# 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

# Acknowledgments

This thesis describes joint work with Elaine Angelino, Margo Seltzer, Cynthia Rudin, and Daniel Alabi. Margo has been my advisor, professor, boss, and research supervisor and I cannot thank her enough for all her mentorship and advice. In particular, she has encouraged my interest in research, introduced me to this specific research project, and has guided me throughout the process of writing a thesis. I would not have been able to write this thesis if not for her help. I would also like to thank Elaine for her encouragement and guidance on this project, as well as her patience with my questions and inexperience. I would like to thank all of my friends for their support, especially my blockmates—CJ Christian, Juanky Perdomo, Renan Carneiro, Matthew DiSorbo, and Demren Sinik. Last, but certainly not least, I would like to thank my brothers Micah Stone and Jeremy Larus and my parents James Larus and Diana Stone for their continual love, support, and advice.

# 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[19]. 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[33]. 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[23,38] 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 en-

countered 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[19]. 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[33], 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[21] to churn prediction[22]. 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[6,32,34,27,12]. For example, in fields such as medical diagnoses[4] and criminal sentencing[19], 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[20]. 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[19] 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 algo-

rithm, CART, to create such trees[7]. 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[31]. 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.

The problem of constructing the optimal binary decision tree has been shown to be NP-Complete[15], where optimal means the fewest number of nodes. So, while most decision trees are constructed greedily, and thus sub-optimally, there has been some work on constructing optimal decision trees[28]. 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[14]. 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[33]. 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[23], suggesting medical treatments[40], and text classification[24].

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[23,38]. 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[23]. 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[38].

Rivest's introduction of decision lists[33] was soon after Valiant's theory of learnability[37]. Much of the work in the years immediately following the invention of decision lists were focused on issues of learnability and complexity rather

than practical implementations. Some of this work has focused on the attribute efficiency of various algorithms[5,9], while others have focused on the relationship between decision lists and classes of boolean functions such as DNF or CNF[33,11]. Other authors focused on a specific case of decision lists such as homogenous decision lists[35] or a more general cases of the decision list problem such as decision committees[30]. Throughout the past 3 decades, the main thrust of the research has been to improve the complexity of both the number of examples and the runtime of their algorithms[16]. However, most of these techniques are focused on finding an approximately correct solution. Despite discussing that the class of problems surrounding learning decision lists was NP-Hard, there seems to be no work on trying to solve the optimality problem. We hypothesize that hardware limitations played a large part in the lack of development of discrete optimization techniques with regards to the problem of decision lists.

Branch and bound was a technique originally developed to solve linear programming problems[18]. 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

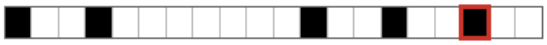
used to great effect in the realm of mixed integer programming[25]. It has also been applied to other NP-hard problems such as the Traveling Salesman Problem[26] and the Knapsack Problem[17]. More recently, it has been applied to machine learning problems such as feature subset selection[29] and clustering[13]. However, over the past few decades, its popularity has declined in favor of convex optimization methods such as neural nets. Researchers found other ways to achieve high accuracy for their predictive models by using large datasets that would take too long for branch and bound to complete. However, the continual improvements in computer hardware has led to a recent resurgence in the use of branch and bound techniques[10].

# 3

# Definitions

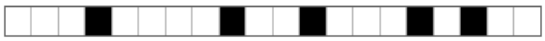
We present definitions and explanations of concepts and terms that are used throughout this work. Throughout this chapter, we provide examples from a hypothetical dataset, Computer. We will use Computer to answer the problem of predicting whether someone uses a Mac or a PC. Computer is a dataset

Rule 1: **if age** $< 25$ **then predict** $Mac$



**Rule 2: if (in college) then predict** $Mac$

**Rule 3: if age** $> 45 \wedge \neg$ **(Software Engineer) then predict** $PC$



**Rule 4: if** $\neg$ **(lives in California) then predict** $PC$

**Figure 3.1:** Example rules from the Computer dataset . The bit vector below the rule is a visual representation of the rule where each box is a data point. Black boxes represent data points that are classified by a rule. Data points that are classified incorrectly are outlined in red.

comprised of 20 individuals, whether they use a Mac or PC, and other personal details that may help with the prediction.

## 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 (or an equivalent label). The boolean antecedents are generated from the rule mining mechanism[23] and are a conjunction of boolean features. 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. A rule's *support* is comprised of all of the data points that are classified by a rule. Rules therefore have an inherent

**Rule List 1**

**if age** $< 25$ **then predict** $Mac$
**else if** $\neg$ **(lives in California) then predict** $PC$
**else predict** $PC$

**Rule List 2**

**if** $\neg$ **(lives in California) then predict** $PC$
**else if age** $< 25$ **then predict** $Mac$
**else predict** $PC$

**Figure 3.2:** Example rule lists from the Computer dataset made using rule 1, rule 4, and a default rule. As in Fig 3.1, black boxes represent data points captured by the rules in the rule list and red marks incorrect classification. All of the white boxes are classified by the default rule. Rule 4 has an accuracy of 50% in Rule List 2, but an accuracy of 100% in Rule List 1 when it comes after rule 1. So, even though the two rule lists are permutations of each other, Rule List 1 performs much better than Rule List 2.

accuracy based on their support. Fig 3.1 provides an example of 4 rules that could be mined from the Computer dataset. Rule 1, has a support of 5 but only classifies 4 of those samples correctly—thus its inherent accuracy is 80%.

## 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. As we combine these rules into rule lists, however, 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. Therefore, when rules are placed into a rule list, their accuracy is based on what data they capture—which is not necessarily 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. For example, we can see in Fig 3.2 that even though Rule 4 only has an inherent accuracy of 50%, in Rule List 1 it has an accuracy of 100% because it is placed below Rule 1.

Our algorithm is focused on finding the list that combines rules in an order that maximizes predictive accuracy. A rule list also has a *default rule*, placed at the end of any rule list, 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 captured by the default rule. We refer to the length of a rule list as the number of pre-mined rules, without including the default rule. Fig 3.2 shows two rule lists of length 2. We define a *prefix* as any subset of the rules at the beginning of a rule list. We will often use prefix to refer to the list of pre-mined rules in a rule list, not including the default rule.

## 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—including the default rule. We define our *objective* function to be the sum of that loss and 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 discouraging extremely long, and therefore uninterpretable, rule lists. While the objective is related to accuracy (a higher accuracy means a lower objective), we will be minimizing over the objective function instead of maximizing the accuracy to reap the benefits of the regularization term. Let RL be a rule list, then:

$$objective(RL) = loss(RL) + \lambda * len(RL)$$

We can calculate the objective for Rule List 1 in Fig 3.2 as follows. The loss from rules 1 and 4 is 0.05, while the loss from the default rule is 0.05. Assuming a regularization constant of $\lambda = 0.01$, the objective for Rule List 1 is $0.05+0.05+2*0.01 = 0.12$.

## 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 this 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 loss of the prefix and assuming that any points not captured 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 never do better than this perfect default rule, we will be able to use this bound to prune our exploration. Therefore, the lower bound increases monotonically—any rules we add can only make mistakes that this perfect default rule does not make. This also

means that our lower bound calculation is always an underestimate of the true lower bound. Much of the theoretical work on this project is focused on creating expressions for the lower bound that is as close as possible to the true lower bound so that it is easier to prune.

$$lower\_bound(prefix) = loss(prefix) + \lambda * len(prefix)$$

The lower bound for Rule List 1 is therefore $0.05 + 2 * 0.01 = 0.07$, which we can see is less than the objective function. The objective function includes the error made by the default rule, but the lower bound takes into account that we might still add a rule that could correctly classify the data point that is misclassified by the default rule.

