Efficient Rashomon Set Approximation for Decision Trees

Zakk Heile
Department of Computer Science
Duke University
Durham, NC 27708
zakk.heile@duke.edu

Varun Babbar
Department of Computer Science
Duke University
Durham, NC 27708
varun.babbar@duke.edu

Hayden McTavish
Department of Computer Science
Duke University
Durham, NC 27708
hayden.mctavish@duke.edu

Cynthia Rudin
Department of Computer Science
Duke University
Durham, NC 27708
cynthia@cs.duke.edu

Abstract

Standard machine learning pipelines often admit many near-optimal models. These "Rashomon sets" pose a range of challenges and opportunities for uncertainty-aware, robust decision making. They allow incorporation of domain knowledge and user preferences that would otherwise be difficult to specify directly in an objective, and they quantify diversity among valid models and their predictions for a given training dataset and objective function. However, the applicability of Rashomon sets has been limited by computational intractability. Computation of Rashomon sets even for simple, interpretable model classes like sparse decision trees continues to require immense memory and runtime resources. We present LicketyRESPLIT, an algorithm to approximate this Rashomon set with orders of magnitude improvement in runtime and memory usage. We validate that LicketyRESPLIT regularly recovers almost all of the full Rashomon set. This work dramatically expands the ability of researchers and practitioners to model the Rashomon set for real-world datasets.

1 Introduction

Model selection is crucial to any machine learning pipeline. The Rashomon effect [5], coined by Leo Breiman, theorizes that there exist many equally good predictive models for a given dataset and objective function. For model selection in the Rashomon paradigm, we to first enumerate the set of all plausible models that fit the data well (i.e., the Rashomon set) and then select the model that aligns best with user needs. Users may want to find the model that maximizes fairness, obeys causal hypotheses, makes use of certain features, and follows certain structural constraints. All these modeling goals becomes simple when the Rashomon set has been enumerated, because only a simple loop through it is required to optimize any secondary objective. The exact method for uncovering this Rashomon set depends on the model class being considered. For instance, Rashomon sets of generalized additive models (GAMs) can be efficiently approximated by sampling around a convex hyperboloid centered at the optimal weight vector [19]. Uncovering the Rashomon set for discrete, non-parametric model classes such as decision trees, however, can be a massive computational undertaking both in runtime and memory owing to the NP-hard nature of the underlying problem. As an example, [10] show that the size of the search space of decision trees of depth 4 with 20 features is $\approx 8.4 \times 10^{18}$ trees.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MLxOR: Mathematical Foundations and Operational Integration of Machine Learning for Uncertainty-Aware Decision-Making.

Although the recent TreeFARMS algorithm [18] can exactly uncover this set, its runtime and memory requirements scale exponentially with the depth budget. Babbar et al. [2] proposed a set of approaches called Sparse Lookahead for Interpretable Trees (SPLIT), LicketySPLIT, and Rashomon Estimation with SPLIT (RESPLIT) which incorporate a lookahead depth in their search to find well-performing decision trees much faster than exact approaches. Among these, RESPLIT was designed to approximate Rashomon sets, while SPLIT and LicketySPLIT return a single tree. We show in this paper, however, that RESPLIT uses more memory than needed and can be sped up substantially while simultaneously producing a more complete approximation of the Rashomon set.

Inspired by the polynomial-time nature of the LicketySPLIT algorithm [2] used to find a single tree, we generalize this approach and propose a new family of polynomial-time Rashomon set approximation algorithms called LicketyRESPLIT. Our algorithms recursively find the set of near-optimal splits conditioned on the subsequent behaviour of easy to compute oracles. LicketyRESPLIT achieves orders-of-magnitude improvements in both runtime and memory efficiency compared to TreeFARMS and RESPLIT, while still recovering nearly the full Rashomon set.

2 Related Work

Decision trees have been established for decades as interpretable, scalable classifiers, with widely cited implementations such as CART [5] and C4.5 [15]. These greedy approaches build trees from the root, adding splits one a time according to heuristics. While these greedy heuristics were useful in the 1980's and 90's, their accuracy is lower than that of modern decision tree methods.

