Exploring the Efficiency of GOSDT

MATH 466: Mathematics of Machine Learning
Duke University

Zachary Robers

July 7, 2025

1 Introduction

Decision trees provide an intuitive approach for both classification and regression, yet they tend to overfit without a mechanism to penalize complexity. By introducing a sparsity parameter penalizing the size of a tree, one can control the trade-off between training loss and model simplicity. This balance is central to interpretable machine learning and raises an important question: is there an algorithm that guarantees the optimal tree—minimizing a loss function that includes a complexity penalty without incurring exponential runtime?

The Generalized Optimal Sparse Decision Trees (GOSDT) algorithm [1] addresses this challenge by seeking the globally optimal tree. In practice, however, the complexity of the tree search space remains a major bottleneck. As the number of features or data points grows, the number of potential subproblems can become enormous, despite pruning strategies. While GOSDT is among the most efficient exact algorithms available, its performance can degrade on large-scale datasets or when the regularization parameter is small, motivating further investigation into the underlying mechanics of GOSDT, including how performance differs with hyperparameters and how its runtime can be improved in practice.

2 Background

2.1 Decision Trees

A decision tree is a hierarchical model used for classification or regression that recursively partitions the input space using feature-based rules. Each internal node represents a binary split on a feature, and each leaf assigns a prediction, typically the majority class in classification or the mean response in regression. In application, decision trees are especially interpretable. Explicit splitting criteria can be used to explain why a certain combination of features yields a predictive label.

Traditional decision tree algorithms such as CART (Classification and Regression Trees) [3] and C4.5 [4] build trees using greedy, top-down procedures. At each node, the algorithm selects the split that locally optimizes a given impurity measure (e.g. Gini impurity or information gain). This process repeats recursively until some stopping criterion is met, such as minimum leaf size or maximum depth.

While these algorithms are fast and often yield reasonable performance, their greedy nature means they do not guarantee global optimality. For example, a locally optimal split early in the tree may preclude

better splits deeper down. As a result, CART and related methods can produce trees that are both suboptimal in predictive accuracy and unnecessarily complex. This has motivated interest in global optimization approaches, such as GOSDT, that aim to find the best tree according to a specified objective function.

2.2 Regularization

Regularization introduces a penalty term to an optimization objective to discourage overly complex models. In the context of decision trees, complexity is often measured by the number of leaves or the depth of the tree. By penalizing tree size, regularization promotes sparsity: fewer splits and shallower trees. Sparsity is a desirable property because it enhances interpretability and mitigates overfitting. Sparse trees are more interpretable because a smaller combination of feature splits can be used to explain a predictive class.

This approach parallels regularization in linear models. For instance, LASSO regression adds an l_1 penalty to the loss function, encouraging coefficients to be exactly zero and thereby selecting a sparse subset of features. Ridge regression, by contrast, uses an l_2 penalty to shrink coefficients without necessarily driving them to zero. Both methods aim to balance model fit with complexity, and similar ideas underpin regularization in decision trees.

2.3 The GOSDT Optimization Problem

GOSDT tackles the problem of finding a sparse, binary decision tree T that minimizes a regularized loss objective:

$$\min_{T} \quad [J(T) := R(T) + \lambda \cdot \text{size}(T)] \tag{1}$$

where $R(T)$ represents the empirical loss (typically misclassification error), size(T) is the number of leaves in the tree, and $\lambda \geq 0$ is a regularization parameter controlling the trade-off between accuracy and sparsity. Additionally, GOSDT enforces a hard constraint on tree depth: depth$(T) \leq D$. This constraint ensures that the optimization is restricted to trees of bounded complexity, which can substantially reduce the search space.

Before optimization begins, GOSDT requires a critical preprocessing step: all features must be binarized. For continuous variables, this is accomplished by identifying all unique values in the dataset, computing midpoints between adjacent values, and creating binary indicators of the form $x \leq \theta$. For example, if a feature x takes values $2.9, 4.6, 6.9, 7.0, 10.4$, then new binary features like $x \leq 3.75$, $x \leq 5.75$, and so on are generated. Categorical variables are one-hot encoded in a similar fashion. The result is a binary feature matrix that enumerates all possible splits a tree could make.

This binarization process transforms the space of potential decision trees into a finite but still combinatorially large set. A naïve exhaustive search over all binary trees of depth at most D would be computationally infeasible, as the number of possible trees grows exponentially with both the number of binary features and the tree depth. Specifically, with p binary features and maximum depth D, the number of distinct trees can easily exceed $O(p^{2^D})$, even for moderate values of p and D.

GOSDT circumvents this brute-force search through a dynamic programming strategy, bounding techniques, and careful reuse of subproblem solutions. Nevertheless, the inherent complexity of the search space makes computational efficiency a central concern in applying GOSDT to large datasets—motivating our exploration of algorithmic and practical improvements.

