Learning Certifiably Optimal Rule Lists: A Case For Discrete Optimization in the 21st Century

A THESIS PRESENTED
BY
NICHOLAS L. LARUS-STONE
TO
THE DEPARTMENT OF COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2017

Thesis advisor: Professor Margo Seltzer Nicholas L. Larus-Stone

Learning Certifiably Optimal Rule Lists: A Case For Discrete Optimization in the 21st Century

Abstract

We demonstrate a new algorithm, CORELS, for constructing rule lists. It finds the optimal rule list and produces proof of that optimality. Rule lists, which are lists composed of If-Then statements, are similar to decision tree classifiers and are useful because each step in the model's decision making process is understandable by humans. CORELS uses the discrete optimization technique of branch-and-bound to eliminate large parts of the search space and turn this into a computationally feasible problem. We use three types of bounds: bounds inherent to the rules themselves, bounds based on the current best solution, and bounds based on symmetries between rule lists. In addition, we use efficient data structures to minimize the memory usage and runtime of our algorithm on this exponentially difficult problem. Our algorithm proves that finding optimal solutions in a search space is a feasible solution using modern computers. Our algorithm therefore allows for the analysis and discovery of optimal solutions to problems requiring human-interpretable algorithms.

Contents

1

Introduction

As computing power continues to grow, combinatorial optimization problems that may have been out of the reach of earlier computational power can now be feasibly executed. The goal of this thesis is to discuss a number of data structure optimizations that allow for the completion of medium to large scale combinatorial optimization problems. This work builds off of the theoretical bounds and implementation found in Angelino et. al[2]. While the techniques found in this thesis are specific to a given machine learning technique, it is our hope that they can be generalized to other combinatorial optimization problems.

We work in the realm of machine learning, specifically focusing on the interpretability of predictive models. Our algorithm produces models that are highly predictive, but in which each step of the model's decision making process can also be understood by humans. Machine learning models, such as neural nets or support vector machines, can achieve stunning predictive accuracy, but the reasons for their predictions remain unintelligible to a human user. This lack of interpretability is important, because models that are not understood by humans may have hidden bias in their predictive decision making. A recent ProPublica article found racial bias in the use of a black box machine learning model used to create risk assessments intended to help judges with criminal sentencing[13]. Northepointe, the company providing COMPAS (the black box model), argues that their use of a black box model is necessitated by the fact that they can achieve better accuracy through the use of that model. This thesis is part of a body of work that attempts to disprove that statement by showing that it is possible to build interpretable machine learning models without sacrificing accuracy.

To achieve interpretability, we use *rule lists*, also known as decision lists, which are lists comprised of *if-then* statements[26]. This structure allows for predictive models that can be easily interpreted, because each prediction is explained by the rule that is satisfied. Given a set of rules associated with a dataset, every possible ordering of rules produces a unique rule list. Since most data points can be classified by multiple rules, changing the order of rules leads to different predictions and therefore different accuracies. Rule list generation algorithms attempt to maximize predictive accuracy through the discovery of different rule lists.

Generally, interpretable models are viewed as less accurate than black box models. Thus, proving the optimality of an interpretable model provides an important upper bound on the accuracy of that model. This helps decision makers decide whether or not to use interpretable models. In our case, we are searching for the rule list with the highest accuracy—-the optimal rule list. A brute force solution to find the the optimal rule list is computationally prohibitive due to the exponential number of rule lists. Our algorithm uses combinatorial optimization to find the optimal rule list in a reasonable amount of time.

Recent work on generating rule lists[17,29] uses probabilistic approaches to generating rule lists. These approaches achieve high accuracy quickly. However, despite the apparent accuracy of the rule lists generated by these algorithms, there is no way to determine if the generated rule list is optimal or how close to optimal it is. Our model, called Certifiably Optimal RulE ListS (CORELS), finds the optimal rule list and also allows us to investigate the accuracy of near optimal solutions. The benefits of this model are two-fold: first, we are able to generate the best rule list on a given data set and therefore will have the most accurate predictions that a rule list can give. Second, since CORELS generates the entire space of potential solutions, we can evaluate the quality of rule lists generated by other algorithms. In particular, we can determine if the rule lists from probabilistic approaches are nearly optimal or whether those approaches sacrifice too much accuracy for speed. This will allow us to bound the accuracy on important problems and determine if interpretable methods should be used.

CORELS achieves these results by optimizing an objective function and placing a set of bounds on the best objective that a rule list can achieve in the future. This allows us to prune that rule list if that bound is worse than the objective value of the best rule list that we have already examined. We continue to examine rule lists until we have either examined every rule list or eliminated all but one from consideration. Thus, when the algorithm terminates, we have found the rule list with the best possible accuracy. Our use of this branch and bound technique leads to massive pruning of the search space of potential rule lists and allows our algorithm to find the optimal rule list on real data sets.

Due to our interest in interpretability, the amount of data each rule captures informs the value of that rule. We want our rule lists to be understandable by humans, so shorter rule lists are more optimal. Therefore, we use an objective function that takes into account both accuracy and the length of the rule list to prevent overfitting. This means we may not always find the highest accuracy rule list—our optimality is over both accuracy and length of rule lists. This requires each rule to classify a minimum amount of data correctly to make it worth the penalty of making a rule list longer. This limits the overall length of our rule lists and avoids overfitting, as well as preventing us from investigating rule lists containing useless rules.

The exponential nature of the problem means that the efficacy of CORELS is partially dependent on how much our bounds allow us to prune. We list a few types of bounds that allow us to drastically prune our search space. The first type of bound is intrinsic to the rules themselves. This category includes bounds such the bound described above that ensures that rules capture enough data correctly to overcome a regularization parameter. Our second type of bound compares the best future performance of a given rule list to the best solution encountered so far. We can avoid examining parts of the search space whose maximum possible accuracy is less than the accuracy of our current best solution. Finally, our last class of bounds uses a symmetry-aware map to prune all but the best permutation of any given set of rules.

To keep track of all of these bounds for each rule list, we implemented a modified trie that we call a prefix tree. Each node in the prefix tree represents an individual rule; thus, each path in the tree represents a rule list where the final node in the path contains metrics about that rule list such as its accuracy and the number of data points already classified. This tree structure facilities the use of multiple different selection algorithms including breadth-first search, a priority queue based on a custom function that trades off exploration and exploitation, and a stochastic selection process. In addition, we are able to limit the number of nodes in the tree and thereby achieve a way of tuning space-time tradeoffs in a robust manner. We propose that this tree structure is a useful way of organizing the generation of rule lists and allows the implementation of CORELS to be easily parallelized.

We applied CORELS to the problem of predicting criminal recidivism on the COMPAS dataset. Larson et al examines the problem of predicting recidivism and shows that a black box model, specifically the COMPAS score from the company Northpointe, leads to racially biased predictions[13]. Black defendants are misclassified at a higher risk for recidivism than occurs in practice, while white defendants are misclassified at a lower risk. The model that produces the COMPAS scores is a black box algorithm, which is not interpretable, and therefore the model does not provide a way for human input to correct for these racial biases. Our model produces accuracies that are similar to standard predictive models and the black-box COMPAS scores while maintaining interpretability.