Researchers have substantially improved the accuracy of individual sparse decision trees via global optimization of performance and sparsity, alongside a range of techniques for computational efficiency [3, 8, 11, 17, 8, 12, 1]. This optimization structure can be extended to find all trees within epsilon of the optimal objective - that is, the Rashomon set of trees [18]. This approach enables a range of powerful downstream applications, from adding robustness to variable importance [9] to allowing customization and control to domain experts and practitioners [16]. These advances are incredible, but struggle to scale to larger practical datasets, being combinatorially complex in memory and runtime, particularly with respect to the number of features in the dataset.

A range of approaches have improved the scalability of optimal decision tree algorithms: through handling continuous features [13, 6], or incorporating carefully founded heuristics [14, 4, 7]. However, relatively few have extended these advances to computational benefits for computing Rashomon sets, which is can be complex given the distinct nature of the optimization task.

Of particular relevance are the LicketySPLIT and RESPLIT algorithms from [2]. The authors extend a two-step tree optimization algorithm to Rashomon set computation - this approach, RESPLIT, improves runtime by an order of magnitude but remains quite computationally complex. The paper also includes a polynomial-time recursive formulation to learn sparse trees that are consistently superior to standard greedy approaches and on par with globally optimal algorithms on practical datasets. This LicketySPLIT approach requires a number of nontrivial adjustments to extend to Rashomon sets, but yields substantial improvements in memory and runtime requirements.

3 Methodology

Let $\mathcal{F}$ be a model class of decision trees limited to some depth d. Let $D = \{(x_i, y_i)\}_{i=1}^n$ be a dataset of size n, where $y_i \in \{0, 1\}$ and each $x_i \in \{0, 1\}^k$ has k binary features (discretized by any binarization method). Let $\text{Obj}(f, D)$ measure the quality of a tree $f \in \mathcal{F}$. Let $S(f)$ denote the number of leaves in f. We use an objective which penalizes the number of misclassifications as well as a constant penalty λ multiplied by the number of leaves in the tree, as done in [12, 14, 2]:

$$\text{Obj}(f, D) = \lambda S(f) + \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{f(x_i) \neq y_i\}.$$

The Rashomon set can be defined as $\mathcal{R}_{\varepsilon_{\text{abs}}}(\mathcal{F}, D) := \{f \in \mathcal{F} | \text{Obj}(f, D) \leq \varepsilon_{\text{abs}}\}$, with cardinality $|R_{\varepsilon_{\text{abs}}}(\mathcal{F}, D)|$. For convenience, we sometimes omit the additional notation and use R and $|R|$. For our empirical experiments, we define a multiplicative epsilon, such that $\varepsilon_{\text{abs}} = (1 + \varepsilon_{\text{mult}}) \min_{f \in \mathcal{F}} \text{Obj}(f, D)$.

Algorithm 1 LICKETYRESPLIT(D, d, λ, B)

Require: Dataset D, remaining depth d, regularization λ, budget B

$\mathcal{A} \leftarrow \emptyset$
for $b \in \{0, 1\}$ **do**
  $C_b \leftarrow$ LEAFCOST(D, λ, b)
  if $C_b \leq B$ **then**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{$LEAF$(b, C_b)\}$ ▷ Leaf in budget: add as valid tree for this subproblem
  end if
end for
if $d = 0$ **or** $B < 2\lambda$ **then**
  return $\mathcal{A}$ ▷ No budget or depth for splits
end if
for each feature j **do**
  $(D_L, D_R) \leftarrow$ PARTITION(D, j)
  $L_L \leftarrow$ LICKETYSPLIT$(D_L, \lambda, d-1)$ ▷ Algorithm 3 in [2] to find a single tree
  $L_R \leftarrow$ LICKETYSPLIT$(D_R, \lambda, d-1)$
  if $L_L + L_R > B$ **then continue** ▷ LicketySPLIT pruning
  end if
  $\mathcal{L} \leftarrow$ LICKETYRESPLIT$(D_L, d-1, \lambda, B - L_R)$ ▷ Recurse on left based on a guess for the right subproblem
  $\mathcal{R} \leftarrow$ LICKETYRESPLIT$(D_R, d-1, \lambda, B -$ MINOBJ$(\mathcal{L}))$ ▷ Recurse on right with exact budget based on left
  if MINOBJ$(\mathcal{R}) < L_R$ **then** ▷ Recurse on left again with larger budget
    $\mathcal{L} \leftarrow$ LICKETYRESPLIT$(D_L, d-1, \lambda, B -$ MINOBJ$(\mathcal{R}))$
  end if
  $\mathcal{A} \leftarrow \mathcal{A} \cup \left\{$NODE$(f, \ell, r,$ OBJ$(\ell)+$OBJ$(r)) \ : \ \ell \in \mathcal{L},\ r \in \mathcal{R},$ OBJ$(\ell)+$OBJ$(r) \leq B\right\}$ ▷ Cartesian product of left/right solutions, filtering to stay within budget
end for
return $\mathcal{A}$ ▷ The final Rashomon Set

Initialize: Call once with $B \leftarrow (1 + \varepsilon) \cdot$ LICKETYSPLIT(D, λ, d).