3 Mathematics Behind GOSDT

GOSDT is a complex, yet elegant, example of dynamic programming with bound relaxation. Due to its complexity, the following description will not be exhaustive, but instead intends to provide a rigorous explanation of key mathematical and algorithmic components.

3.1 Subproblems

Each subproblem can be thought of as the optimization of a smaller tree after some amount of splits have already occurred. In this setting, only a subset of the original data points used to construct the tree are available to be classified. Ergo, this specific subproblem can be represented by the data points it needs to classify. GOSDT represents each subproblem with a bit vector for efficiency purposes and subproblems are stored in a dictionary for quick indexing:

Subproblem Representation. Let n be the total number of training examples. Let $\mathbf{s} \in \{0,1\}^n$ be a bit vector representing the current subproblem, where $s_i = 1$ if the i-th data point is included in the subproblem, and 0 otherwise.

For any subproblem, the loss for that subproblem may be minimized either by splitting on any of the features x_i or by making that tree a leaf. At the time of traversal, the exact loss for these particular cases is unknown, so instead GOSDT stores an upper bound (p.ub) and a lower bound (p.lb) on the loss for each subproblem. These bounds are necessary because initially not all of the losses can be determined as they depend on exploring trees with additional splits. Furthermore, whether or not the algorithm decides to explore addition splits is dependent on the current state of the upper and lower bounds (per 3.2). These upper and lower bounds are initialized and updated as the algorithm traverses the subproblems according to the following scheme:

Initialization. When a new subproblem p with associated bit vector $\mathbf{s}$ is initialized, GOSDT provides the most optimistic upper and lower bounds for the minimum objective value for a tree tasked with classifying these points. Namely, we know that the best tree cannot perform worse than the misclassification error in the event that we just made the tree for that set of data points a lead rather than splitting further, thus providing an upper bound. We also assume that it would be possible, if we were to split once more, that misclassification error would be reduced to zero when deciding on the lower bound:

$$p.\text{ub} = \frac{m_s}{n} + \lambda, \tag{2}$$

$$p.\text{lb} = 2\,\lambda, \tag{3}$$

where n denotes the total number of training examples, and

$$m_s = \min\big(n_s^+,\, n_s^-\big), \quad n_s^+ = \sum_{i:s_i=1} \mathbf{1}\{y_i = 1\}, \quad n_s^- = \sum_{i:s_i=1} \mathbf{1}\{y_i = 0\}.$$

Here, λ is the regularization parameter, and p.lb and p.ub are the lower and upper bounds on the optimal objective value for the subproblem p.

Updation. The subproblem traversal is more thoroughly analyzed in Section 3.3. Abstractly speaking, as further feature splits are explored (by creating left and right subproblems of the data points that fall under each of these categories), the state of a parent subproblem is updated by minimizing the bounds resulting from making additional splits:

$$p.\text{ub} = \min(p.\text{ub}, \min_{(p_l,p_r)_j} (p_l^j.\text{ub} + p_r^j.\text{ub})) \tag{4}$$

$$p.lb = \min(p.\text{lb}, \min_{(p_l,p_r)_j} (p_l^j.\text{lb} + p_r^j.\text{lb})) \tag{5}$$

Where $(p_l, p_r)_j$ denotes the left and right subproblems formed by splitting p on the jth feature.

3.2 Completion Condition

Eventually, assuming the sparsity penalty (regularization value λ) is sufficiently large, the penalty will overtake any possible reduction in misclassification loss from continuing to split a tree as the number of data points considered by the subproblems becomes less and less. In which case, the trivial tree of a single leaf is the optimal solution. This is the case when the following condition, presented in [2], holds on the upper and lower bounds,

Theorem 3.1. *Let p be a subproblem corresponding to a subset of data points indexed by a bit vector $\mathbf{s} \in \{0,1\}^n$. Suppose the upper bound $p.ub$ and the lower bound $p.lb$ for the optimal objective value of p satisfy*

$$p.ub - p.lb \leq 0.$$

Then the trivial tree consisting of a single leaf is an optimal solution to subproblem p; that is,

$$J(\textit{trivial tree}, X_{\mathbf{s}}, Y_{\mathbf{s}}) \leq J(\textit{any child tree}, X_{\mathbf{s}}, Y_{\mathbf{s}}).$$

Proof. By definition, the trivial tree classifies all points in the subproblem as a single class, leading to a misclassification loss equal to the number of minority class labels. Let this number be m_s. Then the objective value of the trivial tree is:

$$J(\text{trivial tree}, X_{\mathbf{s}}, Y_{\mathbf{s}}) = \frac{m_s}{n} + \lambda = p.\text{ub}.$$