CORELS demonstrates a novel approach towards generating interpretable models by identifying and certifying the optimal rule list. While searching for that optimal list, we are able to discover near-optimal solutions that provide insight into how effective other interpretable methods might be. Rule lists have been around for 30 years[26], but computational power has been too limited to use discrete optimization to attack problems of reasonable scale.

There are two major contributions of this work. First, it shows that discrete optimization techniques are computationally feasible with our current set of tools. Additionally, the optimizations performed on our tree structure and symmetry-aware map can be applied more broadly to other discrete optimization problems.

Chapter 2 provides an overview of related work in the fields of discrete optimization, interpretable models, and rule lists. Chapter 3 proves definitions and explanations of the terminology used in the rest of this thesis. Chapter 4 describes the implementation and architecture of CORELS, paying special attention to the data structures used to make this problem tractable. Chapter 5 explains the data structure optimizations performed and the experiments used to validate these optimizations.

This thesis arose out of joint work with Elaine Angelino, Daniel Alabi, Margo Seltzer, and Cynthia Rudin. This joint work involved the development of the implementation of CORELS as well as proofs of the theoretical bounds upon which this work is based. However, the papers about the joint work focus more on the theoretical bounds than the data structure optimizations performed. Therefore, my thesis is intended to provide a different perspective on this work—focusing on the implementation details and trying to generalize to other types of systems.

2

Related Work

The use of classification models is popular in a number of different fields from image recognition[15] to churn prediction[16]. Oftentimes, however, simply receiving a prediction from software is not enough–it is important to have a predictive model that humans can investigate and understand[5,25,27,21,8]. For example, in fields such as medical diagnoses[?] and criminal sentencing[13], it is important to be able to investigate the reasons behind a model's predictions. One reason is that medical experts are unlikely to trust the predictions of these models if they are unable to understand why the model is making certain predictions[14]. Interpretable models also allow users to examine predictions to detect systemic biases in the model. This is especially important in classification problems, such as criminal recidivism prediction, where there are often race-related biases[13] or credit scoring where a justification is necessary for the denial of credit[3].

Tree structured classifiers are a popular technique that combines interpretability with high predictive accuracy. Also called decision trees, these trees are often used as either classification or regression tools. Every node in the tree classifier splits the data into two subsets; these subsets are then recursively split by nodes lower in the tree. Nodes are constructed by choosing an attribute that splits the data to minimize a certain metric. This metric differs from algorithm to algorithm, but it is usually focused on separating similar items into their own groups.Trees are constructed by recursively performing splits on the child subsets until the resulting subset is either entirely homogenous according to the metric or small enough according to some threshold. Methods for constructing decision trees differ primarily based on how they define this metric and what attributes they choose for each node. Breiman et al. laid out an seminal algorithm, CART, to create such trees[6]. CART tries to minimize Gini impurity which is a measure of the probability that any random element taken from a node is mislabeled. Another popular algorithm, C4.5, uses the idea of information gain to make its splits instead[24]. In C4.5, nodes are chosen such that each split minimizes the amount of information necessary to reconstruct the original data. Both algorithms grow the initial tree greedily. However, this leads to extremely large trees, so they perform a post-processing step of pruning to avoid overfitting and maintain interpretability.

While most decision trees are constructed greedily, and thus sub-optimally, there has been some work on constructing optimal decision trees[22]. There has even been the use of a branch and bound technique in an attempt to construct more optimal decision trees. Garofalakis et al. introduce an algorithm to generate more interpretable decision trees by allowing constraints to be placed on the size of the decision tree[10]. They use the branch-and-bound technique to constrain the size of the search space and limit the eventual size of the decision tree. During tree construction, they bound the possible Minimum Description Length (MDL) cost of every different split at a given node. If every split at that node leads to a more expensive tree than the MDL cost of the current subtree, then that node can be pruned. In this way, they were able to prune the tree while constructing it instead of just constructing the tree and then pruning at the end. However, even with the added bounds, this approach did not yield globally optimal decision trees, because they constrained the number of nodes in the tree.

Whereas decision trees are always grown from the top downwards, decision lists are built while considering the entire pool of rules. Thus, while decision trees are often unable to achieve optimal performance even on simple tasks such as determining the winner of a tic-tac-toe game, decision lists can achieve globally optimal performance. Decision lists are a generalization of decision trees since any decision tree can be converted to a decision list through the creation of rules to represent the leaves of the decision tree[26]. Thus, decision list algorithms are a direct competitor to the popular interpretable methods detailed above: CART and C4.5. Indeed, decision list algorithms are being used for a number of real world applications including stroke prediction[17], suggesting medical treatments[30], and text classification[18].

Work in the field of decision lists focuses both on the generation of new theoretical bounds and the improvement of predictive accuracy of models. Recent work on improving accuracy has led to the creation of probabilistic decision lists that generate a posterior distribution over the space of potential decision lists[17,29]. These methods achieve good accuracy while maintaining a small execution time. In addition, these methods improve on existing methods, such as CART or C4.5, by optimizing over the global space of decision lists as opposed to searching for rules greedily and getting stuck at local optima. Letham et al. are able to do this by pre-mining rules, which reduces the search space from every possible split of the data to a discrete number of rules. We take the same approach towards optimizing over the global search space, though we don't use probabilistic techniques. We also want to work in a regime with a discrete number of rules, thus we use the same rule mining framework from Letham et al. to generate the rules for our data sets[17]. This framework creates features from the raw binary data and then builds rules out of those features. Yang et al. builds on this earlier work by placing additional bounds on the search space and creating a fast low-level framework for computation, specifically a high performance bit vector manipulation library. We use that bit vector manipulation library to help perform computations involving calculating accuracy of rules[29].

Branch and bound was a technique originally developed to solve linear programming problems[12]. The branch and bound algorithm recursively splits the data into subgroups, yielding a tree-like structure. Then, by calculating a value corresponding to the end goal of the algorithm (e.g., accuracy), some branches of the tree can be proved to be worse in every case than another branch and therefore can be pruned, reducing the search space. For decades, it has been used to great effect in the realm of mixed integer programming[19]. It has also been applied to other NP-hard problems such as the Traveling Salesman Problem[20] and the Knapsack Problem[11]. More recently, it has been applied to machine learning problems such as feature subset selection[23] and clustering[9]. However, in recent times its popularity has declined in favor of convex optimization methods such as neural nets.

TODO: Complete paragraph about discrete optimization/bounds in 90s

In the 1980s and 90s, soon after the release of Valiant's theory of learnability[28] and Rivest's development of decision lists[26], there were a host of papers concerned with bounding the efficiency of learning boolean functions (which are equivalent to decision lists). Blum[47]

3

Definitions

We present definitions and explanations of concepts and terms that are used throughout this work. We motivate these definitions with a toy dataset attempting to answer the problem of predicting whether or not someone has already eaten lunch. Let us assume that this dataset is composed of 20 individuals and the time that they eat lunch each

if $time < 10 : 00AM$ then predict no
else if $time > 3 : 00PM$ then predict yes
else if $day == Sunday \wedge time > 10 : 30AM$ then predict yes
else if $day == Friday$ then predict no

Figure 3.1: Example rules that could be mined from our Lunch dataset.

day.

