Systems Optimizations for Learning Certifiably Optimal Rule Lists

Nicholas Larus-Stone
Computer Lab, University of Cambridge
nl363@cl.cam.ac.uk

Elaine Angelino
EECS, UC Berkeley
elaine@eecs.berkeley.edu

Daniel Alabi, Margo Seltzer
SEAS, Harvard University
{alabid@g, margo@eecs}.harvard.edu

Aditya Saligrama
Weston High School
20saligramaa@my.weston.org

Cynthia Rudin
Duke University
cynthia@cs.duke.edu

1 INTRODUCTION

Decision trees are an extremely influential prediction technique in AI. They are interpretable, simple to store, and easy to use [4]. However, not enough work has been done on speeding up the creation of decision trees for large datasets. As big data becomes even more common in machine learning applications, decision trees risk falling in disuse due to computationally prohibitive construction costs. This work focuses on a set of optimizations that aids with the construction of a specific subset of decision trees: one-sided decision trees, also known as decision lists or rule lists [12]. Our goal is to find the rule list that is provably optimal with respect to our objective function, the regularized empirical risk, for a given set of pre-mined rules. Our algorithm, Certifiably Optimal RulE ListS (CORELS), uses a combination of tight bounds and systems optimizations to achieve reasonable runtimes for moderately sized real datasets [1, 2]. Our implementation can be found at **https://github.com/nlarusstone/corels**.

Finding the optimal rule list from a set of rules is a computationally hard combinatorial problem with no polynomial-time approximation. Most decision tree construction strategies solve this by quickly finding an approximate solution through greedy methods [4, 11]. Other fields solve similar computationally hard combinatorial problems through the use of branch-and-bound algorithms[5]. Traditionally, branch-and-bound has only been applied to small problems, because the search space is otherwise enormous, even when tight bounds allow for the pruning of the space. Our previous work presents a branch-and-bound search strategy, full description of CORELS, and proofs of the bounds we use [1, 6]. Unlike prior greedy approaches, branch-and-bound allows us to both find and certify the optimal rule list.

A rule list is composed of an ordered list of rules, an example of which can be seen in Fig 1. Rules are features or conjunctions of features that uniformly classify a subset of the observations. We generate rule lists by adding a rule to the end of a pre-existing rule list. For a given rule list, any longer rule list that starts with the same rules will make the same mistakes as the shorter one. To encapsulate this relationship, we call the original rule list a parent and the longer rule list a child. We calculate both how many observations the list incorrectly classifies as well as a lower bound on the number of mistakes that can be made by any child rule lists. Calculating each of these metrics from scratch each time would be inefficient; instead, metrics are calculated incrementally using metadata recorded for the parent rule list. The incremental computation takes advantage of a specialized, space-efficient prefix tree by reusing previous computation from parent rule lists. We systematically search the entire space of rule lists, evaluating them in an order specified by our search policy; for example, a breadth-first search policy examines shorter rule lists before longer ones.

if $(age = 18 - 20)$ **and** $(sex = male)$ **then predict** *yes*
else if $(age = 21 - 22)$ **and** $(priors = 2 - 3)$ **then predict** *yes*
else if $(priors > 3)$ **then predict** *yes*
else predict *no*

Figure 1: Example optimal rule list that predicts two-year recidivism for the ProPublica COMPAS dataset (122 rules), found by CORELS, across 10 cross-validation folds.

Our objective function combines the number of incorrectly classified observations, normalized to measure misclassification error, with a regularization term that penalizes longer rule lists. The maximum number of mistakes that a rule list makes provides a bound for how well the children of that rule list could ever perform. Using those two metrics, we are able to compare the potential performance of all of a rule list's children to the current best rule list we have seen so far. If the best potential performance is worse than the performance of a rule list we have already seen, then we know it cannot be the optimal rule list, and thus we can prune it. We also proved a suite of additional algorithmic bounds that interact to aggressively prune the search space. In our implementation, we exploit these bounds via several data structures. Together with additional systems optimizations, our approach efficiently searches the space of rule lists to find a provably optimal solution.