LicketyRESPLIT is defined in Algorithm 1, though in practice we represent the set of trees using a compact trie-based structure (as in [18]) to share common subtrees and reduce memory usage. At each node, we consider all features and prune any split whose LicketySPLIT-completed cost would exceed the budget; the budget starts at the Rashomon threshold obtained by treating the LicketySPLIT tree as the reference solution. Later, if trees that improve on that are found, the output can be filtered accordingly.

Whereas normally the memory and runtime requirements on Rashomon set construction can be arbitrarily larger than the size of the Rashomon set, LicketyRESPLIT takes memory and runtime linear in the size of the Rashomon set approximation it finds. We demonstrate this in the following theorems (which are proven in the appendix) and empirically in Section 4.

Theorem 3.1. *Given a dataset D of size n with k features, LicketyRESPLIT with $\mathcal{F}$ corresponding to all trees within depth budget d can find a Rashomon set $R_{\varepsilon_{abs}}(\mathcal{F}, D)$ in $\mathcal{O}(|R|nk^3d^3)$ time.*

Theorem 3.1 shows that LicketyRESPLIT's runtime is polynomial in the size of the recovered set, number of samples, binarized features, and maximum depth—a much stronger bound than for TreeFARMS or RESPLIT. The approach also enables scalable memory:

Theorem 3.2. *Given a dataset D of size n with k features, and denoting $S(f)$ as the number of leaves in tree f, a memory-efficient implementation of LicketyRESPLIT can find a Rashomon set $R_{\varepsilon_{abs}}(\mathcal{F}, D)$ using $\mathcal{O}\Big(nk + \sum_{f \in R} S(f)\Big)$ memory, with the runtime complexity in Theorem 3.1.*

Theorem 3.2 tells us that LicketyRESPLIT can be upper bounded with memory complexity corresponding to the size of its input and the size of the set of trees it finds. In practice, our implementation uses additional memory for caching and reuse. While the asymptotic runtime is unchanged, caching accelerates practical runtime by exploiting repeated subproblems in LicketyRESPLIT's execution.

4 Results

We compare LicketyRESPLIT to TreeFARMS and RESPLIT across a range of benchmark datasets, evaluating runtime, peak memory usage, and Rashomon set quality in terms of precision and recall.

Table 1: Comparison of runtime (s) and maximum memory consumption (MB) of LicketyRESPLIT, TreeFARMS, and RESPLIT. LicketyRESPLIT is an order of magnitude faster than TreeFARMS on larger datasets and uses $100\times$ less memory. All results use $\lambda = 0.01$, $\epsilon_{\text{mult}} = 0.01$, max depth $= 5$.

Dataset	LicketyRESPLIT (ours) Time / RAM	TreeFARMS Time / RAM	RESPLIT Time / RAM
Bike	**18.8 / 438**	685 / 51714	184 / 528
Bank	**123 / 776**	OOM	238 / 2079
Compas	8.9 / **390**	50 / 3054	**6.0** / 402
Covertype	**507 / 1793**	1819 / 67635	1295 / 2974
Heloc	58.9 / 571	OOM	**34.3 / 545.2**
Student	**1.7 / 370**	351 / 4673	4.0 / 382
Wine	**14.5 / 425**	2493 / 113244	19.0 / 571

Table 1 shows that LicketyRESPLIT consistency outperforms both TreeFARMS and RESPLIT in terms of runtime and memory efficiency. For instance, on the Bike dataset, LicketyRESPLIT completes in <20 seconds using only 438MB of memory, compared to ~700 seconds and over 50GB for TreeFARMS.