3.1 Rules

A *rule* is an IF-THEN statement consisting of a boolean antecedent and a classification label. We are working in the realm of binary classification, so the label is either a 0 or a 1. The boolean antecedents are generated from the rule mining mechanism[17] and are a conjunction of boolean features. Fig 3.1 These antecedents are *satisfied* by some data points (also called *samples*) and not for others. We say a rule *classifies* a given data point when the antecedent is satisfied for that data point.

As we combine these rules into rule lists, only the first rule that classifies any given data point can make a prediction for that data point. Thus, we say a rule *captures* a given data point if it is the first rule in a rule list to classify that data point.

3.2 Rule Lists

A *rule list* is an ordered collection of rules. As defined above, rules have an inherent accuracy based on what data they classify and how they predict the label. When they are placed

if $(age = 23 - 25) \wedge (priors = 2 - 3)$ then predict *yes*
else if $(age = 18 - 20)$ then predict *yes*
else if $(sex = male) \wedge (age = 21 - 22)$ then predict *yes*
else if $(priors > 3)$ then predict *yes*
else predict *no*

Figure 3.2: An example rule list that predicts two-year recidivism for the COMPAS dataset, found by CORELS.

into a rule list, however, their accuracy is based on what data they capture—which is usually not the same as what data they classify. They can perform better or worse than their inherent accuracy depending on what rules come before them in a given rule list. Thus, our algorithm is focused on finding the list composed of the best rules and the order that maximizes predictive accuracy. A rule list also has a *default rule*, placed at the end of all of these rules, that classifies all data points and predicts the majority label. This allows a rule list to make predictions for all points because any point not captured by the pre-mined rules is therefore captured by the default rule. We refer to the set of rules that compose a rule list, not including the default rule, as a *prefix*.

3.3 Objective Function

Rule lists have a loss function based on the number of points that are misclassified by the rules in the rule list. We define our *objective* function to be the sum of that loss and a regularization term. We use a *regularization* term, which is a constant times the length of the rule list. This has the effect of preventing overfitting on training data sets as well as preventing extremely long, and therefore uninterpretable, rule lists. While the objective is related to accuracy (a higher accuracy means a lower objective), we will be optimizing over the objective function instead of just the accuracy in order to get the benefits of the regularization term.

$$objective(RL) = loss(RL) + c * len(RL)$$

3.4 Bounds

For a set of n rules, there are $n!$ possible rule lists. Finding the optimal rule list using a brute force approach is infeasible for any problem of reasonable size. Our algorithm uses the discrete optimization technique of branch-and-bound to solve the combinatorially difficult problem of finding an optimal rule list. This requires tight bounds that allow us to prune as much of the search space as possible. These bounds are formalized and proved in Angelino et al.[2] and are reproduced in Appendix A. For clarity we present informal summaries of the important bounds here.

3.4.1 Lower Bound

We use the term *lower bound* to mean the best possible outcome for the objective function for a given prefix. We do this by calculating the error of the prefix and assuming that any points uncaptured by the prefix will be predicted correctly. This is equivalent to assuming that the default rule captures all points perfectly. Because any future extensions of the prefix can only ever make mistakes, we will be able to use it to prune our exploration. An

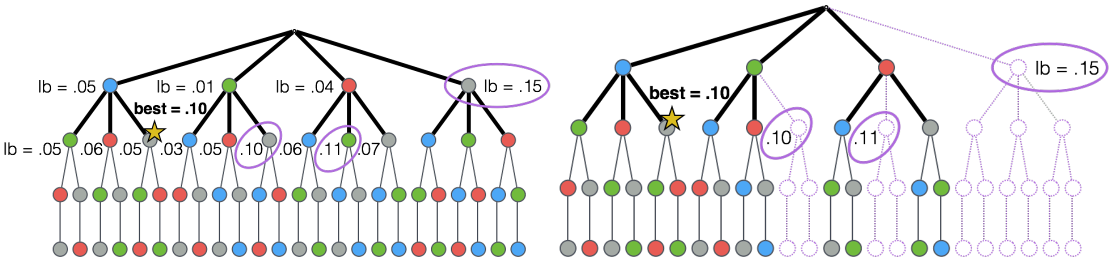

Figure 3.3: This tree shows the hierarchical objective bound in action. Our best objective seen is the prefix (blue, gray) with an objective of 0.10. Any prefixes with lower bound greater than this objective--(gray), (red, green), (green, gray) can be pruned and not ever looked at.

important property of the lower bound is that it increases monotonically.

3.4.2 Hierarchical Objective Bound

The main bound for our algorithm is the *hierarchical objective bound*. It says that we do not need to purse a rule list if it has a lower bound that is worse than the best objective we have already seen. This follows from the fact that lower bounds increase monotonically, so if the lower bound of rule list A is worse than the objective of rule list B, any extensions of rule list A can never be better than rule list B. This allows us to prune large parts of the search space by not pursuing rule lists that could never be better than something we've already seen.

3.4.3 Permutation Bound

As defined above, every sample is captured by precisely one rule–any sample that is caught by rule A in the rule list AB cannot be caught by rule B. Now consider a *permutation* of

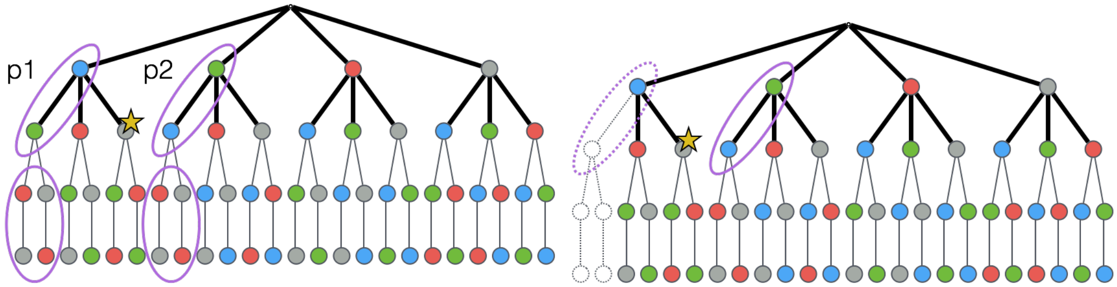

Figure 3.4: This tree shows the use of the permutation bound. Prefixes (blue, green) and (green, blue) are permutations of each other, so their lower bounds can be compared. The prefix (green, blue) has a better lower bound, so (blue, green) can be pruned and none of its children have to be examined.

the rule list AB: the rule list BA. Any samples that are captured by either rule A or B but not both will be captured identically in both rule lists. Samples that are captured by both rules will again be captured the same in both rule lists, though they may be predicted differently in the two rule lists. Thus, regardless of the order in which the rules appear, rule lists AB and BA will capture exactly the same set of data. They will differ only in which rules capture which samples, so their accuracy may differ. We can use this knowledge to create a bound as follows. If we know that the lower bound of AB is better than the lower bound of BA, we can eliminate from consideration all rule lists beginning with BA. This is due to the fact that any corresponding rule list beginning with AB will capture exactly the same samples as the equivalent rule list beginning with BA but will have a better objective score. Generalizing, we can eliminate all but one permutation of a given set of rules using this principle.