Now consider any child tree resulting from a split of the subproblem. Such a tree must have at least two leaves, contributing a regularization term of at least 2λ. The best possible misclassification error (i.e., lower bound on loss) is zero. Thus, for any child tree:

$$J(\text{child tree}, X_{\mathbf{s}}, Y_{\mathbf{s}}) \geq 0 + 2\lambda = p.\mathsf{lb}.$$

If $p.\mathsf{ub} - p.\mathsf{lb} \leq 0$, then:

$$\frac{m_s}{n} + \lambda \leq 2\lambda \Rightarrow \frac{m_s}{n} \leq \lambda.$$

Therefore:

$$J(\text{trivial tree}, X_{\mathbf{s}}, Y_{\mathbf{s}}) \leq J(\text{child tree}, X_{\mathbf{s}}, Y_{\mathbf{s}}),$$

and the trivial tree is at least as good as any child tree. Hence, it is optimal. □

3.3 Traversal Strategy

The prototypical examples of dynamic programming, such as the Knapsack and Palindrome problems, the algorithm addresses subproblems upon which the master problem depends according to some preset uniform traversal strategy. GOSDT, however, traverses its set of subproblems starting at the master problem, then examining the problems on which it most closely depends, revisiting subproblems higher up in this traversal as a relevant update occurs [1].

Whenever the lower or upper bound of a subproblem is decreased, GOSDT propogates this information upward in the traversal structure towards the master problem to see if this modification leads to the completion condition for problems along this path. This upward update occurs at a higher priority than continuing to explore downward in the dependency graph, motivated by a desire to complete the master problem without unnecessary exploration. When a problem is popped off the priority queue, the algorithm

updates the upper and lower bounds for that problem, p, according to the following formulas (originally introduced in 4 and 5):

$$p.\text{ub} = \min(p.\text{ub}, \min_{(p_l,p_r)_j} (p_l^j.\text{ub} + p_r^j.\text{ub}))$$
$$p.lb = \min(p.\text{lb}, \min_{(p_l,p_r)_j} (p_l^j.\text{lb} + p_r^j.\text{lb}))$$

Where $(p_l, p_r)_j$ denotes the left and right subproblems formed by splitting p on the jth feature.

Once evaluated, if either of these equations result in a change to the upper or lower bounds for p, the algorithm places all of the parent problems to p, found via the dependency graph, into the priority queue to launch a similar investigation on problems higher up in the dependency graph. In this sense, GOSDT "updates upward," repeatedly checking if a change to the bounds in a lower subproblem impacts the current state of problems higher up (closer to the master problem) in the dependency graph.

The algorithm continues to check if the problem is now solved (i.e. $p.\text{ub} = p.\text{ub}$, see Section 3.2). If this is the case, then there is no need to revisit the subproblems on which this problem depends as this problem is already solved. If this is not the case, the algorithm adds the subproblems formed by splitting p on each feature into the priority queue to revisit later on to further improve the lower and upper bounds for p. This process continues until the master problem is solved.

4 Runtime Analysis

The best and worst case scenarios for GOSDT's runtime are highly dependent on the information gain provided by splitting on the features inside the dataset. Furthermore, these cases hint at a possible relation between the regularization parameter, λ, and runtime as scenarios like the best case are prone to occur where then the regularization parameter is large and scenarios like the worst case are prone to occur when the regularization parameter is small.

4.1 Best Case

In the best case, GOSDT's architecture enables quick detection of the optimal tree. Such instances occur when the initial split, or (in a less absolute case) the first few splits, improve misclassification error but subsequent splits don't improve the misclassification error by more than the corresponding regularization penalty. For an example, we explore the case of an optimal tree with 2 leaves*.

Proposition 4.1. *Let T_0 be the trivial tree and let T_1 be the two–leaf tree obtained by making the single split that* minimises *the mis-classification error $R(T)$. Assume*

$$\Delta R(T_0, T_1) = R(T_0) - R(T_1) > \lambda, \qquad \Delta R(T', T'') < \lambda$$

for every two–leaf tree T' and every three–leaf tree T'', where $\Delta R(A, B) = |R(A) - R(B)|$ and $\lambda > 0$ is the sparsity parameter in the objective

$$J(T) = R(T) + \lambda |T|. \tag{1}$$

Proof. The argument is by cases on the cardinality of the tree.

*The absolute best case is when the optimal tree is the trivial tree and no other trees are explored as the decrease in the misclassification error does not exceed the regularization parameter. This case is especially clear, so we turn to the two leaf case for a more robust and generalizable example.

Case 1: $J(T_1) < J(T_0)$**.** Because $|T_1| = 2$ and $|T_0| = 1$,

$$J(T_1) = R(T_1) + \lambda < R(T_0) = J(T_0),$$

where the inequality is exactly our first hypothesis $\Delta R(T_0, T_1) > \lambda$. Hence T_1 is preferred to the trivial tree.

