474 Research Meeting

2-10-2025

Objective(s)

- Find a (close to) globally optimal sparse tree with minimal “decision sparsity” (depth)
    - Achieve minimal depth through multi-way splitting so that same level of complexity can be reached with fewer decisions.
    - Can minimize variable repeats in any given path on the tree.
- *Achieve tractability – not possible to enumerate all k splits, must efficiently prune suboptimal branches.*
    - *Store possible k-way splits with efficient data structure design.*
    - *Control combinatorial explosion of potential subproblems.*

Methodology Overview

Approach: Modified SPLIT for accommodating multi-way splits

- **Main Changes:**
  - For GOSDT / Greedy Subroutines, use a subroutine called *getSplits* to find all potential splits to explore in the DP + BnB search.
    - Otherwise, keep algorithms the same (i.e., identical bounds, generalize to take bounds from all *k* children as opposed to binary left/right).
  - Continue to apply regularization penalties as before (i.e., per leaf). However, test trees at limited depth.

Formulation (modified *GOSDT*)

How it Works

- Replace split generation (lines 13-16) with *getSplit* to generate the set of possible multiway splits for GOSDT.
- Note that *get_bounds* will remain the exact same (only now using the multiway greedy approach instead of the 2-way)

Algorithm 1 GOSDT(R, x, y, λ)

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

Generalize these update equations, replace with a *getSplit function*

Subroutine 1 – getSplits (CART)

How it Works

- Run a full, unregularized, univariate CART and generate bins from leaves.
- Then, use DP to find the best (by the weighted Gini) multi-way split for each "way" up to a preset limit *k*.

Algorithm 1 ExtractThresholdsCART

Input: Feature column $\mathbf{x} \in R^n$, labels $\mathbf{y} \in \{0,1\}^n$, min leaf size m
Output: Sorted thresholds $\mathbf{t} = (t_1, \ldots, t_{L-1})$, bin counts $\{(n_\ell, \mathbf{c}_\ell)\}_{\ell=1}^{L}$

Fit univariate CART on $(\mathbf{x}, \mathbf{y})$ with `min_samples_leaf` $= m$
$\mathbf{t} \leftarrow$ sorted unique internal thresholds from CART
$L \leftarrow |\mathbf{t}| + 1$ ▷ number of bins
for $\ell = 1, \ldots, L$ **do**
  $B_\ell \leftarrow \{i : t_{\ell-1} < x_i \le t_\ell\}$ ▷ $t_0 = -\infty,\ t_L = +\infty$
  $n_\ell \leftarrow |B_\ell|$
  $\mathbf{c}_\ell \leftarrow$ (count of class 0 in B_ℓ, count of class 1 in B_ℓ)
end for
return $\mathbf{t}$, $\{(n_\ell, \mathbf{c}_\ell)\}_{\ell=1}^{L}$

Subroutine 1 – getSplits (CART)

How it Works

- Run a full, unregularized, univariate CART and generate bins from leaves.
- Then, use DP to find the best (by the weighted Gini) multi-way split for each “way” up to a preset limit *k*.

Algorithm 2 DP bin collapse: optimal K-way partition of ordered bins

Input: Bin counts $\{(n_\ell, \mathbf{c}_\ell)\}_{\ell=1}^{L}$, max -way $K_{\max}$, total samples N
Output: For each $K = 2, \dots, K_{\max}$: optimal K-way thresholds and cost

Compute prefix sums: $S_n[j] = \sum_{\ell=1}^{j} n_\ell, \quad S_c[j] = \sum_{\ell=1}^{j} \mathbf{c}_\ell$

function $\textsc{Cost}(i, j)$ ▷ Weighted Gini of merged bin $[i, j]$
  $n \leftarrow S_n[j] - S_n[i-1]$
  return $(n - \|\mathbf{S_c}[j] - \mathbf{S_c}[i-1]\|^2/n) \ / \ N$
end function

