THE RECTANGULAR MAXIMUM AGREEMENT PROBLEM: APPLICATIONS AND PARALLEL SOLUTION

by

AI KAGAWA

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Operations Research

Written under the direction of

Professor Jonathan Eckstein

And approved by

____

____

____

____

____

____

New Brunswick, New Jersey

OCTOBER, 2018

ABSTRACT OF THE DISSERTATION

The Rectangular Maximum Agreement Problem: Applications and Parallel Solution

By Ai Kagawa

Dissertation Director:

Professor Jonathan Eckstein

A $\mathcal{NP}$-hard rectangular maximum agreement (RMA) problem finds a "box" that best discriminates between two weighted datasets. We respectively describe a specialized parallel branch-and-bound method and a greedy heuristic to solve RMA exactly or approximately. Our computational results show that a new parallel branch-and-bound method can solve larger RMA problems exactly in less time than a previously implemented parallel branch-and-bound procedure and Gurobi, a mixed integer programming (MIP) solver.

We describe two applications of RMA: a LPBoost-based two-class classification and a rule-enhanced penalized regression. The first classification application constructs a stronger classifier from a set of weighted voting classifiers by maximizing the margin between the two observation classes and penalizing classification errors. The weighted voting classifiers are multidimensional "box"-based rules. The second regression application formulates a L_1-penalized regression model using multidimensional "box"-based rules as additional explanatory variables.

Due to exponentially large number of possible multidimensional rules, they are dynamically generated in both applications. In contrast to prior approaches to solve these problems, we draw heavily on standard (but non-polynomial-time) mathematical programming techniques, enhanced by parallel computing. Our rule-adding procedure is based on the classical column generation method for high-dimensional linear programming. The pricing problem for our column generation procedure reduces to the RMA problem, and it is solved exactly or approximately. This method resembles boosting in machine learning. Furthermore, we propose a discretization method before solving RMA. It reduces the level of difficulty for RMA while still maintaining prediction or classification accuracy in the applications. The prediction accuracy of our models are tested by cross-validation. The resulting classification and regression methods are computation-intensive, but our computational tests suggest that they outperform prior methods at making accurate and stable predictions.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Jonathan Eckstein, for the continuous support of my Ph.D. study and research for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. Without his support and encouragement, this dissertation could not be possible to finish. I could not have imagined having a better advisor and mentor for my Ph.D. study.

Besides my advisor, I would like to thank Prof. Noam Goldberg for his guidance in machine learning. I would like to thank the rest of my thesis committee, Prof. Endre Boros, Prof. Adi Ben-Israel, and Prof. Myong K. Jeong for participating my dissertation.

I thank for my funding of teaching assistantship and instructorship from Rutgers University to complete my Ph.D.

I appreciate my mother for her unconditional love and endless support and my brother for his support and friendship. I thank all my friends for their joy, laughter, humility, support, competition, and inspiration.

Dedication

To my mother.

Table of Contents

Chapter 1

Introduction and Literature Review

1.1 Rectangular Maximum Agreement Problem

This thesis concerns the rectangular maximum agreement (RMA) problem and machine learning. The goal of machine learning is understanding existing patterns in data and predicting outcomes from given input values. We will apply RMA to two problems in machine learning, classification and regression. The RMA problem is based on two types of data samples, one marked positive and the other negative. For example, the positive group might consist of people who bought a specific product online, and the negative group is those who viewed the same product online but did not buy it. Each sample observation contains the classification along with a possibly large number of numerical attributes including categorical information (e.g. gender and ethnicity) and/or numerical values (e.g. age and height). The goal is to use the samples to devise a scheme that will classify new, previously-unseen data points as either positive or negative as accurately as possible. Using the example sample, the classification determines the characteristic of people who bought the product and who did not.

RMA is a $\mathcal{NP}$-hard problem which arises naturally as a subproblem of the machine learning applications studied here. Traditionally, such subproblems have been approximately solved by heuristic methods. Building on earlier work by Eckstein and Goldberg [12] for the maximum monomial agreement (MMA) problem, we have developed a specialized, exact branch-and-bound method for RMA. MMA is the special case of RMA in which explanatory data are binary. MMA can be applied to both classification and regression problems through a binarization procedure, with the drawback the binarization process can be more time consuming than solving the problem itself. Using a RMA solver allows one to directly process non-binary data.

1.1.1 Branch-and-Bound

RMA is solved by a branch-and-bound (B&B) algorithm. B&B is one of the most popular methods to solve $\mathcal{NP}$-hard combinatorial optimization problems [6]. B&B finds an exact optimal solution by searching the entire feasible space using a search tree. The root node of the search tree contains the entire search space, and B&B recursively partitions the search space into smaller disjoint spaces or *subproblems.* As these partitioned subproblems are created, they are stored in a live subproblem list. In each subproblem, B&B tries to find an improved incumbent, the current best solution, and a subproblem bound which is a lower or upper bound in the case of the minimization or maximization, respectively. Any solution in the search space region specified by the subproblem can be no better than this bound. Frequently, a feasible solution is discovered by a heuristic before and during the B&B search, and it is stored as the incumbent. If a bound of a subproblem is not better than the incumbent, then the subproblem should be discarded or *fathomed* from the live subproblem list since any feasible solutions in the subproblem cannot be better than the incumbent. Therefore, using better heuristic incumbents and improved bounding functions speeds up the B&B search process by fathoming more subproblems in the live subproblem list. The B&B algorithm terminates and finds the optimal solution whenever there are no more subproblems in the live subproblem list.

Eager and *lazy* bounding approaches to B&B have a different order of bounding and separating processes. Eager bounding computes a bound as soon as each subproblem is created, and it is then stored in the live subproblem list to be split into further subproblems later. On the other hand, lazy bounding first splits a subproblem without calculating each child's bound, and each child is stored in the live subproblem list by setting its bound to the parent's bound. If the bounding process is easier than the separating process, then eager bounding is preferable; else the lazy bounding is preferable since this choice may improve the run time by processing less time consuming operations first. RMA prefers lazy bounding since its bound computation is time consuming.

Three common methods to select the next subproblem to explore are:

1. *Best-First Search* (BeFS) selects the subproblem with the optimal bound among all live subproblems.

2. *Depth-First Search* (DFS) selects the most recently generated subproblem in the deepest level of the search tree, thereby the live subproblem list as a last-in, first-out stack.

3. *Breath-First Search* (BFS) selects subproblems in shallower levels of the search tree before subproblems in the deeper levels, thereby the live subproblem list as a first-in, first out queue.

Using BFS to select the next subproblem is generally much slower than the other two methods since the size of tree grows exponentially with depth, and it is often necessary to explore deeply into some parts of the tree in order to find an optimal solution or prove its optimality. DFS has an advantage of requiring less memory storage space since it only requires storing information about the current subproblem and its ancestors. BeFS has the memory disadvantage of having to store more live subproblems than DFS. However, using BeFS minimizes the number of subproblems that have to be explored. Since the bounding computation is time consuming in the RMA solver, BeFS is often the preferable method.

1.2 Statistical Learning

Statistical learning utilizes statistical methods to construct a regression function or a classifier using given data. Suppose we have m observation vectors $X_1, \ldots, X_m \in \mathbb{R}^n$, with matching response values $y_1, \ldots, y_m \in \mathbb{Z}$ for classification and $y_1, \ldots, y_m \in \mathbb{R}$ for regression. For a two-class classification problem, each observation is classified as positive or negative, $y_i = +1$ or $y_i = -1$, respectively. The goal is to construct a classifier $\hat{f} : \mathbb{R}^n \to \{-1, +1\}$ which minimizes the number of misclassified instances for unseen samples. For regression, each response y_i is a possibly noisy evaluation of an unknown true prediction function $f : \mathbb{R}^n \to \mathbb{R}$ at X_i, that is, $y_i = f(X_i) + \epsilon_i$, where

$\epsilon_i \in \mathbb{R}$ represents the noise or measurement error. The goal is to estimate f by some $\hat{f} : \mathbb{R}^n \to \mathbb{R}$ for regression such that $\hat{f}(X_i)$ is a good fit for y_i, that is, $|\hat{f}(X_i) - y_i|$ tends to be small. The estimate $\hat{f}$ may then be used to predict the response value y corresponding to a newly encountered observation $x \in \mathbb{R}^n$ through the prediction $\hat{y} = \hat{f}(x)$.

A classical linear regression model is one simple example of the many possible techniques one might employ for constructing $\hat{f}$. The classical regression approach to this problem is to posit a particular functional form for $\hat{f}(X_i)$ (for example, an affine function of X_i) and then use an optimization procedure to estimate the parameters in this functional form. The standard linear regression is:

$$y_i = \beta_0 + \sum_{j=0}^{n} \beta_j x_{ij} + \epsilon_i, \qquad\qquad i = 1, \ldots, m, \tag{1.1}$$

where $\beta_0 \in \mathbb{R}$ is a constant term, each $\beta_j \in \mathbb{R}$ is a linear coefficient, and $\epsilon_i \in \mathbb{R}$ is a random residual for each observation.

1.2.1 Model Evaluation

The performance of classifier and predictor is generally evaluated by a loss function $L(\hat{f}(X), y)$, and also called a *fit measure* in the case of regression problem. For both classification and regression, the goal is to minimize $L(\hat{f}(X), y)$.

The following loss function, called the *hinge loss*, is common for classification problems:

$$\begin{aligned} L(f(X), y) &= \max(0, 1 - y\hat{f}(X)) & (1.2)\\ &= \left|1 - y\hat{f}(X)\right|_{+}. & (1.3) \end{aligned}$$

Here, $y_i\hat{f}(X_i)$ is 1 if X_i is correctly classified and -1 if it is incorrectly classified. Hence, this loss function returns 0 for the correct classification or 2 for the incorrect classification. The square loss function for classification is:

$$L(f(X), y) = (1 - y\hat{f}(X))^2. \tag{1.4}$$

If it is correctly classified, then the loss function is 0; else the loss function is 4.

Two common loss functions in regression are the sum of squared errors of the prediction function,

$$L(\hat{f}(X), y) = (y - \hat{f}(X))^2, \tag{1.5}$$

and the sum of absolute errors of the prediction function,

$$L(\hat{f}(X), y) = \left| y - \hat{f}(X) \right|. \tag{1.6}$$

Their averages are the mean square error (MSE) and the mean absolute error (MAE):

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{f}(X_i))^2 \tag{1.7}$$

$$MAE = \frac{1}{m} \sum_{i=1}^{m} \left| y_i - \hat{f}(X_i) \right|. \tag{1.8}$$

They both measure how well the prediction function $\hat{f}$ predicts the true response values y from the explanatory variables X.

1.2.1.1 MSE Depomposition

Assume that the error term is an i.i.d. (independent and identically distributed) random variable $\epsilon \sim N(0, \sigma_\epsilon^2)$ where σ_ϵ is its standard deviation. The square prediction error can be split into reducible and irreducible errors as:

$$E[(y - \hat{y})^2] = \underbrace{E[(f(X) - \hat{f}(X))^2]}_{\text{Reducible}} + \underbrace{\text{Var}[\epsilon],}_{\text{Irreducible}} \tag{1.9}$$

where $E[(f(X) - \hat{f}(X))^2]$ is a reducible error, and $\text{Var}[\epsilon]$ is an irreducible error [23]. A proof of the decomposition (1.9) is shown below.