3.4.4 Support Bounds

Due to our regularization term in calculating our objective function, adding a rule that does little to help our accuracy will actually be harmful to the overall objective score. This allows us to place bounds that rely on the *support* of the rules we add. It never makes sense to add a rule that increases the objective function, so we only consider adding rules that capture enough points correctly to overcome the regularization penalty. By definition, rules that don't capture enough of the remaining points cannot capture them correctly, so this provides us with two closely related bounds. As our rule lists get longer many rules do not capture enough points that haven't already been captured and this bound begins to play an even larger role.

3.4.5 Equivalent Points Bound

This bound relies on the structure of our dataset. In our dataset, we may encounter two data points that have the same features but different labels. We call the set of these data points *equivalence points*, and describe the label that occurs less often as the *minority* label. Any rule that classifies one point in an equivalence class will also classify all other points in that class. However, it is impossible to correctly predict equivalence points with different labels using a single rule. So, for a given class of equivalence points, we know that we will mispredict all of the points with a minority label. We can thus update our lower bound to be tighter than just assuming that the default rule will capture all remaining points correctly. Now, we assume all remaining points will be captured incorrectly if it is an equivalent point with a minority label. This gives us much tighter lower bounds and in practice allows us to prune much more efficiently.

3.5 Curiosity

There are a number of different ways to explore the search space (see 4.2). Some methods, such as BFS, prioritize exploration—looking at all rule lists of a given length before proceeding to the next length. Others, such as a priority queue ordered by lower bound, focus on purely exploiting the best prefixes that we've seen. We define a new metric, *curiosity*, that blends together both exploration and exploitation. Curiosity is a function of both the lower bound and the number of samples captured. This prioritizes rule lists that still have many samples left to capture (exploration) while also pursuing rule lists with promising lower bounds (exploitation).

$$curiosity(RL) = (lowerBound(RL) - c * (len(RL) - 1)) * (nsamples/|captured(RL)|)$$

3.6 Remaining Search Space

One metric for tracking the efficacy of our optimizations will be seeing how quickly we reduce the remaining search space. We start with a combinatorially large search space, but quickly prune it down using our bounds. We calculate the remaining search space is by looking at all of the potential prefixes in our queue and seeing how much each prefix could potentially be expanded. Due to our regularization term, we are able to bound the maximum length of a prefix as our best objective gets updated. This upper bound on the maximum length of a viable prefix allows us to place an upper bound on the search space as well. We find that the remaining search space decreases rapidly at the beginning of execution, then slowly decreases until the very end of execution when it again rapidly decreases.

4

Implementation

This chapter lays out the design and implementation of the system used to generate optimal rule lists. We begin by describing each of the three main data structures used to run our algorithm–a prefix trie, a symmetry-aware map, and a queue. Then, we provide a walkthrough of the implementation of our main algorithm. We conclude with a discussion of the garbage collection, design tradeoffs, and memory tracking framework used.

4.1 Prefix Trie

The prefix trie is a custom C++ class and is used as a cache to keep track of rule lists we have already evaluated. Each node in the trie contains the metadata associated with that corresponding rule list. This metadata includes bookkeeping information such as what child rule lists are feasible as well as information such as the lower bound and prediction for that rule list. In addition to our base trie class, we implemented two different node types that we use in our algorithm. First, we implemented a class that has an additional field that tracks the curiosity of a given prefix as defined in 3. Curiosity is used by our queue to determine the order that prefixes are explored. Since the curiosity field is just a double, the memory overhead is minimal and the speed-up of using curiosity as opposed to BFS is sizable (see 5.1.2). We also implemented a class that has an additional field keeping track of the captured vector for that prefix. This captured vector has length *nsamples*, which means it is memory expensive. However, using the captured vector allows us to speed up our incremental computations because we would otherwise have to recompute this vector every time we used that prefix.

4.2 Queue

Our queue support many different types of search policies that we can use to explore the search space. We implement a number of different scheduling schemes including BFS, DFS, various priority metrics, and a stochastic exploration process. Due to our use of incremental computation, our queue contains pointers to leaves in the trie. The search process involves selecting which leaf node to explore. The stochastic exploration process bypasses the use of a queue by performing random walks on the trie until a leaf is reached. Our other scheduling schemes use a STL C++ priority queue to hold and order all of the leaves of the trie that still need to be explored. Our priority metrics can be ordered by curiosity, the objective of a prefix, or the lower bound of a prefix. We find that using an exploitation strategy such as ordering by curiosity or lower bound will usually lead to a faster runtime than using BFS.

In C++ the priority queue is a wrapper container that prevents access to the container underlying the queue. Therefore we cannot access elements in the middle of the queue, even if we have know the value that we're trying to access.. Thus, we run into a problem where we may delete something in the prefix trie that is currently in the queue but have no way to update the queue. We deal with this by lazily marking nodes as deleted in the prefix trie without deleting the physical node until it has been popped off of the queue. As Fig 4.1 shows, this leads to a situation where our logical queue is actually smaller than the physical queue.

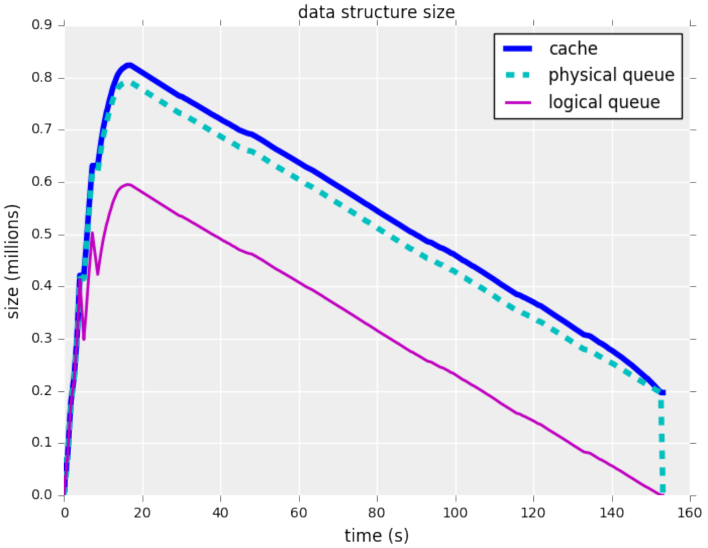

Figure 4.1: Size of prefix trie, logical queue, and physical queue over the execution of our algorithm. Note that the gap between the logical queue and the physical queue is due to our lazy garbage collection

4.3 Symmetry-aware map

We implement our symmetry-aware map as an STL unordered_map. We use this data structure to implement the permutation bound described in ??. We have two different versions of the map with different key types that allow permutations to be compared and pruned. In both cases, the values consist of the best lower bound and the actual ordering of the rules that is best for that permutation. In the first version, keys to the map are represented as the canonical order of the rules: i.e. the lists 3-2-1 and 2-3-1 both map to 1-2-3. The second version has keys that represent the captured vector. Our earlier permutation bound was based off of the fact that different permutations capture the same data, so this type of key will again be equivalent for the rule lists 3-2-1 and 2-3-1. Representing keys with captured vectors could potentially match more permutations since two prefixes may not be permutations of each other but might capture the same data points and therefore fall under the permutation bound. In fact, we find that this occurs only rarely, as the number of rule lists examined is only slightly less in the captured vector case. Despite only supporting one bound, the permutation map plays a significant role in our memory usage.