▷ Base case: one group spanning bins $1 \dots j$
for $j = 1, \dots, L$ **do**
  $\mathrm{dp}[1][j] \leftarrow \textsc{Cost}(1, j)$
end for

▷ Fill DP table
for $k = 2, \dots, \min(K_{\max}, L)$ **do**
  for $j = k, \dots, L$ **do**
    $\mathrm{dp}[k][j] \leftarrow \min_{i \in \{k, \dots, j\}} \mathrm{dp}[k-1][i-1] + \textsc{Cost}(i, j)$
    $\mathrm{ptr}[k][j] \leftarrow \arg\min_i$ (same)
  end for
end for

▷ Backtrack to recover thresholds for each K
for $K = 2, \dots, K_{\max}$ **do**
  Trace $\mathrm{ptr}[K][L] \rightarrow \mathrm{ptr}[K-1][\cdot] \rightarrow \cdots$ to get boundary indices $b_1, \dots, b_{K-1}$
  Thresholds $\leftarrow (t_{b_1}, \dots, t_{b_{K-1}})$
end for

Subroutine 1.1 – getSplits (sampled bins)

How it Works

- From testing, it seemed interesting to instead optimize across all midpoints (in the same way as the GOSDT binarizer works). For tractability, they were subsampled into statistical bins (usually 500 of them).
- Hopefully, this will allow for better candidate multi-way splits to be recovered (closer to SPLIT- or GOSDT-style boundaries).

Algorithm 3 ExtractThresholdsMidpoint

Input: Feature column $\mathbf{x} \in R^n$, labels $\mathbf{y} \in \{0,1\}^n$, max bins B
Output: Sorted thresholds $\mathbf{t} = (t_1, \dots, t_{L-1})$, bin counts $\{(n_\ell, \mathbf{c}_\ell)\}_{\ell=1}^{L}$

```
u ← sorted unique values of x
if |u| > B + 1 then                              ▷ subsample to cap complexity
    u ← B + 1 equally-spaced quantiles of u
end if
t_i ← (u_i + u_{i+1}) / 2 for i = 1, ..., |u| − 1  ▷ midpoints between consecutive values
L ← |t| + 1
for ℓ = 1, ..., L do
    B_ℓ ← {i : t_{ℓ−1} < x_i ≤ t_ℓ}               ▷ t_0 = −∞, t_L = +∞
    n_ℓ ← |B_ℓ|
    c_ℓ ← (count of class 0 in B_ℓ, count of class 1 in B_ℓ)
end for
return t, {(n_ℓ, c_ℓ)}_{ℓ=1}^{L}
```

Subroutine 1.2 – getSplits (greedy beam)

How it Works

- On earlier tests, some datasets seemed to perform worse in many instances than the binary-split trees
  - As such, we also implemented a beam search-style approach, where we can take more potential thresholds for DP optimization.
  - For tractability, we only consider variations in the last index.
- It seems useful to opt for an adaptive widening, where we increase possibilities for smaller splits (i.e., try many more 2-way splits while only a few 5-way splits).

Algorithm 4 DP Backtrack for a "beam"-like approach

Input: DP table dp$[k][j]$ and pointers ptr$[k][j]$ from Alg. 2, bins $\{(n_\ell, \mathbf{c}_\ell)\}_{\ell=1}^{L}$, beam schedule $(B_2, B_3, \dots, B_{K_{\max}})$
Output: For each K: up to B_K distinct K-way partitions

for $K = 2, \dots, K_{\max}$ **do**
  ▷ Score every possible placement of the last boundary
  for $i = K, \dots, L$ **do**
    score$[i] \leftarrow$ dp$[K-1][i-1] + \textsc{Cost}(i, L)$
  end for
  $\mathcal{I} \leftarrow$ indices of the B_K smallest values in score ▷ top-B_K last boundaries

  for each $i^* \in \mathcal{I}$ **do**
    $\mathbf{b} \leftarrow (i^*)$ ▷ start with last boundary
    $j \leftarrow i^* - 1$
    for $k = K-1, \dots, 2$ **do** ▷ backtrack earlier boundaries optimally
      Prepend ptr$[k][j]$ to $\mathbf{b}$
      $j \leftarrow$ ptr$[k][j] - 1$
    end for
    Emit partition with boundaries $\mathbf{b}$ and cost score$[i^*]$
  end for
end for