Lemma 1.2.1. *The squared prediction error can be divided into the reducible error and irreducible error terms as shown in* (1.9).

Proof.

$$E[(y - \hat{y})^2] \tag{1.10}$$

$$= E[(f(X) + \epsilon - \hat{f}(X))^2] \tag{1.11}$$

$$= E[f(X)^2 + \varepsilon^2 + \hat{f}(X)^2 + 2\varepsilon f(X) - 2f(X)\hat{f}(X) - 2\epsilon\hat{f}(X)] \tag{1.12}$$

$$= E[f(X)^2 - 2f(X)\hat{f}(X) + \hat{f}(X)^2] + E[\epsilon^2] + 2E[\epsilon]E[f(X)] - 2E[\epsilon]E[\hat{f}(X)] \tag{1.13}$$

$$= E[(f(X) - \hat{f}(X))^2] + Var[\epsilon]. \tag{1.14}$$

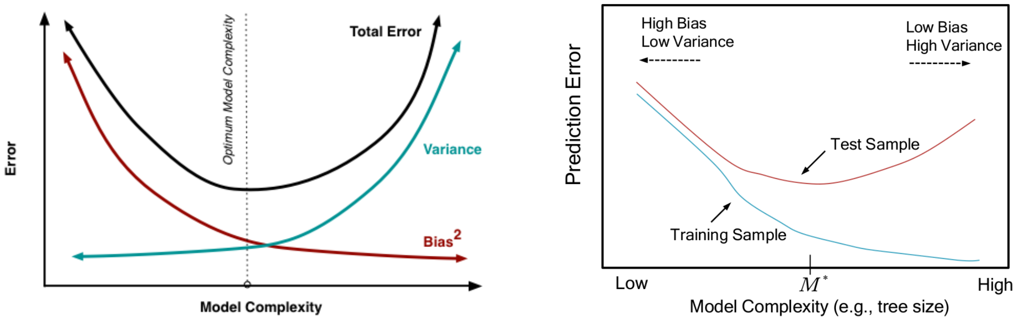
(1.11) is derived by substituting $y = f(X) + \epsilon$ and $\hat{y} = \hat{f}(X)$. By expanding the square in (1.11), (1.12) is obtained. By the linearity of expectation, (1.13) is obtained. (1.14) is obtained by substituting $E[\epsilon] = 0$ and $Var(\epsilon) = E(\epsilon^2) - E[\epsilon]^2 = E(\epsilon^2)$ since $\epsilon \sim N(0, \sigma_\epsilon^2)$. □

Since the irreducible error is the variance associated with the error term of the responses y, we cannot minimize the irreducible error and can only minimize the reducible error. The reducible error can be also split into the bias and variance of the predictor $\hat{f}(X)$. Therefore, the square prediction error is:

$$E[(y - \hat{y})^2] = (f(X) - E[\hat{f}(X)])^2 + \text{Var}[\hat{f}(X)] + \text{Var}[\epsilon] \tag{1.15}$$

$$= \text{Bias}[\hat{f}(X)]^2 + \text{Var}[\hat{f}(X)] + \text{Var}[\epsilon] \tag{1.16}$$

where $\text{Var}[\hat{f}(X)]$ is the variance and $\text{Bias}[\hat{f}(X)] = f(X) - E[\hat{f}(X)]$ is the bias of the prediction function. The proof is shown in Lemma 1.2.2. The bias is the difference between the value given by the unknown true prediction function f and the expected value derived by the prediction function $\hat{f}$. The variance measures the sensitivity of our model. A high variance model has a larger change in the response value with respect



(a) MSE decomposition [15]

(b) Prediction errors for training vs test samples [29]

Figure 1.1: Bias-Variance Tradeoff

to a small change in the input values, *vice versa*.

Lemma 1.2.2. *The reducible error can be split into the squared bias and variance of* $\hat{f}$*, as shown in* (1.16).

Proof.

$$\begin{aligned} E[(f(X)-\hat{f}(X))^2] &= E[(f(X)^2 - 2f(X)\hat{f}(X) + \hat{f}(X)^2] && (1.17) \\ &= E[f(X)]^2 - 2E[f(X)]E[\hat{f}(X)] + E[\hat{f}(X)^2] && (1.18) \\ &= f(X)^2 - 2f(X)E[\hat{f}(X)] + E[\hat{f}(X)]^2 + Var[\hat{f}(X)] && (1.19) \\ &= \underbrace{(f(x) - E[\hat{f}(X)])^2}_{Bias(\hat{f})^2} + \underbrace{Var[\hat{f}(X)]}_{Var(\hat{f})} && (1.20) \end{aligned}$$

(1.17) is obtained by expanding the square. By the linearity of expectation, (1.18) is obtained. Since $f(X)$ is the true prediction function, $E[f(X)] = f(X)$. Also, $E[\hat{f}(X)^2] = E[\hat{f}(X)]^2 + Var[\hat{f}(X)]$ since $Var[\hat{f}(X)] = E[\hat{f}(X)^2] - E[\hat{f}(X)]^2$. Hence, (1.19) is derived, and it can be formulated as the squared bias and variance of the prediction function $\hat{f}$, as shown in (1.20). □

The goal is to find a prediction function $\hat{f}$ that minimizes the sum of errors from the variance and bias of $\hat{f}$. However, there is a tradeoff between the bias and variance of the predictor, as shown in Figure 1.1(a). *Model complexity* indicates the difficulty or sensitivity of the model, and a model involving a relatively high-degree polynomial or more explanatory terms is considered more complex. In Figure 1.1(a) and 1.1(b), the horizontal axis indicates the level of model complexity, the model complexity increases from left to right.

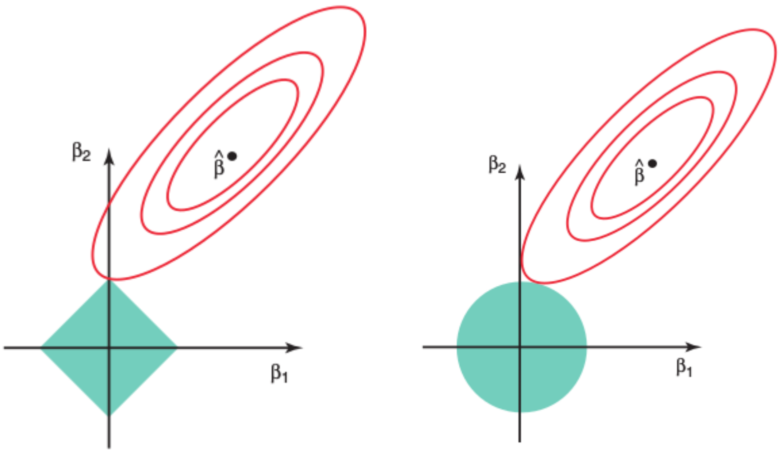
Suppose the original dataset is partitioned into testing and training datasets. A prediction model $\hat{f}$ is trained on the training dataset. A model that minimizes a given loss function over the training dataset may in some cases be over-complex and can be too specific for the given training dataset. Hence, the prediction function has poor performance on the testing dataset even though it has good performance on the training datasets. This phenomenon is called *overfitting* , and is a common problem in machine learning. Overfitting is caused by high variance and low bias of $\hat{f}$. On the other hand, *underfitting* is due to low variance and high bias of $\hat{f}$ on a testing dataset, so the model has poor performance on both the training and testing datasets. Underfitting is the left side of the optimal model complexity M^*, and overfitting is its right side in Figure 1.1(a) and 1.1(b). *TotalError* in Figure 1.1(a) indicates the prediction error on the testing samples in Figure 1.1(b). It is minimum if we choose the optimal model complexity at M^*. Therefore, we want to find the model with close to the optimal model complexity to avoid overfitting and underfitting.

1.2.2 Model Selection: Regularization and Cross-Validation

Two standard techniques to prevent overfitting are *regularization* and *cross-validation*. The regularization technique adds a regularization term $R(f)$ to the loss function, and the goal is to minimize the sum of the loss function and the regularization term, leading to the model of the form:

$$\min_f L(f(x), y) + \lambda R(f), \tag{1.21}$$

where $\lambda \geq 0$. If λ is too large, the coefficients β of the standard regression tends to be small and the model prediction $\hat{f}$ is not close to the true prediction function f even though the model has smaller sensitivity to the explanatory variables. Therefore, the



(a) Lasso penalty (b) Ridge regression penalty

Figure 1.2: Illustration of Penalties in 2D [23]

model has high bias and low variance. If λ is too small, some of the coefficients β can be too large so the model is too sensitive to the explanatory variables even thought its prediction function $\hat{f}$ is close to the true prediction function f. Therefore, choosing λ is important to minimize the prediction error on the testing dataset.

Three common regularization methods in regression are L1, L2, and a combination of L1 and L2 regularized terms of the coefficient vector β. These regularizations make β smaller or sparse. Assume that the loss function is the sum of squared prediction errors, and these three regularized models are [31]:

1. *Lasso* (L1 penalty terms of the coefficients): $R(\beta) = \sum_{j=1}^{n} |\beta_j|$

2. *Ridge Regression* or *Tikhonov Regularization* (L2 penalty terms of the coefficients): $R(\beta) = \sum_{j=1}^{n} \beta_j^2$

3. *Elastic Net* (a combination of Lasso and Ridge Regression):
   $R(\beta) = \lambda_1 \sum_{j=1}^{n} |\beta_j| + \lambda_2 \sum_{j=1}^{n} \beta_j^2$ where $\lambda_1, \lambda_2 \geq 0$.

Figure 1.2 illustrates an example of the lasso and ridge regression penalties in 2 dimensions. Red ellipses indicate the different levels of the square loss function, and the loss function is same for any β chosen on the same ellipse. A model with $\hat{\beta}$ in the center of the ellipses has the smallest square loss, and the loss increases by choosing β

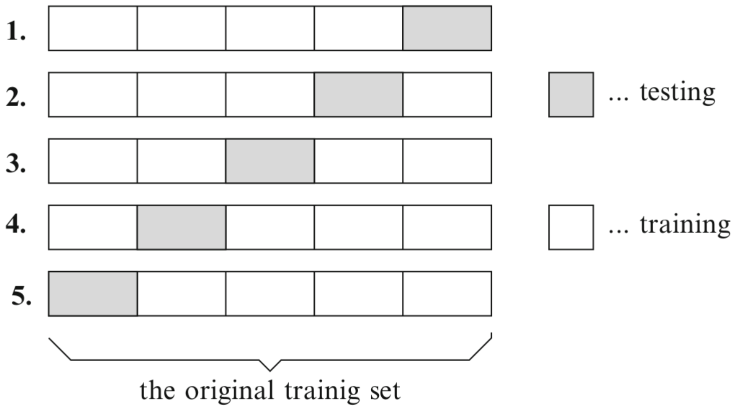


Figure 1.3: An example of 5-fold cross-validation [24]

on a ellipse far from the center. L1 and L2 regularizations constrain β to be on the intersection of the L1 and L2 constraint (within the green region) and the ellipse centered at $\hat{\beta}$. The major difference between L1 and L2 regularizations is that L1 regularization makes β sparse by setting some β_j to 0. On the other hand, L2 regularization tends to make β small but not exactly 0. Since they both construct less complex models, both may prevent overfitting. Since L1 regularization tends to give a sparser β, the L1 regularization model is easier to interpret. Therefore, we use L1 regularization for our regression application.