In previous work, we performed experiments using the ProPublica COMPAS dataset [7], focusing on the prediction problem of whether or not an individual would recidivate within two years. This dataset contains 122 rules; therefore, a brute force search of all possible rule lists up to length 10 would require evaluating 5.0×10^{20} rule lists. Due to our algorithmic bounds, CORELS examines only 28 million rule lists, a reduction of 1.8×10^{13}. Between the bookkeeping required to search and prune the state space and the numerous bit vector operations required to evaluate each rule list, a non-optimized implementation of our algorithm would not complete. However, with CORELS, each evaluation takes only 1.3 μs, allowing us to complete that moderately sized problem in 36s. Systems optimizations, even on just a single processor, allow us to solve a computationally hard problem of reasonable size in under a minute.

2 IMPLEMENTATION

Our algorithm is feasible due to the use of three core data structures. First, we use a prefix tree to represent the search space. The incremental nature of our algorithm requires that we store rule lists that we examined but did not discard. The prefix tree structure allows us to efficiently access values in a parent rule list to calculate bounds for a new rule list. Second, we implement our scheduling policy with a priority queue. Third, we use a symmetry-aware map to implement one of our critical bounds. The map allows us to compare and prune permutations of rule lists.

To support incremental execution, we use a prefix tree to cache rule lists that we have already evaluated and have not pruned. Each node in the tree represents a rule, so any rule list can be reconstructed by following a path from the root to a leaf in the tree. The node stores metadata to aid in the computation of our bounds including: the lower bound, objective value, and a list of unpruned children. Nodes also have parent pointers, so each node is created incrementally by using the parent's bounds to calculate the new bounds. This structure also means that a rule list can be represented and passed around as a pointer to a single node in the tree or as an ordered list of rules. In most cases the pointer is more efficient, though the list representation is useful as a hashable index.

We represent our worklist with a priority queue. This allows us to try different exploration orders simply by changing the priority metric. We implement breadth-first search (BFS), depth-first search (DFS), and a best-first search that can take a custom metric. Our implementation supports ordering by the lower bound and the objective, though any function that maps a prefix to a real value in a stable manner would work. In general, we find that ordering by lower bound (best-first) yields the best results on most datasets.

To take advantage of the symmetry inherent to our problem, we designed a data structure we call a symmetry-aware map. We use this map to keep track of the best possible permutation of a given set of rules. When a new permutation of that set of rules is encountered, we add the new permutation to our prefix tree only if it is strictly better than the permutation we already have stored. We experimented with multiple key types that could represent permutations of rule lists and found that a series of simple optimizations of the key and value types led to a 3x reduction in memory usage for the map [8]. First, we decided that it was unrealistic for our algorithm to operate on datasets with more than 2^{16} rules, so we changed the type that represented rules from a size_t to a short. Next, instead of incurring overhead by using a STL set to represent keys, we created a custom key type that was a chunk of memory just large enough to hold all of the rules and the length of the rule list. Finally, the value type has to record the actual order of the rules in the rule list, but we were able to take advantage of the fact that the key type already contained the ids of the rules. We took advantage of this by using a byte array to remember the ordering of the rules without storing the actual rule ids again.

An important part of our implementation is that we calculate accuracy and other metrics using bit vector operations. Each rule is represented as a bit vector, whose length is equal to the number of items in the training set. Bit i is set to 1 if the rule evaluates true for data item i and 0 otherwise. We use the high performance bit vector rule library [13] that uses the GMP multiple-precision library to represent and manipulate the vectors. This is more efficient than built-in C++ constructs such as a vector of bools or a bitset. CORELS profiling shows that these bit vector operations account for the majority of algorithm's execution time. Thus, improving the performance of these computations will improve algorithm performance and is a promising direction for future work.

Each one of these structures and optimizations is critical for the success of our algorithm. For example, we briefly experimented with a stochastic search policy that did not use a queue at all but instead followed random paths from the queue to a leaf. However, this search strategy performed worse than any strategy involving our queue, so we proceeded only by using the queue.