Case 2: $J(T_1) < J(T)$ for every other two–leaf tree $T \neq T_1$. By construction T_1 attains the minimum possible mis-classification error among all single–split (two-leaf) trees, so $R(T) \geq R(T_1)$ for every such T. Since $|T| = |T_1| = 2$,

$$J(T) = R(T) + \lambda \;\; \geq \;\; R(T_1) + \lambda = J(T_1),$$

with equality only if T coincides with T_1. Thus T_1 dominates every other tree with two leaves.

Case 3: $J(T_1) < J(T)$ for every tree T with $|T| \geq 3$. Let T have $k \geq 3$ leaves. Build a chain of trees

$$T_1 \;\longrightarrow\; T^{(3)} \;\longrightarrow\; \cdots \;\longrightarrow\; T^{(k)} = T,$$

where $T^{(m)}$ has m leaves and differs from $T^{(m-1)}$ by a single additional split. By the second hypothesis each extra split reduces the error by strictly less than λ:

$$R\big(T^{(m)}\big) \;\geq\; R\big(T^{(m-1)}\big) - \lambda \quad (m \geq 3).$$

Inductively,

$$R(T) \;\geq\; R(T_1) - (k-2)\lambda.$$

Therefore

$$J(T) = R(T) + k\lambda \;\geq\; R(T_1) - (k-2)\lambda + k\lambda \;=\; R(T_1) + \lambda + (k-3)\lambda \;>\; R(T_1) + \lambda \;=\; J(T_1),$$

because $\lambda > 0$ and $k \geq 3$. Hence every tree with more than two leaves has larger objective value than T_1.

Combining the three cases, T_1 outperforms T_0, every other two–leaf tree, and every tree with more than two leaves. Consequently T_1 minimizes the objective (1). □

GOSDT is well suited for detecting an optimal tree of this type. In practice, the algorithm would identify T_1 almost instantly: the very first level of its branch-and-bound search evaluates all candidate one-split trees, and the moment the gap $R(T_0) - R(T_1)$ exceeds λ the hierarchical lower bounds prune *every* deeper expansion. Hence the optimality certificate for T_1 is produced after visiting at most the root and its two children.

4.2 Worst Case

In the opposite extreme GOSDT can be forced to explore essentially the entire combinatorial search space, so that its running time degenerates to the exponential complexity of a brute-force enumeration. Intuitively, this happens when every admissible split removes only a small amount of misclassification error (so that the hierarchical lower bounds never certify non-optimality), and the sparsity parameter λ is set so small that even this minute decrease always outweighs the new regularization charge.

A dataset that realizes this scenario can be constructed as follows. Assume the training sample has N distinct observations and M binary attributes. Order the examples arbitrarily and, for $k = 1, \ldots, N$, define a binary feature

$$f_k(x) = \mathbf{1}\{x = x^{(k)}\},$$

i.e. f_k is true on the k-th training point and false elsewhere. Label the first N_+ examples positive and the remaining $N_- = N - N_+$ negative. Set the regularization parameter

$$0 < \lambda < \frac{1}{N}.$$

Proposition 4.2. *Let T_0 be the trivial one-leaf tree and, for $h = 1, \dots, N$, let T_h denote the h-leaf tree obtained by consecutively splitting on $f_1, \dots, f_h$ (isolating one observation per split) and classifying each terminal node by majority vote. Then:*

(a) $R(T_{h-1}) - R(T_h) = \dfrac{1}{N}$, *for all* $h = 1, \dots, N$;

(b) the objective (1) *satisfies*

$$J(T_h) < J(T_{h-1}) \quad \textit{for every } h;$$

(c) consequently, the globally optimal tree is the pure *tree T_N with N leaves, and no hierarchical bound employed by GOSDT can prune any node generated along the (unique) search path*

$$T_0 \;\to\; T_1 \;\to\; \cdots \;\to\; T_N.$$

Proof. (a) Each split isolates exactly one previously misclassified observation, so the empirical error drops by $1/N$.

(b) Since $\lambda < 1/N$, every step satisfies

$$\Delta R(T_{h-1}, T_h) = \frac{1}{N} > \lambda, \qquad \text{hence } J(T_h) = R(T_h) + h\lambda < J(T_{h-1}).$$

(c) Part (b) shows the objective is strictly decreasing along the entire chain, so T_N is the unique minimizer. Because at stage h the best current bound equals $J(T_h)$, while every sibling of T_h has objective at least $J(T_{h-1})$, none of GOSDT's bounding tests rule out the descendants of T_h. Ergo, the algorithm must visit every intermediate tree. □

During the search, GOSDT creates a node for every non-empty subset of the N observations except the full set (handled at the root). Hence the number of sub-problems equals $2^N - 1$, and the total work is exponential in N. Proposition 4.2 thus shows that, in the worst case, the sophisticated branch-and-bound machinery of GOSDT offers no speed-up over naive exhaustive search.