### 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 pursue 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 2 is equal to or worse than the objective of Rule List 1, any extensions of Rule List 2 can never be better than Rule List 1. This allows us to prune large parts of

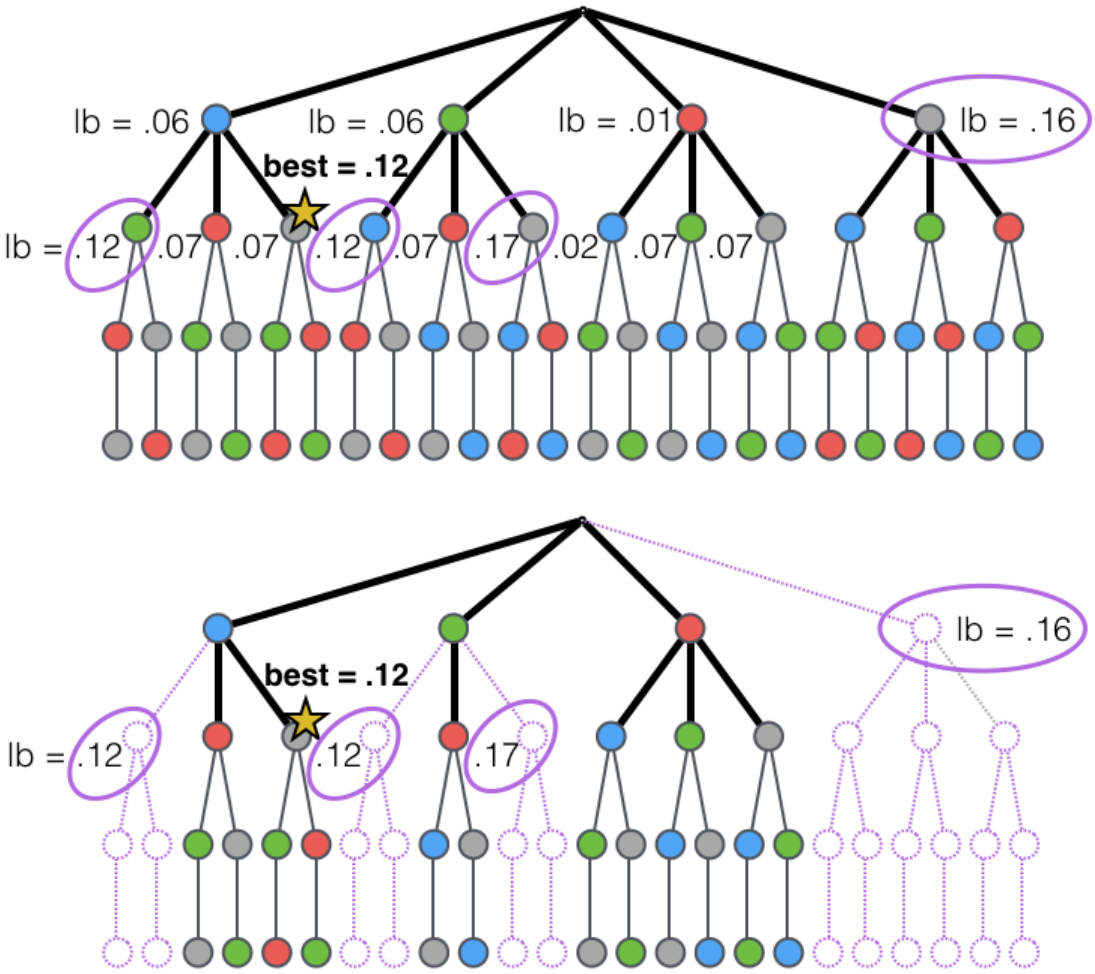



**Figure 3.3:** This tree shows the hierarchical objective bound in action. Each color represents one of our 4 rules from Fig 3.1: blue = Rule 1, green = Rule 2, red = Rule 3, and gray = Rule 4. Thus, each path in the tree represents a rule list. Our best objective seen is the rule list (1, 4, default) with an objective of $0.12$. Any prefixes with lower bound greater than this objective---all prefixes beginning with (4), (1, 2), (2, 1), or (2, 4)---can be pruned and not examined.

the search space by not pursuing rule lists that could never be better than something we have already seen. Fig 3.3 provides an example of our pruning mechanism in action. It shows how a potential exploration of the search space could

lead to a difference in the best observed objective and the lower bounds of some rule lists. Our algorithm will use that difference to prune parts of the search space and turn a combinatorially large problem into a manageable one.

### 3.4.3 Permutation Bound

As defined above, every sample is captured by precisely one rule—in the prefix (1,4), any sample that is captured by rule 1 cannot be captured by rule 4. Now consider a *permutation* of the prefix (1, 4): the prefix (4, 1). Any samples that are captured by either rule 1 or 4 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, prefixes (1, 4) and (4, 1) will capture exactly the same set of data—this can be seen in the bit vectors of Fig **??**. They will differ only in which rules capture which samples, so their accuracies may differ. We can use this knowledge to create a bound that we call the *permutation bound*. If we know that the lower bound of the prefix (1, 4) is better than the lower bound of (4, 1), we can eliminate from consideration all rule lists beginning with (4, 1). This is due to the fact that any rule list beginning with (1, 4) will capture exactly the same samples as the equivalent rule list beginning with (4, 1), but will have a better objective score. Generaliz-

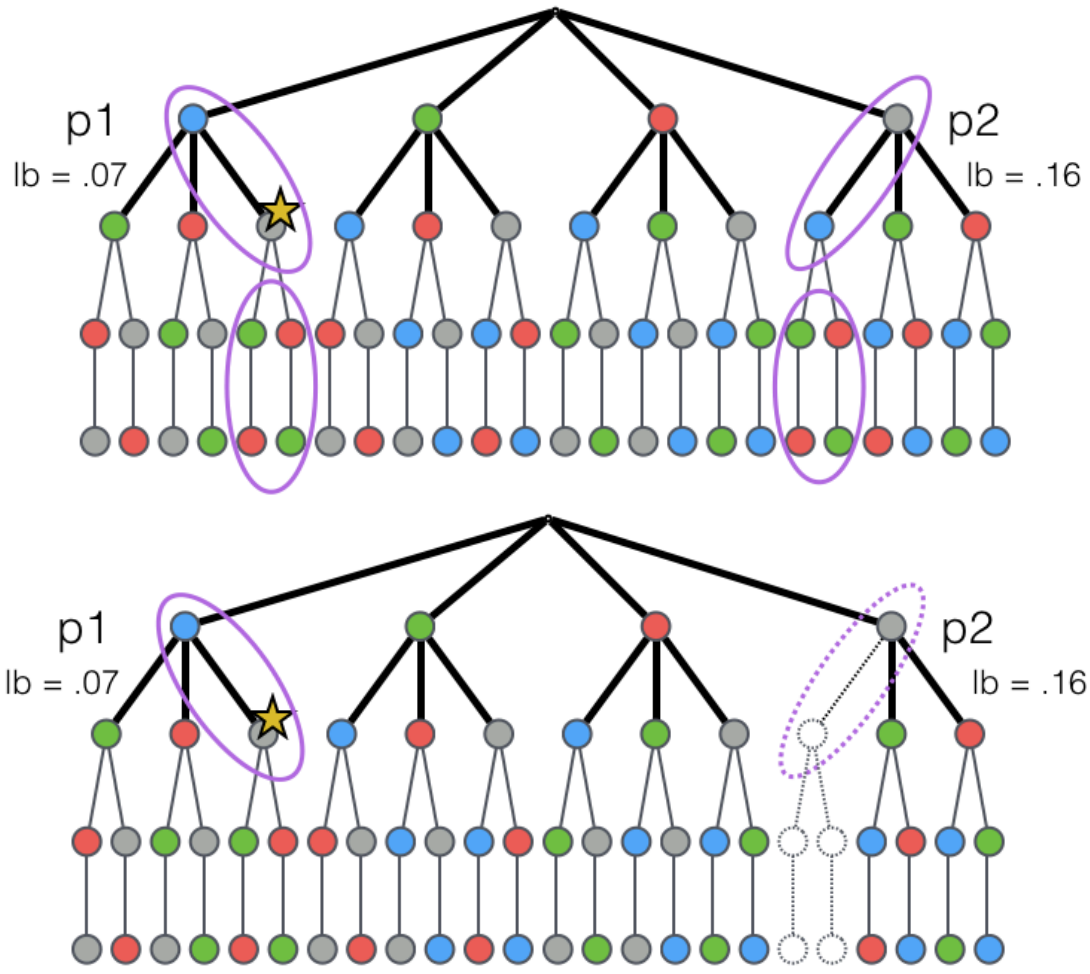



**Figure 3.4:** This tree shows the use of the permutation bound in action. Prefixes (1, 4) and (4, 1) are permutations of each other, so they capture identical data (see Fig ??). When we compare their lower bounds, we see the prefix (1, 4) has a better lower bound. Any rule list beginning with (4, 1) would be worse than the corresponding rule list beginning with (1, 4), so (4, 1) can be pruned and none of its children have to be examined.

ing this principle allows us to eliminate all but one permutation of a given set of rules. When we are dealing with these permutations, it is helpful to map all permutations to a single ordering by ordering the rules in numerical order. We

call this the *canonical ordering* of a prefix. For example, the prefixes (1, 4) and (4, 1) both map to the canonical ordering (1, 4).

