Cost-Sensitive Generalized Optimal Sparse Decision Trees

Marcus Lai, Hayden McTavish, Cynthia Rudin, Margo Seltzer

February 2025

Abstract

Decision Trees have.... We extend the GOSDT framework to optimize for both the accuracy and the expected decision-making cost. We prove that our approach preserves optimality guarantees from GOSDT and achieves lower cost than existing methods that use greedy approaches to reduce the cost of decision-making.

1 Introduction

Machine learning has long been an integral part of our decision-making process and are known to be effective for tasks such as classification and prediciton. In socially sensitive domains such as medical and judicial decision-making, decision trees are especially reliable for their combination of classification accuracy, simplicity and interpretability.

Many decision tree algorithms focus solely on maximizing classification accuracy. However, Turney points out that, in practice, misclassification is just one many forms of decision-making costs, and that others such as the testing cost, waiting cost (i.e., the time to acquire a particular feature value), and teaching costs [1]. In the medical setting, for example, misclassifying a positive diagnosis could be a lot more costly than a negative diagnosis. Feature values, which could represent blood test results, MRI data, or a temperature reading, could also incur various amounts of testing costs to perform. It is often the case that tests with more expensive costs could be more informative as well, making the problem of... [2]. Some tests might introduce a long delay to obtain results, producing a *waiting cost*. Waiting costs are particularly important to . Specifically, misclassification cost and testing cost are noted to be the most important costs that should be reflected in the training process of decision trees to improve interpretability and incorporate decision trees into practical workflows.

Cost-sensitive classification arises in applications such as automatic diagnosis, forensic classification, ... In automatic diagnosis, decision trees are a natural choice as a cost-sensitive and accurate classification technique, since each medical test can be modeled as a split and feature values can be obtained lazily

instead of proactively [3]. While our approach can apply to a wide variety of settings, we use medical diagnosis as a motivating example throughout the paper.

Although the problem of optimal decision trees (ODT) is computationally hard [4], recent advances in algorithms and pruning techniques have made it possible to find optimal decision trees on small to mid-sized datasets [5][6][7][8]. It is therefore reasonable to ask whether optimality could be exploited to obtain more accurate feature-cost-sensitive decision trees. One optimal formulation of the feature-cost-sensitive training problem is STreeD [9] which extends Murtree – a state-of-the-art dynamic programming formulation to the ODT problem. ...

We extend the GOSDT framework to optimize for both accuracy and the expected testing cost to generate feature-cost sensitive, optimal sparse decision trees. We show that our extension preserves the optimality guarantees of the pruning boundaries of GOSDT and introduce two new boundaries – the **Extended Lowerbound on Incremental Accuracy** and the **Feasibility Bound on Feature Costs**. We evaluate our approach against STreeD[9] across 14 datasets.

1.1 Related Work

Feature Cost Sensitive Algorithms Perhaps most relevant to our work is STreeD [9] which extends Murtree's [8]. Lin et al. define a dynamic-programming subproblem, or *optimization task*, broadly as a state with two possible actions: make a prediction or split on a feature; splitting depends on the two subproblems produced by such a split. An optimization task is *separable* if the optimal solution of the subproblem can be resolved independently from tasks other than the ones it may eventually generate. It was shown that optimizing over misclassification costs and feature costs yields a separable problem formulation and could thus necessarily be solved by an extension to MurTree. Both STreeD and our work offer optimality guarantees to our cost-sensitive solutions.

Bayer Zubek and Dietterich propose a Partially Observable Markov Decision Process (POMDP) formulation to find the optimal feature-cost sensitive decision tree [10]. Maliah and Shani extend this work with a modification to reduce the observation space and by exploiting a more powerful solver for the POMDP [2]. However, both algorithms struggle with scalability as the state space and action space grow exponentially with data size. To resolve this issue, Maliah and Shani additionally propose an approximate MDP algorithm and demonstrate superior performance against state-of-the-art heuristics.

Most other feature cost-sensitive optimization algorithms are greedy[4]. C4.5 is a top-down inductive algorithm where the feature that maximizes information gain is used to split a node [11]. Many feature-cost sensitive algorithms such as CSID3[12], EG2[13], and IDX[14] extend C4.5 by optimizing over a modified object that depends on both the information gain term and feature costs of the splits. ICET extends EG2 with a genetic algorithm that biases feature costs and retrains the decision tree [3]. After several iterations, the tree with the best average misclassification cost is returned. Although biasing feature costs could

sacrifice interpretability, ICET can potentially find lower-cost solutions that elude the greedy algorithm due to getting stuck in local optimums [3]. (Talk abt the 2006 one here). Our method differs from these by directly optimizing feature costs thereby guaranteeing optimality. In our experiments, we compare our approach with ... algorithms. We refer readers to this survey [15] by Lomax and Vadera for a discussion and comparison of many more variants of heuristic methods.

Optimal Decision Trees Optimal Decision trees was shown to be NP-hard [4]. Bertsimus and Dunn gave a Mixed Integer Programming formulation of the optimal decision problem and demonstrate optimal solutions on small datasets using MIP solvers [5]. BinOCT extends this work [6]. Dynamic Programming formulations are also explored in DL8[16], DL8.5[17] and MurTree[8] and achieve speedup by caching computed subproblems and bounding the objective. Most relevant to our work is GOSDT [18] which we directly extend to incorporate feature costs. GOSDT is a branch and bound algorithm that achieves competitive runtimes by extensively pruning the search space and reusing subproblems. One drawback of the existing ODT approaches is that they do not consider feature costs and thus could be less interpretable for settings such as medical diagnosis. Our work hopes to combine the value of feature costs with the predictive powers of ODTs.

(Incomplete Section; meant for related problems) Another relevant problem is the Bayesian Active Learning problem where the goal is to identify the hypothesis h^* that perfectly labels a set of objects X from a set of hypotheses H. Each object $x \in X$ is associated with an evaluation cost of $c(x)$, and the problem is to minimize the total expected or minimum cost of retrieving h^*. This problem is demonstrated to be equivalent to finding the perfect decision tree classifier that minimizes the expected testing cost. By assuming that $P \neq NP$, it can be shown that a heuristic algorithm is an optimal approximation if it is within $O(\log(n))$ of the optimal solution, which is attained by multiple groups (cite cite cite). Our work differs from these approaches since we consider both the misclassification cost and the expected testing cost. Furthermore, unlike these algorithms, our approach guarantees optimality.

Another similar problem is the prediction-time cost reduction problem where the goal is to minimize classification accuracy given a budget on the total testing cost.

Acknowledgements We acknowledge the help of the Interpretable Machine Learning Lab at Duke University for their ....

2 Preliminaries

We follow the notation of OSDT and GOSDT. A leaf set $d = (l_1, l_2, ..., l_{H_d})$ contains H_d distinct leaves where each leaf is associated with

3 Our Approach

3.1 A New Objective

Let N be the number of samples and M be the number of features. We let l_k be a leaf and let $d = (d_{fix}, \delta_{fix}, d_{split}, \delta_{fix}, K, H)$ be a tree. We represent a leaf, l_k, as a boolean function over the samples of our dataset ($l_k : \{0,1\}^M \to \{0,1\}$), where each leaf is associated with a unique feature bitstring $v^{l_k} \in \{0,1\}^M$ where a 1 in the ith entry indicates that feature m_i was split on the path of this leaf. We use a feature cost vector $\mathbf{w} \in \mathbb{R}^M$ such that $w_i \geq 0$ to represent the associated weight or cost of splitting on feature m_i. For a loss function, l, we define the weighted loss function, l_w, over a tree and our dataset as

$$l_w(d, \mathbf{x}, \mathbf{y}) = l(d, \mathbf{x}, \mathbf{y}) + \left(\frac{1}{N} \sum_{i=1}^{M} \sum_{k=1}^{H} \sum_{n=1}^{N} w_i \left[v_i^{l_k} \wedge cap(l_k, x_n) \right] \right)$$

The complete weighted objective is then

$$R_w(d, x, y) = l(d, \mathbf{x}, \mathbf{y}) + \left(\frac{1}{N} \sum_{i=1}^{M} \sum_{k=1}^{H} \sum_{n=1}^{N} w_i \left[v_i^{l_k} \wedge cap(l_k, x_n) \right] \right) + H\lambda$$

or

$$R_w(d, x, y) = \bar{l}(FP, FN) + \left(\frac{1}{N} \sum_{i=1}^{M} \sum_{l=1}^{H} \sum_{n=1}^{N} w_i \left[v_i^{l_k} \wedge cap(l_k, x_n) \right] \right) + H\lambda$$

Where FP, FN denote the false positives and false negatives respectively. For simplicity, we let $W(d, \mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{M} \sum_{l=1}^{H} \sum_{n=1}^{N} w_i \left[v_i^{l_k} \wedge cap(l_k, x_n) \right]$ to obtain:

$$R_w(d, x, y) = \bar{l}(FP, FN) + W(d, \mathbf{x}, \mathbf{y}) + H\lambda$$

3.2 Onto Continuous Features

We note that feature costs are a one-time cost incurred on a sample only when its unique root-to-leaf path splits on the feature. Some decision tree classifiers, such as GOSDT, train only on binary datasets. In this case, multiple binary features could map to the same original feature which we should only incur cost for once. In this section we extend our weighted objective to encompass this setting.