Cross-validation is another method to test for and prevent overfitting. First, the original dataset is randomly partitioned into N nearly equal size subsets. Initially, the last block of the partitions is a testing dataset and the rest of the partitions is a training dataset. The model is trained using the training dataset and tested using the testing dataset. Second, the $(N-1)$st partition is the testing dataset and the rest of the partitions form the training dataset. This process is repeated N times, and called N-fold cross-validation. Figure 1.3 shows an example of 5-fold cross validation. The best regularization parameter λ is often chosen using cross-validation.

Bootstrapping denotes randomly selecting testing and training datasets from the original dataset, and repeating the N times. This procedure may often select the same observations to be testing or training data unequal numbers of times, and some observations may never be selected to be testing samples. On the other hand, cross-validation gives all observations equal frequency of being testing and training data. Therefore, cross-validation is a more accurate testing procedure for overfitting since it tests a given model using all samples.

1.2.3 Ensemble Learning

For our classification and regression applications, we are interested in cases in which a concise candidate functional form for $\hat{f}$ is not readily apparent, and we wish to estimate $\hat{f}$ by searching over a very high-dimensional space of parameters. We use boosting techniques to build our models, and boosting is an ensemble learning technique. Ensemble learning means combining multiple models to obtain a better prediction or classification model. In particular, boosting is iteratively adding one or more "weak learner" models, obtained by solving a subproblem formulated using the current classification or regression errors, to the current model. Boosting is generally time consuming due to the sequential iterative procedure, but it may construct a useful model. A weighted voting classifier takes a weighted sum of the outputs of multiple constituent weak learner models, and then classifies an observation depending on whether the weighted sum exceeds some threshold.

AdaBoost (*Adoptive Boosting*) [17] by Freund and Schapire is a boosting algorithm to solve a classification problem. Adaboost iteratively finds a weak learner which minimizes the sum of misclassification errors, and constructs the combination of weighted voting classifiers after adding the weak learner to the current ensemble. The most popular way to obtain each weak learner is using a decision tree, and each observation weight is initially $1/m$. In each iteration, the weights are updated based on an exponential loss function, and then they are normalized.

LPBoost is another boosting algorithm for a classification problem. It maximizes a margin of the separating boundary between the two observation classes while decreasing the classification errors by a combination of weighted voting classifiers with the margin. The weak classifiers are generally constructed by a decision tree. Since the number of possible weak classifiers is exponentially large, LPBoost is generally solved via a column generation method [11]. Our first application of RMA is an extension of LPBoost using the RMA B&B algorithm to solve each subproblem exactly, instead of using a decision tree, to improve our LPBoost performance.

Bagging or *bootstrap aggregating* is creating K sets of $m' < m$ samples, where the m' samples are selected randomly with replacement. Using each set of samples, it creates a classifier or prediction function. The final model takes the average of K classifiers or prediction functions. Breiman [5] proposed the method of *random forests*, which constructs $\hat{f}$ by training regression trees on multiple random subsamples of the data, and then averaging the resulting predictors. Bagging and random forests can be applied for both classification and regression. Random forests is a modified version of bagging, differing from bagging is that it randomly chooses $n' < n$ features to build models when creating decision trees. A typical choice is $n' = n/3$ for regression and $n' = \sqrt{n}$ for classification. Bagging and random forests are bootstrapping, not boosting. Since bootstrapping does not have any sequential iterative procedures, the bootstrapping procedure can be parallelized. Therefore, bootstrapping has a potential run time advantage over boosting.

Another boosting algorithm for regression is RuleFit by Friedman *et al.* [19], which enhances L_1-regularized regression by generating box-based rules to use as additional explanatory variables. This work empirically demonstrated that the absolute error of linear model is larger than that of rule-based model, and that combined linear and rule-based model performed better than these two. Given $a, b \in \mathbb{R}^n$ with $a \leq b$, the rule function $r_{(a,b)} : \mathbb{R}^n \rightarrow \{0, 1\}$ is given by

$$r_{(a,b)}(x) = I\left(\wedge_{j \in \{1,\ldots,n\}}(a_j \leq x_j \leq b_j)\right), \tag{1.22}$$

that is $r_{(a,b)}(x) = 1$ if $a \leq x \leq b$ (componentwise) and $r_{(a,b)}(x) = 0$ otherwise. An example of such a rule (for a standard blood workup) might be "$0.6 \leq$ Creatinine ≤ 0.8 and Hematocrit ≤ 35". RuleFit generates rules through a two-phase procedure: first, it determines a regression tree ensemble, and then, in a second phase, it decomposes these trees into rules and determines the regression model coefficients (including for the rules).

The approach of Dembczyski et al. [9] generates rules more directly (without having to rely on an initial ensemble of decision trees) within gradient boosting [18] for non-regularized regression. In this scheme, a greedy procedure generates the rules within a gradient descent method that runs for a predetermined number of iterations. Aho et al. [1] extended the RuleFit method to solve more general multi-target regression problems. For the special case of single-target regression, however, their experiments suggest that random forests and RuleFit outperform several other methods, including their own extended implementation and the algorithm of [9]. Compared with Random Forests and other popular learning approaches such as kernel-based methods and neural networks, rule-based approaches have the advantage of generally being considered more accessible and easier to interpret by domain experts. Rule-based methods also have a considerable history in classification settings, as in for example [30], [7], and [10].

Our second application of RMA is a rule-enhanced penalized linear regression procedure. We call this iterative optimization-based regression procedure *REPR* (Rule-Enhanced Penalized Regression). Its output models resemble those of RuleFit, but our methodology draws more heavily on exact optimization techniques from the field of mathematical programming. While it is quite computationally intensive, its prediction performance appears promising. As in RuleFit, we start with a linear regression model (in this case, with L_1-penalized coefficients to promote sparsity), and enhance it by synthesizing rules of the form (1.22). We incrementally adjoin such rules to our (penalized) linear regression model as if they were new observation variables. Unlike RuleFit, we control the generation of new rules using the classical mathematical programming technique of column generation. Our employment of column generation roughly resembles its use in the LPBoost ensemble classification method of [11].

Column generation dates back to [14, 20]; see for example Section 7.3 of [21] for a recent textbook treatment. Column generation involves cyclical alternation between optimization of a *restricted master* problem (in our case a linear or convex quadratic program) and a *pricing problem* that finds the most promising new variables to adjoin to the formulation. In our case, the pricing problem is equivalent to the RMA problem.

Chapter 2

Rectangular Maximum Agreement

2.1 Problem Definition

Suppose there are m observations $X_1, \ldots, X_m \in \mathbb{R}^n$, each having n attributes. Let x_{ij} be the $(i,j)^{\text{th}}$ element of the resulting matrix X, that is, the value of attribute j in observation i. Each observation $i = 1, \ldots, m$ has a weight $w_i \in \mathbb{R}$. For any set $S \subseteq \{1, \ldots, m\}$, let $w(S) = \sum_{i \in S} w_i$. For the purpose of the RMA problem, zero-weight observations may be discarded. Non-zero-weight observations are partitioned into the "positive" and "negative" subsets, $\Omega^+ \subset \big\{i \in \{1, \ldots, m\} \mid w_i > 0\big\}$ and $\Omega^- \subset \big\{i \in \{1, \ldots, m\} \mid w_i < 0\big\}$.

Given two vectors $a, b \in \mathbb{R}^n$, let $B(a,b)$ be the "box" $\{x \in \mathbb{R}^n \mid a \leq x \leq b\}$. Figure 2.1 shows a simple box in two dimension. Given input data as described previously, the "coverage" of $B(a,b)$ consists of the indices of the observations falling within $B(a,b)$, that is $\text{Cvr}_X(a,b) = \{i \in \{1, \ldots, m\} \mid a \leq x_i \leq b\}$. Figure 2.1 shows an example in which observation 1 is covered but observation 2 is not covered by the box

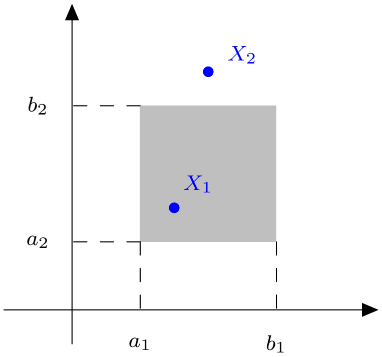


Figure 2.1: A box in two dimensions: Observation 1 is covered by the box, but observation 2 is not covered by the box.

Algorithm 1 Discretization algorithm

```
Input: $X \in \mathbb{R}^{m \times n}$, $\delta \geq 0$
Output: $\overline{X} \in \mathbb{N}^{m \times n}$, $\ell \in \mathbb{N}^n$
ProcessData:

for $j = 1$ to $n$ do
    $\ell_j \leftarrow 0$
    Sort $x_{1j}, \ldots, x_{mj}$ and set $(k_1, \ldots, k_m)$ such that $x_{k_1 j} \leq x_{k_2 j} \leq \cdots \leq x_{k_m j}$
    $\bar{x}_{k_1,j} \leftarrow 0$
    for $i = 1$ to $m - 1$ do
        if $x_{k_{i+1} j} - x_{k_i j} > \delta \cdot \mathrm{CI}_{95\%}(x_j)$ then $\ell_j \leftarrow \ell_j + 1$
        $\bar{x}_{k_{i+1} j} \leftarrow \ell_j$
    end for
    $\ell_j \leftarrow \ell_j + 1$
end for
return $(\overline{X}, \ell)$
```

$B(a, b)$. The rectangular maximum agreement (RMA) problem is then formulated as:

$$\begin{array}{ll} \max & \left| w\big(\operatorname{Cvr}_X(a, b)\big) \right| \\ \text{s.t.} & a, b \in \mathbb{R}^n. \end{array}$$

Essentially implicit in this formulation is the constraint that $a \leq b$, since if $a \not\leq b$ then $\operatorname{Cvr}_X(a, b) = \emptyset$ and the objective value is 0, the smallest possible. Moreover, the Monomial Maximum Agreement (MMA) problem is the special case that $X_1, \ldots, X_m \in \{0, 1\}^n$ [12].

2.1.1 Preprocessing and Restriction to Natural Numbers

Any RMA problem instance may be converted to an equivalent instance in which all the observation data are integer. Essentially, for each coordinate $j = 1, \ldots, n$, one may simply record the distinct values of x_{ij} and replace each x_{ij} with its ordinal position among these values. Algorithm 1, with its parameter δ set to 0, performs exactly this “discretization” procedure, outputing an equivalent data matrix $\overline{X} \in \mathbb{N}^{m \times n}$ and a vector $\ell \in \mathbb{N}^n$ whose j^{th} element is $\ell_j = \left| \bigcup_{i=1}^{m} \{x_{ij}\} \right|$. Algorithm 1’s output values $\bar{x}_{ij}$ for attribute j vary between 0 and $\ell_j - 1$.