4.4 Incremental execution

We detail the execution of our program below. Our program ends when all leaves of the trie have been explored and there is nothing else in our queue. We can also choose to not receive the full certificate of optimality by exiting execution when a certain number of rule lists are in the trie. While there are still leaves of the trie to be explored, we use our scheduling policy to select the next rule list to evaluate. Then, for every rule that is not already in this parent rule list, we calculate the lower bound, objective, and other metrics for the potential child rule list consisting of the new rule appended to the parent rule list. We check the support bounds, the hierarchal objective bound, the permutation bound, and the lookahead bound. If the child rule list has a lower bound that is better than all of these bounds, then we insert it into the permutation map, the tree, and the queue and update the minimum objective if necessary. Otherwise, we do not insert it into any of our data structures and we

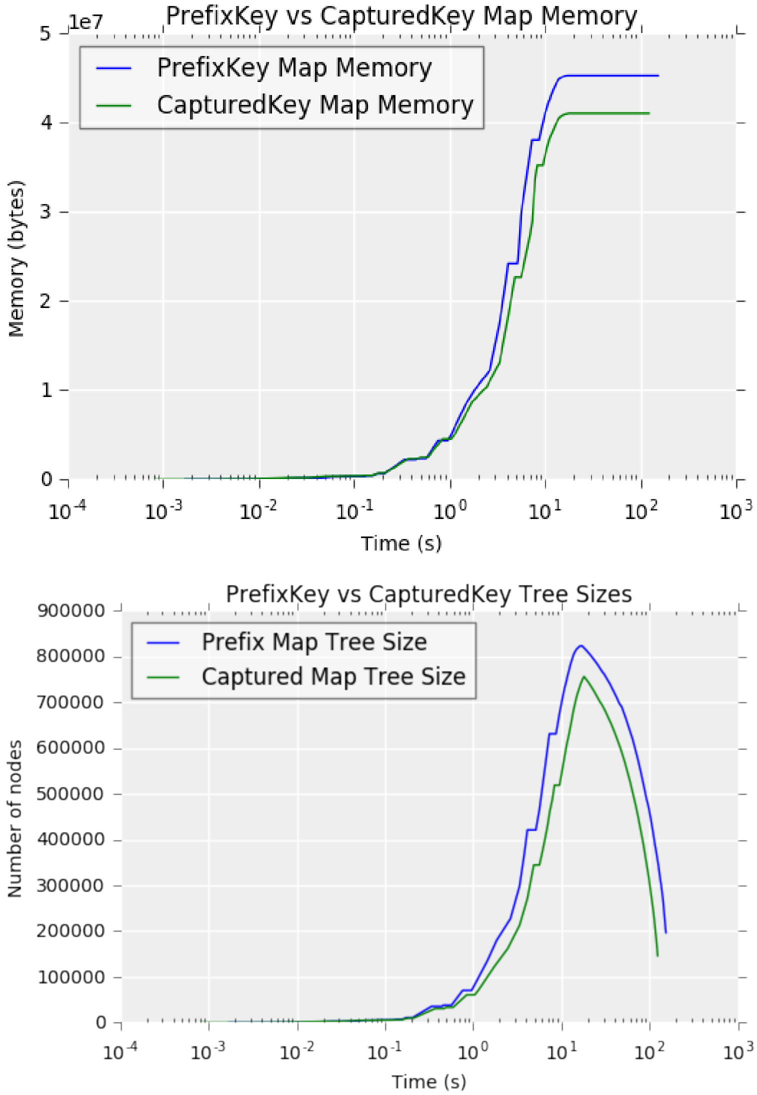

Figure 4.2: This graph shows the memory usage of the map when using CapturedKeys and PrefixKeys. We see that CapturedKeys provide minimal space savings over PrefixKeys due to the slight decrease in nodes inserted into the trie.

continue to the next potential rule to add, having excluded a rule list from being optimal. After evaluating all the rules we could add to our parent rule list, we return to the main loop and look at the next leaf of the trie.

Algorithm 1 Branch-and-bound for learning rule lists

```
Output: Outputs the best rule list and its objective

$opt \leftarrow NULL$
$T \leftarrow initializeTree()$
$Q \leftarrow queue(\ [T.root()\ ])$
$P \leftarrow map(\{\})$
while $Q$ not empty do
    $node \leftarrow Q$.pop( )
    $oldObj \leftarrow T$.minObjective( )
    incremental($node$, $T$, $Q$, $P$)                ▷ Evaluate all of node's children
    if $T$.minObjective( ) $<$ $oldOb$j then
        $opt \leftarrow T$.bestRuleList( )
        $T$.garbageCollect( )
return $opt$, $T$.minObjective( )
```

4.5 Garbage Collection

Every time we update the minimum objective, we garbage collect on the trie. We do this by traversing from the root to all of the leaves. Any time we encounter a node with a lower bound larger than the minimum objective, we delete its entire subtree. In addition, if we encounter a node with no children, we prune upwards—deleting that node and recursively traversing the tree towards the root, deleting any childless nodes. This garbage collection allows us to limit the memory usage of the trie. In practice, though, the minimum objective is not update that often, so garbage collection is triggered only rarely.

Algorithm 2 Incremental evaluation of a prefix

```
Input: Node to be evaluated $parent$, prefix tree $T$, queue $Q$, symmetry-aware map $P$
Output: None—destructively updates best objective in the tree

$prefix \leftarrow parent$.prefix( )
$t \leftarrow c*$ nsamples                          ▷ This forms the threshold for our support bounds
for $rule \notin prefix$ do
    $rlist \leftarrow prefix$.append($rule$)
    $cap \leftarrow parent$.notCaptured( ) $\wedge$ $rule$.notCaptured( )
    if $|cap| < t$ then                                          ▷ Minimum support bound
        continue
    $capZero \leftarrow cap \wedge T$.zeroLabel( )      ▷ Record how many zeros the new rule captures
    $corr \leftarrow max\{|capZero|, |cap| - |capZero|\}$
    if $corr < threshold$ then                               ▷ Minimum correct support bound
        continue
    $lb = parent$.bound( ) $-$ $parent$.minority( ) $+ \frac{|cap|-corr}{nsamples} + c$   ▷ Calculate lower bound
    if $lb >= T$.minObjective( ) then                                         ▷ Objective bound
        continue
    $notCap \leftarrow parent$.notCaptured( ) $\wedge \neg cap$
    $notCapZero \leftarrow notCap \wedge T$.zeroLabel( )
    $defaultCorr \leftarrow max\{|notCapZero|, |notCap - notCapZero|\}$
    $obj \leftarrow lb + \frac{|notCap|-defaultCorr}{nsamples}$                                 ▷ Calculate objective
    if $obj < T$.minObjective( ) then                            ▷ Update minimum objective
        $T.minObjective(obj)$
    if $lb + c >= T$.minObjective( ) then ▷ Lookahead bound—don't add to queue if its children will all be worse than the minimum objective
        continue
    $n \leftarrow P.insert(rlist)$          ▷ Symmetry-aware map handles permutation bound for us
    if $n$ then                          ▷ $n$ will be NULL if it failed the permutation bound check
        $T$.insert($n$)
        $Q$.push($n$)
```

4.6 Templates vs Inheritance

Our system has a lot of different modular parts–various priority metrics, symmetry-aware map types, and different types of information stored in trie nodes. Since we were using

C++, we took advantage of its templating system to achieve this modularity. We believed that it would be the easiest way to switch out different components of the system, but did not think about the ramifications of this choice. Due to code duplication, large amount of templates can lead to a much larger execution. Indeed, with a pure templating system, our executable was 253732 bytes. Execution time took about 126s.