### 3.4.4 Support Bounds

Due to the regularization term in our objective function, adding a rule that does little to improve accuracy will harm 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 consider adding only those rules that capture sufficiently many points correctly to overcome the regularization penalty. By definition, rules that do not capture enough of the remaining points cannot capture them correctly, so this provides us with two closely related bounds. As our rule lists grow, many rules do not capture enough points that have not 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 that any remaining points with a minority label will be captured incorrectly. This gives us much tighter lower bounds and, in practice, allows us to prune more efficiently.

In our Computer dataset, for instance, we might have 3 people who are in college and form an equivalence class. Now, 2 of those people use Macs and will be correctly classified by Rule 1. However, we will never correctly classify that 3rd person because any rule will always predict Mac to maximizes its accuracy. So, our lower bound should take into account the fact that we will always mispredict that person.

## 3.5 Curiosity

There are a number of different ways to explore the search space (see 4.3). 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 ordering by lower

bound, focus purely on exploiting the best prefixes that we have seen. We define a new metric, *curiosity*, that combines exploration and exploitation. Lower values of curiosity mark prefixes that we explore first. 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).

$$curiosity(RL) = (lowerBound(RL) + \lambda * len(RL)) * (nsamples/|captured(RL)|)$$

We can see that the curiosity of Rule List 1 is $(.07 + 0.01 * 2) * (20/8) = 0.225$, while the curiosity of Rule List 2 is $(.22 + .01 * 2) * (20/8) = 0.6$. Thus, curiosity prioritizes extending the more promising rule list.

### 3.6 Remaining Search Space

One metric for tracking the efficacy of our optimizations will be observing how quickly we reduce the remaining search space. We start with a combinatorially large search space, but quickly reduce it using our bounds. We calculate the remaining search space by determining how much each prefix could be expanded. We do this for every prefix we have evaluated and not eliminated. Due to our regularization term, we are able to bound the maximum length of any optimal

rule list as our best objective gets updated. This upper bound on the maximum length of an optimal rule list allows us to place an upper bound on the remaining search space as well. The search space of our problem can be visualized as a tree with fanout $n - depth$. From the root, there are $n$ rules we can add, and then we can add $n - 1$ rules to each of those rules, and so forth. However, as the best objective we have seen gets better, there are more paths of the tree that we can eliminate and so our search tree grows smaller.

## 3.7 Datasets

All of our datasets are split into a training set and a test dataset to test for accuracy. We then divide the training and test sets into 10 folds for cross validation. For our analysis that does not involve accuracy calculations, we choose a single fold on which to run and get our performance numbers.

The COMPAS dataset is a list of criminal offenders and information about them and their records. The classification problem that we're trying to solve is whether or not a given offender will commit another crime within 2 year of being arrested. As mentioned in the introduction, this problem was previously solved through the use of a black-box model because the authors of that model said that interpretability would harm accuracy. However, this black-box model

has been accused of racially biased predictions[19]. We explore this dataset with the goal of providing an interpretable, non-biased model that has accuracy comparable to state of the art black-box models.

COMPAS has 7214 individuals, so our fold has 6489 data points. 2947 (45.1%) of these individuals are labeled "yes", meaning they have committed a second crime after being arrested—while the other 3542 (54.9%) individuals are labeled "no". Analysis of COMPAS scores by the creator of the score showed an AUC of .68 for predicting any offense on a smaller dataset of 2300 individuals[8]. On this particular dataset, COMPAS scores achieve 61% accuracy for predicting recidivism. In addition, the creator of the COMPAS score says that it's difficult to get good accuracy without using racially influenced features:

> "...it is difficult to construct a score that doesn't include items that can be correlated with race — such as poverty, joblessness and social marginalization. 'If those are omitted from your risk assessment, accuracy goes down,' he said."

[19]. We are able to extract 155 rules from the dataset, including rules that involve the race of the individual.

The Stop and Frisk dataset is a list of stops made by the New York City Police Department (NYPD) that contains information about the outcome of that stop: whether an individual was frisked, searched, or found carrying a weapon.

The two classification problems that we are trying to solve are whether someone is carrying a weapon when stopped (which we refer to from now on as Weapon) and whether someone is frisked (referred to as Frisk). Stop and Frisk has been a controversial program and recent work has alleged that these stops are racially motivated[36]. Goel, Rao, and Shroff suggest that police officers cannot evaluate a complex statistical model when deciding whether or not to make a stop and that a simple, interpretable heuristic model would be ideal. We hope to provide a model that is short and easy to remember, but also has high predictive accuracy when it comes to finding weapons or other contraband.

This dataset is composed of 45787 individuals of whom 3.3% are carrying a weapon and 66% of whom are frisked. For Weapon, we are able to extract 46 rules, while Frisk yielded 64 rules.

# 4

# Implementation

This chapter lays out the design and implementation of the system used to generate optimal rule lists. We begin by providing an overview of our algorithm. Next, we explore the details of each of the three main data structures used to run our algorithm–a prefix trie, a symmetry-aware map, and a queue.

We conclude with a more thorough walkthrough of the implementation of our main algorithm, including a discussion of our garbage collection and memory tracking.

## 4.1 Algorithm overview

Algorithm 1 details a pseudocode version of our algorithm, which is based on the a basic branch-and-bound algorithm[18]. First, we perform the branching step where we choose a new branch to explore on our search tree by selecting the next rule list to evaluate. Next, we evaluate the objective and bounds on this rule list and prune it if possible, allowing us to prune some rule lists and not examine the entire search space. Finally, we update the optimal rule list if this rule list is better than the best rule list we have seen—this allows us to perform the bounding step more efficiently. We continue with these 3 steps until we have no more prefixes to evaluate. At the end of execution, we return the optimal rule list, which is the rule list with the best objective.

## 4.2 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 represents a rule—see Fig 4.1 for our class definition of a node–and contains the id and prediction for the rule

**Algorithm 1** Branch-and-bound Algorithm for Rule Lists

**Input:** A data set and a binary classification problem.

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

```
Initialize data structures
while rule lists to examine do
    Get next rule list
    Evaluate rule list bounds, discard if possible
    if rule list is better than previous best then
        Update best rule list
        Garbage collection
return Best rule list, best objective
```

that the node represents. Each leaf node in the trie contains the metadata associated with that corresponding rule list. This metadata includes the following bookkeeping information: viable child rule lists, lower bound, objective, minority misclassification, length, number captured, prediction for default rule, and whether or not this node should be pruned. 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 Section 3.5. Curiosity is one of several ways we use to order our search space. 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 Section 5.2.2).

```
class Node {
  protected:
    std::map<unsigned short, Node*> children_;
    Node* parent_;
    double lower_bound_;
    double objective_;
    double minority_;
    size_t depth_;
    size_t num_captured_;
    unsigned short id_;
    bool prediction_;
    bool default_prediction_;
    bool done_;
    bool deleted_;
};
```

**Figure 4.1:** Our custom C++ class defining a node in the prefix trie.

## 4.3 Queue

The queue is a worklist that orders our exploration over the search space of possible rule lists. We implement a number of different scheduling schemes including a stochastic exploration process, BFS, DFS, and priority metrics of curiosity, objective, or lower bound. Our queue contains pointers to leaves in the trie to leverage incremental computation. The search process involves selecting which leaf node to explore. The stochastic exploration process bypasses the use of a