The number of distinct values ℓ_j for each explanatory variable j strongly influences the difficulty of the RMA problem. To obtain easier problem instances at the cost of implicitly searching a smaller set K of possible box rules in our column generation application, one may set the parameters of Algorithm 1 to combine nearby values into common "bins". Essentially, if $\delta > 0$, the algorithm bins together consecutive values x_{ij} that are within relative tolerance δ, resulting in a smaller number of distinct values ℓ_j for each explanatory variable j. The comparison between successive variable values on line 9 of Algorithm 1 uses δ scaled by the 95% central confidence interval of x_j, denoted by $\mathrm{CI}_{95\%}(x_j)$ and defined to be the difference between the 2.5% and 97.5% quantiles of x_j. This comparison procedure provides some problem-adaptive scaling but limits the influence of extreme values of explanatory variables.

In cases where an attribute has a large number of closely-spaced values, Algorithm 1 can aggregate the input data into excessively large bins. To avoid this problem, we use the following procedure: if the values in any bin span a range greater than $\rho\,\mathrm{CI}_{95\%}(x_j)$ for some parameter ρ, the bin is broken into sub-bins by recursively applying Algorithm 1 to just the data within the bin. This process is repeated recursively until no sub-bin spans more than a fraction ρ of the 95% central confidence interval of its containing bin. Our recommended value of ρ is $\rho = 0.05$.

2.2 RMA MIP Formulation

Figure 2.2 shows a mixed integer programming (MIP) formulation for the RMA problem. Constraints (2.2) and (2.3) ensure that the objective value is the maximum absolute value of the sum of the covered observation weights. If $s = 1$, constraint (2.2) is inactive; else if $s = 0$, constraint (2.3) is inactive since adding $2(\sum_{i=1}^{m} |w_i|)$ to $\pm \sum_{i=1}^{m} w_i q_i$ provides a loose bound on the objective. If a value x_{ij} is not covered for attribute j, then $z_{j,x_{ij}} = 0$, and constraint (2.4) ensures $q_i = 0$. If the observation i is covered in all dimensions, the right-hand side of constraint (2.5) leads to $q_i = 1$. For each attribute j, constraints (2.6) and (2.7) require that there is exactly one distinct value for the respective lower and upper bounds for each attribute. Constraints (2.8)-(2.13) implicitly ensure that the lower bound of an attribute is less than equal to its upper bound. For each attribute j, $z_{jk} = 1$ if $k \in \{0, \ldots, \ell_j - 1\}$ is a covered value of attribute j, else it

$$\max \quad \phi \tag{2.1}$$

$$\text{ST} \quad \phi \leq \sum_{i=1}^{m} w_i q_i + 2s \left(\sum_{i=1}^{m} |w_i| \right) \tag{2.2}$$

$$\phi \leq -\sum_{i=1}^{m} w_i q_i + 2(1-s) \left(\sum_{i=1}^{m} |w_i| \right) \tag{2.3}$$

$$q_i \leq z_{j,x_{ij}} \qquad \forall i, j \tag{2.4}$$

$$q_i \geq \sum_{j=1}^{n} z_{j,x_{ij}} - (n-1) \qquad \forall i \tag{2.5}$$

$$\sum_{k=0}^{\ell_j - 1} l_{jk} = 1 \qquad \forall j \tag{2.6}$$

$$\sum_{k=0}^{\ell_j - 1} u_{jk} = 1 \qquad \forall j \tag{2.7}$$

$$z_{j,-1} = 0 \qquad \forall j \tag{2.8}$$

$$z_{j,\ell_j} = 0 \qquad \forall j \tag{2.9}$$

$$z_{jk} \leq z_{j,k-1} + l_{jk} \qquad \forall j, k \tag{2.10}$$

$$z_{jk} \leq z_{j,k+1} + u_{jk} \qquad \forall j, k \tag{2.11}$$

$$l_{jk} \leq z_{jk} \qquad \forall j, k \tag{2.12}$$

$$u_{jk} \leq z_{jk} \qquad \forall j, k \tag{2.13}$$

$$l_{jk} \in \{0, 1\} \qquad \forall j, k \tag{2.14}$$

$$u_{jk} \in \{0, 1\} \qquad \forall j, k \tag{2.15}$$

$$0 \leq z_{jk} \leq 1 \qquad \forall j, k \tag{2.16}$$

$$0 \leq q_i \leq 1 \qquad \forall i \tag{2.17}$$

where

ϕ	objective value
s	1 for positive objective value and 0 for negative
q_i	1 if observation i is covered, else 0
u_{jk}	1 if k is the upper bound of attribute j, else 0
l_{jk}	1 if k is the lower bound of attribute j, else 0
z_{jk}	1 if k is a covered value of attribute j, else 0

Figure 2.2: A MIP formulation of the RMA problem

k	-1	0	1	2	3	4	5	6
z_{jk}	0	0	1	1	1	1	0	0
u_{jk}	—	0	0	0	0	1	0	—
l_{jk}	—	0	1	0	0	0	0	—

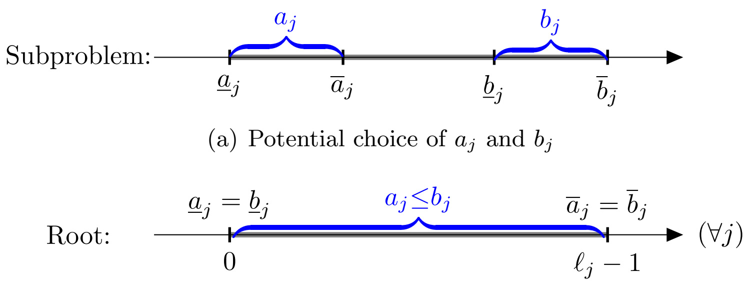
Table 2.1: An example of MIP formulation, for $\ell_j = 6$

is 0; $z_{jk} = 0$ if k is 0 or ℓ_j. Table 2.2 shows an example of how constraints (2.8)-(2.13) are satisfied.

The number of variables and constraints in the formulation are $m+2n+3\sum_{j=1}^{n} \ell_j+2$ and $mn + m + 4n + 4\sum_{j=1}^{n} \ell_j + 2$, respectively. Due to the large number of variables and constraints for the MIP formulation, Gurobi [22] could not solve some larger RMA instances we experimented with. Therefore, we developed a customized parallel parallel branch-and-bound algorithm for the RMA problem class. Section 2.7.3 compares the performance of Gurobi on the MIP formulation with the algorithms we now describe.

2.3 A Branch-and-Bound Algorithm for RMA

In the specialized branch-and-bound scheme, each subproblem is defined by upper and lower bounds on the vectors a and b defining the box. Specifically, each subproblem P is characterized by four vectors $\underline{a}(P), \overline{a}(P), \underline{b}(P), \overline{b}(P) \in \mathbb{N}^n$, and represents the portion of the search space of vector pairs (a, b) for which $\underline{a}(P) \leq a \leq \overline{a}(P)$ and $\underline{a}(P) \leq b \leq \overline{b}(P)$. Figure 2.3(a) depicts the potential choices of a_j and b_j for a given $\underline{a}(P), \overline{a}(P), \underline{b}(P), \overline{b}(P)$. Henceforth, assume that the data X have already been preprocessed as described in Section 2.1.1. A valid subproblem conforms to the conditions $\underline{a}(P) \leq \overline{a}(P)$, $\underline{b}(P) \leq \underline{b}(P)$, $\underline{a}(P) \leq \underline{b}(P)$, and $\overline{a}(P) \leq \overline{b}(P)$. However, $\overline{a}(P) \leq \underline{b}(P)$ is not necessary. The root problem R of the branch-and-bound tree is given by $\underline{a}_j(R) = \underline{b}_j(R) = 0$ and $\overline{a}_j(R) = \overline{b}_j(R) = \ell_j - 1$, $j = 1, \ldots, n$, where ℓ_j is the number of effectively distinct values in attribute j, as output from the recursive discretization method of Section 2.1.1. Figure 2.3(b) depicts the potential choices of a_j and b_j for the root problem. At the root subproblem, a_j and b_j may be any natural number within $[0, \ell_j - 1]$, as long as $a_j \leq b_j$. Writing each subproblem as $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$, the root subproblem may be expressed as $R = (\mathbf{0}, \ell - \mathbf{1}, \mathbf{0}, \ell - \mathbf{1})$, where $\ell \in \mathbb{Z}^n$ is the vector consisting of the ℓ_j.



(a) Potential choice of a_j and b_j

(b) Potential choice of a_j and b_j for the root problem

Figure 2.3: A graphical view of potential choice of a_j and b_j for a given subproblem subproblem $\underline{a}, \overline{a}, \underline{b}, \overline{b}$

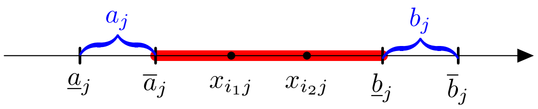


Figure 2.4: Graphical view of the inseparability region for attribute j when $\overline{a}_j < \underline{b}_j$

2.3.1 Inseparability and the Bounding Function

In branch-and-bound methods, the bounding function provides an upper bound (when maximizing) on the best possible objective value in the region of the search space corresponding to a subproblem. The method uses a bounding function based on an extension of the inseparability notion developed for the MMA problem in [12]. Consider any subproblem $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$ and two observations i_1 and i_2. As shown in Figure 2.4, if $\overline{a}_j < \underline{b}_j$, any values in $[\overline{a}_j, \underline{b}_j]$ are always covered by any permitted selection of a_j and b_j. If $x_{i_1 j} = x_{i_2 j}$ or $\overline{a}_j \leq x_{i_1 j}, x_{i_2 j} \leq \underline{b}_j$ for each $j = 1, \ldots, n$, then $X_{i_1}, X_{i_2} \in \mathbb{N}^n$ are *inseparable* with respect to $\overline{a}, \underline{b} \in \mathbb{N}^n$, in the sense that any box $B(a, b)$ with $a \leq \overline{a}$ and $b \geq \underline{b}$ must either cover both of X_{i_1}, X_{i_2} or neither of them. When $\overline{a} \geq \underline{b}$, the inseparability condition reduces to $X_{i_1} = X_{i_2}$.

Inseparability with respect to $\overline{a}, \underline{b}$ is an equivalence relation; denote the equivalence classes it induces among the observation indices $1, \ldots, m$ by $\mathcal{E}(\overline{a}, \underline{b})$. That is, observation indices i_1 and i_2 are in the same equivalence class of $\mathcal{E}(\overline{a}, \underline{b})$ if X_{i_1} and X_{i_2} are inseparable with respect to $\overline{a}, \underline{b}$.

Our bounding function for each subproblem $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$ is

$$g(\underline{a}, \overline{a}, \underline{b}, \overline{b}) = \max \left\{ \sum_{C \in \mathcal{E}(\overline{a}, \underline{b})} \left[w(C \cap \mathrm{Cvr}_X(\underline{a}, \overline{b})) \right]_+ , \sum_{C \in \mathcal{E}(\overline{a}, \underline{b})} \left[w(C \cap \mathrm{Cvr}_X(\underline{a}, \overline{b})) \right]_- \right\}, \tag{2.18}$$

where $[d]_+ = \max\{d, 0\}$ and $[d]_- = \max\{-d, 0\}$ denote the positive and negative parts of a number, respectively. The reasoning behind this bound is that each possible box allowed by $(\underline{a}, \overline{a}, \underline{b}, \overline{b})$ must either cover or not cover the entirety of each $C \in \mathcal{E}(\overline{a}, \underline{b})$. The first argument to the "max" operation reflects the situation that every equivalence class C with a positive net weight is covered, and no classes with negative net weight are covered; this is the best possible situation if the box ends up covering a higher weight of positive observations than of negative. The second "max" argument reflects the opposite situation, the best possible case in which the box covers a greater weight of negative observations than of positive ones.