Our other alternative way to maintain modularity was to take advantage of C++'s class system and use inheritance and polymorphism. Now, instead of having a template argument for each data structure, we can have a base data structure type and then implement each of our extensions as subclasses. Therefore, we can write all of our functions to take the base class as arguments and then pass in the specialized subclasses based on command line arguments. This requires the use of virtual functions, which are potentially a bit slower than regular functions because they require a vtable lookup. With a pure inheritance framework, our executable was 171288 bytes. However, even with the inheritance framework, the runtime of our algorithm was not significantly different from the template runtime. Thus, since inheritance provided a much cleaner code base to work with, we decided to switch to an inheritance based framework.

4.7 Memory tracking

We track memory through the use of C++11 custom allocators. Our allocators are simple malloc wrappers that also log which data structure the allocation is coming from–the trie, the map, or the queue. We validated the accuracy of this memory tracking by running the program under Valgrind's Massif heap profiler tool and comparing the outputs. Our outputs matched Valgrind's to within a few tenths of a percentage points, so we concluded that our memory tracking was successful. On a trial run, Valgrind's tool Massif reported that we used 98,732,685 bytes, while our data structure tracking recorded 98,495,580 bytes, a difference of 0.2% which is easily explained by miscellaneous heap allocations. The heap profiling of Valgrind has a large overhead, while our memory logging is more limited and therefore overhead is very minimal. This allows us to write out our memory usage per data structure on a regular basis without incurring the large overhead of Valgrind.

5

Experiments

All of the following experiments were executed on a MacBook Air with a 1.4 GHz Intel Core i5 processor and 8GB of RAM. This is a very basic setup and just further goes to show the potential ubiquity of using discrete optimization techniques. Our algorithm is run on one of the training folds of the COMPAS dataset. The fold has 6489 data points from which we're able to extract 155 rules. 2947 of these individuals are labeled ”yes”, meaning they have committed a second crime after being arrested—the other 3542 individuals are labeled ”no”.

Naively evaluating all prefixes of up to length length 5 would require examining 84,382,025,575 different prefixes. However, with our solution, we examine only 99,891,878 prefixes in total. This is a reduction of 845x. Note that this is a lower bound since any brute force solution would have to examine prefixes of longer than length 5 in order to certify optimality. It takes us 0.000002s to evaluate a single prefix. Thus, a naive solution would take 168,764s—about 2 days. While this is a long time, it is still not a completely unreasonable amount of time, though it clearly wouldn’t scale to larger problems.

However, if we look at how processor speeds have changed over the last 25 years, we can see that computers are about 1,000,000 faster now than they were in 1993.[Sup] Thus, when we combine our algorithmic improvements with the increased processor speeds, we see that our algorithm runs almost 1 billion times faster than a naive implementation would have in 1993. This explains why there has been a dearth of work on algorithms focused on optimality.

We first examine how much each of our optimizations helps. We will show that without our algorithmic and data structure improvements, it would be intractable to solve a real-world problem. We will also show that even with our improvements, this algorithm would not have been able to be run just 25 years ago. Next, we will examine the improvements made to our symmetry-aware map to reduce the memory overhead on our algorithm. Finally, we will explore a parallel implementation.

5.1 Ablation–how much does each bound/optimization help?

We wished to determine how much each theoretical bound and data structure optimization helped, so that we could determine which one was the most helpful. In addition, we wanted to find out how much of an overall speed-up our work has given us. In order to do that, we ran a number of experiments where we took out a single component from our system and measured the effects on the runtime and memory usage of our algorithm. We find that the equivalent points bound is our most important optimization for running in a reasonable amount of time. Each other optimization and bound plays an important, but not crucial, role in the speed of our algorithm.

5.1.1 Full CORELS system

Our baseline system with all of our improvements is the CORELS system. Running the full CORELS system on the COMPAS dataset yields the optimal solution and its certificate within 121s. The maximum memory usage is 150MB, with the majority of that coming from our prefix trie.

With about 1 million items active at the peak, this comes out to about 150 bytes per item. This includes copies of that item that are kept across the prefix trie, symmetry-aware map,

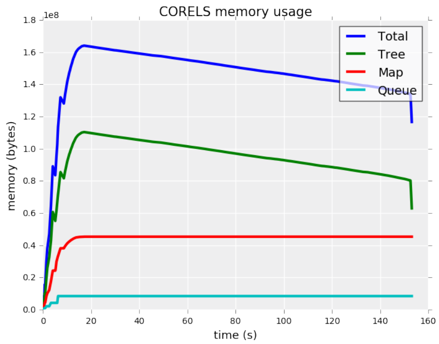

Figure 5.1: Memory usage of full CORELS system. Our prefix tree accounts for the bulk of the memory usage

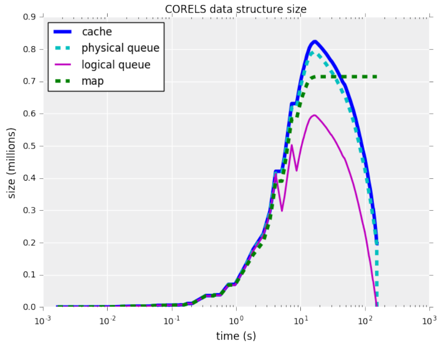

Figure 5.2: The size (in number of items) of each of our main data structures. Corresponds to the amount of memory used, as shown above.

and queue.

5.1.2 Priority Queue

One of our first optimizations was the use of a priority queue to utilize different exploration techniques. Removing the priority queue and simply using BFS allows us to find the optimal solution and certificate in 142s. So, the queue gives us a slight speed-up without requiring much memory at all. On other problems, we've found that using a priority queue

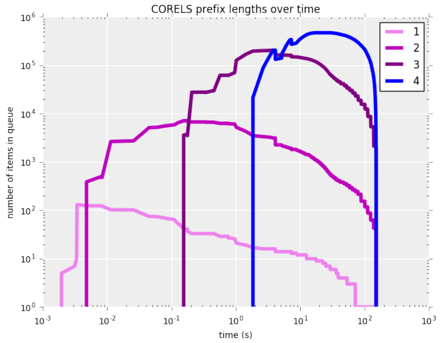

Figure 5.3: Tracks the number of prefixes of a given length active in the queue for the full CORELS system

can lead to a large speed-up.