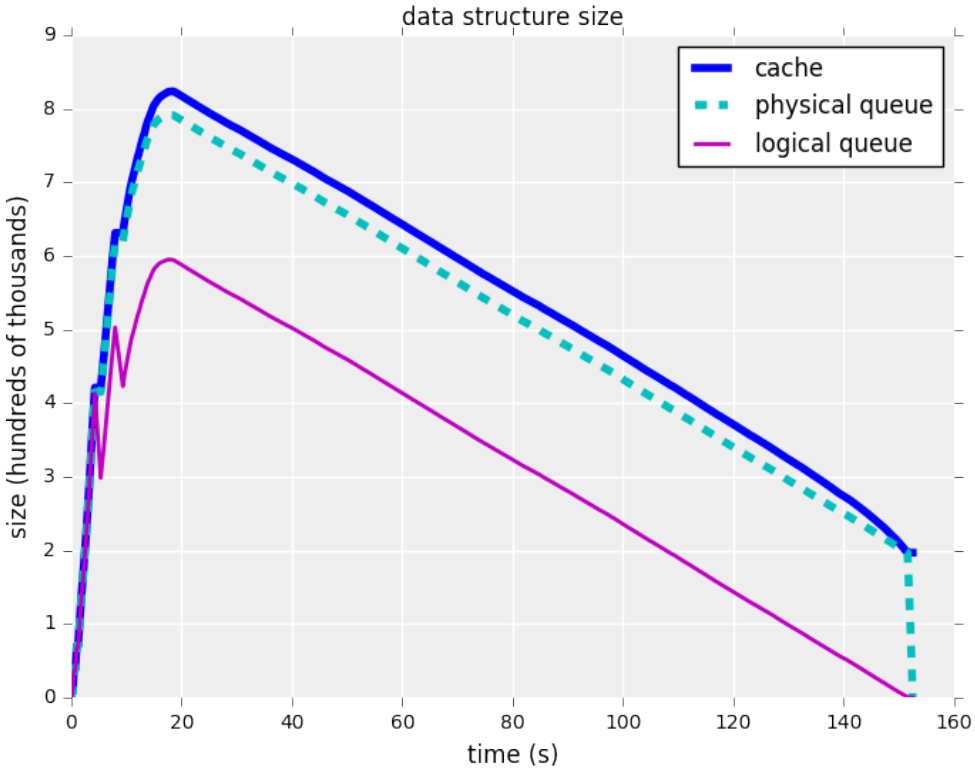



**Figure 4.2:** Size of prefix trie, logical queue, and physical queue over the execution of our algorithm on COMPAS data set. This dataset has 155 rules and 6489 samples (see Section 3.7). The gap between the logical queue and the physical queue is due to our lazy garbage collection

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, or the lower bound. We find that using an exploitation strategy, such as ordering by curiosity or lower bound, usually leads 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, as we have no way to update the queue without iterating over every element. We address this by lazily marking nodes as deleted in the prefix trie without deleting the physical node until it has been removed from the queue. As Fig 4.2 shows, this leads to a situation where our logical queue is actually smaller than the physical queue.

## 4.4 Symmetry-aware map

We implement our symmetry-aware map as an STL unordered_map, to apply the permutation bound described in Section 3.4.3. 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, called PrefixKeys. The second version has keys that represent the captured vector, which we call CapturedKeys. Our permutation bound is based on the fact

that different permutations capture the same data, so both types of key is equivalent for the rule lists (1, 4) and (4, 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. Indeed, in Fig 4.3, we find that this occurs in only a few cases, so the size of the map is mostly consistent across different key types. As can be seen, CapturedKeys are much more memory intensive than PrefixKeys because they're storing all of the data points instead of just a few rules, so we used PrefixKeys for our later analyses. Despite supporting only one bound, the permutation map plays a significant role in our memory usage.

## 4.5 Incremental execution

Algorithm 2 shows the macrostructure of our algorithm. It is very similar to the general structure provided in Algorithm 1, but it replaces the pseudocode with our specific data structures. Our program terminates when all leaves of the trie have been explored and there is nothing else in our queue. We can also opt for early termination based on the number of leaves in the trie. This prevents us from achieving the certificate of optimality, but can still lead to us finding a good or even optimal rule list. Most executions find the optimal rule list

**Algorithm 2** CORELS

**Input:** A data set and a binary classification problem.

**Output:** Outputs the best rule list and its objective.

```
opt ← NULL
T ← initializeTree()
Q ← queue( [T.root() ])
P ← map({})
while Q not empty or niter == termination do
    node ← Q.pop( )
    newObj ← incremental(node, T, Q, P)                ▷ Defined below
    if newObj < T.minObjective( ) then
        opt ← T.bestRuleList( )
        T.garbageCollect( )
return opt, T.minObjective( )
```

quickly, but then certification requires a long time and a large amount of memory. Early termination allows us to trade the guarantee of optimality for lower time and memory costs. While there are still leaves of the trie to be explored, we use our scheduling policy to select the next prefix to evaluate. We pass that prefix to our incremental function, detailed in Algorithm 3. For every rule that is not already in this prefix, i.e. rules that we could potentially add to the list, we calculate whether adding that rule to that prefix would create a viable rule list. First, we compute how many points the new rule would capture and how many of those it would capture correctly so that we can prune this rule list if it does not capture enough samples correctly. Then, we calculate the lower bound

and prune this rule list if the lower bound is larger than the best objective we have seen. Next, we determine the rule list's objective and update our current best objective if necessary. If our lower bound is large enough that adding any additional rule will cause the lower bound to be larger than our current best objective, we apply the lookahead bound and do not add the rule list to any of our data structures. Finally, we try to add the rule list to our various data structures. The symmetry-aware map will take care of the permutation bound, so we only insert the rule list into the prefix trie and queue if it successfully passes the permutation bound. Otherwise, we do not insert it into any of our data structures, and we continue to the next potential rule.After evaluating all the rules we could add to the current rule list, we return to the main loop and examine the next element in the queue.

## 4.6 Garbage Collection

Every time we update the minimum objective, we garbage collect the trie, by traversing from the root to 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 updated that often, so garbage collection is triggered only rarely.

## 4.7 Memory tracking

Throughout our development of CORELS, we often ran into the problem that larger datasets would run us out of memory. Therefore, we wanted to institute data structure optimizations that would reduce the memory burden of our algorithm. In order to achieve this goal, we needed to implement a way of tracking how much memory our program was using and where it was being used. We track memory through the use of C++11 custom allocators. Our allocators are simple malloc wrappers that 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 limited run of the bcancer data set, Valgrind's tool Massif reported that we used 98.7 MB, while our data structure tracking recorded 98.5 MB, a difference of 0.2% which is easily explained by miscellaneous heap allo-

cations. The heap profiling of Valgrind has a large overhead, while our memory logging has minimal overhead. This allows us to output our memory usage per data structure on a regular basis without incurring the large overhead of Valgrind.