Definition 3.1 (Feature Group)**.** A feature group of a feature is m is the set of all binary features $m_1, m_2, ..., m_i$ generated from m with some binarization scheme. In this case we say that $m_1, m_2, ..., m_i$ belong to the same feature group. The feature group of a binary feature m_b is the set $\{m_b\}$.

We let $G_1, G_2, ..., G_M$ be the set of feature groups generated from the features of our dataset $m_1, m_2, ..., m_M$ using a binarization scheme. We generalize our objective onto continuous features by redefining W as:

$$W(d, \mathbf{x}, \mathbf{y}) = \left(\frac{1}{N} \sum_{i=1}^{M} \sum_{k=1}^{H} \sum_{n=1}^{N} w_i \Big[\big(\bigvee_{\{i | m_i \in G_i\}} v_i^{l_k} \big) \wedge cap(l_k, x_n) \Big] \right)$$

which means that if a sample goes through any feature in the feature group, it incurs the one-time feature cost w_i.

3.3 Adapted and Extended Boundaries

We now show how we adapt the GOSDT pruning bounds to GOSDT with Weights. We first prove that the new feature-cost objective function is amenable to GOSDT's pruning bounds. Fortunately, most of GOSDT's pruning bounds apply directly, because feature costs are modeled as a linear term in the objective. The one exception is the similar support bound. These adaptation proofs demonstrate that GOSDT works out-of-the-box if we replace the original objective function with the weighted objective. These proofs are straight forward and follow the GOSDT and OSDT proofs closely so we leave them in Appendix A for the interested reader.

Slightly more novel are the extended boundaries that take advantage of feature costs for pruning. We present the proofs of extended bounds in Appendix B.

Lastly, we present an upper bound on feature costs to suggest a sensible limit for feature costs beyond which we cannot hope to split on a feature. We note that this bound is loose so to make a feature feasible, the feature cost should be much smaller than the bound. We present the proof of Feasibility of feature costs in Appendix C.

Theorem 3.2 (Feasibility of Feature Costs)**.** *Let N be the number of samples in the dataset and λ be the sparsity constant. Then for a feature m_i with corresponding weight w_i, if*

$$w_i \geq N(\frac{1}{2} - \lambda)$$

then any tree that splits on m_i is suboptimal.

As an immediate corollary, we have

Corollary 3.3 (Upper bound on w_i)**.** *An upper bound on any feasible weight w_i is*

$$N(\frac{1}{2} - \lambda) > w_i$$

3.4 Implementation

For each subproblem, we maintain a bitvector indicating the feature groups for which we have already incurred a weight. We call this the*weight bit-string* or

incurred weight bit-string. A 0 on the ith entry of the bit-string denotes that the subproblem has incurred cost for the ith feature group G_i, and 1 denotes that it hasn't. We also persist the value of feature costs incurred by a subproblem. To compute the feature costs of between a parent and its children, we use Algorithm 1.

Algorithm 1 ComputeFeatureCost

1: scaled-weight$\leftarrow$ (|child-capture|/|parent-capture|)$\times$parent-weight
2: feature-weight $\leftarrow 0$
3: **if** new-feature *not incurred* **then**
4: feature-weight $\leftarrow \frac{|\text{child-capture}|}{N}\times$ new-feature's weight
5: new-feature's feature-group $\leftarrow$ *incurred*
6: **end if**
7: **return** scaled-weight + feature-weight

In Algorithm 1, line 1 computes the fraction of the parent's weight that belongs to the child's capture-set. To see that Algorithm 1 is correct, consider splitting a node l_M into a left node l_L and right node l_R with capture-sets $C_M, C_L, C_R \subseteq \mathbf{X}$ respectively. Also let $w_1, w_2, ..., w_k$ be the set of feature costs incurred on the path to the parent node l_M, and w_α be the feature cost of the split at l_M to create l_L and l_R. Without loss of generality, consider the feature cost of the left node l_L:

$$\begin{aligned}\text{cost of } l_L =& \frac{|C_L|}{N}\sum_{i=1}^{k} w_i + |C_L|w_\alpha \\ =& \frac{|C_L|}{|C_M|}\times\frac{|C_M|}{N}\sum_{i=1}^{k} w_i + |C_L|w_\alpha \\ =& \frac{|C_L|}{|C_M|}\times \text{cost of } l_M + |C_L|w_\alpha\end{aligned}$$

which is indeed line 7 in our algorithm. The right child's cost can be computed similarly using C_R.

We note that by including feature costs in GOSDT, we can no longer identify subproblems uniquely by their capture-sets. Indeed, it is possible for two distinct sets of splits to yield the same capture-set but incur two different feature costs. Thus, in our implementation we extend GOSDT to use both the capture-set bit-string and the weights bit-string for subproblems identification during caching. To minimize the changes necessary on the existing data structures in GOSDT, we persist the weights bit-string as a nested bit-string within the capture-set bit-string. The alternative would require using the weights bit-string as a secondary key to the GOSDT data structures and would require changes in the structures themselves and calls to the structure. The **Bitmask**'s (a bit-string in GOSDT) constructors, copy constructor, destructor, $=, \geq, \leq$ comparators are also extended to involve the nested weights bit-string. Finally,

we also extend the hashing function of our data structures to additionally hash with contents of the weights vector.

3.5 Feature Evaluation Heuristic

With the addition of feature costs, one natural heuristic for feature evaluation-order emerges. It makes sense that costly features would occur lower in the optimal tree since the cost of a split is determined by the capture-set of that node which is generally larger the higher the node is in a tree. Thus, a least-cost-first feature evaluation order could contribute to pruning by finding an early near-optimal solution.

One simple way to implement this is to using the priority metric of the subproblem queue. In this approach, features wouldn't be evaluated completely in increasing order, but our priority queue can guarantee that a new task that splits on a less costly feature will be dispatched before an existing task that splits on a more costly feature. With this approach, it may be worthwhile to consider a weighted formula between weights, depth, and subproblem capture size as the metric.

We also propose an algorithm that completely ensures features are enqueued in increasing order.

Algorithm 2 ExploreFeatures

Require: features-ll ← sorted linked list of feature-group **Bitmask** by cost
Require: capture-set
1: **evaluate subproblem**
2: curr-group ← pointer to features-ll
3: **while** curr-group not *null* **do**
4: split-feature ← **next-available-feature**(curr-group)
5: left-features-ll ← **copy**(features-ll) except curr-group node
6: right-features-ll ← **copy**(features-ll) except curr-group node
7: **if** split-feature is last index of head **then**
8: curr-group← **next**(curr-group)
9: **else**
10: left-features-ll← **copy-node**(curr-group) + left-features-ll
11: right-features-ll← **copy-node**(curr-group) + right-features-ll
12: ▷ move curr-group node to front
13: **end if**
14: capture-set-left ← split capture-set left on split-feature
15: capture-set-right ← split capture-set right on split-feature
16: **ExploreFeature**(capture-set-left, left-features-ll)
17: **ExploreFeature**(capture-set-right, right-features-ll)
18: **end while**

In Algorithm 2, we persist a linked list of bit-strings, each representing a feature group that hasn't been completely explored. As a preprocessing step, we sort the linked list by feature-group costs in increasing order, breaking ties

arbitrarily. We note that our recursive calls to Algorithm 2 preserves feature-ll's non-decreasing cost order. The cost order is guaranteed since the linked list is initially sorted increasingly and we only change the order of features on lines 10, 11 where we move the feature group we just split on to the front of the list. Since feature costs are one-time across feature groups, the new head of the linked list would have cost 0 – a lower bound on all feature costs.

We note that this algorithm and the original GOSDT algorithm has the same complexity. The original algorithm does a one-time pass of the feature vector to enqueue all possible splits and uses an optimized search algorithm for bit-strings to identify the next available feature. Our algorithm similarly does a one-time pass of available features in the least costly feature group (note we drop any feature group with no more available in line 7), with each split performing additional constant work to remove and prepend a node to the linked list. Our algorithm also uses the optimized bit-string search algorithm to find the next available split (line 4).

It is unclear how large the overhead of this algorithm would be compared to the original algorithm without an empirical evaluation.

3.6 Variable Feature Costs

A more generalized approach to implement and represent linear objective terms can be derived by realizing that in our proofs and definitions, nothing is inherently restricting feature costs to be constant. Namely, it is possible to replace our feature cost vector with a feature cost *function* vector whose output depends on the weights bit-string of a subproblem. More specifically, we can replace the constant feature cost vector with a vector of pointers to light-weight functions that depend on the weights bit-string of a subproblem. To satisfy our proofs, we note that these cost functions must be 1) non-negative and 2) non-increasing in each bit of the weights bit-string. We state these properties more formally.

Definition 3.4 (Feature Cost Functions)**.** A feature cost function for a feature i is a function $f_i : \{0,1\}^M \to \mathbb{R}$ where M is the number of original features. Furthemore, we say a feature cost function f_i is *valid* if the following holds true:

$$\forall x \in \{0,1\}^M, f_i(x) \geq 0 \tag{1}$$

$$\forall x, y \in \{0,1\}^M, \text{ if } \forall k \in [M] \text{ such that } x_k = 1 \text{ we have } y_k = 1, \text{ then } f_i(y) \geq f_i(x) \tag{2}$$

We say that a feature cost function f_i is *partially valid* if only the first property holds true, and invalid if neither is satisfied.

The second property is essential in the proofs of Bound A.1, Bound A.3, and Bound A.5 and their extensions where we rely on that fact that we cannot perform a split to produce a tree with a smaller total feature cost. This becomes possible if we allow invalid feature costs functions.