LicketyMultiSPLIT

How it Works

- With the initial prototype, setting the lookahead depth to 3 and beyond led to generally intractable trees (not converging within a time limit).
- This made the polynomial-time LicketySPLIT algorithm appealing to experiment with for multiway splits.
- From tests, the single-level update of LicketySPLIT seemed to degrade performance with effective multiway splits. We implemented <u>multistep lookahead</u> (i.e., optimize every 2) which showed improved performance.

Algorithm 3 LicketySPLIT(ℓ, D, λ, d)

Require: ℓ, D, λ, d {loss function, samples, regularizer, full depth}

$t_{lookahead} = \text{SPLIT}(\ell, D, \lambda, 1, d, 0)$ {Call SPLIT with lookahead depth 1 and no post-processing}
if $t_{\text{lookahead}}$ is not a leaf **then**
  for child $u \in t_{\text{lookahead}}$ **do**
    $D(u)$ = subproblem associated with u
    $\lambda_u = \lambda \frac{|D|}{|D(u)|}$ {Renormalize λ for the subproblem in question}
    $t_u = \text{LicketySPLIT}(\ell, D(u), \lambda_u, d-1)$
    Replace u with subtree t_u
  end for
end if
return $t_{\text{lookahead}}$

Experimented with an extension to go every two levels (generate a two-level-optimized tree and then go to the grandchildren for the next round of optimization)

Results

We tested the algorithms’ performance on four datasets (with binary classification tasks):
1) HELOC, 2) COMPAS, 3) Electricity, and 4) Eye-State. 1/2 seem to be commonly used by previous DP-BnB papers while 3/4 were benchmarked by ShapeCART.

Results

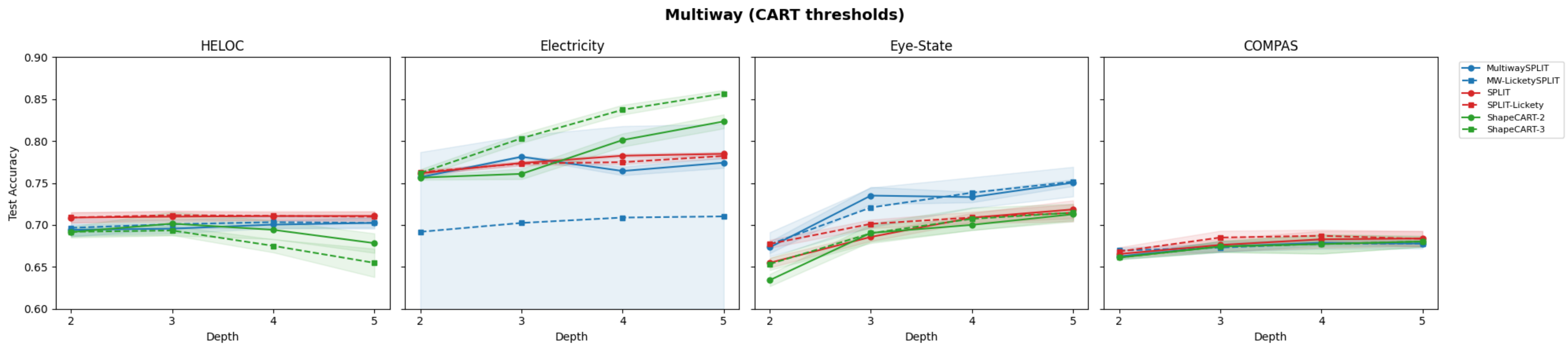
We tested the algorithms' performance on four datasets (with binary classification tasks): 1) HELOC, 2) COMPAS, 3) Electricity, and 4) Eye-State. 1/2 seem to be commonly used by previous DP-BnB papers while 3/4 were benchmarked by ShapeCART.