Even in this adversarial setting, the leaf count of any optimal tree is bounded by

$$|T^\star| \;\le\; \min\big(\lfloor R(T_0)/\lambda \rfloor,\, 2^M\big),$$

a direct consequence of Theorem B.4 in [2]. Once $R(T) = 0$ the objective can no longer improve, so the chain in Proposition 4.2 stops after at most $\lceil 1/(\lambda N) \rceil$ steps; with our choice $\lambda < 1/N$ this cap equals N. Thus the regularization parameter still puts a *finite* ceiling on how far the blow-up can go at the price of exponential growth up to that ceiling.

This investigation exemplifies that GOSDT's practical speed-ups rely on hierarchical bounds that prune unpromising branches early. If every split looks just promising enough (relative to λ) these bounds fail to limit the search space, and the algorithm degrades to worst-case exponential time. Choosing λ well above $1/N$ and eliminating near-duplicate features in preprocessing are key to preventing this adversarial instance.

5 Parallelizability

At first glance, GOSDT may appear to be an inherently parallelizable algorithm. Given that GOSDT works with subproblems, one may reason that each subproblem can be computed on a separate thread, and these subproblems can later be joined from the bottom up to solve the master problem. However, this design runs

counter to the top-down (starting at the master problem) traversal strategy with dynamic programming that makes GOSDT computationally feasible in the first place.

If one were to revise the GOSDT algorithm to instead operate from the bottom up by first exploring how to best classify subsets of 2 data points then 3, then 4 and so forth, the algorithm could easily be implemented in parallel by introducing new threads for each subproblem with the same number of data points repeatedly as then number of data points in the subset increases. This strategy would be valid since the optimal tree for a set of data points of size k depends on the optimal trees for strict subsets of those data points. However, this is exactly the kind of brute force exponential blow-up that GOSDT seeks to avoid. While parallelizing at the subset size level like this would enable the algorithm to make use of parallelization, it would need to explore each of the 2^n possible subproblems where n is the size of the dataset, making the tree construction practically infeasible at large scales.

To implement parallelization with the current traversal and dynamic programming techniques, there are two central obstacles to surmount:

1. *Updating Upwards*: Per Section 3.3, when a lower or upper bound is lowered, the GOSDT architecture propogates this information upward in the traversal structure towards the master problem to see if this modification leads to the completion condition for problems along this path. This process goes between multiple subproblems, making it difficult for them to run on separate threads.
2. *Reusing Subproblems*: In order to gain efficiency improvements by not recalculating solved subproblems, GOSDT needs access to a centralized dictionary of subproblems to find the optimal tree for a subset of the data that has already been encountered.

The latter problem of reusing subproblems can likely be circumvented through locking and unlocking mechanisms to avoid race conditions between threads on the dictionary which stores the current state of subproblems. However, the complexity of GOSDT's traversal strategy prohibits any straightforward subproblem parallelization as GOSDT does not solve subproblems to the point in which the termination condition is met in isolation; rather, it is constantly revisiting and introducing new suproblems using the dynamic bounding mechanisms discussed in Section 3.3. For this reason, the rest of this investigation focuses on means by which one can make GOSDT efficient by analyzing hyperparameter tradeoffs and exploring approximate variants.

6 Hyperparameter Sensitivity

The best and worst case scenario investigation of Section 4 hints at the possibility of a strong correlation between runtime and the regularization hyperparameter, λ. Concurrently, the discussion on regularization from Section 2.2 notes the traditional balance of overfitting and underfitting which the regularization parameter seeks to accomplish. These theoretical impacts of regularization motivate the following empirical investigation into how the λ influences the runtime, performance, and tree structure of GOSDT in practice.

Furthermore, considering that GOSDT indexes subproblems according to the data points in each subproblem (see Section 3.1), the following section also includes a similar investigation on how the size of the dataset (n) affects runtime and performance in practice. Lastly, the impact of the max depth hyperparameter (D), which limits the depth of the tree generated by GOSDT, is also investigated.

6.1 Setup

The GOSDT Python Library [6], offers an easy-to-use implementation of the GOSDT algorithm. The library offers support for modifying the hyperparameters of the GOSDT optimization function. To test

how the efficiency and accuracy of the algorithm respond to changes in possibly impactful hyperparameters, we trained the algorithm on different hyperparameter combinations using the wine dataset from the UCI machine learning repository [7].