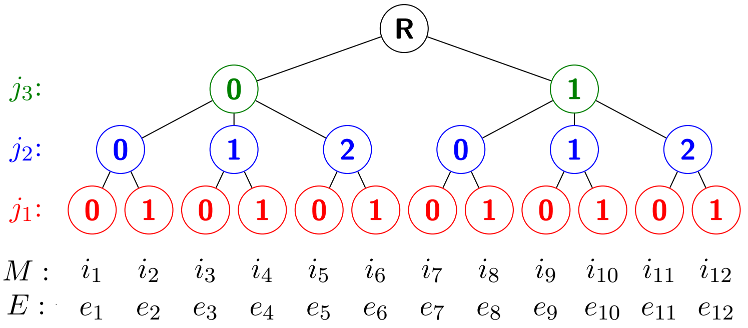
2.3.1.1 Equivalence Class Computation

This section describes the algorithm to compute equivalence classes for a subproblem $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$. The algorithm starts with a set of indices $M \subseteq \{1, 2, \ldots, m\}$ containing all non-zero-weight observation indices, since zero-weight observations have no effect on the objective function. For each attribute $j = 1, 2, \ldots, n$, the algorithm performs a stable bucket sort of M based on the column vector x_j and discards observations with $x_{ij} < \underline{a}_j$ or $\overline{b}_j < x_{ij}$, since they are not covered by the current subproblem. Moreover, $x_{i_1 j}, x_{i_2 j}$ are treated as being equal if $\overline{a}_j \le x_{i_1 j}, x_{i_2 j} \le \underline{b}_j$, due to inseparability. Since each bucket sort takes $\mathrm{O}(m)$ time and there are n attributes, the total run time is $\mathrm{O}(mn)$. Then, by scanning the sorted vector M containing non-zero-weight observation indices covered by the current subproblem, the equivalence classes $\mathcal{E}$ can be created. Each equivalence class $\mathcal{E}_e$ stores a representative observation index "obsIdx" and a total weight "wt". The procedure to create equivalence classes is shown in Algorithm 2. The **createEquivClass** function in line 5 and 11 of Algorithm 2 initializes a new equivalence class $\mathcal{E}_e$ with its observation index and weight.

	Observation											
Attribute	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{10}	i_{12}
j_3	0	0	0	0	0	0	1	1	1	1	1	1
j_2	0	0	1	1	2	2	0	0	1	1	2	2
j_1	0	1	0	1	0	1	0	1	0	1	0	1

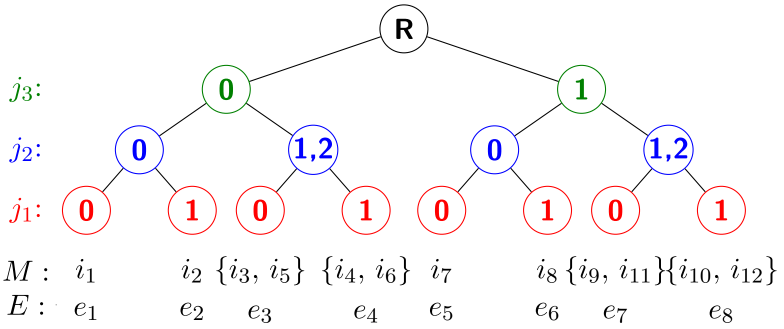
(a) An example dataset

Attribute	$\underline{a}$	$\overline{a}$	$\underline{b}$	$\overline{b}$
j_3	0	1	0	1
j_2	0	2	0	2
j_1	0	1	0	1



(b) An initial equivalence class tree for the root node of the search tree

Attribute	$\underline{a}$	$\overline{a}$	$\underline{b}$	$\overline{b}$
j_3	0	1	0	1
j_2	0	**1**	**2**	2
j_1	0	1	0	1



(c) An equivalence class tree for a subproblem having $\overline{a}_j < \underline{b}_j$ for some attribute j (in this case, j_2)

Figure 2.5: Equivalence class construction examples

Algorithm 2 Creating an Initial Equivalence Class

```
1: Input: $M$ (sorted, covered observation indices), $X \in \mathbb{R}^{m \times n}$ (data),
   $w \in \mathbb{R}^m$ (observation weights), $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$ (subproblem)
2: Output: $\mathcal{E}$ (Equivalence Classes)
3: createInitEquivClasses:

4: $e \leftarrow 1$ ; $i_1 \leftarrow$ dequeue($M$); $i_2 \leftarrow$ dequeue($M$)
5: $\mathcal{E}_e \leftarrow$ createEquivClass($i_1, w_{i_1}$)
6: loop
7:    if isInseparable($X_{i_1}, X_{i_2}, P$) then
8:       $\mathcal{E}_e$.wt $\leftarrow \mathcal{E}_e$.wt $+ w_{i_2}$
9:    else
10:      $e \leftarrow e + 1$; $i_1 \leftarrow i_2$
11:      $\mathcal{E}_e \leftarrow$ createEquivClass($i_1, w_{i_1}$)
12:   end if
13:   if $M = \varnothing$ break
14:   else $i_2 \leftarrow$dequeue($M$)
15: end loop
16: return $\mathcal{E}$
```

A tree can be used to visualize the structure of the equivalence classes. For example, suppose that the dataset has 12 observations and 3 attributes as shown in Figure 2.5(a). Now, consider the root subproblem, with $\underline{b} \leq \overline{a}$, as shown in Figure 2.5(b). After the sequence of the stable bucket sorts described above, we may envision **createEquivClass** as constructing the equivalence-class tree and the vector M shown in Figure 2.5(b). The **isInseparable** function in line 7 of Algorithm 2 checks whether two observations are inseparable; if the two observations are inseparable, the function returns true, otherwise false. Since observations in the same equivalence classes must be contiguous in the sorted observation index vector M, it is sufficient to compare adjacent element of M to construct the equivalence classes. The **isInseparable** function checks whether or not the consecutive two observations in M are in the same equivalence class by comparing attribute values in order from the leaf level of the tree to the root level. As soon as it detects that two observations are not in the same equivalence class, the algorithm no longer compares the rest of the attribute values. It is relatively efficient to compare two consecutive observations' attribute values in order from the leaf level to the root level, since deeper-level attributes change more frequently than shallower-level attributes. The resulting initial equivalence-class tree for the example data is shown in Figure 2.5(b). Let E be a vector of sorted and covered equivalence class indices. Computing the bounding function only requires scanning E and adding each equivalence class's weight to one of two possible accumulators, depending on its sign. Even though we use a tree to depict the equivalence classes, an actual tree structure does not have to be stored. Instead, only the vectors M and E and the equivalence classes $\mathcal{E}$ are stored. Now, suppose there is a subproblem with $\underline{b} \nleq \overline{a}$ as shown in Figure 2.5(c), within $[\overline{a}_{j_2}, \underline{b}_{j_2}] = [1, 2]$. If any observations which have the exact same values for all attributes except values within $[1, 2]$ in attribute j_2 and values of attribute j_2 are within $[1, 2]$, they are inseparable and in the same equivalence class, as shown in Figure 2.5(c).

After creating the sorted observation list M, the running time to create initial equivalence classes is $\mathrm{O}(mn)$ per subproblem, but it is usually much less. For example, in Figure 2.5(b), since all consecutive pair values at the leaf level of the equivalence-class tree are different, it takes $\mathrm{O}(m)$ time to find the initial equivalence classes using the sorted M.

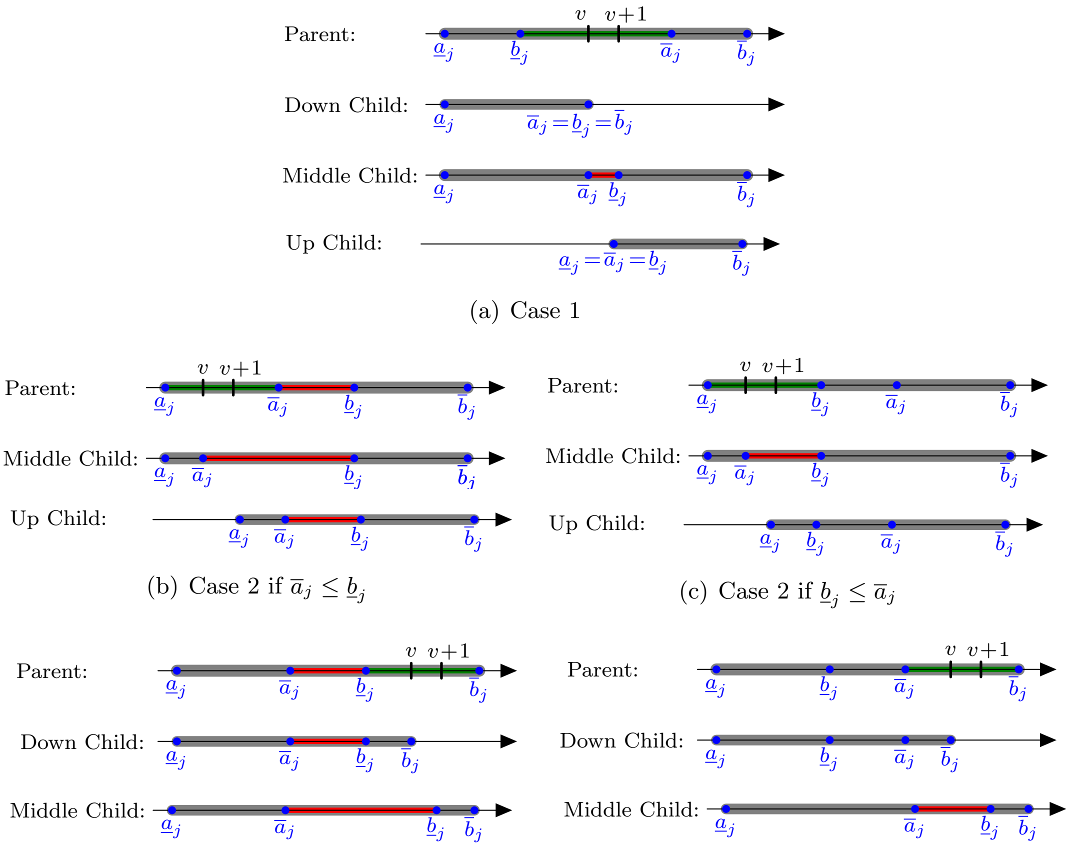
2.3.2 Branching

This section describes branching procedures in each subproblem, $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$, for the algorithm. Branching a subproblem involves choosing a coordinate $j \in \{1, \ldots, n\}$ and a *cutvalue* $v \in \{\underline{a}_j, \ldots, \overline{b}_j - 1\}$. Let a *cutpoint* be such a pair (j, v). There are three possible cases, one generating three children, and the others generating two children:

Case 1: If $\underline{b}_j < \overline{a}_j$ and $v \in \{\underline{b}_j, \ldots, \overline{a}_j - 1\}$, then P is split into three children as shown in the table below:

Child	$\underline{a}_j$	$\overline{a}_j$	$\underline{b}_j$	$\overline{b}_j$	Explanation	
Down	$\underline{a}_j$	v	v	v	$a_j \leq v$, $b_j = v$	The box falls below v.
Middle	$\underline{a}_j$	v	$v+1$	$\overline{b}_j$	$a_j \leq v$, $b_j > v$	The box spans $[v, v+1]$.
Up	$v+1$	$v+1$	$v+1$	$\overline{b}_j$	$a_j = v+1$, $b_j > v$	The box falls above $v+1$.