Table 2: Precision and recall of LicketyRESPLIT and RESPLIT versus TreeFARMS. Metrics are reported as mean $\pm$ standard error across 10 bootstraps. All results use $\lambda = 0.01$, $\epsilon_{\text{mult}} = 0.01$, and maximum depth $= 5$. The slack column allows for trees to be an additive 0.01 outside the true Rashomon set and be counted in it: $\varepsilon_{\text{abs, slack}} = 0.01 + (1 + \varepsilon_{\text{mult}}) \min_{f \in \mathcal{F}} \text{Obj}(f, D)$.

Dataset	LicketyRESPLIT (ours)			RESPLIT		
	Precision	$\text{Prec}_{\text{Slack}}$	Recall	Precision	$\text{Prec}_{\text{Slack}}$	Recall
Adult	1 ± 0	1 ± 0	1 ± 0	0 ± 0	1 ± 0	0 ± 0
Bike	1 ± 0	1 ± 0	1 ± 0	0.201 ± 0.133	0.230 ± 0.129	0.368 ± 0.066
Compas	0.910 ± 0.030	1 ± 0	0.920 ± 0.035	0.911 ± 0.049	1 ± 0	0.374 ± 0.137
Covertype	1 ± 0	1 ± 0	1 ± 0	1 ± 0	1 ± 0	1 ± 0
Spambase	1 ± 0	1 ± 0	0.964 ± 0.023	0 ± 0	0.914 ± 0.046	0 ± 0
Wine	1 ± 0	1 ± 0	0.9 ± 0.071	1 ± 0	1 ± 0	0.9 ± 0.071

Table 2 shows that LicketyRESPLIT was able to enumerate almost all of the Rashomon set on all of our benchmark datasets. While LicketyRESPLIT has no guarantee of optimality, it reliably achieves nearly perfect precision and recall whenever TreeFARMS is tractable. In contrast, RESPLIT frequently fails to recover any of the true Rashomon set (see Spambase, Adult), though it does return a large number of slightly lower-quality trees that fit within the slack given.

5 Conclusion

We introduce LicketyRESPLIT, a novel polynomial time, memory efficient Rashomon set estimation algorithm. LicketyRESPLIT performs one step lookahead at every node in the search process, finding splits which are within the Rashomon bound given the performance of an oracle subroutine. We show that using the high performing $\mathcal{O}(nk^2d^2)$ LicketySPLIT algorithm oracle from [2] enables efficient search whilst retaining near-optimality. We empirically demonstrate that LicketyRESPLIT is an order of magnitude faster than TreeFARMS and often 1-2$\times$ as fast as RESPLIT on a wide variety of datasets. It also uses $100\times$ less memory than TreeFARMS and 1-2$\times$ less memory than RESPLIT. LicketyRESPLIT also recovers almost the entire Rashomon set, demonstrating near perfect precision and recall across all datasets tested. We expect future work to consider generalizations of LicketyRESPLIT, including performance under different oracle subroutines and a broader theoretical and empirical characterization of runtime and memory improvements for larger scale tabular datasets.

Acknowledgments

We acknowledge funding from the the U.S. Department of Energy under DE-SC0023194, and the National Institute On Drug Abuse of the National Institutes of Health under R01DA054994. Content does not necessarily represent official views of the United States Government nor any agency thereof.

References

[1] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3146–3153, 2020.

[2] Varun Babbar, Hayden McTavish, Cynthia Rudin, and Margo Seltzer. Near-optimal decision trees in a SPLIT second. In *Forty-second International Conference on Machine Learning*, 2025.

[3] Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106:1039–1082, 2017.

[4] Guy Blanc, Jane Lange, Chirag Pabbaraju, Colin Sullivan, Li-Yang Tan, and Mo Tiwari. Harnessing the power of choices in decision tree learning. *Advances in Neural Information Processing Systems*, 36, 2024.

[5] Leo Breiman. *Classification and regression trees*. Routledge, 1984.

[6] Cătălin E Brița, Jacobus GM van der Linden, and Emir Demirović. Optimal classification trees for continuous feature data using dynamic programming with branch-and-bound. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 11131–11139, 2025.

[7] Emir Demirović, Emmanuel Hebrard, and Louis Jean. Blossom: an anytime algorithm for computing optimal decision trees. In *International Conference on Machine Learning*, pages 7533–7562. PMLR, 2023.

