, denoted as `LGDT`, employed a reversible sparse bitset structure, while the second, `LGDT`*, followed the original [11] algorithm using a double loop and boolean data view. A scikit-learn[3] implementation of C4.5 was included for comparison. DL8.5 was run with a 10-minute time limit. All the algorithms on an Intel i5-1245U machine with 16 GB RAM running Arch Linux. The experiments encompassed 23 discretized datasets from CP4IM[4], with a minimum support of 1. Figure 3 illustrates the cumulative termination count of each algorithm. Notably, C4.5 exhibited the fastest performance, consistently producing decision trees within a second, regardless of depth. At a depth limit of 2, `LGDT` and DL8.5 displayed similar performance, as both returned optimal trees at this depth. DL8.5 and `LGDT` performed similarly due to their shared utilization of the level-two specialized algorithm and reversible sparse bitset data structure. `LGDT`* lagged behind the RSBS implementation, attributed to individual example checks for supported examples during each projection. With increasing depth, C4.5 maintained its speed edge, while DL8.5 progressively slowed and ultimately couldn't solve all instances within a 10-minute window starting from depth 4. Furthermore, when comparing LGDT implementations across depths, `LGDT` outperformed `LGDT`* by an average factor of 10.

We proceed with an experiment to gauge the proximity of LGDT tree errors to optimality. Using a performance profile [9], we contrast error rates of two LGDT versions (`LGDT+error` using a level-two specialized optimal tree algorithm, and `LGDT+IG` with level-two information gain optimization) with those of C4.5 and DL8.5 on the aforementioned datasets, based on the training set. This aims to ascertain LGDT's viability as an alternative to C4.5 and to gauge its divergence from DL8.5. Figure 2 illustrates performance profiles for instances with maximum depths of 2, 3, and 4. The performance profile is a cumulative

[2] https://github.com/aia-uclouvain/pydl8.5
[3] https://scikit-learn.org/
[4] https://dtai.cs.kuleuven.be/CP4IM/datasets/

distribution of an algorithm's $s \in S$ enhanced performance versus other algorithms in set S across a problem set P: $p_s(\tau) = \frac{1}{|P|} \times |\{p \in P : r_{p,s} \leq \tau\}|$, with the performance ratio $r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} | s \in S\}}$ where $t_{p,s}$ signifies each algorithm's error rate. Results reveal that, regardless of depth, LGDT consistently achieves superior error rates compared to C4.5 across all datasets. At depth 2, both `LGDT+error` and DL8.5 immediately secure the lowest error rates as they yield optimal trees in this context. `LGDT+IG` holds the lowest error rate for about 50% of instances, while C4.5 achieves it on around 30%. Moreover, permitting an error rate roughly double the best, `LGDT+IG` solves all instances, whereas C4.5 remains unable. With increasing depth, DL8.5 consistently outperforms others with the lowest error rate, except for depth 4, where it times out on some. The error gaps widen with depth, reflecting escalating performance ratios. Notably, the LGDT-C4.5 gap widens faster than DL8.5-LGDT, showcasing LGDT's greater reliability than C4.5. This may be due to C4.5's inclination to make erroneous or sub-optimal decisions with deeper trees, heightening the risk of mitigating underfitting.

To evaluate LGDT's effectiveness compared to optimal decision trees on large datasets, we conducted a final experiment using 15 classification datasets from the UCI Repository. These datasets had at least 30 features, binarized to create datasets with a minimum of 300 features. Experiments covered tree depths of 4, 5, and 6, with DL8.5 constrained to a 30-second runtime to highlight challenging scenarios. The performance profile in Figure 4 revealed that at depth 4, DL8.5 outperformed in about 40% of instances, but this advantage diminished and disappeared at depth 6. DL8.5 consistently lagged behind LGDT, and as depth increased, the error gap widened, showcasing LGDT's superior reliability for deeper trees. DL8.5's declining performance with depth is attributed to its tendency to become trapped in deeper search tree sections, prioritizing optimal features over comprehensive search space coverage. This focus results in substantial unexplored segments, leading to suboptimal outcomes compared to LGDT approaches.