For cases 2 and 3, $v \in \big[\underline{a}_j, \overline{b}_j\big] \setminus \big[\min\{\overline{a}_j, \underline{b}_j\}, \max\{\overline{a}_j, \underline{b}_j\}\big]$.


Parent:
v $v{+}1$
$\underline{a}_j$ $\underline{b}_j$ $\overline{a}_j$ $\overline{b}_j$
Down Child:
$\underline{a}_j$ $\overline{a}_j = \underline{b}_j = \overline{b}_j$
Middle Child:
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$
Up Child:
$\underline{a}_j = \overline{a}_j = \underline{b}_j$ $\overline{b}_j$
Parent:
v $v{+}1$
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$
Middle Child:
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$
Up Child:
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$
Parent:
v $v{+}1$
$\underline{a}_j$ $\underline{b}_j$ $\overline{a}_j$ $\overline{b}_j$
Middle Child:
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$
Up Child:
$\underline{a}_j$ $\underline{b}_j$ $\overline{a}_j$ $\overline{b}_j$
Parent:
v $v{+}1$
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$
Down Child:
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$
Middle Child:
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$
Parent:
v $v{+}1$
$\underline{a}_j$ $\underline{b}_j$ $\overline{a}_j$ $\overline{b}_j$
Down Child:
$\underline{a}_j$ $\underline{b}_j$ $\overline{a}_j$ $\overline{b}_j$
Middle Child:
$\underline{a}_j$ $\overline{a}_j$ $\underline{b}_j$ $\overline{b}_j$

(a) Case 1

(b) Case 2 if $\overline{a}_j \leq \underline{b}_j$

(c) Case 2 if $\underline{b}_j \leq \overline{a}_j$

(d) Case 3 if $\overline{a}_j \leq \underline{b}_j$

(e) Case 3 if $\underline{b}_j \leq \overline{a}_j$

Figure 2.6: Graphical View of Branching: cutpoints are selected from the green range in each parent subproblem, and the red range is inseparable in each child problem.

Case 2: If $v \in \left[\underline{a}_j, \ldots, \min\{\overline{a}_j, \underline{b}_j\} - 1\right]$, then P is split into two children, as follows:

Child	$\underline{a}_j$	$\overline{a}_j$	$\underline{b}_j$	$\overline{b}_j$	Explanation
Middle	$\underline{a}_j$	v	$\underline{b}_j$	$\overline{b}_j$	$a_j \leq v$
Up	$v+1$	$\overline{a}_j$	$\underline{b}_j$	$\overline{b}_j$	$a_j > v$

Case 3: If $v \in \left[\max\{\overline{a}_j, \underline{b}_j\}, \ldots, \overline{b}_j - 1\right]$, then P is split into two children as shown in the table below:

Child	$\underline{a}_j$	$\overline{a}_j$	$\underline{b}_j$	$\overline{b}_j$	Explanation
Down	$\underline{a}_j$	$\overline{a}_j$	$\underline{b}_j$	v	$b_j \leq v$
Middle	$\underline{a}_j$	$\overline{a}_j$	$v+1$	$\overline{b}_j$	$b_j > v$

(a) Computed bounds of children for each cutpoint

(j, v)	B_D	B_M	B_U
(j_1, v_1)	.2	.7	.55
(j_2, v_2)	.6	.7	.3

(b) Set a bound $-\infty$ if it is lower than the incumbent

(j, v)	B_D	B_M	B_U
(j_1, v_1)	$-\infty$	.7	.55
(j_2, v_2)	.6	.7	$-\infty$

(c) Descending sort on bounds of children for each cutpoint

(j, v)	B_1	B_2	B_3
(j_1, v_1)	.7	.55	$-\infty$
(j_2, v_2)	.7	.6	$-\infty$

Table 2.2: An example of the lexicographic comparison procedures to choose the optimal cutpoint to branch when the incumbent is 0.5

If no v meeting any of these three cases exist for any attribute j, then the subproblem represents a single possible box, that is, $\underline{a} = \overline{a}$ and $\underline{b} = \overline{b}$. Such a subproblem is a terminal node of the branch-and-bound tree, and in such a case the RMA objective value computed for $a = \underline{a} = \overline{a}$ and $b = \underline{b} = \overline{b}$ is substituted for its bound. When more than one possible cutpoint pair (j, v) exists, as is typically the case, the algorithm must select one. This dissertation tests four branching methods: strong branching, cutpoint caching, binary search, and hybrid search.

2.3.2.1 Branching Rules

Generally, RMA (our algorithm to solve the RMA problems) selects the cutpoint that minimizes the maximum bound of the resulting two or three children. Since multiple cutpoints may be tied in this comparison, RMA ranks (or "scores") multiple candidate cutpoints in a lexicographic manner as follows: when a cutpoint does not have an up or down child, then the bound of the missing child is set to negative infinity. Next, any bound lower than the incumbent is set to negative infinity. Now, three bounds are sorted from the highest to lowest for each cutpoint. Then, RMA compares the largest bound of cutpoints and selects one with the minimum value. If multiple cutpoints are tied after comparing their highest bounds, then RMA chooses one with the smallest value among their second highest bounds. If there are still ties among some cutpoints, RMA similarly chooses one with the smallest value among their smallest (third highest) bounds.

Table 2.2 shows an example of the lexicographic comparison on given two cutpoints when the current incumbent is 0.5. B_D, B_M, and B_U respectively denote a bound value of down, middle, and up child. First, all bounds for potential children created by these two cupoints were computed as shown in Table 2.2(a). Second, RMA sets the bounds less than the incumbent to negative infinity as shown in Table 2.2(b). Third, the bounds of 3 children for each cutpoint are sorted in descending order and denoted by B_1, B_2, and B_3, as shown in Table 2.2(c). Finally, in this example, the cutpoint (j_1, v_1) is chosen since its second highest bound is smaller than the cutpoint (j_2, v_2) even though their highest bounds are the same. In some cases, more than one cutpoint may be tied in lexicographic comparison. Let such cutpoints be called "optimal tied cutpoints". RMA has three methods to select one cutpoint to branch among optimal tied cutpoints. The first and second methods are choosing the first and the last optimal cutpoint discovered, respectively. The third (default) method is randomly selecting one of the optimal tied cutpoints. The first and second methods are straightforward to implement. An efficient implementation of the third method is as follows: while computing bounds of potential children for each cutpoint sequentially, the optimal cupoint is updated by lexicographic comparison. If a new optimal tied cutpoint is found, it replaces the current cutpoint with probability $1/I$, where I is the number of the optimal tied cutpoints found so far. Algorithm 3 shows pseudocode for this random cutpoint selection method. Let B contains 3 sorted bound values (B_1, B_2, B_3) for children potentially created by a cutpoint (j, v). The "<" symbol on line 10 of Algorithm 3 denotes lexicographic comparison. This method does not need to store a list of optimal tied cutpoints.

Lemma 2.3.1. *Under Algorithm 3, each optimal tied cutpoint has equal probability of being chosen.*

Proof. We proceed by induction on k, the number of optimal tied cutpoints. The result is true here for $k = 1$. Now, assume that k optimal tied cutpoints have been found, and the probability of each was chosen is $\frac{1}{k}$. Next, we show that when a $(k+1)$th tied cutpoint is found, the probability of each tied cutpoint to be chosen is $\frac{1}{k+1}$. If the random number in line 6 is less than equal to $\frac{1}{k+1}$, the $(k+1)$th tied cutpoint is chosen. Hence, the probability that the $(k+1)$th tied cutpoint is not chosen is $\frac{k}{k+1}$. Furthermore, the probability that each of the previous k tied cutpoints is chosen is $\frac{k}{k+1} \cdot \frac{1}{k} = \frac{1}{k+1}$. □

Algorithm 3 Selecting the best cutpoint randomly among the optimal tied cutpoint by the lexicographical comparison

```
Input:
    $(B, j, v)$ (current cutpoint to inspect and its bounds),
    $(B^*, I, j^*, v^*)$ (current optimal cutpoint, its bounds and # of tied cutpoints)
Output: $(B^*, I^*, j^*, v^*)$ (the best cutpoint to branch)
RandomBestCutointSelection:

if $B = B^*$ then
    $I \leftarrow I + 1$;
    Generate $r$ (a uniform random number between 0 and 1)
    if $r \leq \frac{1}{I}$ then
        $(B^*, j^*, v^*) \leftarrow (B, j, v)$
    end if
else if $B < B^*$ then
    $(B^*, I^*, j^*, v^*) \leftarrow (B, 1, j, v)$
end if
return $(B^*, I^*, j^*, v^*)$
```

2.3.2.2 Strong Branching

In strong branching, all applicable cutpoints (j, v) in each subproblem are tested, and the algorithm selects one by the lexicographic comparison, and tiebreaking procedures described in the previous section. Strong branching is a standard technique in branch-and-bound algorithms: it involves evaluating the bounds of all the potential children of the current search node and is the strategy used in [12] for the related MMA problem. The strong branching process may be improved by using the equivalence class representation in Section 2.3.1.1 and analyzing the branching possibilities in a particular order dictated by that representation; see Section 2.3.2.6 below.

2.3.2.3 Cutpoint Caching

Strong branching can be much more time consuming for the RMA problem than for MMA, because the number of potential cutpoints is often much larger. Therefore, several alternative branching procedures are considered. Our experiments showed that randomly sampling cutpoints greatly inflated the search tree. Eventually, an effective heuristic of “cutpoint caching” was discovered. It is based on the tendency of strong branching to select the same cutpoints at many different points in the search tree.

Essentially, every time strong branching is performed, the selected cutpoint is stored in a cache. For each new subproblem, the heuristic checks what fraction of all possible cutpoints for the subproblem are already in the cache. If this fraction is above a threshold parameter τ, only the cached cutpoints are checked, compared using the same method as for strong branching; otherwise, the algorithm performs strong branching and potentially adds another cutpoint to the cache. This method significantly accelerated the branch-and-bound search: the branching selection is considerably faster than strong branching on average, but the search tree did not inflate significantly. As the cutpoint threshold τ decreases, the run time improved for most datasets, as shown in Section 2.7.3. In practice, it appeared advantageous to consider only the cached cutpoints if there is at least one applicable cached cutpoint.

2.3.2.4 Binary Cutpoint Search

For attributes j with a large number of distinct values (that is, large ℓ_j), binary cutpoint search is a second alternative to strong branching. This method begins by choosing a cutpoint v which is close as possible to $(\underline{a}_j + \overline{b}_j)/2$, subject to the restriction that it cannot lie within $[\overline{a}_j, \underline{b}_j)$. The next step is to generate the next candidate in a list of possible cutpoints by "diving" to either the left or right of v. Let g_- and g_+ be the bounds of down and up child obtained by branching at v, respectively. For a cutpoint without a down or up child, the missing child bound is replaced by that of the middle child. If $g_- \leq g_+$, then the next cutpoint to be evaluated is the midpoint of $\underline{a}_j$ and v, rounded to an integer; conversely, if $g_- > g_+$, then the next cutpoint to be evaluated is the midpoint of v and $\overline{b}_j$, rounded to an integer. A graphical depiction of this process is shown in Figure 2.7.