Evaluation Procedure

- Test accuracy was evaluated by 5-fold stratified CV (sklearn's *StratifiedShuffleSplit*) to ensure general class proportion consistency.
- Set lookaheads to *(d+1) / 2* – in other words, for a tree of depth 5, had a lookahead of 3 (optimize the first two sets globally and then generate greedily from there). Set regularization to *lambda = 0.01*
- Ran both *ShapeCART* and *Shape-3-CART* with mostly default settings (20 as the minimum support size of a leaf)
- Record the following statistics:
  - Current: Test (and train) accuracy, # Leaves
  - In Progress: Decision Sparsity, Runtime (with further optimizations)

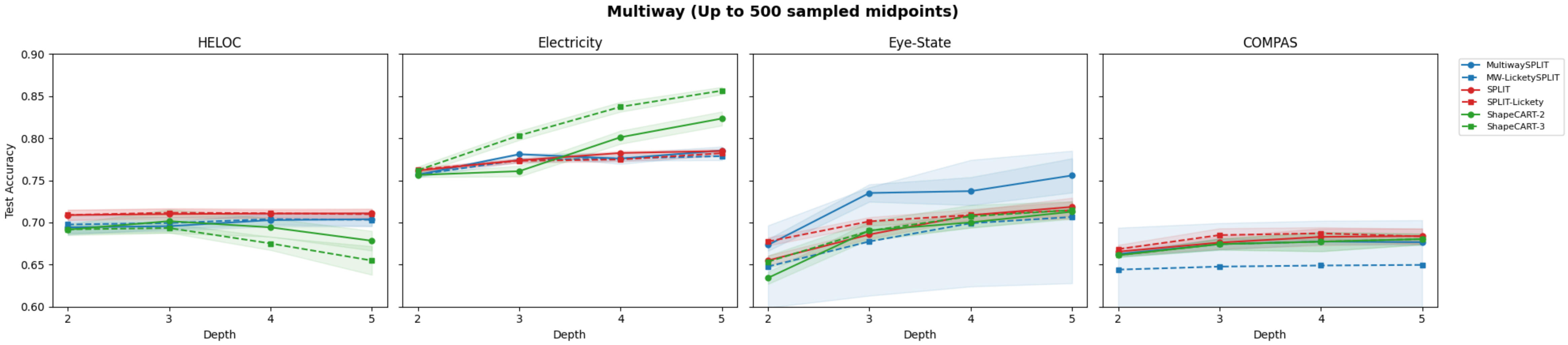
Results (CART-split)

We tested the algorithms' performance on four datasets (with binary classification tasks): 1) HELOC, 2) COMPAS, 3) Electricity, and 4) Eye-State. 1/2 seem to be commonly used by previous DP-BnB papers while 3/4 were benchmarked by ShapeCART. Generally, the performance appears relatively in line with other models, not doing well on the *Electricity* dataset while doing incredibly well on *Eye-State*.