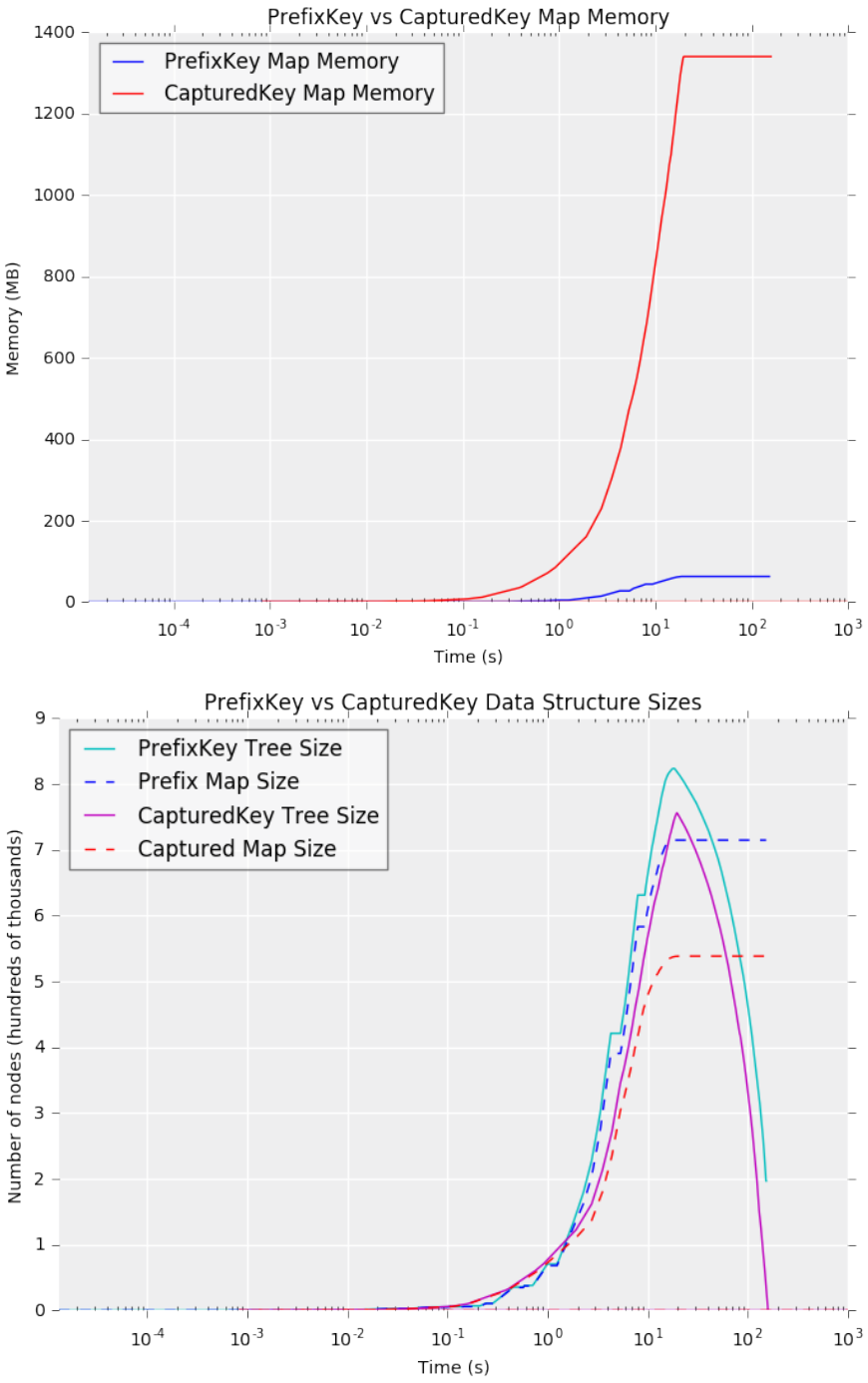



**Figure 4.3:** This graph shows the memory usage of the map when using CapturedKeys and PrefixKeys. We see that CapturedKeys provide minimal space savings over PrefixKeys due to the slight decrease in nodes inserted into the trie.

**Algorithm 3** 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( )
$t \leftarrow c \cdot nsamples$ ▷ This forms the threshold for our support bounds
**for** $rule \notin prefix$ **do**
  $rlist \leftarrow prefix$.append($rule$)
  $cap \leftarrow parent$.notCaptured( ) $\wedge$ $rule$.captured( )
  **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** $corr < t$ **then** ▷ Minimum correct support bound
    continue
  $lb = parent$.bound( ) $-$ $parent$.minority( ) ▷ Lower bound is equal to parent's lower bound
    $+\frac{|cap| - corr}{nsamples} + c$ ▷ plus the number of mistakes made by the new rule.
  **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 based on lower bound and default rule
  **if** $obj < T$.minObjective( ) **then** ▷ Update minimum objective
    $T.minObjective(obj)$
  **if** $lb + c >= T$.minObjective( ) **then** ▷ Lookahead bound
    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($n$)

# 5

# Experiments

All of the following experiments were executed on a commodity laptop, a MacBook Air with a 1.4 GHz Intel Core i5 processor and 8GB of RAM. The lowend nature of our platform illustrates the potential ubiquity of using discrete optimization techniques.

On the COMPAS dataset, where $n = 155$, naively evaluating all prefixes of up

to 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 on the reduction since any brute force solution would have to examine prefixes of longer than length 5 to certify optimality. It takes us about 2$\mu$s to evaluate a single prefix. Thus, a naive solution would take 168,764s—about 2 days—while CORELS takes only 2 minutes. While the naive solution will take a long time but eventually complete in this case, it is clear that brute force 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] So, even with our algorithmic and data structural improvements, CORELS would have taken over 120,000,000s in 1993—an unreasonable amount of time. This explains why there has been a dearth of work on algorithms focused on optimality in the past. When we combine our algorithmic improvements with the increased processor speeds, though, we see that our algorithm runs almost 1 billion times faster than a naive implementation would have in 1993.

We first examine how much each of our optimizations helps. We show that without our algorithmic and data structure improvements, it would be intractable

**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 5.1:** The optimal rule list that predicts two-year recidivism for a fold of the COMPAS dataset, found by CORELS.

**if** $stop - reason = suspicious - object$ **then predict** $yes$
**else if** $location = transit - authority$ **then predict** $yes$
**else if** $stop - reason = suspicious - bulge$ **then predict** $yes$
**else predict** $no$

**Figure 5.2:** The optimal rule list that predicts whether or not someone who is stopped by the NYPD has a weapon for one fold of the Stop and Frisk dataset, found by CORELS.

to solve a real-world problem. We also show that even with our improvements, this algorithm would not have been able to be run just 25 years ago. Next, we examine the improvements made to our symmetry-aware map to reduce the memory overhead on our algorithm. Finally, we explore a parallel implementation.

## 5.1 Accuracy

We can see from Fig **??** that CORELS performs just as well as any other method on the COMPAS and Weapon datasets. We can see that CORELS is not much better than other rule list methods, so these other methods are fairly close to optimal. However, if we look at Fig 5.1 and Fig **??** we can see that neithor of the optimal rule lists for COMPAS and Weapon include race related rules. Both

datasets contained rules pertaining to race, which means that those rules were not predictive enough to be in the optimal rule list. This presents a reasonable heuristic for judges and police officers to evaluate these problems without racially biased reasoning.

## 5.2 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 ones were the most helpful. In addition, we wanted to find out how much of an overall speed-up our work has given us. 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.

### 5.2.1 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,

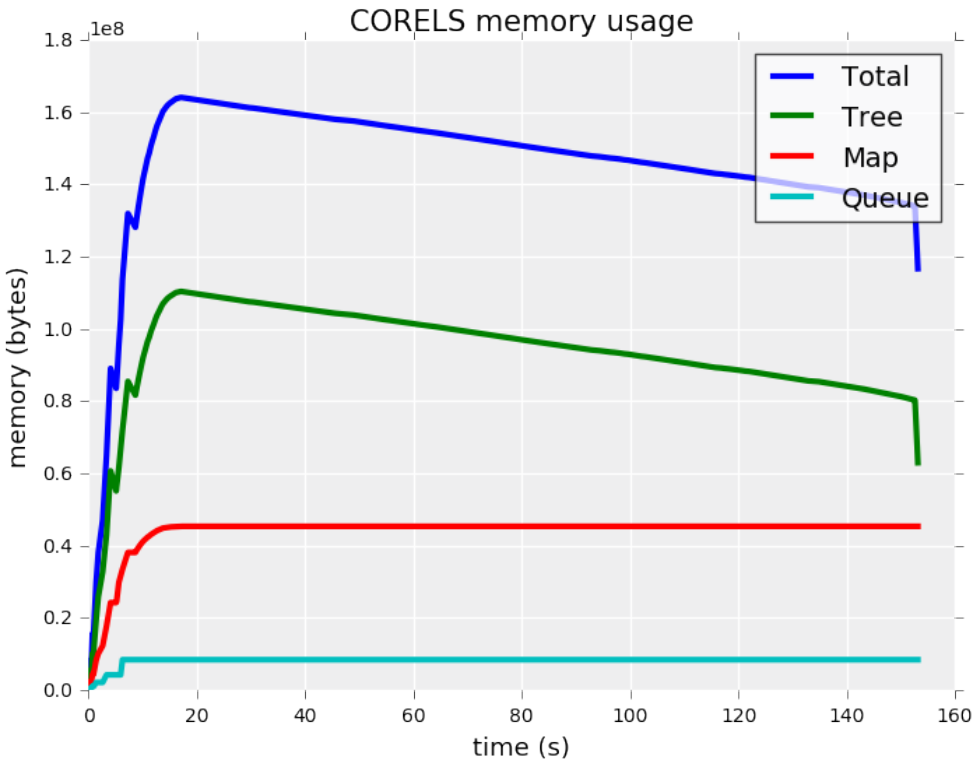



**Figure 5.3:** Memory usage of full CORELS system. Our prefix tree accounts for the bulk of the memory usage

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, and queue.