The wine dataset has 13 continuous features, expressing differing chemical, physical, and taste properties of various wine samples. Each wine sample falls into one of three classes corresponding to the cultivar labels of the wine. The wine dataset is commonly used to evaluate model performance on multi-class classification tasks because it exhibits a clear non-linear relationship between the features and the classes.

Since we are interested in studying the effects of varying a specific hyperparameter in isolation rather than finding the maximum accuracy we define a default setting of the parameters and run parameter combinations and adjust each parameter individually rather than a typical grid search. The default setting and values of each hyperparameter tested are presented in the tables below:

	Regularization (λ)	Dataset Size (n)	Hard Depth Constraint (D)
Default Parameters	0.05	25	4

Table 1: Default settings for the regularization parameter, dataset size, and maximum tree depth used in experiments.

λ	0.01	0.05	0.10	0.20	0.30
n	25	50	80	120	150
D	2	3	4	5	6

Table 2: Hyperparameter values explored for regularization λ, dataset size n, and hard depth constraint D.

Before any of the trees are trained, we perform an 80-20 train-test split on the entire dataset. For the sample size (n), the five datasets (corresponding to the five values of n) were randomly sampled from the training dataset after the split. Each model, regardless of n, was evaluated using the entirety of the original test dataset.

Accuracy is well-suited to be used as the performance metric for evaluating the GOSDT trees since classes are relatively balanced within the wine dataset. In addition to studying the runtime for fitting the tree, we also keep track of the number of the leaves in each tree to see how the hyperparameters impact the complexity of the resulting model.

6.2 Regularization (λ)

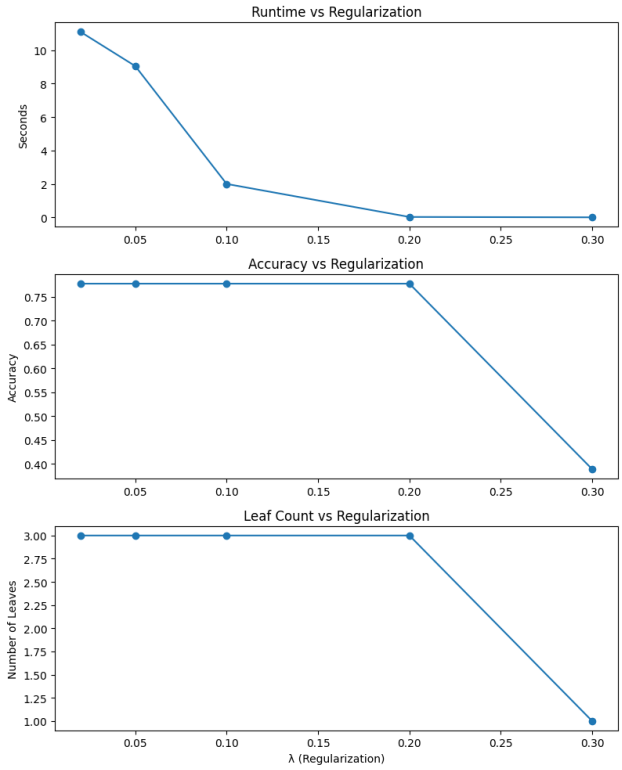

Figure 1: Accuracy, Runtime, and Number of Leaves for Different Values of λ

The investigation of changing the regularization parameter λ shows that an increase in λ is generally associated with lower runtime, lower accuracy, and a less complex (lower leaf count) resulting model. This aligns with both the formulation of the algorithm as well as the general purpose of regularization. Since regularization penalizes making additional splits in the tree, it reduces the number of trees that GOSDT explores in the fitting process as the penalty impedes the possibility for a tree with more leaves to be optimal, thus decreasing runtime. In cases where the model is overfit to the data, adding regularization can increase test accuracy. However, in our case, even with the lowest value of regularization the tree was not overfit to the data, meaning that regularization did not have a positive impact on model performance. Of notable interest is the fact that neither accuracy nor leaf count decreased in all cases except for $\lambda = 30$. In fact, oftentimes in experimentation, GOSDT produced the same tree even after substantial increases to the regularization parameter which also greatly lowered runtime. This suggests that in application it may oftentimes be possible to increase the efficiency of GOSDT by increasing λ without sacrificing performance even if the model is not overfit to the training dataset.