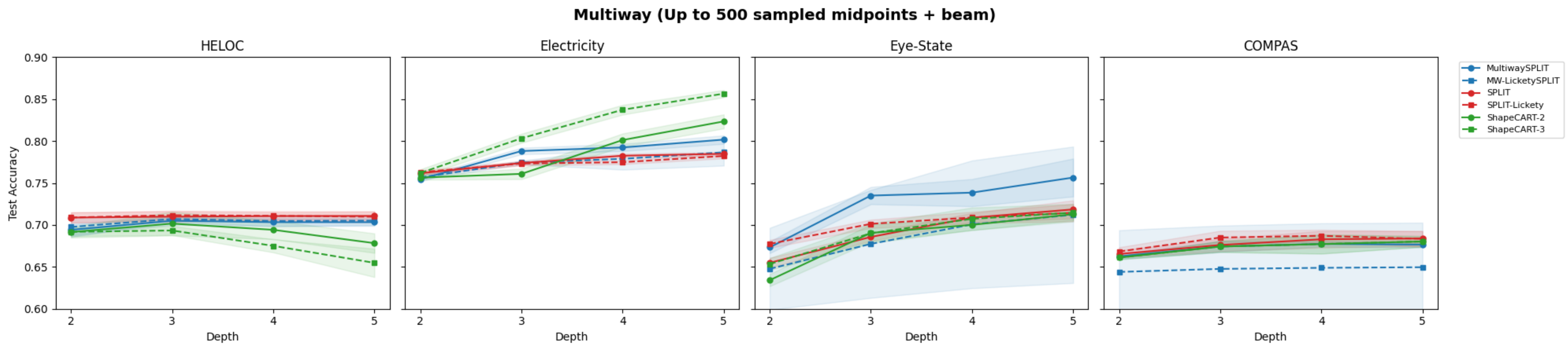
Results (sample-split)

We tested the algorithms' performance on four datasets (with binary classification tasks): 1) HELOC, 2) COMPAS, 3) Electricity, and 4) Eye-State. 1/2 seem to be commonly used by previous DP-BnB papers while 3/4 were benchmarked by ShapeCART. Generally, the performance appears relatively in line with other models, not doing well on the *Electricity* dataset while doing incredibly well on *Eye-State*.



Results (beam-split)

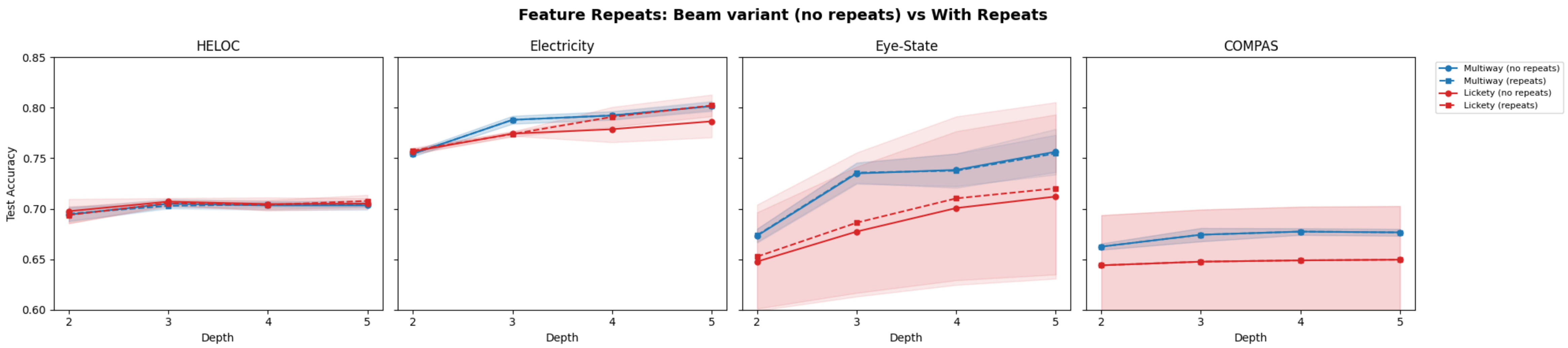
We tested the algorithms' performance on four datasets (with binary classification tasks): 1) HELOC, 2) COMPAS, 3) Electricity, and 4) Eye-State. 1/2 seem to be commonly used by previous DP-BnB papers while 3/4 were benchmarked by ShapeCART. Generally, the performance appears relatively in line with other models, not doing well on the *Electricity* dataset while doing incredibly well on *Eye-State*.