[8] Emir Demirović, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J Stuckey. Murtree: Optimal decision trees via dynamic programming and search. *Journal of Machine Learning Research*, 23(26):1–47, 2022.

[9] Jon Donnelly, Srikar Katta, Cynthia Rudin, and Edward P. Browne. The rashomon importance distribution: Getting rid of unstable, single model-based variable importance, 2024.

[10] Xiyang Hu, Cynthia Rudin, and Margo Seltzer. Optimal sparse decision trees. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 7265–7273. Curran Associates, Inc., 2019.

[11] Xiyang Hu, Cynthia Rudin, and Margo Seltzer. Optimal sparse decision trees. *Advances in neural information processing systems*, 32, 2019.

[12] Jimmy Lin, Chudi Zhong, Diane Hu, Cynthia Rudin, and Margo Seltzer. Generalized and scalable optimal sparse decision trees. In *International Conference on Machine Learning*, pages 6150–6160. PMLR, 2020.

[13] Rahul Mazumder, Xiang Meng, and Haoyue Wang. Quant-BnB: A scalable branch-and-bound method for optimal decision trees with continuous features. In *International Conference on Machine Learning*, volume 162, pages 15255–15277. PMLR, 17–23 Jul 2022.

[14] Hayden McTavish, Chudi Zhong, Reto Achermann, Ilias Karimalis, Jacques Chen, Cynthia Rudin, and Margo Seltzer. Fast sparse decision tree optimization via reference ensembles. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 9604–9613, 2022.

[15] J Ross Quinlan. *C4.5: Programs for Machine Learning*. Elsevier, 2014.

[16] Cynthia Rudin, Chudi Zhong, Lesia Semenova, Margo Seltzer, Ronald Parr, Jiachang Liu, Srikar Katta, Jon Donnelly, Harry Chen, and Zachery Boner. Amazing things come from having many good models. *arXiv preprint arXiv:2407.04846*, 2024.

[17] Jacobus van der Linden, Mathijs de Weerdt, and Emir Demirović. Necessary and sufficient conditions for optimal decision trees using dynamic programming. *Advances in Neural Information Processing Systems*, 36:9173–9212, 2023.

[18] Rui Xin, Chudi Zhong, Zhi Chen, Takuya Takagi, Margo Seltzer, and Cynthia Rudin. Exploring the whole rashomon set of sparse decision trees. *Advances in neural information processing systems*, 35:14071–14084, 2022.

[19] Chudi Zhong, Zhi Chen, Jiachang Liu, Margo Seltzer, and Cynthia Rudin. Exploring and interacting with the set of good sparse generalized additive models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

A Proofs

Lemma A.1. *Given a dataset of size n and a depth budget d, the runtime cost of evaluating all initial splits given LicketySPLIT completions is $\mathcal{O}(nk^3d^2)$ where k is the number of candidate features considered for splitting.*

Proof of Lemma A.1. Theorem 6.4 in [2] shows that the cost of finding the optimal LicketySPLIT tree for a dataset of size n is $\mathcal{O}(nk^2d^2)$. We need to run LicketySPLIT for each of the left and right subproblems for each of the k candidate splits at the start, leading to a runtime of $\mathcal{O}(\sum_{i=1}^{k} n_i^{left}k^2d^2 + n_i^{right}k^2d^2)$ for finding the best split, where n_i^{left} and n_i^{right} are the sizes of the left and right subproblems respectively induced by the i^{th} split. Since $n_i^{left} + n_i^{right} = n$, the runtime expression simplifies to $\mathcal{O}(nk^3d^2)$. □

Theorem 3.1. *Given a dataset D of size n with k features, LicketyRESPLIT with $\mathcal{F}$ corresponding to all trees within depth budget d can find a Rashomon set $R_{\varepsilon_{abs}}(\mathcal{F}, D)$ in $\mathcal{O}(|R|nk^3d^3)$ time.*

Proof of Theorem 3.1. We first prove this result in a special case of Algorithm 1 where line 20 does not run; that is, we do not recurse with a larger budget.

Consider the search trie created by LicketyRESPLIT. In this case, each node corresponds to a subproblem defined by the sequence of splits leading to it, and there is no evaluation of the same subproblem with different budgets. By construction, LicketyRESPLIT only expands those splits whose LicketySPLIT completions remain in the Rashomon set (based on the budget allocation procedure in Algorithm 1).