### 5.2.2 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 al-

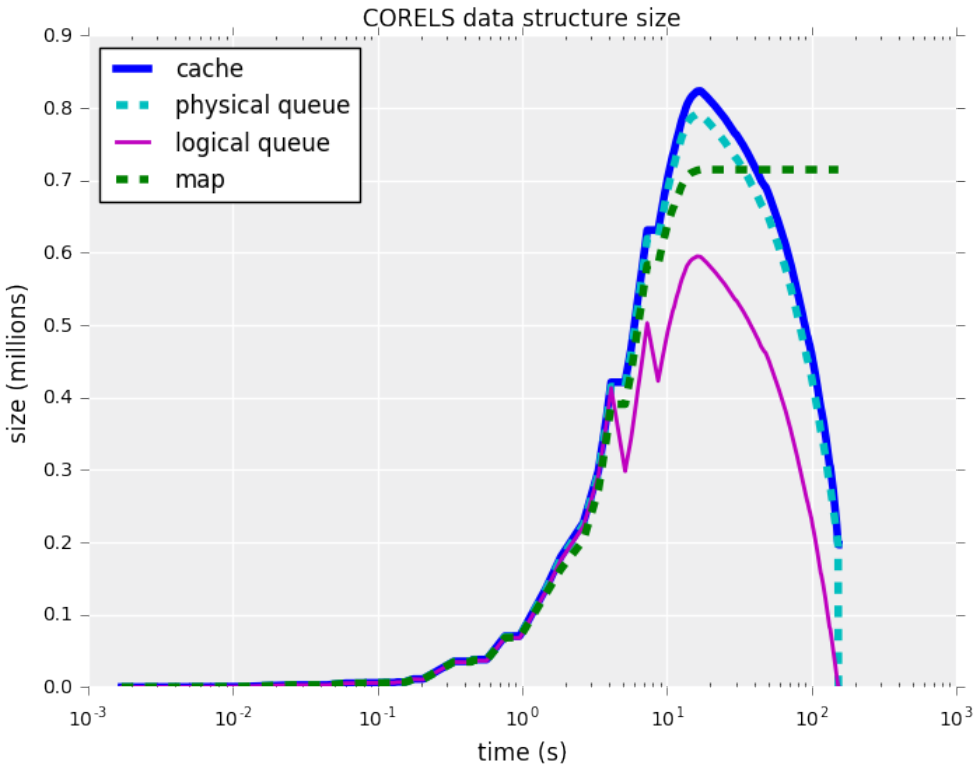



**Figure 5.4:** The size (in number of items) of each of our main data structures. Corresponds to the amount of memory used, as shown above.

lows 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 can lead to a large speed-up.

### 5.2.3 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 com-

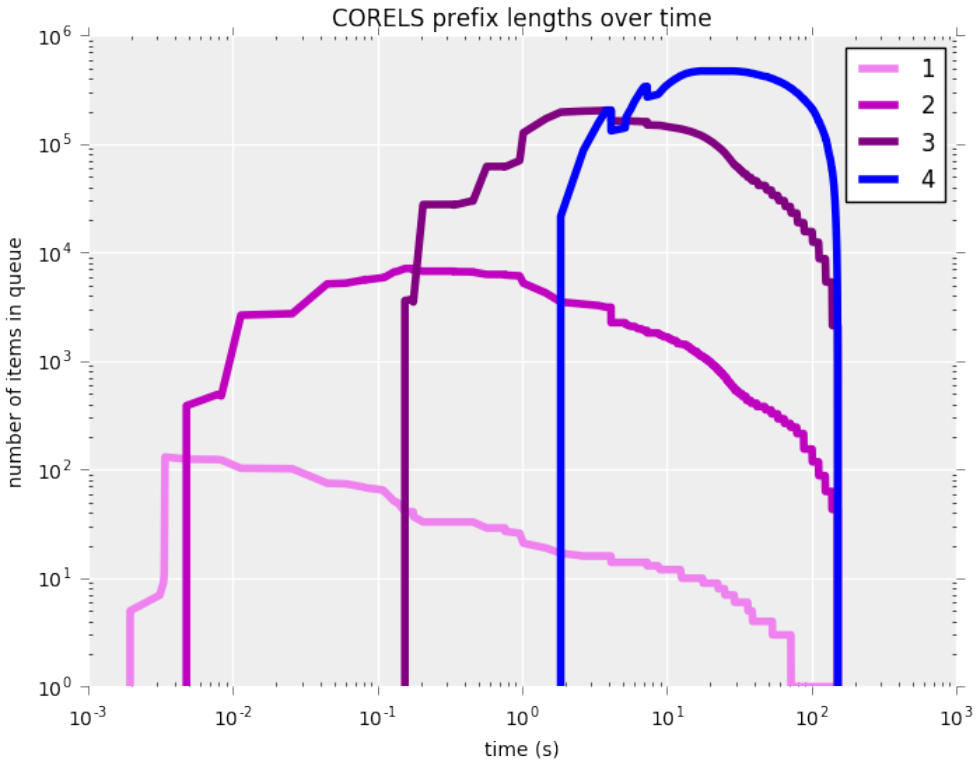



**Figure 5.5:** Tracks the number of prefixes of a given length active in the queue for the full CORELS system

putation belies the fact that these bounds prevent the pursuit of useless rules. Without these bounds we complete the certification in 261s.

### 5.2.4 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 pursue only one prefix out of all of its permutations. As our prefixes grow to length 4 and beyond, that means we can

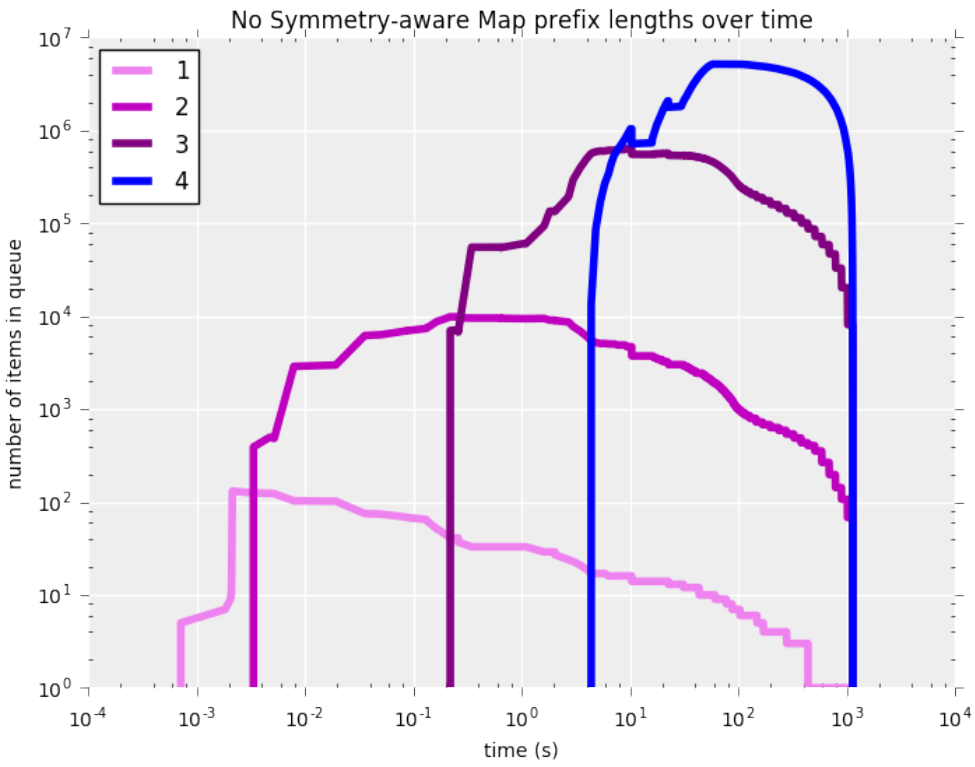



**Figure 5.6:** Tracks the number of prefixes of a given length active in the queue for CORELS without a symmetry-aware map

eliminate at least 23 prefixes. This is an important optimization and removing it takes a long time to complete the certification—1147s.