3.7 Handling discounts

In the medical setting, it is common for multiple tests to be ordered as a group for a discount. We call a set of features a *discount group* if splitting on more than two features in the set makes each feature incur a discounted version of their costs. Section 3.6 hints at certain types of costs, including discounted costs, that violates certain GOSDT weight Weights boundaries by allowing splits to reduce the overall feature costs incurred on a tree. To see why this might be problematic, consider two features a and b, individually with $w_a = w_b = \infty$ but is discounted to be free when split together. Without loss of generality, suppose we first split on feature a. Then the feature cost of a would block GOSDT from exploring further splits, causing it to miss the discount when splitting on both a and b.

So to incorporate discounts to GOSDT, we inevitably turn off certain bounds. We note however that we can still solve for the optimal discounted tree since certain critical boundaries are not affected by discounts, such as Bound A.1 and Bound A.2.

Rather than a clever way to realize discounts in GOSDT, this section demonstrates how feature cost functions can be used to implement discount, a new type of cost, with minimal changes. It is also worth noting that we can detect that a new type of cost violates certain boundaries by checking whether a cost function that represents it in valid. Indeed, we will see that the feature cost function that handles discounts is only partially valid.

One approach to incorporate discounts into GOSDT is by adding checks somewhere in the GOSDT algorithm for whether splitting on a new feature gives discounts to any splits on the path so far. This may involve writing additional helper functions to determine whether two features can give discounts and changes to the C++ and Python interfaces to pass maps and structures to keep track of discount groups.

Suppose instead we have an implementation of featured cost functions. Suppose we have two features, features 1 and 2, that initially have cost w_1 and w_2 but if both are split on would have incur the discounted costs w_1' and w_2' such that $w_1' < w_1$ and $w_2' < w_2$. Then we can define a partially valid feature cost function that takes the weights bit-string $\mathbf{W}$ as the following.

Algorithm 3 DiscountFunctionFeature1

```
1: if W_1 is 1 and W_2 is 1 then
2:     return w_1'
3: else
4:     return w_1 × W_1
5: end if
```

Algorithm 3 can be used as the feature cost function for feature 1 to compute the discounted cost. Feature 2 has an analogous discount function, and pointers to the two functions would make up the feature cost function vector. Using feature cost functions, we do not need to make any additional changes into the

code base to extend GOSDT with Weights with discounts (outside of disabling the violated pruning bounds). Perhaps the sparsity coefficient, λ, can also be incorporated into the feature cost functions.

4 Case Study and Numerical Experiments

We compare our approach with Pystreed across 14 datasets. Overall, we demonstrate that GOSDT with Weights achieves identical objective values as Pystreed, although with greater runtimes.

4.1 Experimental Setup

Unless otherwise specified, all of our experiments are ran according to the following setup. We run our experiments with Pystreed's cost-sensitive example datasets which are preprocessed and binarized according to instructions by S. Lomax et al. [19]. In total, there are 15 datasets, each pre-split into training and testing sets 100 times at random. We omit the nursery dataset since we run into errors when training Pystreed on it.

For small datasets, we run our experiments with depth limit set to $\{2, 3, 4, 5\}$. For larger datasets, we consider only trees with max depth in $\{2, 3\}$. Each of the 100 split is ran once per dataset per depth with timeout set to 60 seconds, and the training time is computed as the mean over the 100 dataset splits. The sparsity coefficient is set to 0 for all experiments except the sparsity experiment. We run all experiments on Intel(R) Xeon(R) Silver 4309Y CPUs.

We adapt the feature costs and misclassification costs in Pystreed's repository for the datasets. From S. Lomax et al.'s feature costs, Pystreed generate three cost-matrices with misclassification cost that are low, middle, and high relative to the feature costs. In our experiments, we find that low and middle misclassification costs mostly produce the trivial tree, so we run all experiments with high classification costs which encourages splitting.

To port real dollar costs into GOSDT with Weights, we normalize the feature and misclassification costs by dividing the cost matrix C and the feature cost vector by the $\max_{ij} C_{ij} \times N$. Since our objective is linear, scaling by a positive constant factor would not affect the minimizer. To compare results across GOSDT with Weights and Pystreed, we extract the generated trees into a Python Tree class to compute the objective value within the same numerical context.

Since Pystreed only supports a clock with resolution in seconds, we add a clock within their code to extract precise training times. In Pystreed's classifier, we also change the feature search order from "gini" to "in-order" to be consistent with GOSDT. In our experiments, we set the "discount" column of Pystreed's feature cost configuration files to be the same cost as the regular feature costs, since GOSDT with Weights doesn't currently handle discounts. We propose discounts as a future direction in Section 5.1.

4.2 Runtime Experiment

Since both GOSDT with Weights and Pystreed hold optimal guarantees, we are primarily interested in a runtime comparison. Overall, we observe that GOSDT with Weights is significantly slower than Pystreed. We refer readers to Table 1 for detailed results of the basic runtime experiment.

Disparity In Objective Value In the majority of cases, GOSDT with Weights and Pystreed either attain the identical tree or yield the same objective value, demonstrating optimality. However, in a small number of cases, we observe a disparity between the objective value of GOSDT and Pystreed. We document these cases and summarize them in Table 2 and Table 3. The objective disparity value is computed by subtracting the objective value of GOSDT with Weights with that of Pystreed. Hence, a negative disparity indicates that GOSDT with Weights outperforms Pystreed and vice versa. We observe is that a positive disparity is only ever detected when training on depth = 5.

We are currently unsure of the cause of the disparity, but we note some insights. Firstly, we can possibly rule out numerical error as the source of disparity. One might suspect this to be a source of variance since the normalization procedure in our experiments can be dramatic, at times scaling the weights by $\frac{1}{10^{-9}}$. Furthermore, while Python by default uses double-precision floating point numbers, many of GOSDT's C++ calculations are performed with **float**s (instead of **double**s) which are single-precision to may use fewer digits than we need. However, a sanity check demonstrates that GOSDT with Weight's native (C++) objective value and the objective value computed in Python is never different by more than 10^{-8}, while some of our disparities are much greater than that (up to around 10^{-3}). In our tables, we remove cases with absolute disparities smaller than 10^{-7}, but we have all cases well documented. The disparities we see are at most 1% of the optimal objective values, and these cases represent less than 10% of all solves across datasets and depth limits. It is perhaps worth attempting to train Pystreed over our normalized feature and misclassification costs to further verify that the error we see is not an artifact of numerical approximation.

It is worth noting that another reason we extract the trees from both models to compare in Python is because we cannot make sense of the Pystreed's native objective value. When we ran Pystreed on *breast-1* with depth = 5, we obtain a feature costs of 22.97 which we cannot interpret (it is neither the cost incurred per sample nor the total cost). This may be a point of concern since if Pystreed is optimizing over a different objective then it is certainly possible to have disparities between the two objective values. However, we dismiss this since in most cases, both models obtain the same optimal objective value if not the same tree.

4.3 Overhead of GOSDT with Weights

To measure the overhead of our implementation, we run GOSDT with Weights against GOSDT on the same training sets, depths, and with λ set to 0. Overall, we observe that our implementation roughly doubles the original runtime, although for some datasets, such as the *diabetes* and *hepatitis* and *heart* at depth=5, we observe a faster solve due to feature costs pruning. We also observe that the overhead on runtime decreases with depth, possibly because a larger feature space enables more pruning by the weighted objective. Our results are summarized in Table 5

Fundamentally, our implementation differs minimally from GOSDT. We suspect that primary source of additional overhead is in the additional copying operations required to pass the weight vector, whose frequency grows exponentially with the number of subproblems and the size of the dependency graph. Another possible factor is reduced subproblem reuse. Whereas two identical capture-set bit-string previously identified to the same subproblem, a subproblem is now defined by both the capture-set bit-string and the incurred weights bit-string. Hence, it is more difficult to save on computation by caching if at all.

4.4 Optimizing GOSDT - The Terminal Solver

Through our experiments, we find that Pystreed's performance over GOSDT is an implementational advantage. We observe that their runtime advantage is largely due to their specialized terminal-solver, which uses an optimized algorithm to quickly compute the optimal depth two tree of a subproblem. In Table 4, we can see that enabling the terminal solver gives Pystreed a signicant runtime improvement. Without the terminal solver, Pystreed's performance is within the same order of magnitude as GOSDT with Weights.

As part of this work we also implemented the terminal solver algorithm to investigate its runtime benefits on GOSDT with Weights. We refer to GOSDT with Weights and terminal solver as *GOSDT with Weights Optimized* in our tables and we showcase our results in Table 1. Overall, we observe a similar speedup of up to one of magnitude on average.

We suspect that our speedup is achieved primarily through reducing the problem graph size and hence the frequency of small operations such as copying bit-strings between Tasks and Messages. When performing a depth = 5 solve of *breast-1*, we observe that the total dependency graph size is reduced by an order of magnitude (from 10^5 to 10^4), resulting in a five-times speedup.