6.3 Dataset Size (n)

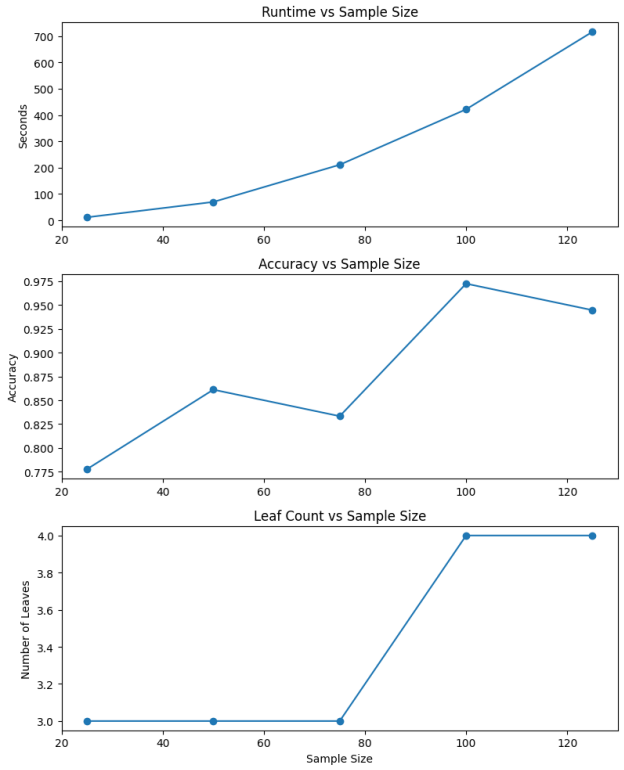

Figure 2: Accuracy, Runtime, and Number of Leaves for Different Values of n

Dataset size (n) is the largest inhibitor of GOSDT's widespread adoption. The default setting of $n = 25$ was chosen because higher valuables prohibited GOSDT's completion in practice as runtime exceeded several hours for individual trials. However, most machine learning datasets include several orders of magnitude more data points, making GOSDT an impractical choice of algorithm. The top chart shows that the relationship between runtime and sample size appears to be of higher asymptotic order than linear as the slope of each line segment becomes larger as the sample size increases. Theoretically, this suggests that the worst case of exponential blow-up in the number of subproblems GOSDT explores (4.2) may be closer to the relationship we see in practice. While the general trend suggests that higher sample size leads to higher accuracy, a lack of monotonicity in that relationship suggests that at times trees trained on a larger dataset may be overfit to the particular set of data points, prohibiting generalization.

6.4 Max Depth (D)

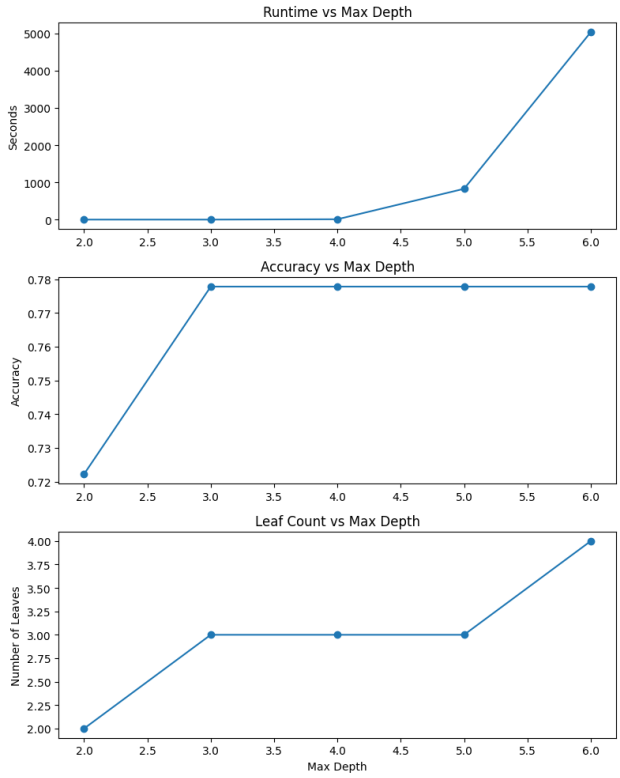

Figure 3: Accuracy, Runtime, and Number of Leaves for Different Values of D

The final study on the impact of the hard depth constraint (D) shows that the depth constraint actively limits the number of subproblems GOSDT explores as the runtime increases when the depth constraint increases. This means that certain trees that could be optimal simply are not being explored by GOSDT because they would be deeper than the depth constraint allows. However, the plots of accuracy and number of leaves suggest that although these deeper trees have the possibility to be optimal, they seldom end up being the optimal tree or wielding higher accuracy. This means that imposing a depth constraint may be a worthwhile trade-off considering the great improvement in runtime.

7 Efficient Variants

There are several ways in which the GOSDT architecture could be modified to enhance its efficiency. However, these approaches come at the cost of maintaining optimality, which is central to the appeal of GOSDT. On most datasets, GOSDT cannot outperform more standard machine learning algorithms such as XGBoost and Random Forest in terms of purely its performance metrics on a test dataset. However, GOSDT's advantage stems from the fact that one can guarantee that the tree produced by GOSDT is optimal (according to 1) and GOSDT produces a single tree which tends not to have a large number of leaves, meaning its classification and regression outputs are easily interpretable.