### 5.2.5 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 exam-

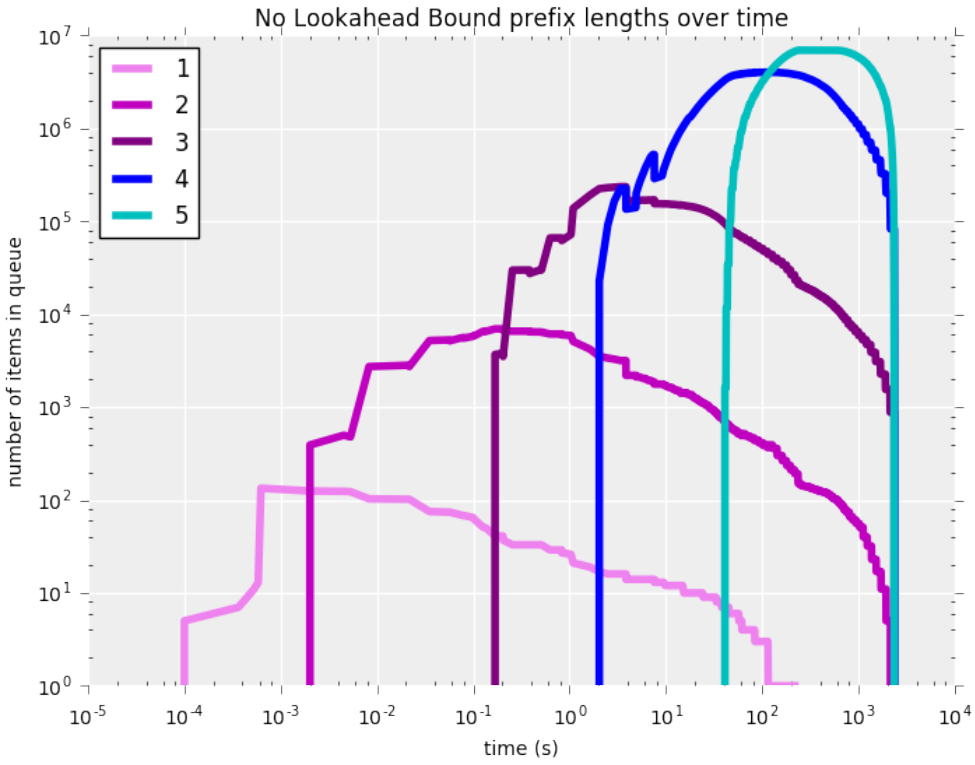



**Figure 5.7:** Tracks the number of prefixes of a given length active in the queue without our lookahead bound

ine 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.

### 5.2.6 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 prob-

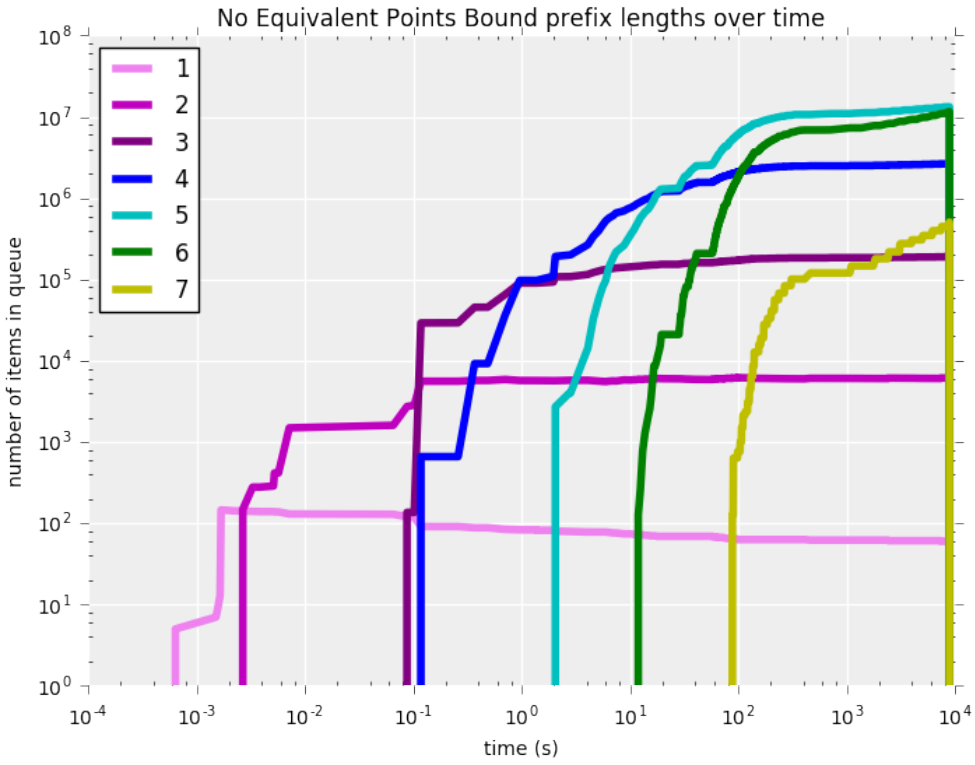



**Figure 5.8:** Tracks the number of prefixes of a given length active in the queue without the equivalent points bound

lems. We stop execution after multiple hours (8500 s) and show that there is still a large portion of the search space to be explored.

## 5.3 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. In early versions of this algorithm, the symmetry-aware map was the most memory-intensive data struc-

| Removed component | $t_{\text{total}}$ (s) | $t_{\text{opt}}$ (s) | $i_{\text{total}}$ ($\times 10^6$) | $Q_{\max}$ ($\times 10^6$) | $K_{\max}$ | $t/t_{\text{CORELS}}$ |
|---|---|---|---|---|---|---|
| none (CORELS) | 121 | 7.3 | 0.83 | 0.59 | 4 | 1 |
| 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 | $\geq$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

ture and we would often run out of memory before certifying the optimal rule list. Section 5.2.4 shows us that running without a symmetry-aware map leads to a much longer runtime, so we needed to find a way to use the symmetry-aware map while reducing memory usage. In the course of a normal execution, 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 excess memory usage and ran us out of memory. This section explores some of the techniques we used to address this memory bloat.

Our first instantiation of the symmetry-aware map was an STL map with keys of type STL sets and values of type a pair of an STL vector 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 chil-

dren. 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 it has much less overhead than the two pointers per node of overhead of the STL map.

Our original scheme also used the 8 byte size_t to represent rule ids. By assuming that we'll never have more than 65000 rules—which would overtax our algorithm anyways—we were able to use 2 byte unsigned shorts for our rule ids. This means that our key type, value type, and our node type were able to use only 2 bytes per rule id instead of 8 bytes. Since our prefixes often get to be length 5 or greater, saving 6 bytes per rule in every representation of the prefix is a large space saving.

After these initial optimizations, we worked on optimizing our key type. The only criteria for our key type is that two different prefixes that are permutations of each other should map to the same key. We began by using a STL set as the key because it allowed us to easily determine if two prefixes were permutations of each other by simply inserting into the set. However, this carries a lot of overhead because a set needs to support insertions, lookups, and deletions in $O(\log n)$ time. This overhead varies between compilers, but my machine allocated 32 bytes in total for each 2 byte item inserted into the set, plus 24 bytes

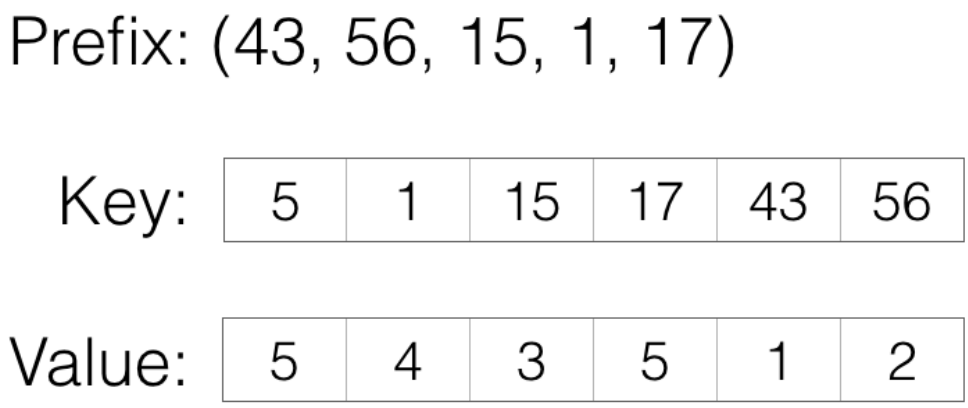



**Figure 5.9:** An example prefix (43, 56, 15, 1, 17) and the associated representations in the key and the value types. The key is an array of 6 unsigned shorts where the first entry demarcates the length of the prefix. Each the rest of the array are the rule ids of the prefix in canonical order. The value is an array of 6 unsigned chars where the first entry again represents the length of the prefix. The remaining entries in the array map the rule ids in the canonical order to their actual order in the prefix. For instance, the first entry in the key is 1 and the first entry in the value is 4, so rule 1 is the 4th rule in the prefix.

for the set itself. For a prefix of length 5, the corresponding key therefore takes up 160 bytes. Sets support far more operations than we need, though, so we transitioned to a sorted STL vector. This still allows us to compare prefixes as permutations, and it reduced overhead because a STL vector is essentially just a dynamically resizing array with a little bit of extra bookkeeping information. This translates to a prefix of length 5 taking up only 40 bytes. However, STL vectors still support a broader range of operations than we needed because they need to know when to allocate more space. All we needed was some way to compare the canonical orders and determine if they were the same. We created a custom key class by allocating a chunk of memory that held the length of the

prefix and the sorted order of the ids. This kept the same idea that the STL vector had in order to compare two prefixes, but it removed the overhead. In the end, we use only 12 bytes for the key for a prefix of length 5. An example of the custom key class is shown in Fig 5.9.