This diving process is repeated recursively until no more than three possible cutpoints remain. Of these remaining cutpoints and all those encountered earlier in the diving process, the lexicographically best of these choices is selected using the same ranking and selection procedures already described. Across all attributes j, the lexicographically best of these choices is selected, using the same tiebreaking procedures.

When each attribute has a large number of potential cutpoints ℓ_j, this binary search

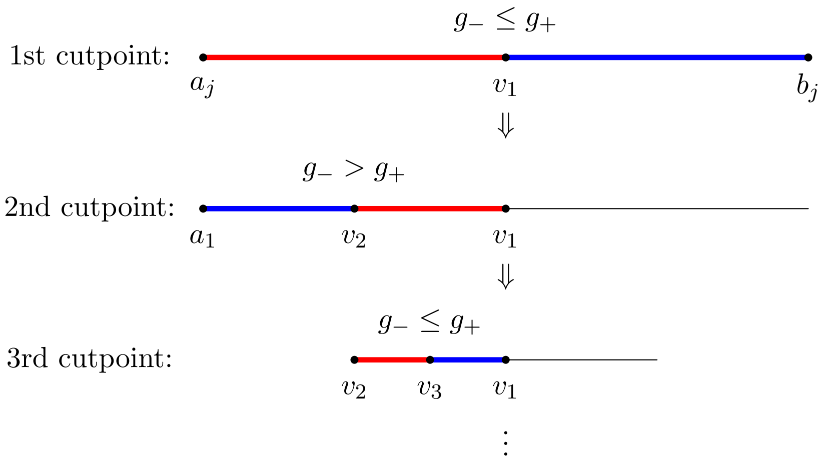


Figure 2.7: An example of binary cutpoint search method: cutpoints in red region are the potential cutpoints to branch, and the cutpoints in blue region are no longer considered to branch in the current subproblem.

method tends to decrease running time, even though it tends to increase the size of the branch-and-bound tree.

2.3.2.5 Hybrid Branching

A method combining both cutpoint caching and binary cutpoint search may be implemented by setting a threshold parameter η and proceeded as follow: let ℓ'_j the total number of applicable cutpoints for attribute j. For attributes with $\ell'_j \geq \eta$, RMA performs binary cutpoint search. During binary cutpoint search, the cutpoint selected in each subproblem is always stored in the cache. For attributes j having $\ell'_j < \eta$, RMA implements cutpoint caching, which may in turn "fall back" on strong branching when a sufficient number of applicable cutpoints are not found in the cache.

2.3.2.6 Rotation Algorithm

Strong branching requires computation of the equivalence classes for each potential child. Brute-force computation of the equivalence classes for all potential children requires creation of equivalence classes for each child, taking $\mathrm{O}(mn)$ time per child. There are at most $\mathrm{O}(\hat{L}n) \subseteq \mathrm{O}(mn)$ children, where $\hat{L} = \max_i \left\{\overline{b}_j - \underline{a}_j + \max\{0, \underline{b}_j - \overline{a}_j\}\right\}$. The overall run time to compute the bounds of all potential children by brute force is

Case	Child	(	$\underline{a}_j$,	$\overline{a}_j$,	$\underline{b}_j$,	$\overline{b}_j$	)	Drop or Merge Procedure
1	Down	(	$\underline{a}_j$,	v,	v,	v	)	Drop some equivalence classes
	Middle	(	$\underline{a}_j$,	v,	$v+1$,	$\overline{b}_j$	)	Merge some equivalence classes
	Up	(	$v+1$,	$v+1$,	$v+1$,	$\overline{b}_j$	)	Drop some equivalence classes
2	Down	(	$\underline{a}_j$,	$\overline{a}_j$,	$\underline{b}_j$,	v	)	Drop some equivalence classes
	Middle	(	$\underline{a}_j$,	$\overline{a}_j$,	$v+1$,	$\overline{b}_j$	)	Merge some equivalence classes
3	Middle	(	$\underline{a}_j$,	v,	$\underline{b}_j$,	$\overline{b}_j$	)	Merge some equivalence classes
	Up	(	$v+1$,	$\overline{a}_j$,	$\underline{b}_j$,	$\overline{b}_j$	)	Drop some equivalence classes

Table 2.3: A summary of the procedures to construct equivalence classes for each possible child of a parent subproblem $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$

therefore O(m^2n^2) per subproblem.

A "rotation" algorithm reduces the effort required to recompute equivalence classes. For each subproblem $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$, equivalence classes are created as explained in Section 2.3.1.1. To obtain equivalence classes for each child produced by each cutpoint, it is sufficient to drop or merge equivalence classes from the parent subproblem. If a child's $\overline{a}_j$ or $\underline{b}_j$ changes and $\overline{a}_j < \underline{b}_j$, some equivalence classes are required to be merged; else if a child's $\underline{a}_j$ or $\overline{b}_j$ changes, it is sufficient to drop some equivalence classes no longer covered by the child. Table 2.3 describes the drop or merge procedure required for each potential child for each case described in Section 2.3.2. When both up and down children are available for a cutpoint, as in Case 1 of Table 2.3, the parent bound is equal to the sum of bounds of the up and down children. Thus, it is sufficient to calculate the bound of only one of these children.

For each possible cutpoint (j, v), the above procedures are repeated to evaluate the bounds of the 2 or 3 possible children. As explained above, E contains the sorted equivalence-class indices for the parent subproblem. When dropping or merging equivalence classes, it is necessary to create a new equivalence-class index list $\hat{E}$ for each child since the original E must be kept for subsequent calculations. The dropping procedure eliminates equivalence-class indices no longer covered by a down or up child. The computations for dropping equivalence classes take O($\ell_j\,|E|$) time per attribute, since they require constant pass through the sorted equivalence-class list E for at most $\ell_j - 1$ candidate cutvalues. To merge equivalence classes when the number of cutvalues for each attribute is greater than 1, it is efficient to create a vector I, of the same length as E,

Dropping classes:	$\mathrm{O}(\ell_j \lvert E \rvert)$	$\subseteq$	$\mathrm{O}(m^2)$
Merging classes:	$\mathrm{O}(n \lvert E \rvert + \ell_j \lvert E \rvert)$	$\subseteq$	$\mathrm{O}(mn + m^2)$
Bounds of all children:	$\mathrm{O}(\ell_j \lvert E \rvert)$	$\subseteq$	$\mathrm{O}(m^2)$
Overall:	$\mathrm{O}(n \lvert E \rvert + \ell_j \lvert E \rvert)$	$\subseteq$	$\mathrm{O}(mn + m^2)$

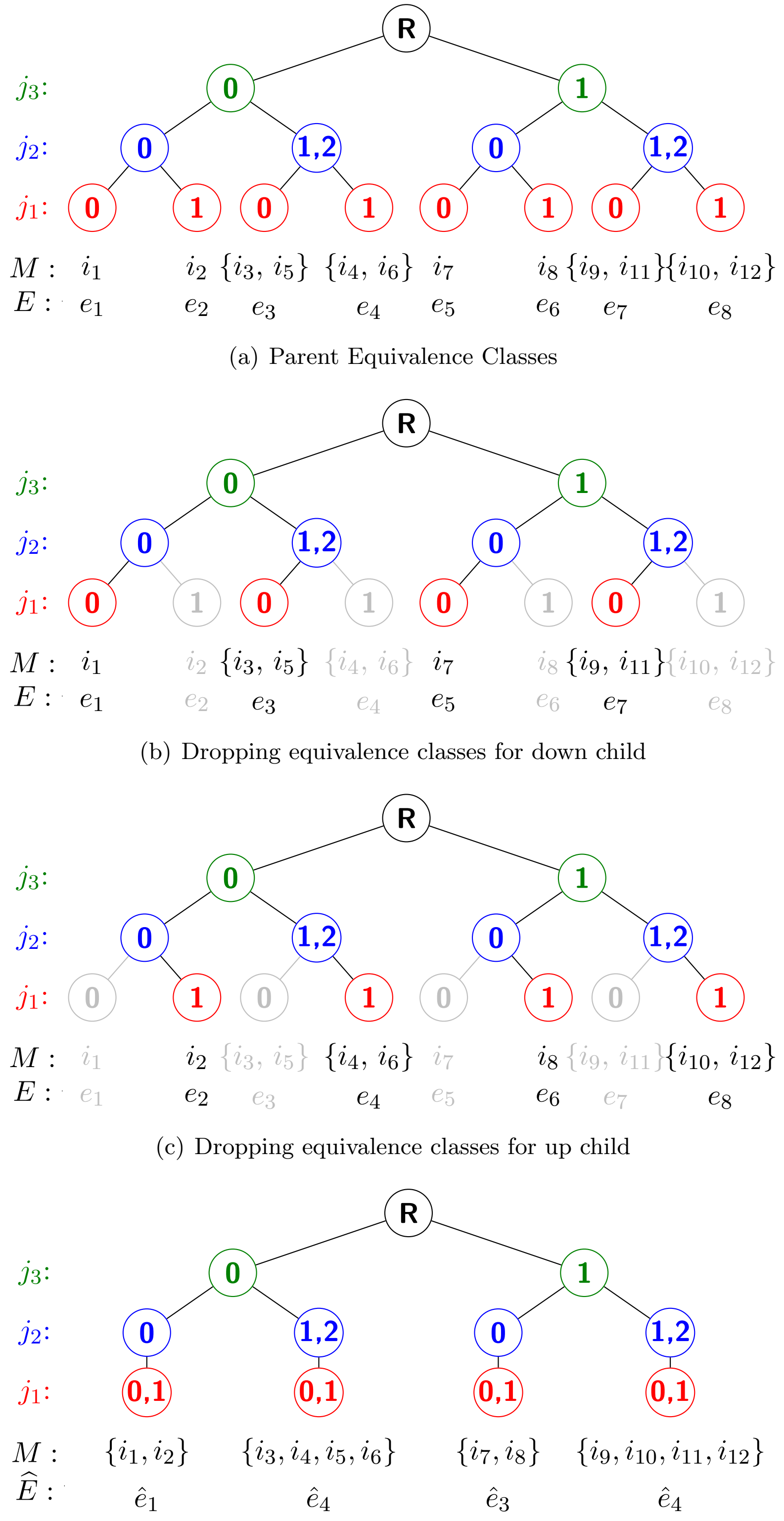
Table 2.4: Time to compute bounds of children for each attribute

that indicates whether or not each leaf node has the same parent node as the next leaf in the equivalence-class list. If there is only one applicable cutvalue for an attribute, it is not necessary to store I. Creating I takes $\mathrm{O}(n \lvert E \rvert)$ time, since the algorithm compares attribute values from the leaf level of the equivalence-class tree to the top level. It is efficient to stop comparing attribute values as soon as the algorithm detects that consecutive leaf nodes have a different parent nodes. For each middle child, the algorithm must scan the vectors E and I to detect whether or not consecutive equivalence class indices in E have to be merged. If consecutive equivalence classes values in attribute j are both within the child's $[\overline{a}_j, \underline{b}_j]$ and I indicates that the leaf nodes of these two equivalence classes have the same parent node in the equivalence-class tree, the two equivalence classes should be merged. After constructing I, the merging process takes $\mathrm{O}(\lvert E \rvert)$ per cutvalue for each attribute. Since there are at most ℓ_j potential cutvalues for each attribute, this process takes $\mathrm{O}(\ell_j \lvert E \rvert)$ time per attribute. Thus, the overall run time to merge equivalence classes for each attribute is $\mathrm{O}(n \lvert E \rvert + \ell_j \lvert E \rvert) \subseteq \mathrm{O}(nm + m^2)$. After merging or dropping equivalence classes, computing bounds for all children takes $\mathrm{O}(\ell_j \lvert E \rvert)$ per attribute, since it involves just adding the weight of each equivalence class in E for at most ℓ_j potential cutvalues.