Note: selected top 5 k=2 splits, top 4 k=3, top 3 k=4 splits, and top 2 k=5 splits

Results (beam-split with repeats)

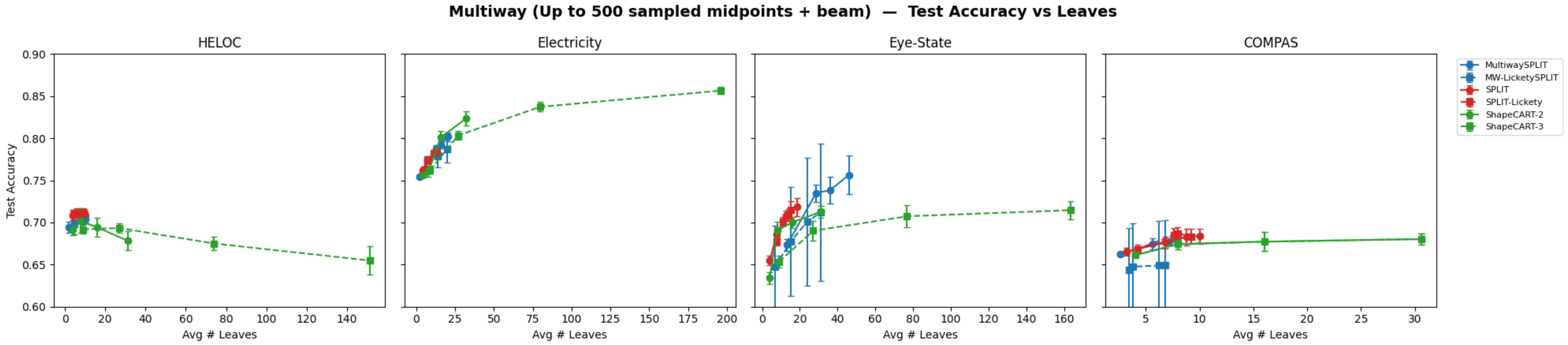
We tested the algorithms' performance on four datasets (with binary classification tasks): 1) HELOC, 2) COMPAS, 3) Electricity, and 4) Eye-State. 1/2 seem to be commonly used by previous DP-BnB papers while 3/4 were benchmarked by ShapeCART. Generally, the performance appears relatively in line with other models, not doing well on the *Electricity* dataset while doing incredibly well on *Eye-State*.



Oddly, it seems that double-splitting on variables gave a performance boost to the multiway trees (perhaps if one variable is particularly explanatory)

Results (sparsity)

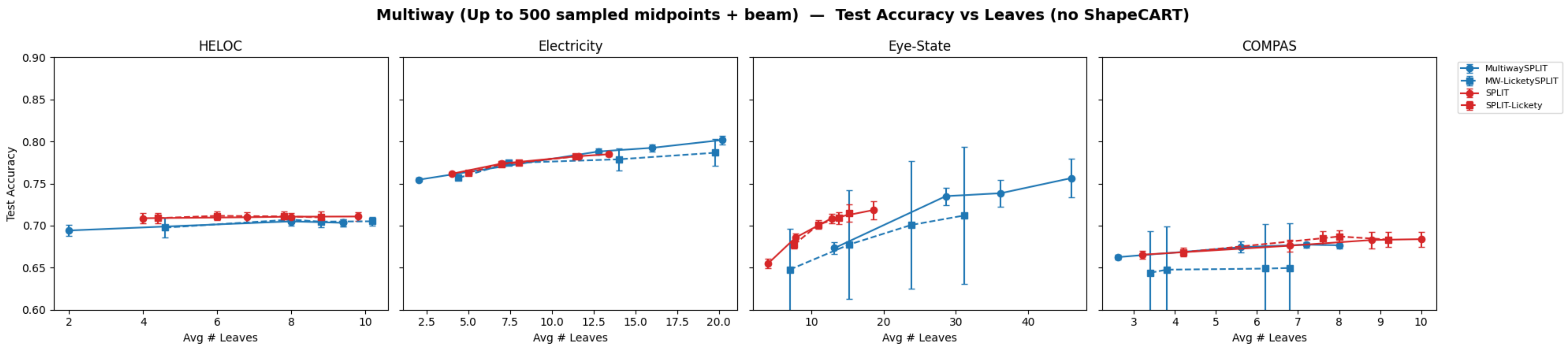
We tested the algorithms' performance on four datasets (with binary classification tasks): 1) HELOC, 2) COMPAS, 3) Electricity, and 4) Eye-State. 1/2 seem to be commonly used by previous DP-BnB papers while 3/4 were benchmarked by ShapeCART. Generally, the performance appears relatively in line with other models, not doing well on the *Electricity* dataset while doing incredibly well on *Eye-State*.