Approximate variants lose the optimality aspect of GOSDT's appeal but maintain the model structure and the associated interpretability benefit. Furthermore, depending on the structure of an approximate variant of GOSDT it is possible that the variant maintains model performance in spite of losing optimality.

Given these trade-offs suggest that efficient approximate variants could still have a use case, the final section of this work proposes two approximate variants of GOSDT. While this work does not venture as far as to implement these variants, the motivation for both of the variants is discussed and their differences from the original GOSDT algorithm are formulated.

7.1 Subset Traversal

One way to reduce the search burden is to explore only a subset of the candidate splits at each node rather than all of them. In particular, this work proposes three possible means by which to select the subset of feature splits to explore for a given unsolved subproblem within GOSDT's traversal process:

- *Random sampling* of a fixed fraction of the available split thresholds, in the spirit of the Random Forests algorithm which selects the best possible feature split of a random subset of features when it constructs its tree ensembles [5].
- *Learned ranking*: train a fast surrogate model (e.g. a small neural network or gradient-boosted tree) to predict the most promising splits, and only explore the top–k.
- *Subset prediction*: use a lightweight scoring rule to identify groups of features whose splits are likely to yield the largest bound improvements, and ignore the rest.

In each case one trades off the completeness of the branch-and-bound search for a drastic reduction in branching factor, at the price of losing the global optimality guarantee.

7.2 Relaxing Upper and Lower Bound Difference

In its provably optimal form (see 3.2), GOSDT requires that the lower bound matches or exceeds the upper bound in order to certify that no further splitting can improve the objective (i.e. the trivial tree is best). A simple approximate variant is to introduce a slack parameter $\varepsilon \geq 0$, and terminate a node whenever for any subproblem p

$$p.\text{ub} - p.\text{lb} \;\leq\; \varepsilon.$$

Choosing $\varepsilon > 0$ allows the algorithm to bail out early on nodes whose potential improvement is negligible, thus dramatically pruning the search tree. The user can control the trade–off: larger ε yields faster runtime but possibly suboptimal trees whose loss lies within ε of the true optimum.

8 Conclusion

In this work, we have taken an in-depth look at the mechanics that make GOSDT both powerful and, in many cases, potentially expensive, prohibiting its widespread adoption. We began by casting decision-tree learning as a regularized combinatorial optimization problem and showing how GOSDT's dynamic programming with bounds framework uses subproblems, priority-driven traversal, and hierarchical bounds to prune vast regions of the search space. In the best and worst case analysis, we showed that, when each split yields a jump in misclassification error surpassing the regularization weight, GOSDT stops almost immediately, visiting only the root and its two children. On the other hand, in the converse adversarial regime of infinitesimal gains and very small λ, it must explore essentially every possible subset of observations. We also discussed why attempts at naïve parallelization conflict with the upward-updating, master-problem-first strategy that underpins GOSDT's efficiency. Furthermore, the empirical investigation of hyperparameter sensitivity highlighted the delicate balance between λ, tree sparsity, accuracy, and runtime.

Looking forward, a particularly promising avenue is the design of efficient approximate variants to GOSDT, like the subset traversal and relaxed bound variants discussed in the final section. While these variants do not preserve global optimality, they still produce a single sparse tree with potentially only marginal differences in test accuracy. Furthermore, depending on the extent to which these variants rely on approximation, they are capable of offering significant enhancements to efficiency, which has been shown in this work to be a major inhibitor to GOSDT's usefulness. For these reasons, it is suggested that further work be carried out in the realm of developing these variants, in particular evaluating the extend to which the resulting tree varies in structure and performance to the globally optimal tree as well as resulting speed-up.

9 Git Repository

Code used for empirical investigations throughout this project can be found here.

References

[1] J. Lin, C. Zhong, D. Hu, C. Rudin, and M. Seltzer, "Generalized and scalable optimal sparse decision trees," in *Proceedings of the 37th International Conference on Machine Learning*, 2020, pp. 6150–6160.

[2] C. Rudin, "Modern Decision Tree Optimization with Generalized Optimal Sparse Decision Trees," Duke Course Notes, 2021.

[3] L. Breiman, J. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*, Wadsworth Int. Group, 1984.

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

[5] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[6] `gosdt`: Generalized Optimal Sparse Decision Trees Python package, version 0.2.0, PyPI, `https://pypi.org/project/gosdt/`, accessed 2025-04-27.

[7] M. Lichman, "UCI Machine Learning Repository," University of California, Irvine, 2013. `https://archive.ics.uci.edu/ml/datasets/wine`, accessed 2025-04-27.