6 Conclusion

In this paper, we proposed an efficient lookahead algorithm for constructing decision trees that can capture feature interdependencies within binary datasets. To offer the best computation time, and become a viable and practical alternative over pure greedy methods, the algorithm relies on two algorithmic ideas: the Murtree level-two specialized algorithm and the reversible sparse-bitset data structure also used in DL8.5. Through experiments on various datasets, we compared the performance of our algorithm with two state-of-the-art decision tree methods, C4.5 and DL8.5. Our results suggest that lookahead decision trees can be a valuable addition to the toolkit of data scientists. In future research, we aim to explore effective techniques for managing continuous features instead of binarizing them in advance.

References

1. Aghaei, S., Gómez, A., Vayanos, P.: Strong optimal classification trees. arXiv preprint arXiv:2103.15965 (2021)
2. Aglin, G., Nijssen, S., Schaus, P.: Learning optimal decision trees using caching branch-and-bound search. In: Proceedings of AAAI. vol. 34, pp. 3146–3153 (2020)
3. Bertsimas, D., Dunn, J.: Optimal classification trees. Machine Learning **106**(7), 1039–1082 (2017)
4. Boutilier, J.J., Michini, C., Zhou, Z.: Shattering inequalities for learning optimal decision trees. In: International Conference On Integration Of Constraint Programming, Artificial Intelligence, And Operations Research. pp. 74–90. Springer (2022)
5. Breiman, L., Friedman, J., Olshen, R., Stone, C.: Classification and regression trees. wadsworth int. Group **37**(15), 237–251 (1984)
6. Burdick, D., Calimlim, M., Gehrke, J.: Mafia: A maximal frequent itemset algorithm for transactional databases. In: Proceedings 17th international conference on data engineering. pp. 443–452. IEEE (2001)
7. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.C., Schaus, P.: Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In: International Conference on Principles and Practice of Constraint Programming. pp. 207–223. Springer (2016)
8. Demirović, E., Lukina, A., Hebrard, E., Chan, J., Bailey, J., Leckie, C., Ramamohanarao, K., Stuckey, P.J.: Murtree: Optimal decision trees via dynamic programming and search. Journal of Machine Learning Research **23**(26), 1–47 (2022)
9. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical programming **91**, 201–213 (2002)
10. Donick, D., Lera, S.C.: Uncovering feature interdependencies in high-noise environments with stepwise lookahead decision forests. Scientific Reports **11**(1), 9238 (2021)
11. Esmeir, S., Markovitch, S.: Lookahead-based algorithms for anytime induction of decision trees. In: Proceedings of the twenty-first international conference on Machine learning. p. 33 (2004)
12. Holsheimer, M., Kersten, M.L., Mannila, H., Toivonen, H.: A perspective on databases and data mining. In: KDD. vol. 95, pp. 150–155 (1995)
13. Iba, W., Langley, P.: Induction of one-level decision trees. In: Machine Learning Proceedings 1992, pp. 233–240. Elsevier (1992)
14. Lin, J., Zhong, C., Hu, D., Rudin, C., Seltzer, M.: Generalized and scalable optimal sparse decision trees. In: ICML. pp. 6150–6160. PMLR (2020)
15. Narodytska, N., Ignatiev, A., Pereira, F., Marques-Silva, J., Ras, I.: Learning optimal decision trees with sat. In: Ijcai. pp. 1362–1368 (2018)
16. Nijssen, S., Fromont, E.: Mining optimal decision trees from itemset lattices. In: KDD. pp. 530–539 (2007)
17. Norton, S.W.: Generating better decision trees. In: IJCAI. vol. 89, pp. 800–805 (1989)
18. Quinlan, J.R.: C4.5: Programs for machine learning. Elsevier (2014)
19. Ragavan, H., Rendell, L.A.: Lookahead feature construction for learning hard concepts. In: ICML (1993)
20. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C.G., Schaus, P.: Learning optimal decision trees using constraint programming. Constraints **25**(3), 226–250 (2020)
21. Verwer, S., Zhang, Y.: Learning optimal classification trees using a binary linear program formulation. In: Proceedings of AAAI. vol. 33, pp. 1625–1632 (2019)