Fix a level $d' < d$ of the trie. Let $u_{d'}$ denote the number of candidate splits selected by LicketyRESPLIT for expansion. For each split $1 \leq i \leq u_{d'}$ with subproblem size n_i, we must (i) generate its left and right subproblems, and (ii) evaluate LicketySPLIT completions over all k possible splits on each side (in order to select the next batch of candidate splits). From Lemma A.1, this requires $\mathcal{O}(n_ik^3(d-d')^2)$ time per split, so the total cost at level d' is:

$$\mathcal{O}\left(\sum_{i=1}^{u_{d'}} n_ik^3(d-d')^2\right) \tag{1}$$

Since only splits corresponding to trees guaranteed to be in the Rashomon set appear in the trie, the combined subproblem sizes at any level d' cannot exceed $n|R|$. We can see this as follows:

$$\sum_{i=1}^{u_{d'}} n_i \leq \sum_{\text{tree } t \in R} \sum_{\text{splits in } t \text{ at level } d'} n_{u_t} \tag{2}$$

$$= \sum_{\text{tree } t \in R} n \tag{3}$$

$$= n|R|. \tag{4}$$

Therefore, the total runtime is bounded by

$$\sum_{d'=1}^{d} \sum_{i=1}^{u_{d'}} n_i k^3 (d-d')^2 \leq \sum_{d'=1}^{d} |R| n k^3 (d-d')^2 \tag{5}$$

$$\leq |R| n k^3 d^3. \tag{6}$$

Now, consider the case where line 20 runs—that is, LicketyRESPLIT is run on the left subproblem with a larger budget (note that if the minimum objective is still the LicketySPLIT objective, the enumeration on the left subproblem is not rerun). In order for this larger budget to have been set, it implies that there exist at least two trees for the right subproblem (the LicketySPLIT tree and the new minimum-objective tree) that could be combined with the LicketySPLIT tree for the left to be within the budget for the subproblem.

Recall that LicketyRESPLIT has a key invariant: any solution to a subproblem not pruned is part of at least one tree in the Rashomon set (this follows from LicketyRESPLIT's construction - it only expands splits whose LicketySPLIT completions are within the budget). So in our new call of LicketyRESPLIT, we know that every subproblem visited is a part of a solution with the new minimum-objective tree on the right. And every subproblem visited in the original LicketyRESPLIT call on the left is part of a solution with the LicketySPLIT tree on the right, so we know we're not double counting solutions.

Thus, even though we may solve a subproblem multiple times with different budgets, the number of times a subproblem (a sequence of splits) is solved is never more than the number of trees it appears in.

This fact implies that the combined subproblem sizes at any level d' cannot exceed $n|R|$—the sum of all subproblem sizes across the Rashomon-set trees found for that depth—because any distinct subproblem is solved once, and any subproblem shared by t trees is solved no more than t times. Thus, even in this case, the size of subproblems solved is bounded by the same quantity, and as such, the asymptotic runtime is as well.

□

Lemma A.2. *Given a dataset of size n with k binary features, the memory cost of LicketySPLIT[2] algorithm can be limited to $\mathcal{O}(nk)$.*

Proof. We can show this by induction:

Inductive hypothesis: Let c be some fixed constant. Given that LicketySPLIT called with remaining depth $d-1$ and any dataset of dimension $n_1 \times k$, with $n_1 \leq n$ takes memory no greater than $c(n_1 k + 1)$, we want to show that LicketySPLIT with depth d and any dataset of dimension $n_2 \times k$, with $n_2 \leq n$, takes memory no greater than $c(n_2 k + 1)$.

Pick c such that the original dataset size is $\leq cnk$ (and all subsets of size n_i are similarly of size $\leq cn_i k$) and the storage required for a few constants is $\leq c$.

Base case: When the remaining depth budget is 0, LicketySPLIT requires no additional memory beyond its input dataset (size $\leq cn_2 k$) and a constant (size $\leq c$); all that is required is to compute the leaf objective, corresponding to the minimum of the number of positive vs negative entries in label y. So the memory cost is not greater than $c(nk+1)$.