Table 1 empirically validates the importance of data structure design (specifically the symmetry-aware map) on several datasets [3, 9, 10]. We conducted trials on a small personal laptop.[1] The method for both the tests using the map and the tests without the map used lower-bound ordering (best-first search) in the queue as described above.[2] The symmetry-aware map led to an average speedup of 8.83x in runtime and a reduction in memory usage of 3.15x. On more computationally complex datasets, such as the breast cancer dataset, the symmetry-aware map allows the problem to be solved to optimality using less than 4GB of RAM, versus about 8.6GB without this optimization. This enables such problems to be solved on almost all recent laptops.

Table 1: Average runtime (s) and memory usage (MB) for three executions, on different datasets, with and without the symmetry-aware map, tested on a laptop with 2.6 GHz i7-6700HQ and 16GB RAM, capped at 12GB. Standard deviations are within 1% of reported values in all cases.

Dataset	Time w/map (s)	Memory w/map (MB)	Time no map (s)	Memory no map (MB)	Speedup	Memory savings	Number of samples
B. cancer	3,206	3,078	14,039	8,598	4.4	2.8	684
Haberman	8.5	61	33	104	3.9	1.7	307
MONKS-1	0.29	<0.1	0.4	<0.1	1.2	~ 1.0	432
MONKS-3	0.10	<0.1	0.1	<0.1	1.4	~ 1.0	432
Votes	8.3	44	24	79	2.9	1.8	436
NYPD	80	26	15,073	1,343	188.4	51.6	566,839
COMPAS	0.86	2.7	1.45	3.0	1.7	1.1	6,217
Average	472	459	4,167	1,447	8.8	3.2	—

3 CONCLUSION

We demonstrated that it is possible to learn certifiably optimal rule lists, but only through a combination of algorithmic bounds, systems optimizations, and efficient data structures. While much of the theory in this area was developed in the 1980s and 1990s, to the best of our knowledge there have been few dedicated systems approaches that take advantage of modern hardware to work on real problems. Future work on CORELS involves parallelizing the algorithm both for a single machine with many cores and for distributed systems. Additionally, further scaling could be realized by porting the algorithm to an FPGA. We are optimistic that other classic AI algorithms may benefit from investigations of systems level optimizations.

[1] 2.6 GHz i7-6700HQ and 16GB of RAM.
[2] We set the regularization parameter λ to 0.01.

REFERENCES

[1] E. Angelino, N. Larus-Stone, D. Alabi, M. Seltzer, and C. Rudin. Learning certifiably optimal rule lists for categorical data. *Preprint at arXiv:1704.01701*, November 2017.

[2] E. Angelino, N. Larus-Stone, D. Alabi, M. Seltzer, and C. Rudin. Learning certifiably optimal rule lists for categorical data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'17)*, 2017.

[3] K. Bache and M. Lichman. UCI machine learning repository, 2013. http://archive.ics.uci.edu/ml.

[4] L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and regression trees. 1984.

[5] Jens Clausen. Branch and bound algorithms - principles and examples. 1999.

[6] S. Ertekin and C. Rudin. Learning customized and optimized lists of rules with mathematical programming. Unpublished, 2017.

[7] J. Larson, S. Mattu, L. Kirchner, and J. Angwin. How we analyzed the COMPAS recidivism algorithm. *ProPublica*, 2016.

[8] N. L. Larus-Stone. *Learning Certifiably Optimal Rule Lists: A Case For Discrete Optimization in the 21st Century*. 2017. Undergraduate thesis, Harvard College.

[9] O. L. Mangasarian and W. H. Wolberg. Cancer diagnosis via linear programming. 23(5):1–18, 1990.

[10] New York Civil Liberties Union. Stop-and-frisk data, 2014. http://www.nyclu.org/content/stop-and-frisk-data.

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

[12] R. L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, November 1987.

[13] H. Yang, C. Rudin, and M. Seltzer. Scalable Bayesian rule lists. In *Proceedings of the 34th International Conference on Machine Learning*, ICML '17, 2017.