The trees generated by ShapeCART, without any pruning, form quite aggressively deep trees that would probably fall outside the realm of "easily interpretable."

Results (sparsity – no ShapeCART)

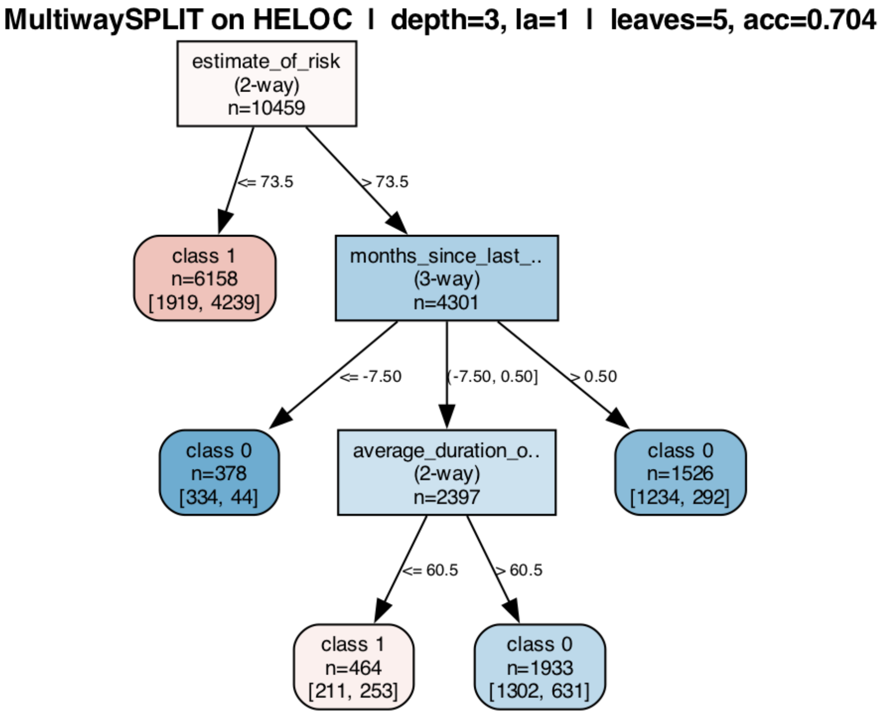
We tested the algorithms' performance on four datasets (with binary classification tasks): 1) HELOC, 2) COMPAS, 3) Electricity, and 4) Eye-State. 1/2 seem to be commonly used by previous DP-BnB papers while 3/4 were benchmarked by ShapeCART. Generally, the performance appears relatively in line with other models, not doing well on the *Electricity* dataset while doing incredibly well on *Eye-State*.



SPLIT is really good, especially at generating small trees that perform nearly as well as the multi-way's deeply extended trees.