Inductive step: When depth is > 0, LicketySPLIT must call a greedy subroutine for the left and right children from every possible binary feature split. For a given potential split, the required memory is no more than $cn_2 k + c$: we need only provide the left and right training data subsets to the greedy algorithms, and store the sum of the resulting objectives. By going through one split at a time, and just tracking the best objective so far and the corresponding feature, LicketySPLIT can do this without persisting more than constant memory. Once the optimal split is known, LicketySPLIT then constructs the left and right subproblems corresponding to that split (still with total size matching the original dataset size, which is $\leq cn_2 k$). LicketySPLIT then must run LicketySPLIT with one fewer depth for the left and right subproblems of the selected split. Note that each of the two subproblems has a number of samples no greater than $n_2 - 1$, because the optimal split must place at least one sample in each subproblem. So, by the inductive hypothesis, each split requires $\leq c((n_2-1)k+1) = c(n_2 k)$ memory to be solved individually. After one subproblem is solved, the memory used for it can be

freed except for a single constant, and we can solve the other subproblem with $\leq c(n_2 k)$ memory. In total, we use $\leq c(n_2 k + 1)$ memory, as required. (Note that once these two LicketySPLIT approaches are ready to be called, no other information needs to be persisted in memory - LicketySPLIT will just return the sum of these two calls. So the total amount of information in memory remains bounded by $cnk + c$).

Therefore, by induction the total memory use for any LicketyRESPLIT call is bounded by $cnk + c$, and thus in $O(nk)$. □

Theorem 3.2. *Given a dataset D of size n with k features, and denoting $S(f)$ as the number of leaves in tree f, a memory-efficient implementation of LicketyRESPLIT can find a Rashomon set $R_{\varepsilon_{abs}}(\mathcal{F}, D)$ using $\mathcal{O}\left(nk + \sum_{f \in R} S(f)\right)$ memory, with the runtime complexity in Theorem 3.1.*

Proof. First note from Lemma A.2 that the LicketySPLIT subroutine (from [2]) requires only $\mathcal{O}(nk)$ memory to run. Also note that its result can be given as just a single float corresponding to the objective.

Now, when we run LicketyRESPLIT, at each node we visit, we just need to know which splits result in LicketySPLIT objectives below the epsilon bound. So for each potential split, we need to run LicketySPLIT on the left and right subproblems (with $O(nk)$ memory total), then check if that falls within the absolute epsilon bound. If it does, then we will save the resulting subproblems from this split, and later visit those nodes to continue LicketyRESPLIT. This will be an additional two nodes we need to persist in the dependency graph. However, every time this happens (increasing the total storage used by 2), the total sum of nodes in trees across the entire Rashomon set will also increase by at least 2, since at least one tree with this split (the one found by LicketyRESPLIT) will fall within the absolute epsilon bound, and that tree will have at least one internal or leaf node corresponding to each of these two nodes, since it includes this split.

In order to keep the information stored in each node efficient, we use the following structure (assuming we have one global copy of the entire dataset provided as input). Consider a node to be active only if we are visiting that node currently (that is, we are mid-execution of Algorithm 1 at that node), or if we are visiting one of its child nodes. When a node is active, we persist information about the row indices corresponding to the data subset used in that node. We can still efficiently determine the row indices relevant for any child just using these row indices, the original dataset, and the binary splitting feature, so the runtime is not affected. We also maintain memory efficiency: we only have $O(d)$ nodes active at once, so the total memory required to persist these row indices is $O(nd)$; since we cannot have a depth greater than the number of binary features, that means the memory required is under $O(nk)$, and does not affect asymptotic complexity. Once we are done visiting a node, we don't need to visit it again and can safely stop persisting the memory needed for its row indices.[1]

So the total information we persist is just:

1. A single copy of the original dataset
2. the dependency graph structure, which can be constructed to only include nodes with a constant amount of storage space, where there are no more nodes in the dependency graph than there are nodes (split nodes or leaves) across the whole Rashomon set.
3. Row indices for currently active subproblems in the dependency graph.

Since the number of leaves and internal nodes is bounded by twice the number of leaves, we know object (2) is $O(\sum_{t \in R} S(t))$. We know object (1) is $O(nk)$, so combining them we have proven our claimed memory complexity. (Note that (3) is also within $O(nk)$ so does not affect the complexity).

□

[1]This is slightly complicated by the third pass of LicketyRESPLIT, i.e. line 20 of Algorithm 1; however, it does not affect the asymptotic complexity if we reconstruct the relevant row indices a second time when revisiting the left side with a second pass.