5.1.3 Support Bounds

The support bounds are intrinsic to the rules and are the first bounds we check in our execution because they are the simplest to compute. This ease of computation belies the fact that these bounds prevent the pursuit of useless rules. Without these bounds we complete the certification in 261s.

5.1.4 Symmetry-aware map

The symmetry-aware map is a novel way of approaching branch-and-bound and it plays a large role in our elimination of search space. Using the symmetry-aware map means that we are able to only pursue one prefix out of all of its permutations. As our prefixes grow to length 4 and beyond, that means we can eliminate at least 23 prefixes. This is an important optimization and removing it takes a long time to complete the certification—1147s.

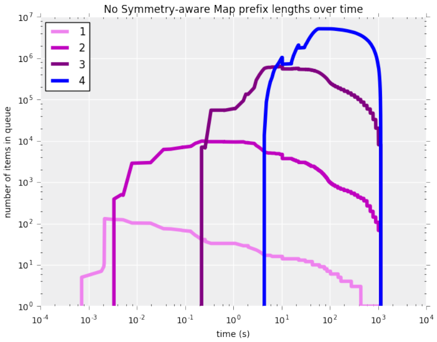

Figure 5.4: Tracks the number of prefixes of a given length active in the queue for CORELS without a symmetry-aware map

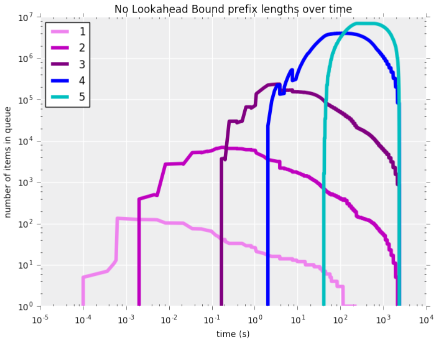

Figure 5.5: Tracks the number of prefixes of a given length active in the queue without our lookahead bound

5.1.5 Lookahead Bound

Our lookahead bound is useful for preventing us from examining longer prefixes than we need to. From running our full CORELS system, we know that our optimal rule list is of length 4. With our lookahead bound, we never have to examine prefixes of length 5. However, removing that bound means that we do look at many prefixes of length 5 which drastically slows our computation to 2360s.

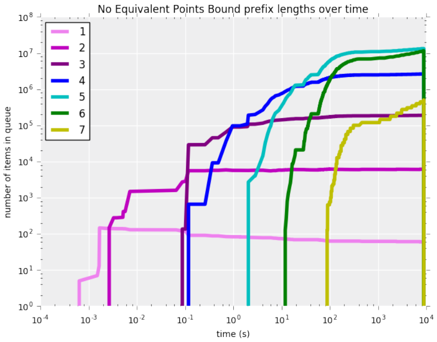

Figure 5.6: Tracks the number of prefixes of a given length active in the queue without the equivalent points bound

5.1.6 Equivalent points bound

The equivalent points bound is our optimization that provides the largest benefit. Since all of our other bounds eliminate prefixes contingent on the lower bound, the equivalent points bound is important because it tightens the lower bound. Removing this bound makes it impractical to complete real world problems. We stop execution after multiple hours (8500 s) and show that there is still a large portion of the search space to be explored.

TODO: Re-run BFS because numbers seem wrong

5.2 Symmetry-aware Map Optimization

As one of our two main data structures, the memory usage of our symmetry-aware map is something that was very important to us. We saw above that running without a symmetry-aware map leads to a much longer runtime, so it was crucial that we cut down on this overhead. We initially found that we often had trouble running large data sets because we would

Removed component	t_{total} (s)	t_{opt} (s)	i_{total} ($\times 10^6$)	$Q_{\max}$ ($\times 10^6$)	$K_{\max}$
none (CORELS)	121	7.3	0.83	0.59	4
priority queue (BFS)	142	0.14	0.73	0.19	5
support bounds	261	11	1.3	0.98	4
symmetry-aware map	1147	23	6.5	5.6	4
lookahead bound	2360	7.6	7.6	10.7	5
equivalent pts bound	>8500	>42	>29	>28	≥7

Table 5.1: Per-component performance improvement. The columns report total execution time, time to optimum, number of queue insertions, maximum queue size, and maximum evaluated prefix length. The first row shows CORELS; subsequent rows show variants that each remove a specific implementation optimization or bound. (We are not measuring the cumulative effects of removing a sequence of components.) All rows represent complete executions, except for the final row, in which each execution was terminated due to memory constraints

run out of memory before certifying the optimal rule list. In these long runs, our permutation map would grow to be hundreds of thousands or millions of entries large. Thus, carrying a lot of memory overhead with each node led to serious memory bloat and ran us out of memory. This section explores some of the techniques we used to handle this memory bloat.

Our symmetry-aware map was originally an STL map with keys as STL sets of size_t and values as pairs of STL vectors of size_t and a double. The first problem is that the STL map is implemented as a red-black tree, meaning every node had the overhead of multiple pointers pointing to its children. This can be solved using a STL unordered map, which is implemented as a hashtable. That has some overhead, since it will allocate more buckets than are filled, but nowhere near to the two pointers per node of overhead of the STL map.

Next, the representation of the rule ids no longer had to be size_t. We made an assumption that we'll never have more than 65000 rules, so we used unsigned shorts for our rule ids. This means that our final data structure was able to use only 2 bytes per rule id instead of 8 bytes for a size_t. Since we're keeping two different representations of the prefix–the canonical order and the actual order of the prefix, and these prefixes can be of length on the order of 5 rules, saving 6 bytes per rule translates to a lot of

Our initial implementation used an STL set as the key, which carries a lot of overhead because of all the operations that a set needs to support. For a prefix of length 6, the corresponding key therefore takes up After starting with the STL set, we transition to a sorted STL vector. This reduced overhead because a vector is essentially just an array with a little bit of overhead. However, all of these STL containers support a broader range of operations than we needed. All we needed was some way to compare the canonical orders and determine if they were the same. We eventually achieved this simply by allocating a chunk of memory that held the length of the prefix and the sorted order of the ids. This means that, for a prefix of length 6, we use 14 bytes for the key.

We had a similar set of issues with our values for the permutation map. Beginning with a STL vector to keep track of the actual order of the rules in the prefix, we again wanted to remove the overhead incurred by the fact that STL containers need to support a wider array of operations than we needed. Thus, we could use a similar technique to what we did with the keys to keep track of the prefix order. However, in fact, we can do even better

Map version	Key version	Value version	Total Memory (b)	t_{total} (s)
Map	Set (size_t)	Vector (size_t)	190751744	154
Unordered Map	Set (size_t)	Vector (size_t)	185897576	150
Unordered Map	Set (unsigned short)	Vector (unsigned short)	148611880	148
Unordered Map	Vector (unsigned short)	Vector (unsigned short)	68713960	147
Unordered Map	Custom Key	Custom Value	62865376	139

Table 5.2: Symmetry aware map improvements. The columns report the type of map, type of key, type of value, total memory used by the symmetry-aware map, and time of execution. Each row represents a different version of the symmetry aware map that we tested. We see that

because we already have a canonical representation of the rules–all we need in the value was the ordering of those rules. Thus, we can use unsigned chars instead of unsigned shorts to record the index of where the rules in the canonical order are in the prefix.