We note a few issues that haven't been resolved in our implementation. Firstly, our terminal solver is able to compute the optimal depth two tree, but we are unable to extract this tree efficiently with clean changes to the code. The initial idea to incorporate the terminal solver was to treat it as a pruning boundary where the solver would compute the 3 splits that uniquely identifies the optimal depth two tree of a subproblem when its depth budget runs out. We note that by the optimal depth two tree of a subproblem, we refer to **the optimal tree with depth at most two**. Using these splits, we would prune all

other features in the feature bit-string of the subproblem. GOSDT would in turn proceed as usual and find the optimal tree, enqueuing only a series of exploration tasks that permute the splits defining the optimal depth two tree. Although this strategy is modular and minimizes changes to the GOSDT algorithm, we find that it can yield a suboptimal tree since GOSDT uses the feature bit-string to identify both the features to explore and features to propagate updates to. Pruning all other features except the optimal splits could prune the features from which the subproblem originated, thereby blocking updates to the parents of the subproblem.

A natural solution to this was to retain the feature whose split resulted in the subproblem to ensure correct updates. However, we observe that our performance speedup disappears with this method and that our dependency graph returns to its original size.

Our current implementation uses the terminal solver to compute the optimal depth two tree at a node when the depth budget is 3. Since the terminal solver gives us the optimal depth two tree rooted at the subproblem, we can compute the optimal objective value of the subproblem and set its upper and lower bound to the optimal value. This resolves the subproblem after a final upwards propagation of the objective value since GOSDT with Weights would treat the problem as "converged". This implementation retains the speed up from the terminal solver while still letting us get the optimal objective value. However, the downside is that there isn't a clean way to extract the optimal depth two trees from the terminated nodes. One approach is to persist the defining splits of the optimal depth two tree in the subproblem bitmask, but this change makes little semantic sense and puts more load on the Bitmask class which is we wish to keep lightweight and uncomplicated.

Another drawback of using the terminal solver is that it works poorly with multi threading. In preliminary experiments, we observe that running multi threaded GOSDT with the terminal solver can remove the benefits on runtime and even slowdown GOSDT.

Lastly, we currently have some unresolved bugs with our terminal solver. While we often obtain the optimal objective, on certain datasets, enabling the terminal solver leads to a lower objective value than optimal, suggesting a bug in our objective calculations or updates. We document all of these cases.

4.5 Performance on sparse trees

One way GOSDT with Weights can achieve speed up is with a sparsity constant. Indeed many of the adapted and extended pruning bounds of GOSDT with Weights involve the lambda constant, which become invaluable as the search space grows exponentially. Our experiments demonstrate that while a sparsity constant always rewards GOSDT by contributing to pruning, that sparsity can punish the performance of Pystreed, potentially increasing performance by 2-fold.

There isn't good way to compare between GOSDT's and Pystreed's approaches to solve for sparse trees. While GOSDT directly involves the sparsity

constant in its objective function by penalizing each new leaf, Pystreed's cost sensitive classifier controls sparsity with a hard limit on the number of tree nodes. It is worth noting that Murtree, from which Pystreed is extended, is able to handle a sparsity constant. However, upon manually enabling the sparsity constant parameter in the Pystreed Cost Sensitive classifier, we find that the sparsity constant is not handled since GOSDT is always able to find superior trees. Thus, in our sparsity experiment, we first train the optimal cost-sensitive sparse GOSDT tree with $\lambda = \frac{1}{N}$, then use the number of nodes of the resulting tree as the node-limit for Pystreed. We note that this may give Pystreed a small advantage by allowing it to optimize over a smaller search space. We refer readers to Table 6 for detailed results.

5 Discussion

5.1 Future Work

Delayed Costs A clear next step is furthering the cost-sensitivity of GOSDT. Our work addresses feature-cost sensitivity by including it in the calculations of the objective function. However, many more types of costs are possible to make GOSDT more adaptable to practical workflows. We proprose a way to handle discounts in Section 3.7, which Pystreed handles. One particularly difficult cost to consider is delayed costs, which is reasonable when the values of a feature or test, such as blood test results, may not be immediately available. In this case, a doctor is forced to order all successive tests in the subtree below the delayed test regardless of its result, which corresponds as incurring the feature cost of the entire subtree rooted at the delayed split.

An inherent challenge of delayed tests is that when splitting on a delayed test, the objective of a fixed left subtree is variable depending on the right subtree since a sample would not incur the cost of a certain feature if that feature's cost is already incurred in the right subtree. In particular, the left and right subproblem would no longer be independent if their parent node is a delayed test. Since this conflicts with the inherent, independent, subproblem-based design of GOSDT, resolving this problem may require a different optimization framework.

A Recipe for Extending GOSDT It is also a good engineering challenge to consider whether it is possible to design a systematic way to implement costs that presents as linear terms in the GOSDT objective function. Such a recipe can reduce unnecessary repetition of the same shotgun surgery to include another type of cost. More specifically, to include feature costs into GOSDT, we extended many functionalities of the **Bitmask**, **Task** and **Messages** classes which combine to represent subproblems throughout the program. It seems intuitive that adding other similar types of linear cost to GOSDT would require a similar set of changes, which motivates the design challenge.

Feature Cost Sensitive Rashemon Sets Another direction of future work is to extend cost-sensitivity into TreeFarms. Rather than a single optimal tree, TreeFarms is an optimal decision tree framework extended from GOSDT that generates the Rashemon set of all trees within an ϵ of the optimal objective value. Our experiment demonstrates that it is completely possible to find different trees that are both cost-sensitively optimal. Thus, it is reasonable to have access to these different solutions and give the user a choice for good trees which may be more suitable for the context.

One consideration to note is that Treefarms relies on the discrete nature of the objective space. Rui et. al highlight that the efficiently of Treefarms stems from the fact that the loss function takes on a discrete number of values (roughly n distinct values). However, this property is violated with arbitrary feature costs, which could potentially make this extension infeasible. A neat solution is to discretize the values of misclassification cost, feature cost, and lambda. For example, we could enforce that the costs and lambda must be multiples of $\frac{1}{c}$ for some positive constant c. Then it is straight forward to show that the loss function can take on around cn values. If we require more resolution, we can simply increase c.

Implemention of Bounds and Algorithms Due to our time constraint, our implementation of GOSDT with Weights is the minimal extension of GOSDT. In particular, we did not implement the extended bounds in Appendix B and the feature evaluation order heuristic in Section 3.5. One future direction is to implement these ideas and evaluate them. Our intuition is that the increasing-cost evaluation order heuristic in 2 may generate too much overhead for noticeable speedup. The extended bounds in Appendix B could be potentially helpful, especially when feature costs are non-trivial relative to misclassification costs. These extended bounds requires the subproblem to efficiently compute the weight of the least costly available split and all splits that are more costly than a certain split, and may require some thought to implement. Perhaps ideas in Section 3.5 could be adapted.

References

[1] P. D. Turney, "Types of cost in inductive concept learning," 2002.

[2] S. Maliah and G. Shani, "Using pomdps for learning cost sensitive decision trees," *Artificial Intelligence*, vol. 292, p. 103400, 2021.

[3] P. D. Turney, "Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm," 1995.

[4] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is np-complete," *Information Processing Letters*, vol. 5, no. 1, pp. 15–17, 1976.

[5] D. Bertsimas and J. Dunn, "Optimal classification trees," *Machine Learning*, vol. 106, no. 7, pp. 1039–1082, 2017.

[6] S. Verwer and Y. Zhang, "Learning optimal classification trees using a binary linear program formulation," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 1625–1632, Jul. 2019.

[7] R. Mazumder, X. Meng, and H. Wang, "Quant-BnB: A scalable branch-and-bound method for optimal decision trees with continuous features," in *Proceedings of the 39th International Conference on Machine Learning* (K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, eds.), vol. 162 of *Proceedings of Machine Learning Research*, pp. 15255–15277, PMLR, 17–23 Jul 2022.

[8] E. Demirović, A. Lukina, E. Hebrard, J. Chan, J. Bailey, C. Leckie, K. Ramamohanarao, and P. J. Stuckey, "Murtree: Optimal classification trees via dynamic programming and search," 2022.

[9] J. G. M. van der Linden, M. M. de Weerdt, and E. Demirović, "Necessary and sufficient conditions for optimal decision trees using dynamic programming," 2025.

[10] V. Zubek and T. Dietterich, "Pruning improves heuristic search for cost-sensitive learning," 03 2002.

[11] S. L. Salzberg, "C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993," *Machine Learning*, vol. 16, no. 3, pp. 235–240, 1994.

[12] M. Tan, "Cost-sensitive learning of classification knowledge and its applications in robotics," *Machine Learning*, vol. 13, no. 1, pp. 7–33, 1993.

[13] M. Núñez, "The use of background knowledge in decision tree induction," *Machine Learning*, vol. 6, no. 3, pp. 231–250, 1991.

[14] S. W. Norton, "Generating better decision trees," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, (San Francisco, CA, USA), p. 800–805, Morgan Kaufmann Publishers Inc., 1989.

[15] S. Lomax and S. Vadera, "A survey of cost-sensitive decision tree induction algorithms," *ACM Comput. Surv.*, vol. 45, Mar. 2013.

[16] S. Nijssen and E. Fromont, "Optimal constraint-based decision tree induction from itemset lattices," *Data Mining and Knowledge Discovery*, vol. 21, no. 1, pp. 9–51, 2010.

[17] G. Aglin, S. Nijssen, and P. Schaus, "Learning optimal decision trees using caching branch-and-bound search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 3146–3153, Apr. 2020.

[18] J. Lin, C. Zhong, D. Hu, C. Rudin, and M. Seltzer, "Generalized and scalable optimal sparse decision trees," 2022.

[19] S. Lomax, S. Vadera, and M. Saraee, "A multi-armed bandit approach to cost-sensitive decision tree learning," pp. 162–168, 12 2012.