We went through a similar process with our values for the symmetry-aware map. Our only requirement for the values were that they keep track of which prefix was the current best for the given permutation. We began with a STL vector to keep track of the actual order of the rules in the prefix, but 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. We reduced our memory usage through a similar technique of what we did with the key type: we moved from a STL vector to a chunk of memory. However, we realized that we could do even better because we already had the rule ids—all we need in the value was the ordering of those rules. Since we don't need to represent the rule ids, we can use unsigned chars instead of unsigned shorts to record the actual ordering of the rule ids stored in the key type. This is shown in Fig 5.9

In total, Table 5.2 shows us that these improvements give us a 3x memory reduction on this problem. These reductions take the symmetry-aware map from being the largest data structure to being smaller than the prefix trie. Reducing

| Map version | Key version | Value version | Total Memory (MB |
|---|---|---|---|
| Map | Set (size_t) | STL Vector (size_t) | 190.7 |
| Unordered Map | Set (size_t) | STL Vector (size_t) | 185.9 |
| Unordered Map | Set (unsigned short) | STL Vector (unsigned short) | 148.6 |
| Unordered Map | STL Vector (unsigned short) | STL Vector (unsigned short) | 68.7 |
| Unordered Map | Custom Key | Custom Value | 62.9 |

**Table 5.2:** Symmetry aware map improvements when running on the COMPAS dataset. 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 each optimization reduced the total memory consumption while maintaining or slightly reducing the overall runtime.

the memory footprint of CORELS was helpful for running on COMPAS, but on larger problems this memory improvement is even more pronounced.

In Section 4.4 we described two different key types for the symmetry-aware map, but all of the above optimizations pertained only to the PrefixKey type. Dealing with the CapturedKey type required a different approach. We used the bit vector manipulation library from Yang et al.[38] which was built on top of the GMP library. These bit vectors are of the GMP defined type mpz_t, which stands for multiple precision integer. This type allows us represent and perform operations with very large bit vectors in an efficient manner. Since unordered_map is implemented as a hash table, it requires a custom hash for non built-in types. We initially wrote a function to convert between mpz_t and a $std :: vector < bool >$ so that we could use STL's built-in hash function. This conversion turned out to be very slow, especially when we were running on data

sets with many samples. These mpz_t types were fairly large, so copying the information from one place to another was inefficient and impractical. Instead, we adapted the sdbm hash function to our mpz_t type.[Yigit] Once we used our own hash function and didn't have to copy between types, CapturedKeys became much faster. However, as Fig 4.3 showed, they were still less space efficient that PrefixKeys so we did not pursue them further.

## 5.4 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, templates can lead to a much larger executable. Indeed, with a pure templating system, our executable was 253,732 bytes. Execution time took about 126s on the COMPAS data set.

An alternative way to maintain modularity was to take advantage of C++'s class system and use inheritance and polymorphism. 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. A vtable, or virtual method table, is how C++ deals with the fact that the compiler might not know what class a variable is at compile time, so at runtime the vtable is used to run the appropriate function. With a pure inheritance framework, our executable was 171,288 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.

## 5.5 Parallelization

Almost every modern machines supports parallelism through the use of multiple cores or simultaneous multi-threading. This is a crucial fact that modern discrete optimization programs ought to take advantage of. We show that the structure of this problem nicely supports a parallel implementation of our algorithm. Furthermore, we show that we get significant, though not linearly-

scaling, speed-ups with our parallel implementation.

Analysis of our log files showed that the majority of the time of our program is spent in the incremental section of our execution, shown in Algorithm 3. So, we began by trying to parallelize this inner loop of our incremental evaluation. This inner loop involves trying to extend a prefix by calculating the bounds for all possible rules we could add to this prefix. In order to add these child 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 where one thread is trying to insert a child rule into the parent's representation of the children and another thread is trying to read the parent's representation of the children. This leads to a segfault in the STL code and could be fixed by locking the appropriate fields in the parent. Locking the parent has too much contention, however, because all of our threads needed to own the parent lock at the same time—essentially rendering the loop sequential.

We realized, however, that the tree structure of our search space, encapsulated in 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 con-

tention on parents since only one thread can access a node at a time. The shared state only consists of the minimum objective and the symmetry-aware map, and these are kept in the master thread. No locking is necessary on the minimum objective because race conditions don't lead to incorrect execution, just to slightly less efficient execution. For the symmetry-aware map, we lock on insertions and lookups because insertions can trigger a rehash of the underlying hashtable, which is problematic if multiple threads are accessing the map. 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. One theory we had was that since we had to add locking to the symmetry-aware map, there might be contention that is slowing down the multi-threaded implementation. However, when we preallocate a large symmetry-aware map and don't lock, we see the same type of slowdown, implying that contention is not the issue.

# 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. Discrete optimization has become less popular as other techniques such as convex optimization dominate the machine learning landscape. However, we showed that due to dramatic increases in processor speed and computer mem-

ory, discrete optimization techniques can be applied to real problems and complete them in a reasonable amount of time. In particular, we hope that this work will inspire further work on discrete optimization techniques for other methods such as decision trees or SVMs.

We also showed that memory reduction

Finally, this work parallelized

Further optimizations involving distance sensitive hashing and bit-packing could reduce runtime and memory usage even further.

Some of the computations in this paper were run on the Odyssey cluster supported by the FAS Division of Science, Research Computing Group at Harvard University.

# References


[Sup] Top500 supercomputer sites, directory page for top500 lists. Accessed: 3/18/2017.

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

[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] Bellazzi, R. & Zupan, B. (2008). Predictive data mining in clinical medicine: Current issues and guidelines. *International Journal of Medical Informatics*, 77(2), 81 – 97.

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

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

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

[8] Brennan, T., Dieterich, W., & Ehret, B. (2009). Evaluating the predictive validity of the compas risk and needs assessment system. *Criminal Justice and Behavior*, 36(1), 21–40.

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

[10] Dimitris Bertsimas, A. K. & Mazumder, R. (2016). Best subset selection via a modern optimization lens. *The Annals of Statistics*.

[11] Eiter, T., Ibaraki, T., & Makino, K. (2002). Decision lists and related boolean functions. *Theor. Comput. Sci.*, 270(1-2), 493–524.

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

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

[14] Garofalakis, M., Hyun, D., Rastogi, R., & Shim, K. (2000). Efficient algorithms for constructing decision trees with constraints. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00 (pp. 335–339). New York, NY, USA: ACM.

[15] Hyafil, L. & Rivest, R. L. (1976). Constructing optimal binary decision trees is np complete. *Information Processing Letters*.

[16] Klivans, A. R. & Servedio, R. A. (2006). Toward attribute efficient learning of decision lists and parities. *JMLR*.

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

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

[19] Larson, J., Mattu, S., Kirchner, L., & Angwin, J. (2016). How we analyzed the compas recidivism algorithm. *ProPublica*. Accessed 20 December 2016.

[20] 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.

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

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

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

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

[30] Nock, R. & Gascuel, O. (1995). On learning decision committees. In *Proceedings of ICML-95, International Conference on Machine Learning* (pp. 413–420).

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

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

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

[34] Rüping, S. (2006). *Learning interpretable models*. PhD thesis, Universitat Dortmund am Fachbereich Informatik.

[35] Segal, R. & Etzioni, O. (1994). Learning decision lists using homogeneous rules. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence* (pp. 619–625).: AAAI.

[36] Sharad Goel, J. M. R. & Shroff, R. (2016). Precinct or prejudice? understanding racila disparities in new york city's stop-and-frisk policy. *The Annals of Applied Statistics*.

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

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

[Yigit] Yigit, O. Hash functions. Accesssed: 2/5/2017.

[40] 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.

# A

# Proof of Bounds

The following bounds and proofs are adapted from the joint work upon which this thesis is based[2].