In total, these improvements give us a 3x memory reduction on this problem. This takes the permutation map from being the largest data structure to being smaller than the tree. On larger problems, this memory decrease is even more important.

However, these optimizations all pertained to the symmetry-aware map with prefix keys. Dealing with the captured vector keys required a different approach. Our captured vectors were of type mpz, and since unordered_map requires a custom hash for unsupported types, we had to find a way around that. STL has some hash functions for built in types, so we initially wrote a function to convert between our VECTOR type and a $std :: vector < bool >$. This conversion turned out to be very slow, especially as we were running on data sets with many samples, meaning these mpz types were fairly memory intensive. Once we wrote our own hash function and were able to use our mpz type again, it became much faster.

5.3 Parallelization

The majority of the time of our program is spent in the incremental section of our execution. Trying to extend prefixes by calculating the bounds and inserting into our various data structures is the vast majority of our program.

So, we began by trying to parallelize the inner loop of our incremental evaluation. This inner loop involves trying to add all possible rules to the prefix we're currently trying to extend. In order to add these rules, though, we need to calculate the bounds based on characteristics of the parent prefix. Without locking the parent, we run into a race condition that leads to a segfault in the STL code. Locking the parent had too much contention, however, because all of our threads needed to own the parent lock for the majority of the loop—essentially rendering the loop sequential.

We realized, however, that the tree structure of our prefix trie lends itself nicely to parallelization. Instead of parallelizing the evaluation of a single prefix, we can parallelize our search over the tree itself. We do this by creating the tree in the master thread and then spawning worker threads to work in different parts of the tree. Thus, there is no contention on parents since only one thread can access a node at a time. The shared state only consists of the minimum objective, and this is kept in the master thread. Deletion and garbage collection is handled by the master thread as well.

We find a linear speed up when moving from 1 to 2 threads, but diminishing returns as we increase the number of threads.

6

Conclusion

Through the use of tight theoretical bounds and clever data structure optimizations, we are able to find and certify the optimal rule list on real-world problems. This idea of deterministically finding the optimal rule list has been around for a while, but it has not been feasible until recently. Due to dramatic increases in processor speed and computer memory, discrete optimization techniques can be applied to real problems and complete them in a reasonable amount of time. Our novel bounds and use of branch-and-bound technique applied to the problem of discovering the optimal rule list show the renewed usefulness of applying discrete optimization approaches to problems. Further optimizations involving distance sensitive hashing and bit-packing could reduce runtime and memory usage even further.

A

Proof of Bounds

This is adapted from joint work[2].

References

[Sup] Top500 supercomputer sites, directory page for top500 lists.

[2] Angelino, E., Larus-Stone, N., Alabi, D., Seltzer, M., & Rudin, C. (2017). Learning certifiably optimal rule lists. *KDD*.

[3] Baesens, B., Mues, C., De Backer, M., Vanthienen, J., & Setiono, R. (2005). *Building Intelligent Credit Scoring Systems Using Decision Tables*, (pp. 131–137). Springer Netherlands: Dordrecht.

[4] Blum, A. (1992). Learning boolean functions in an infinite attribute space. *Mach. Learn.*, 9(4), 373–386.

[5] Bratko, I. (1997). *Machine Learning: Between Accuracy and Interpretability*, (pp. 163–177). Springer Vienna: Vienna.

[6] Breiman, L., Friedman, J., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. CRC press.

[7] Dhagat, A. & Hellerstein, L. (1994). Pac learning with irrelevant attributes. In *Proceedings 35th Annual Symposium on Foundations of Computer Science* (pp. 64–74).: IEEE.

[8] Freitas, A. A. (2014). Comprehensible classification models: A position paper. *SIGKDD Explor. Newsl.*, 15(1), 1–10.

[9] Fukunage, K. & Narendra, P. M. (1975). A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comput.*, 24(7), 750–753.

[10] Garofalakis, M., Hyun, D., Rastogi, R., & Shim, K. (2000). Efficient algorithms for constructing decision trees with constraints. (pp. 335–339).

[11] Kolesar, P. J. (1967). A branch and bound algorithm for the knapsack problem. *Management science*, 13(9), 723–735.

[12] Land, A. H. & Doig, A. G. (1960). An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, (pp. 497–520).

[13] Larson, J., Mattu, S., Kirchner, L., & Angwin, J. (2016). How we analyzed the compas recidivism algorithm. *ProPublica*.

[14] Lavrač, N. (1999). Selected techniques for data mining in medicine. *Artificial Intelligence in Medicine*, 16(1), 3 – 23. Data Mining Techniques and Applications in Medicine.

[15] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.

[16] Lemmens, A. & Croux, C. (2006). Bagging and boosting classification trees to predict churn. *Journal of Marketing Research*, 43(2), 276–286.

[17] Letham, B., Rudin, C., McCormick, T. H., & Madigan, D. (2015). Interpretable classifiers using rules and Bayesian analysis: Building a better stroke prediction model. *Annals of Applied Statistics*, 9(3), 1350–1371.

[18] Li, H. & Yamanishi, K. (2002). Text classification using esc-based stochastic decision lists. *Information Processing and Management*, 38(3), 343 – 361.

[19] Linderoth, J. T. & Savelsbergh, M. W. P. (1999). A computational study of search strategies for mixed integer programming. *INFORMS J. on Computing*, 11(2), 173–187.

[20] Little, J. D., Murty, K. G., Sweeney, D. W., & Karel, C. (1963). An algorithm for the traveling salesman problem. *Operations research*, 11(6), 972–989.

[21] Martens, D., Vanthienen, J., Verbeke, W., & Baesens, B. (2011). Performance of classification models from a user perspective. *Decision Support Systems*, 51(4), 782 – 793.

Recent Advances in Data, Text, and Media Mining and Information Issues in Supply Chain and in Service System Design.

[22] Moret, B. M. E. (1982). Decision trees and diagrams. *ACM Comput. Surv.*, 14(4), 593–623.

[23] Narendra, P. M. & Fukunaga, K. (1977). A branch and bound algorithm for feature subset selection. *IEEE Trans. Comput.*, 26(9), 917–922.

[24] Quinlan, J. R. (1993). *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[25] Quinlan, J. R. (1999). *Some Elements of Machine Learning*, (pp. 15–18). Springer Berlin Heidelberg: Berlin, Heidelberg.

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

[27] Rüping, S. (2006). *Learning interpretable models*. PhD thesis.

[28] Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11), 1134–1142.

[29] Yang, H., Rudin, C., & Seltzer, M. (2016). Scalable Bayesian rule lists. *Preprint at arXiv:1602.08610*.

[30] Zhang, Y., Laber, E. B., Tsiatis, A., & Davidian, M. (2015). Using decision lists to construct interpretable and parsimonious treatment regimes. *Biometrics*, 71(4), 895–904.