A Proofs for Adapated Boundaries

In this section we have all adapted proofs of GOSDT pruning bounds. There proofs consider only additive losses that are monotonically increasing on false positives (FP) and false negatives (FN). We first recall some definitions from previous papers:

Definition A.1. For two fixed leaf sets d_{fix}, d'_{fix} from trees d, d' respectively, we say d_{fix} starts with d'_{fix} if the leaves of d_{fix} are contained in d'_{fix}, and we write $d_{fix} \subseteq d'_{fix}$. The set of all such d' for d is denoted as $\sigma(d)$.

Definition A.2 (Objective Lower Bound)**.** For a tree $d = (d_{fix}, \delta_{fix}, d_{split}, \delta_{split}, K, H)$, we define

$$b(d_{fix}, \mathbf{x}, \mathbf{y}) := l_p(d_{fix}, \delta_{fix}, \mathbf{x}, \mathbf{y}) + \lambda H$$

From previous works, we know that this is a lower bound on the objective, namely:

$$b(d_{fix}, \mathbf{x}, \mathbf{y}) \leq R(d, \mathbf{x}, \mathbf{y})$$

We also introduce two new definitions.

Definition A.3 (Bound on Objective Weight Term)**.** For a tree $d = (d_{fix}, \delta_{fix}, d_{split}, \delta_{split}, K, H)$, we define a lower bound on the weight term by only applying the splitting cost on samples that are captured by the fixed leaves. Namely,

$$W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) = \left(\frac{1}{N} \sum_{i=1}^{M} \sum_{l=1}^{K} \sum_{n=1}^{N} w_i \Big[\Big(\bigvee_{\{i | m_i \in G_i\}} v_i^{l_k} \Big) \wedge cap(l, x_n) \Big] \right)$$

Definition A.4 (Weighted Objective Lower Bound)**.** We define a new function b_w. We show later on that b_w can be used as a lower bound on the weighted objective.

$$\begin{aligned} b_w(d_{fix}, \mathbf{x}, \mathbf{y}) &:= b(d_{fix}, x, y) + W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) \\ &= l_p(d_{fix}, \delta_{fix}, \mathbf{x}, \mathbf{y}) + W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) + \lambda H \end{aligned}$$

A.1 Hierarchical weighted objective lower bound for arbitrary monotonic losses

Theorem A.5 (Hierarchical weighted objective lower bound for arbitrary monotonic losses)**.** *Let the loss function $l(d, x, y)$ be monotonically increasing in FP, FN. Thus, we may rewrite the loss function as $\bar{l}(FP, FN)$. Let $d = (d_{fix}, \delta_{fix}, d_{split}, \delta_{split}, K, H)$ be a labelled tree with fixed leaves d_{fix} and let FP_{fix}, FN_{fix} be the false positives and false negatives of d_{fix}. Then*

$$b_w(d_{fix}, \mathbf{x}, \mathbf{y}) \leq R_w(d, \mathbf{x}, \mathbf{y})$$

Furthermore, $\forall d' \in \sigma(d)$,

$$b_w(d_{fix}, \mathbf{x}, \mathbf{y}) \leq R_w(d', \mathbf{x}, \mathbf{y})$$

Proof.

$R_w(d, \mathbf{x}, \mathbf{y}) = R(d, \mathbf{x}, \mathbf{y}) + W(d, \mathbf{x}, \mathbf{y}) \geq b(d_{fix}, x, y) + W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) = b_w(d_{fix}, x, y)$

We also have:

$$R_w(d', \mathbf{x}, \mathbf{y}) \geq b(d'_{fix}, x, y) + W_{fix}(d'_{fix}, \mathbf{x}, \mathbf{y}) \geq b(d_{fix}, x, y) + W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y})$$
$$= b_w(d_{fix}, x, y)$$

as required. □

Namely, the weighted objective lower bound on a parent is a lower bound for its children. Thus, if a parent's lower bound exceeds our current best weighted objective, we may prune the parent and all its children.

A.2 Objective lower bound with one-step lookahead

Theorem A.6 (Objective lower bound with one-step lookahead)**.** *Let d be a H leaf tree with K leaves fixed and let R^c_w be the current best objective. If $b_w(d_{fix}, \mathbf{x}, \mathbf{y}) + \lambda \geq R^c_w$, then for any child tree $d' \in \sigma(d)$, its fixed leaves d'_{fix} include d_{fix} and $H' > H$, it follows that $R(d', \mathbf{x}, \mathbf{y}) \geq R^c_w$.*

Proof. The original version applies trivially. We let $FP_{fix}, FN_{fix}, FP_{fix'}, FN_{fix'}$ denote the false positives and negatives of d, d' respectively. Suppose $b(d_{fix}, x, y) + \lambda \geq R^c_w$. Then,

$$\begin{aligned}
&R_w(d', \mathbf{x}, \mathbf{y}) \\
&\geq \bar{l}(FP_{fix'}, FN_{fix'}) + W_{fix}(d'_{fix}, \mathbf{x}, \mathbf{y}) + \lambda H' \\
&\geq \bar{l}(FP_{fix}, FN_{fix}) + W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) + \lambda H' \\
&= \bar{l}(FP_{fix}, FN_{fix}) + \lambda H + W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) + \lambda(H' - H) \\
&\geq b_w(d_{fix}, \mathbf{x}, \mathbf{y}) + \lambda \\
&\geq R^c_w
\end{aligned}$$

Since $H' - H \geq 1$. □

A.3 Hierarchical objective lower bound for subtrees and additive losses

Theorem A.7 (Hierarchical objective lower bound for subtrees and additive losses)**.** *Let the loss functions $l(d, \mathbf{x}, \mathbf{y})$ be monotonically increasing in FP and FN, and let the loss of a tree d be the sum of the losses of the leaves. Also let R^c_w be the current best weighted objective. For some tree d, we consider an split k with weight w_k that generates two sub-trees d_{left} and d_{right} with H_{left} leaves and H_{right} respectively. The data captured by the left tree is $(\mathbf{x}_{left}, \mathbf{y}_{left})$ and the data captured by the right tree is $(\mathbf{x}_{right}, \mathbf{y}_{right})$. Let $b_w(d_{left}, \mathbf{x}_{left}, \mathbf{y}_{left})$ and $b_w(d_{right}, \mathbf{x}_{right}, \mathbf{y}_{right})$ be the weighted objective lower bound of the left sub-tree and right sub-tree respectively. If $b_w(d_{left}, \mathbf{x}_{left}, \mathbf{y}_{left}) + b_w(d_{right}, \mathbf{x}_{right}, \mathbf{y}_{right}) > R^c_w$ then d is not optimal.*

Proof.

$$
\begin{aligned}
& R_w(d, \mathbf{x}, \mathbf{y}) \\
&= l(d, \mathbf{x}, \mathbf{y}) + W(d, \mathbf{x}, \mathbf{y}) + \lambda H \\
&\geq l(d_{left}, \mathbf{x}_{left}, \mathbf{y}_{left}) + l(d_{left}, \mathbf{x}_{left}, \mathbf{y}_{left}) + \\
& \quad W(d_{left}, \mathbf{x}_{left}, \mathbf{y}_{left}) + W(d_{right}, \mathbf{x}_{right}, \mathbf{y}_{right}) + \lambda H_{left} + \lambda H_{right} \\
&\geq b_w(d_{left}, \mathbf{x}_{left}, \mathbf{y}_{left}) + b_w(d_{right}, \mathbf{x}_{right}, \mathbf{y}_{right})
\end{aligned}
$$

Then we have

$$R_w(d, \mathbf{x}, \mathbf{y}) \geq b_w(d_{left}, \mathbf{x}_{left}, \mathbf{y}_{left}) + b_w(d_{right}, \mathbf{x}_{right}, \mathbf{y}_{right}) > R_w^c$$

Therefore d is suboptimal. We note that that in the first inequality, we deliberately leave out the weight of feature k. We would have equality if the root split of both d_{left} and d_{right} is k. If we include the weight term of k, we would not necessarily have equality since the weights $W(d_{right}, \mathbf{x}_{right}, \mathbf{y}_{right})$ and $W(d_{left}, \mathbf{x}_{left}, \mathbf{y}_{left})$ don't have knowledge of the split k so they may reincur the cost of feature k again on certain samples. □

A.4 Upper bound on the number of leaves

Theorem A.8 (Upper bound on the number of leaves)**.** *Proof.* We note that the original proof follows even for our weighted objective function. Thus, we refer readers to the GOSDT paper for a complete proof. □

Recall that L is a function that counts the number of leaves of a given complete tree.