Example Visualization

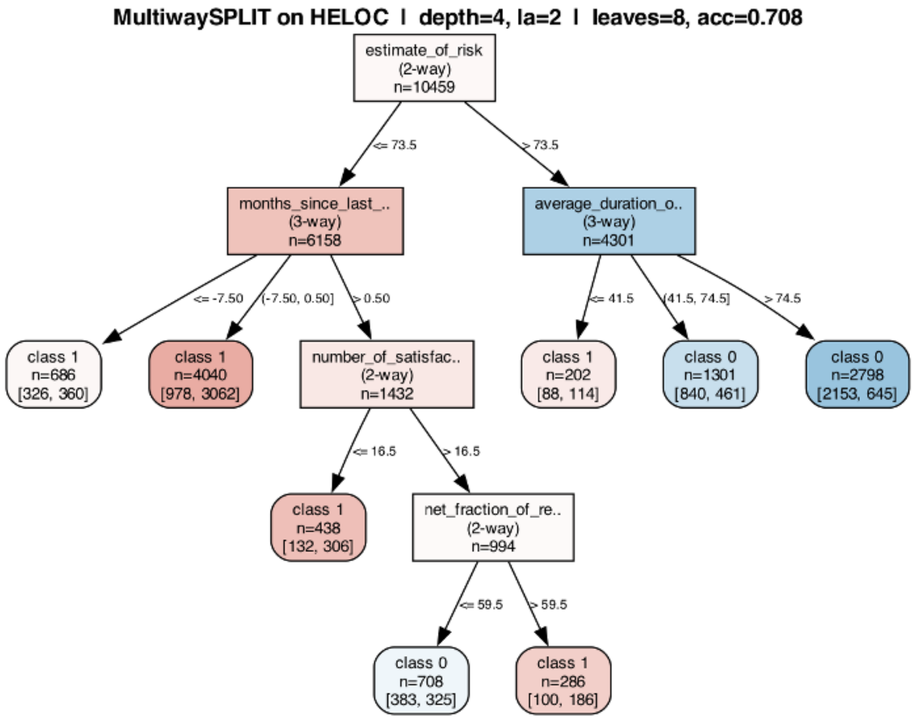
Here are a few examples of trees generated by Multi-SPLIT, which seem to provide some of the benefits of ShapeCART with different regimes (i.e., a 0, 1, 0 pattern) being captured



depth = 3, lookahead depth = 2 (i.e., optimizes level 1, starts generating greedy after)

Example Visualization

Here are a few examples of trees generated by Multi-SPLIT, which seem to provide some of the benefits of ShapeCART with different regimes (i.e., a 0, 1, 0 pattern) being captured



depth = 4, lookahead depth = 3 (i.e., optimizes levels 1-2, starts generating greedy after)
Weirdly, it appears that the much simpler binary splits weren't in the candidate set.

Noted Optimizations + Ideas

In SPLIT / GOSDT (WIP)

- (SPLIT) Leaf post-processing of trees with the fully optimal GOSDT.
- (SPLIT) threshold guessing with boosted trees (would it provide significantly better candidates than the current approach?)
- (GOSDT) Similar support bounds

Other Optimization Ideas

- Tighter initialization of upper/lower bounds (i.e., leveraging a boosting-based model to guess a lower bound and a greedy model to guess the upper bound, or similar).
- Selecting *m* best splits (by some heuristic) as opposed to doing DP + BnB on all valid splits.
- Allow multiway splits early but constrain to 2-way later.
- Post-processing to further compact tree size.

Conclusions / Where to Go

Interpretation of results

- ShapeCART still has the advantage of being able to have non-axis-aligned splits, could capture difficult-to-evaluate structures. However, it has no pruning / regularization and appears to overfit strongly (with trees having 100+ leaves).
- Seems to show some initial promise and reasonable performance, but is suffering vs. SPLIT due to the constrained set of splits it can explore.

Conclusions / Where to Go

Interpretation of results

- ShapeCART still has the advantage of being able to have non-axis-aligned splits, could capture difficult-to-evaluate structures. However, it has no pruning / regularization and appears to overfit strongly (with trees having 100+ leaves).
- Seems to show some initial promise and reasonable performance, but is suffering vs. SPLIT due to the constrained set of splits it can explore.

Next Steps

- Implementing more GOSDT-style optimizations: tractability up to 3 levels of multiway splitting could create cleaner trees (especially at higher depths).
- Experimenting with other methods of selecting thresholds
  - Using boosted threshold guessing (not sure if it would make a significant difference over the current two ends of threshold selection with one from CART and the other sampling all).
  - Sampling from some Rashomon set of optimal sparse trees and collapsing shared feature splits (possibly would give good guidance for existing sparse split combos).