B Trie Representation

In practice, rather than explicitly forming the filtered Cartesian product of all feasible left/right subtrees at a given subproblem, we represent each subproblem solution as a trie structure. This lets us represent the entire Rashomon set implicitly, without materializing every tree in memory, while still supporting efficient operations like extracting a specific tree by index or sampling from the set when it is large.

Each subproblem stores a *Trie node* T with: (i) a budget $B(T)$, (ii) a multiset $\mathsf{Obj}(T)$ mapping objective values to counts of trees achieving them, (iii) a minimum objective $\mathsf{MinObj}(T)$, and (iv) children consisting of either $\textsc{Leaf}(b, \ell)$ (prediction $b \in \{0, 1\}$, loss ℓ) or $\textsc{Split}(f, L, R)$ (feature f, left/right subtries L, R). A split's count of feasible trees is the number of left/right pairs whose summed loss is $\leq B(T)$.

By storing the distribution of objective values in each Trie node, we can efficiently retrieve the i^{th} tree according to a canonical ordering. During extraction, we impose an ordering based on increasing objective value, using feature order and other attributes as tiebreakers.

We replace line 5 and line 20 in Algorithm 1 with a call to the AddLeaf and AddSplit subroutines, respectively.

Algorithm 2 $\textsc{AddLeaf}(T,\ b,\ \ell)$

Require: Trie node T, label $b \in \{0, 1\}$, loss ℓ
1: Append $\textsc{Leaf}(b, \ell)$ to $T.\text{children}$
2: $T.\mathsf{MinObj} \leftarrow \min(T.\mathsf{MinObj},\ \ell)$
3: $T.\mathsf{Obj}[\ell] \leftarrow T.\mathsf{Obj}[\ell] + 1$

Algorithm 3 $\textsc{AddSplit}(T,\ f,\ L,\ R)$

Require: Trie node T with budget $B(T)$; feature f; left/right subtries L, R
1: Append $\textsc{Split}(f, L, R)$ to $T.\text{children}$
2: $T.\mathsf{MinObj} \leftarrow \min\left(T.\mathsf{MinObj},\ \mathsf{MinObj}(L)+\mathsf{MinObj}(R)\right)$
3: $c \leftarrow 0$
4: **for each** $(\ell_L,\ n_L)$ in $L.\mathsf{Obj}$ **do**
5: **for each** $(\ell_R,\ n_R)$ in $R.\mathsf{Obj}$ **do**
6: $\ell \leftarrow \ell_L + \ell_R$
7: **if** $\ell \leq B(T)$ **then**
8: $T.\mathsf{Obj}[\ell] \leftarrow T.\mathsf{Obj}[\ell] + n_L \cdot n_R$
9: $c \leftarrow c + n_L \cdot n_R$
10: **end if**
11: **end for**
12: **end for**
13: The appended split stores `num_valid_trees` $\leftarrow c$

The TreeFARMS algorithm uses a similar storage mechanism with efficient algorithms to sample trees from the Rashomon set in the event that the Rashomon set is too large to store in memory [18]. We store very similar metadata to be able to facilitate this sampling, and as such, extraction algorithms follow directly from the paper.

C Caching

In our experiments, we make extensive use of caching to reuse oracle objectives from previously solved subproblems. While this caching mechanism can be disabled, Table 1 shows that LicketyRE-SPLIT requires less than 1.8 GB of memory even with caching enabled. Consequently, we retain caching in all reported experiments, as it improves runtime with negligible memory overhead.

The LicketySPLIT algorithm and the greedy subroutine described in [2] are both best implemented recursively. Given this, when we query the LicketySPLIT oracle, we can cache the numerical objective for each subproblem the recursive procedures explore.

In practice, this means that when a split calls the LicketySPLIT oracle to evaluate its objective, the result and intermediate results are stored for reuse. If LicketyRESPLIT does not prune that split, then those left and right LicketySPLIT calls will be guaranteed to be queried, and thus we can save computation. Additionally, this will save computation anytime the same subproblem is reached from a different sequence of splits.

Likewise, as any trie returned for a (subproblem, depth, budget) is guaranteed to be in the Rashomon set, we can store a pointer to it for that subproblem. If the exact triple is obtained via a path of LicketyRESPLIT recursive calls, we can reuse it. Additionally, we can reuse tries at the same subproblem solved with a greater depth and budget by truncating them.