A.5 Incremental Progress Bound to Determine Splitting

Theorem A.9 (Incremental Progress Bound to Determine Splitting)**.** *Let d^* be any optimal tree with objective R^*. Consider tree d' derived from d^* by deleting a pair of leaves l_i and l_{i+1} and adding their parent leaf l_j, $d' = (l_1, ..., l_{i-1}, l_{i+2}, ..., l_H, l_i)$. Let $\tau := \bar{l}(FP_{d'}, FN_{d'}) - \bar{l}(FP_{d'} - FP_{l_j}, FN_{d'} - FN_{l_j})$. Then τ must at least be λ.*

Proof. $l(d', \mathbf{x}, \mathbf{y}) = \bar{l}(FP_{d'}, FN_{d'}), l(d^*, \mathbf{x}, \mathbf{y}) = \bar{l}(FP_{d'} + FP_{l_i} + FP_{l_{i+1}} - FP_{l_j}, FN_{d'} + FN_{l_i} + FN_{l_{i+1}} - FN_{l_j})$. The difference between them is maximized when l_i, l_{i+1} correctly classifies all captured data, so the difference between the losses is upper bounded by τ.

$$l(d', \mathbf{x}, \mathbf{y}) - l(d^*, \mathbf{x}, \mathbf{y}) \leq \tau \tag{3}$$

$$l(d', \mathbf{x}, \mathbf{y}) \leq \tau + l(d^*, \mathbf{x}, \mathbf{y}) \tag{4}$$

which gives

$$\begin{aligned}
&l(d', \mathbf{x}, \mathbf{y}) + W_{fix}(d'_{fix}, \mathbf{x}, \mathbf{y}) + \lambda(H-1) \\
\leq\ & l(d^*, \mathbf{x}, \mathbf{y}) + W_{fix}(d'_{fix}, \mathbf{x}, \mathbf{y}) + \lambda(H-1) + \tau \\
\leq\ & l(d^*, \mathbf{x}, \mathbf{y}) + W_{fix}(d^*_{fix}, \mathbf{x}, \mathbf{y}) + \lambda(H-1) + \tau \\
\Rightarrow & R(d', \mathbf{x}, \mathbf{y}) \leq R(d^*, \mathbf{x}, \mathbf{y}) - \lambda + \tau \\
\Rightarrow & 0 \leq R(d', \mathbf{x}, \mathbf{y}) - R(d^*, \mathbf{x}, \mathbf{y}) \leq -\lambda + \tau \\
\Rightarrow & \lambda \leq \tau
\end{aligned}$$

To get first implication, we recall that d^* is of size H and d' is of size $H-1$. □

Thus any split that would result in less than λ of improvement in loss is not in the optimal tree.

A.6 Lower bound on incremental progress

Theorem A.10 (Lower bound on incremental progress extended)**.** *Let d^* be any optimal tree with objective R^*. Let d^* have leaves d_{fix}. Consider the tree d' derived from $d*$ by deleting a pair of leaves l_i, l_{i+1} and adding their parent leaf l_j. Define a_i as the incremental objective of splitting l_j to get l_i, l_{i+1}.*

$$a_i = l(d', \mathbf{x}, \mathbf{y}) - l(d^*, \mathbf{x}, \mathbf{y})$$

Assuming l_j splits on w_α to obtain l_i, l_{i+1}. Then:

$$a_i \geq \lambda + \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(l_j, x_i)$$

If w_α is from a feature group that we have used on the path to l_j, we obtain the original bound:

$$a_i \geq \lambda$$

Proof.

$$\begin{aligned}
&R_w(d', \mathbf{x}, \mathbf{y}) \\
=& l(d', \mathbf{x}, \mathbf{y}) + \lambda(H-1) + W(d', \mathbf{x}, \mathbf{y}) \\
=& \alpha_i + l(d^*, \mathbf{x}, \mathbf{y}) + \lambda(H-1) - \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(l_j, x_i) + W(d^*, \mathbf{x}, \mathbf{y}) \\
=& R_w(d^*, \mathbf{x}, \mathbf{y}) + \alpha_i - \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(l_j, x_i) - \lambda
\end{aligned}$$

By optimality of d^*, we have $0 \leq R_w(d', \mathbf{x}, \mathbf{y}) - R_w(d^*, \mathbf{x}, \mathbf{y})$, which gives

$$\lambda + \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(l_j, x_i) \leq \alpha_i$$

Indeed, if w_α is from a feature group that we have used on the path to l_j, we do not incur addition cost from the split. Namely,

$$W(d^*, \mathbf{x}, \mathbf{y}) = W(d', \mathbf{x}, \mathbf{y})$$

So instead, we have

$$\begin{aligned} & R_w(d', \mathbf{x}, \mathbf{y}) \\ =& l(d', \mathbf{x}, \mathbf{y}) + \lambda(H-1) + W(d', \mathbf{x}, \mathbf{y}) \\ =& \alpha_i + l(d^*, \mathbf{x}, \mathbf{y}) + \lambda(H-1) + W(d^*, \mathbf{x}, \mathbf{y}) \\ =& R_w(d^*, \mathbf{x}, \mathbf{y}) + \alpha_i - \lambda \end{aligned}$$

Similarly, by optimality of d^*, we can rearrange to get

$$a_i \geq \lambda$$

as required. □

A.7 Permutation Bound

Theorem A.11 (Permutation Bound)**.** *Certainly if the paths to leaves are invariant, each sample will incur the same weighted cost. So trees that are permutations of each other have identical lower bound and weighted objective.*

A.8 Equivalent Points Bound

Theorem A.12 (Equivalent Points Bound)**.** *This bound is one on the loss function, which is independent to the weights in the objective. So the theorem still applies directly:*

$$l(d', \mathbf{x}, \mathbf{y}) \geq \bar{l}(FP_{fix} + FP_e, FN_{fix} + FN_e)$$

And any application of this bound would still hold.

A.9 Similar Support Bound

Theorem A.13 (Similar Support Bound)**.** *Proof.* The maximum difference between d and D is achieved if one misclassifies all and the other one correctly classifies all. Further, that the one that misclassifies incurs the maximum weight of all unseen feature groups on the path to the node (a more loose but general bound is possible by just using the maximum weight), denoted w_{max}, while the other node incurs no split cost. Let the internal node be l.

$$\gamma := \max_{a \in \{0,\ldots,|\omega|\}} [\bar{l}(FP_{-\omega}+a, FN_{-\omega}+|\omega|-a)] + \frac{w_{max}}{N} \sum_{i=1}^{N} cap(l, x_i) - \bar{l}(FP_{-\omega}, FN_{-\omega})$$

Then certainly

$$|R_w(d, \mathbf{x}, \mathbf{y}) - R_w(D, \mathbf{x}, \mathbf{y})| \leq \gamma \iff \gamma \geq R_w(d, \mathbf{x}, \mathbf{y}) - R_w(D, \mathbf{x}, \mathbf{y}) \geq -\gamma$$

We also have (letting d^* be the optimal tree that's a descendent of d, and letting D' be d^* but using a different split at l)

$$\begin{aligned}
&\min_{d^+ \in \sigma(d)} R_w(d^+, \mathbf{x}, \mathbf{y}) \\
=&R_\omega(d^*, \mathbf{x}, \mathbf{y}) \\
\geq&R(D', \mathbf{x}, \mathbf{y}) - \gamma \\
\geq& \min_{D^+ \in \sigma(D)} R(D^+, \mathbf{x}, \mathbf{y}) - \gamma
\end{aligned}$$

Using the same logic, we get

$$\min_{D^+ \in \sigma(d)} R_w(D^+, \mathbf{x}, \mathbf{y}) \geq \min_{d^+ \in \sigma(D)} R(d^+, \mathbf{x}, \mathbf{y}) - \gamma$$

So

$$|\min_{d^+ \in \sigma(d)} R_w(d^+, \mathbf{x}, \mathbf{y}) - \min_{D^+ \in \sigma(D)} R(D^+, \mathbf{x}, \mathbf{y})| \leq \gamma$$

overall.

□

B Proofs for Extended Boundaries

Theorem B.1 (node-specific feature-pruning by weight, leveraging look-ahead)**.** *Let d be a H-leaf tree with K leaves fixed such that $H > K$ (d_{split} is non-empty) and let R_w^c be the current best weighted objective. Consider a node $l_k \in d_{split}$. Let $I \subseteq \{1, ..., M\}$ be an index set for the available list of features for l_k (not seen on the path so far). Suppose for some index $j \in I$,*

$$b(d_{fix}, \mathbf{x}, \mathbf{y}) + w_j \sum_{n=1}^{N} cap(l_k, x_n) + \lambda \geq R_w^c \tag{5}$$

Then for any child d' of d that splits on l_k with feature α, if $w_\alpha \geq w_j$ then $R_w(d', \mathbf{x}, \mathbf{y}) \geq R^c$.

Proof. We let $FP_{fix}, FN_{fix}, FP_{fix'}, FN_{fix'}$ denote the false positives and neg-

atives of d, d' respectively.

$$
\begin{aligned}
& R_w(d', \mathbf{x}, \mathbf{y}) \\
&\geq \bar{l}(FP_{fix'}, FN_{fix'}) + W_{fix}(d'_{fix}, \mathbf{x}, \mathbf{y}) + \lambda H' \\
&\geq \bar{l}(FP_{fix}, FN_{fix}) + \lambda H + W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) + \frac{w_\alpha}{N} \sum_{n=1}^{N} cap(l_k, n) + \lambda(H' - H) \\
&\geq b(d_{fix}, \mathbf{x}, \mathbf{y}) + \frac{w_j}{N} \sum_{n=1}^{N} cap(l_k, n) + \lambda \\
&\geq R_w^c
\end{aligned}
$$

□

The implication for this theorem is that when we look at an internal node to split, we can find the minimum feature that satisfies Equation 5. Then any unseen feature with a greater weight cannot be considered by the internal node.

Theorem B.2 (Extended Incremental Progress Bound to Determine Splitting)**.** *Let d^* be any optimal tree with objective R^*. Consider tree d' derived from d^* by deleting a pair of leaves l_i and l_{i+1} and adding their parent leaf l_j, $d' = (l_1, ..., l_{i-1}, l_{i+2}, ..., l_H, l_j)$. We additionally assume that l_j splits into l_i and l_{i+1} through a feature that is in an unvisited feature group G_α with weight w_α by l_i, namely the feature cost satisfies the strict inequality:*

$$W_{fix}(d^*_{fix}, \mathbf{x}, \mathbf{y}) > W_{fix}(d'_{fix}, \mathbf{x}, \mathbf{y}) = W_{fix}(d^*_{fix}, \mathbf{x}, \mathbf{y}) - \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(x_i, l_i)$$

Let $\tau := \bar{l}(FP_{d'}, FN_{d'}) - \bar{l}(FP_{d'} - FP_{l_j}, FN_{d'} - FN_{l_j})$. Then

$$\tau \geq \lambda + \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(x_i, l_i)$$

Proof. Using Equation 3, we again write:

$$
\begin{aligned}
& l(d', \mathbf{x}, \mathbf{y}) + W_{fix}(d'_{fix}, \mathbf{x}, \mathbf{y}) + \lambda(H - 1) \\
&\leq l(d^*, \mathbf{x}, \mathbf{y}) + W_{fix}(d'_{fix}, \mathbf{x}, \mathbf{y}) + \lambda(H - 1) + \tau \\
&= l(d^*, \mathbf{x}, \mathbf{y}) + W_{fix}(d^*_{fix}, \mathbf{x}, \mathbf{y}) - \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(x_i, l_i) + \lambda(H - 1) + \tau \\
&\Rightarrow R(d', \mathbf{x}, \mathbf{y}) \leq R(d^*, \mathbf{x}, \mathbf{y}) - \lambda - \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(x_i, l_i) + \tau \\
&\Rightarrow 0 \leq R(d', \mathbf{x}, \mathbf{y}) - R(d^*, \mathbf{x}, \mathbf{y}) \leq -\lambda - \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(x_i, l_i) + \tau \\
&\Rightarrow \lambda + \frac{w_\alpha}{N} \sum_{i=1}^{N} cap(x_i, l_i) \leq \tau
\end{aligned}
$$

□

Thus, if we are splitting any node with a feature from a feature group not seen in the path so far, we have a stricter bound on the improvement.

Theorem B.3 (Upper bound on the number of leaves weighted)**.** *We sort the feature groups by their associated weights, namely* $w_{G_1} \leq w_{G_2} \leq ... \leq w_{G_M}$*, handling ties arbitrarily. The number of splits of a tree is the number of leaves - 1, so tree* d^* *has* $L(d^*) - 1$ *splits. Our binarization schemes yields* $|G_i|$ *distinct splits from each original feature* i*. The minimum weight penalty this would incur is*

$$w_{G_1} + \frac{2}{N} \sum_{i=1}^{k} w_{G_i}$$

where $k \in [M]$*, such that* k *is the maximum integer that satisfies.*

$$\sum_{i=1}^{k} |G_i| \leq \text{ number of leaves - 1} \tag{6}$$

Proof. In the trivial case, the number of leaves would be 1 and k would be 0. This correctly forces the minimum weight penalty to be 0.

Note that the coefficient on w_{G_1} is 1 since **some** split must have been used for the root node. All samples must incur the cost of this feature, which is lower bounded by w_{G_1}. The $\frac{2}{N}$ constant in from of the rest of the weights indicates that each internal node must captures at least two sample to split, since otherwise the node would have perfect accuracy by prediction the label of the sole sample and would not split further.

□

C Proofs for New Boundaries

Theorem C.1 (Feasibility of w_i)**.** *Let* N *be the number of samples in the dataset and* λ *be the sparsity constant. Then for a feature* m_i *with corresponding weight* w_i*, if*

$$w_i \geq N(\frac{1}{2} - \lambda)$$

then any tree that splits on m_i *is suboptimal.*

Proof. Consider any dataset and let N be the number of samples and let m_i be an arbitrary feature with corresponding feature weight w_i. Then the a priori upper bound for the weighted objective is $\frac{1}{2} + \lambda$, which is the objective of the base case 1-leaf tree with no splits. Now consider some tree d that splits on m_i with lower subsec:

$$b_w(d, \mathbf{x}, \mathbf{y}) = \bar{l}(NP, NF) + W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) + \lambda H$$

Note that this lower bound is minimized when the loss is 0 (wording?) so any such tree d must satisfy:

$$b_w(d, \mathbf{x}, \mathbf{y}) \geq W_{fix}(d_{fix}, \mathbf{x}, \mathbf{y}) + \lambda H \geq \frac{w_i}{N} + 2\lambda$$

By branch and bound, we know that d is suboptimal when

$$\frac{w_i}{N} + 2\lambda \geq \frac{1}{2} + \lambda$$

We rearrange to get

$$w_i \geq N(\frac{1}{2} - \lambda)$$

□

Dataset	Max Depth	GOSDT Weighted	GOSDT Optimized	Pystreed
annealing	2	72.9	**1.0**	2.51
annealing	3	6440.0	174.0	**46.7**
breast	2	8.05	**0.0**	0.202
breast	3	421.0	16.0	**3.22**
breast	4	8960.0	905.0	**58.7**
breast	5	60500.0	34200.0	**653.0**
car	2	5.02	**0.0**	1.06
car	3	174.0	**3.0**	6.8
car	4	2470.0	128.0	**55.7**
car	5	19600.0	3090.0	**298.0**
diabetes	2	**0.0**	**0.0**	0.415
diabetes	3	2.1	**0.0**	0.727
diabetes	4	5.51	3.0	**1.68**
diabetes	5	7.13	15.1	**3.1**
flare	2	5.06	**0.0**	0.309
flare	3	169.0	5.0	**2.39**
flare	4	2360.0	184.0	**20.7**
flare	5	18800.0	4400.0	**141.0**
glass	2	4.13	**0.0**	0.367
glass	3	88.5	**1.0**	2.01
glass	4	787.0	43.8	**13.5**
glass	5	3470.0	629.0	**62.4**
heart	2	2.59	**0.0**	0.268
heart	3	69.0	2.0	**1.29**
heart	4	701.0	88.3	**7.54**
heart	5	3840.0	1580.0	**33.5**
hepatitis	2	1.98	**0.0**	0.0953
hepatitis	3	61.9	2.0	**0.693**
hepatitis	4	663.0	85.0	**5.4**
hepatitis	5	3970.0	1530.0	**29.0**
iris	2	0.17	**0.0**	0.172
iris	3	4.22	**0.0**	0.416
iris	4	9.42	5.51	**1.17**
iris	5	10.4	21.2	**2.51**
krk	2	366.0	**1.0**	56.9
krk	3	24000.0	**115.0**	589.0
mushroom	2	176.0	**4.0**	14.9
mushroom	3	16500.0	929.0	**94.8**
soybean	2	185.0	**1.0**	8.95
soybean	3	21300.0	**157.0**	540.0
tictactoe	2	5.07	**0.0**	0.904
tictactoe	3	270.0	6.0	**5.59**
tictactoe	4	6040.0	326.0	**66.6**
tictactoe	5	60400.0	12800.0	**796.0**
wine	2	8.15	**0.0**	0.288
wine	3	384.0	10.0	**3.66**
wine	4	6120.0	563.0	**29.5**
wine	5	50000.0	19500.0	**137.0**

Table 1: **Runtime Experiment** - Comparison Between GOSDT with Weights, GOSDT with Weights Optimized, and Pystreed in milliseconds. Timeout = 60000 milliseconds.

Dataset	Depth	Split Id	Objective Disparity
car	4	13	-3.07576510177876e-07
flare	2	28	-0.0004474975154045
flare	3	32	-0.000479020195754
flare	3	36	-0.0002099294054282
flare	3	37	-0.0003708752829233
flare	4	14	-5.31771404389484e-05
flare	4	72	-0.0002010995537345
flare	5	17	-0.0001209280525259
flare	5	4	-1.4288115298046604e-05
flare	5	44	-0.0001239993812555
flare	5	49	-2.1681381883950634e-06
flare	5	59	-0.0001819388180378
flare	5	99	-0.0001192929642667
glass	5	1	4.336279599431725e-05
glass	5	100	0.0010869375448322
glass	5	16	0.0007618825303035
glass	5	17	0.0010788525367472
glass	5	18	0.0007699675383885
glass	5	19	0.0014621163673795
glass	5	20	0.0014140675088043
glass	5	22	0.0003286991444886
glass	5	24	0.000119696214433
glass	5	28	0.0004529975319449
glass	5	3	0.0010788525367473
glass	5	37	0.0007618825303035
glass	5	4	0.0010869375448322
glass	5	40	0.0012344052554578
glass	5	41	0.0002733109627845
glass	5	42	0.0012344052554578
glass	5	46	0.0010788525367472
glass	5	47	0.0004449125238599
glass	5	50	0.0002267518530675
glass	5	51	0.0010869375448322
glass	5	54	0.0009174352490142
glass	5	59	0.0010788525367472
glass	5	6	0.0010788525367472
glass	5	61	0.0016628729944519
glass	5	63	0.0007618825303035
glass	5	65	0.0004529975319449
glass	5	66	0.0007618825303035
glass	5	67	0.0010788525367472
glass	5	73	0.0010788525367472
glass	5	76	0.0008923247133769
glass	5	80	0.0003510186299659
glass	5	85	0.0012344052554578
glass	5	88	0.0010788525367472
glass	5	89	0.0007699675383885

Table 2: **Objective Disparity** - Difference in Objective Values between GOSDT with Weights and Pystreed (GOSDT with Weights - Pystreed). Positive means GOSDT has a larger objective, negative means smaller.

Dataset	Depth	Split Id	Objective Disparity
heart	3	12	-1.5892313682486225e-05
heart	3	2	-1.5892313682486225e-05
heart	3	24	-1.5892313682486225e-05
heart	3	29	-1.5892313682486225e-05
heart	3	34	-1.5892313682486225e-05
heart	3	42	-1.5892313682486225e-05
heart	3	5	-1.5892313682486225e-05
heart	3	52	-1.589231368251398e-05
heart	3	62	-7.946156841243113e-06
heart	3	67	-1.5892313682486225e-05
heart	3	73	-7.946156841243113e-06
heart	5	11	-0.000230438548396
heart	5	15	0.0006936994922405
heart	5	19	0.0006936994922405
heart	5	20	0.0006047025356186
heart	5	21	0.0019706468966284
heart	5	24	0.0005220625044697
heart	5	25	0.0006245679277217
heart	5	27	0.0005633825200442
heart	5	30	0.0007644202881276
heart	5	55	0.0004394224733207
heart	5	69	0.0004481632458461
heart	5	72	0.0019706468966284
heart	5	78	5.323925083630665e-05
heart	5	85	0.0006936994922405
heart	5	91	0.0019706468966284
heart	5	99	0.0006936994922405
hepatitis	4	11	0.000106974753958
hepatitis	4	21	0.0005348737697903
hepatitis	4	22	0.0008557980316645
hepatitis	4	43	0.001711596063329
hepatitis	4	86	0.0023534445870773
soybean	2	68	-0.0018779963006298
wine	5	44	0.0012812737599129
wine	5	54	0.0011695899491832
wine	5	79	0.0010181656731902
wine	5	80	0.0013857343813193

Table 3: **Objective Disparity** - Difference in Objective Values between GOSDT with Weights and Pystreed (GOSDT with Weights - Pystreed). Positive means GOSDT has a larger objective, negative means smaller.

Dataset	Max Depth	Pystreed	Pystreed NTerm	Runtime Increase
annealing	2	2.51	38.4	+1430%
annealing	3	46.7	1480.0	+3070%
breast	2	0.202	5.44	+2590%
breast	3	3.22	156.0	+4740%
breast	4	58.7	3410.0	+5710%
breast	5	653.0	53900.0	+8150%
car	2	1.06	7.48	+606%
car	3	6.8	99.3	+1360%
car	4	55.7	897.0	+1510%
car	5	298.0	5770.0	+1840%
diabetes	2	0.415	0.714	+72%
diabetes	3	0.727	1.97	+171%
diabetes	4	1.68	3.99	+138%
diabetes	5	3.1	5.92	+91%
flare	2	0.309	2.49	+706%
flare	3	2.39	41.9	+1650%
flare	4	20.7	496.0	+2300%
flare	5	141.0	4760.0	+3280%
glass	2	0.367	1.34	+265%
glass	3	2.01	16.0	+696%
glass	4	13.5	137.0	+915%
glass	5	62.4	842.0	+1250%
heart	2	0.268	1.73	+546%
heart	3	1.29	17.0	+1220%
heart	4	7.54	121.0	+1500%
heart	5	33.5	595.0	+1680%
hepatitis	2	0.0953	1.19	+1150%
hepatitis	3	0.693	12.2	+1660%
hepatitis	4	5.4	93.7	+1640%
hepatitis	5	29.0	525.0	+1710%
iris	2	0.172	0.343	+99%
iris	3	0.416	1.3	+213%
iris	4	1.17	3.17	+171%
iris	5	2.51	5.18	+106%
krk	2	56.9	376.0	+561%
krk	3	589.0	11000.0	+1770%
mushroom	2	14.9	363.0	+2340%
mushroom	3	94.8	1640.0	+1630%
soybean	2	8.95	46.4	+418%
soybean	3	540.0	2640.0	+389%
tictactoe	2	0.904	8.6	+851%
tictactoe	3	5.59	169.0	+2920%
tictactoe	4	66.6	2280.0	+3320%
tictactoe	5	796.0	25400.0	+3090%
wine	2	0.288	2.75	+855%
wine	3	3.66	42.1	+1050%
wine	4	29.5	267.0	+805%
wine	5	137.0	979.0	+615%

Table 4: **Terminal Solver Evaluation** - Comparison Between Pystreed and Pystreed without Terminal Solver in milliseconds. Timeout = 60000 milliseconds.

Dataset	Max Depth	GOSDT Weighted	GOSDT	Overhead
breast	2	8.05	**3.0**	+168%
breast	3	421.0	**175.0**	+141%
breast	4	8960.0	**4580.0**	+96%
breast	5	60500.0	60600.0	-
car	2	5.02	**2.0**	+151%
car	3	174.0	**87.3**	+99%
car	4	2470.0	**1360.0**	+82%
car	5	19600.0	**11600.0**	+69%
diabetes	2	0.0	0.0	-
diabetes	3	**2.1**	7.03	-71%
diabetes	4	**5.51**	55.1	-90%
diabetes	5	**7.13**	224.0	-97%
flare	2	5.06	**1.72**	+194%
flare	3	169.0	**58.8**	+187%
flare	4	2360.0	**819.0**	+188%
flare	5	18800.0	**7140.0**	+163%
glass	2	4.13	**1.0**	+313%
glass	3	88.5	**35.1**	+152%
glass	4	787.0	**322.0**	+144%
glass	5	3470.0	**1560.0**	+122%
heart	2	2.59	**1.0**	+159%
heart	3	69.0	**37.4**	+84%
heart	4	701.0	**586.0**	+20%
heart	5	**3840.0**	5760.0	-34%
hepatitis	2	1.98	**1.0**	+98%
hepatitis	3	61.9	**34.6**	+79%
hepatitis	4	663.0	**512.0**	+29%
hepatitis	5	**3970.0**	5070.0	-21%
iris	2	0.17	**0.0**	∞
iris	3	4.22	**3.55**	+19%
iris	4	9.42	**7.0**	+35%
iris	5	10.4	**7.7**	+35%
tictactoe	2	5.07	**2.0**	+154%
tictactoe	3	270.0	**115.0**	+135%
tictactoe	4	6040.0	**2650.0**	+128%
tictactoe	5	60400.0	**35000.0**	$\geq$+73%
wine	2	8.15	**3.02**	+170%
wine	3	384.0	**155.0**	+148%
wine	4	6120.0	**2710.0**	+126%
wine	5	50000.0	**27400.0**	+82%

Table 5: **Overhead Experiment** - Comparison Between GOSDT with Weights and GOSDT in milliseconds. Timeout = 60000 milliseconds.

Dataset	Depth	GOSDT Weighted	GOSDT Weighted Optimized	Pystreed
annealing	2	57.8	**1.0**	2.64
annealing	3	4640.0	121.0	**46.3**
breast	2	7.68	**0.0**	0.155
breast	3	361.0	14.3	**3.05**
breast	4	7070.0	759.0	**79.9**
breast	5	60500.0	27200.0	**302.0**
car	2	5.0	**0.0**	1.12
car	3	165.0	**3.0**	6.59
car	4	2150.0	117.0	**52.2**
car	5	14900.0	2730.0	**621.0**
diabetes	2	**0.0**	**0.0**	0.345
diabetes	3	2.0	**0.0**	0.786
diabetes	4	4.96	3.0	**2.02**
diabetes	5	5.94	12.8	**3.48**
flare	2	4.01	**0.0**	0.254
flare	3	98.4	3.18	**2.42**
flare	4	829.0	110.0	**21.1**
flare	5	2880.0	1960.0	**136.0**
glass	2	3.0	**0.0**	0.288
glass	3	68.8	**1.0**	1.74
glass	4	456.0	33.8	**18.0**
glass	5	1290.0	440.0	**124.0**
heart	2	2.05	**0.0**	0.286
heart	3	55.8	2.0	**1.17**
heart	4	425.0	82.1	**9.86**
heart	5	1590.0	1200.0	**79.7**
hepatitis	2	1.66	**0.0**	0.0943
hepatitis	3	32.1	2.0	**0.825**
hepatitis	4	193.0	56.4	**6.55**
hepatitis	5	553.0	623.0	**29.1**
iris	2	0.02	**0.0**	0.207
iris	3	4.0	**0.0**	0.449
iris	4	7.91	4.11	**1.53**
iris	5	8.88	14.8	**3.61**
krk	2	357.0	**1.0**	50.3
krk	3	23900.0	**115.0**	543.0
mushroom	2	154.0	**4.0**	14.6
mushroom	3	14200.0	746.0	**83.9**
soybean	2	181.0	**1.0**	10.1
soybean	3	19600.0	**150.0**	525.0
tictactoe	2	5.04	**0.0**	0.665
tictactoe	3	264.0	6.0	**5.12**
tictactoe	4	5820.0	325.0	**69.4**
tictactoe	5	60400.0	12700.0	**325.0**
wine	2	7.02	**0.0**	0.281
wine	3	282.0	8.86	**3.4**
wine	4	3400.0[32]	416.0	**44.7**
wine	5	18800.0	11600.0	**260.0**

Table 6: **Sparsity Experiment** - Runtime Comparison Between GOSDT with Weights, GOSDT with Weights Optimized, and Pystreed in milliseconds. Timeout = 60000 milliseconds, $\lambda = \frac{1}{N}$.