A summary of the computations for each attribute is shown in Table 2.4. Its estimates are generally pessimistic since the number of equivalence classes and the number of potential cutvalues become smaller as the search proceeds deeper level of the branch-and-bound tree.

Suppose that the bound computations are processed for attribute $j_1, \dots, j_n$ in order. After computing all bounds for each child corresponding to attribute j_1, all cutpoints in attribute j_2 are evaluated next. To evaluate them, the algorithm performs a stable bucket or counting sort of E on the value of attribute j_1, Section 8.2 and 8.4 of the textbook [8] respectively explain counting and bucket sorts. considering the inseparability induced by the parent's $\overline{a}_{j_1}$ and $\underline{b}_{j_1}$ values. We call this process "rotating" the equivalence-class tree. After the equivalence classes are sorted by attribute j_1, the values of attributes j_1 are represented in the shallowest level of the tree, immediately below the root node, and the values of attributes other than j_1 are "shifted down", that is, shown one level lower, so that the leaf nodes of the tree correspond to values of attribute j_2. If attribute j_2 has no available cutvalues, then this sorting process is skipped. Next, the algorithm considers the cutvalues for attributes j_3. The rotation procedures repeats until attribute j_{n-1}, to evaluate cutvalues in attribute j_n. Each bucket or counting sort takes $\mathrm{O}(|E|)$ time and it is repeated for j_{n-1} attributes. Thus, the total time to for rotating the tree is $\mathrm{O}(n\,|E|)$ per subproblem. Since the computations in Table 2.4 are repeated for n attributes, the overall running time of the bound computations is $\mathrm{O}\big(n \cdot (mn + m^2)\big) = \mathrm{O}(mn^2 + m^2 n)$ per subproblem. Even though this bound is very pessimistic, it is still better than the $\mathrm{O}(m^2 n^2)$ bound for brute-force strong branching.

Considering the example shown in Figure 2.8(a), which is the same example shown in Section 2.3.1.1, let the current cutpoint be $(j, v) = (j_1, 0)$. Then, its down child is $(\underline{a}_{j_1}, \overline{a}_{j_1}, \underline{b}_{j_1}, \overline{b}_{j_1}) = (0, 0, 0, 0)$. Equivalence classes which are not covered by the down child are dropped from E as shown in Figure 2.8(b). For the same cutpoint, the up child is $(\underline{a}_{j_1}, \overline{a}_{j_1}, \underline{b}_{j_1}, \overline{b}_{j_1}) = (1, 1, 1, 1)$, so the algorithm drops different uncovered equivalence class from E as shown in Figure 2.8(c). As explained above, the up and down children's bounds sum to the parent's, so only one of the them is computed. The middle child for this cutpoint is $(\underline{a}_{j_1}, \overline{a}_{j_1}, \underline{b}_{j_1}, \overline{b}_{j_1}) = (0, 0, 1, 1)$. To compute the bound of this child, some equivalence classes must be merged, and these indices are adjacent in E when $j = j_1$. The merged equivalence class list $\hat{E}$ is shown in Figure 2.8(d). After computing bounds for all potential children created by the cutpoints for attribute j_1, the next step is to compute bounds for children in attribute j_2. After the stable bucket or counting sort on attribute j_1, all equivalence classes that might need to be merged to equivalence classes that might need to be merged to equivalence based on branching on attribute j_2 are adjacent in E. This transaction is illustrated in Figure 2.3.2.6. This


R
j3:
j2:
j1:
0
1
0
1,2
0
1,2
0
1
0
1
0
1
0
1
0,1
M : i1 i2 {i3, i5} {i4, i6} i7 i8 {i9, i11}{i10, i12}
E : e1 e2 e3 e4 e5 e6 e7 e8
M : {i1, i2} {i3, i4, i5, i6} {i7, i8} {i9, i10, i11, i12}
Ê : ê1 ê4 ê3 ê4

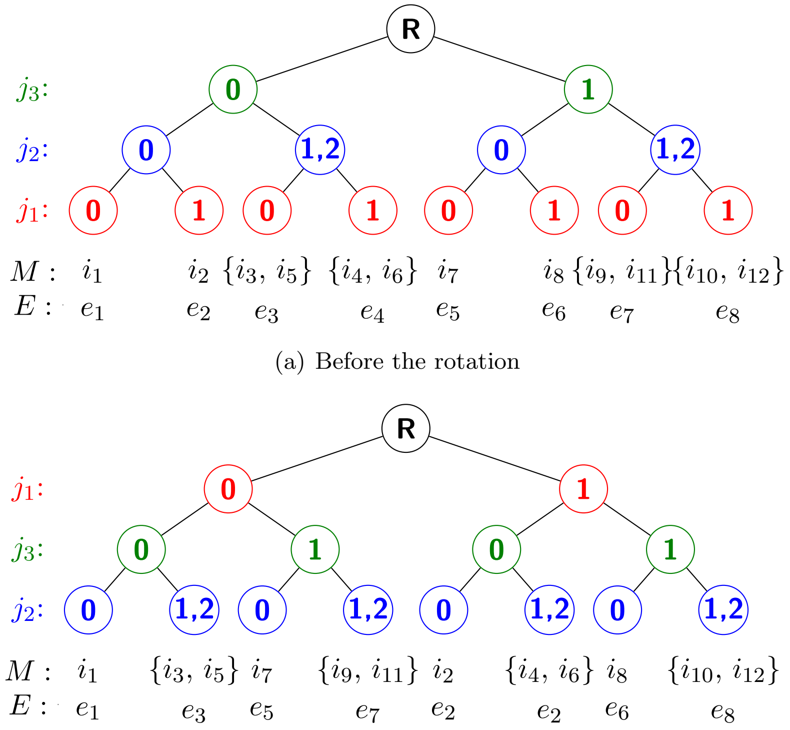
(a) Parent Equivalence Classes

(b) Dropping equivalence classes for down child

(c) Dropping equivalence classes for up child

(d) Merging equivalence classes for middle child

Figure 2.8: Graphical representation of dropping and merging equivalence classes


R
j_3:
0
1
j_2:
0
1,2
0
1,2
j_1:
0
1
0
1
0
1
0
1
M : i_1 i_2 $\{i_3, i_5\}$ $\{i_4, i_6\}$ i_7 i_8 $\{i_9, i_{11}\}$ $\{i_{10}, i_{12}\}$
E : e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8
R
j_1:
0
1
j_3:
0
1
0
1
j_2:
0
1,2
0
1,2
0
1,2
0
1,2
M : i_1 $\{i_3, i_5\}$ i_7 $\{i_9, i_{11}\}$ i_2 $\{i_4, i_6\}$ i_8 $\{i_{10}, i_{12}\}$
E : e_1 e_3 e_5 e_7 e_2 e_2 e_6 e_8

(a) Before the rotation

(b) Roted the rotation

Figure 2.9: Rotating equivalence-class tree

procedure then repeats for attributes $j_3, j_4, \cdots, j_n$ in order.

2.4 Incumbent computation for each subproblem

Throughout the branch-and-bound procedure, the incumbent, or current maximum objective value, has to be maintained. For each subproblem $P = (\underline{a}, \overline{a}, \underline{b}, \overline{b})$, the initial incumbent value is the objective value given by setting $(a, b) = (\underline{a}, \overline{b})$. Then, we employ a heuristic that attempts to improve on this value by introducing an additional restriction $(a_j, b_j) = (\hat{a}_j, \hat{b}_j)$ for one attribute j, where $\underline{a}_j \leq \hat{a}_j \leq \overline{a}_j$ and $\underline{b}_j \leq \hat{b}_j \leq \overline{b}_j$.

To efficiently find a range in each dimension which might improve the incumbent, our heuristic uses the minimization and maximization versions of the linear-time Kadane's continuous subarray sum algorithm, once for each attribute, and then selects the range constraint which improves the incumbent the most. Algorithm 4 shows the maximization version of Kadane's algorithm. The minimization version follows a similar procedure. For each attribute j, the algorithm tests the $\hat{\ell}_j = \overline{b}_j - \underline{a}_j + 1 + \max\{0, \underline{b}_j - \overline{a}_j\}$

Algorithm 4 Kadane's Largest Sum Continuous Subarray

```
1:  Input: $x_j \in \mathbb{R}^m$, $(\underline{a}_j, \overline{a}_j, \underline{b}_j, \overline{b}_j), \mathcal{E}$
    $E$ (currently covered equivalence-class indices sorted by attribute $j$)
2:  Output: $\hat{z} \in \mathbb{R}, l, u \in \mathbb{Z}$,
3:  MaxKadaneAlgo:

4:  $\hat{z} \leftarrow -\infty; \bar{z} \leftarrow 0; s \leftarrow \underline{a}_j$
5:  for $v \in \{\{\underline{a}_j, \ldots, \overline{b}_j\} \setminus \{\overline{a}_j + 1, \ldots, \underline{b}_j\}\}$ do
6:      $z \leftarrow$ getObjectiveValue$(v, x_j, E, \mathcal{E})$
7:      $\bar{z} \leftarrow \bar{z} + z$
8:      if $\bar{z} > \hat{z}$ then $\hat{z} \leftarrow \bar{z}$, $(l, u) \leftarrow (s, v)$
9:      if $\bar{z} < 0$ then $\bar{z} \leftarrow 0$, $s \leftarrow v + 1$
10: end for
11: return $\hat{z}, l, u$
```

distinct values that are not inseparable.

The **getObjectiveValue** function in line 6 of Algorithm 4 computes the objective value where $(a, b) = (\underline{a}, \overline{b})$ with an additional restriction of $a_j = b_j = v$ where $v \in \left\{\{\underline{a}_j, \ldots, \overline{b}_j\} \setminus \{\overline{a}_j + 1, \ldots, \underline{b}_j\}\right\}$. In each subproblem, a vector M, contains nonzero weight observation indices which are covered by $(\underline{a}, \overline{b})$. Computing the **getObjectiveValue** function takes $\mathrm{O}(|M|) \subseteq \mathrm{O}(m)$ time, since it only needs to scan M once to compute the objective value. However, its complexity can be improved to $\mathrm{O}(|E|) \subseteq \mathrm{O}(|M|) \subseteq \mathrm{O}(m)$ if Kadane's algorithm is properly embedded within the rotation algorithm. Since the total weight of each equivalence class is calculated for the bound computation and the equivalence classes are sorted by attribute j considering the inseparability of $(\overline{a}_j, \underline{b}_j)$ in each rotation step, as shown in Section 2.3.2.6, the **getObjectiveValue** function is computed by scanning the sorted equivalence class indices E once for each attribute. After running the minimum and maximum Kadane's algorithms, the incumbent is updated if the absolute value of the minimum or maximum objective value is better than the current incumbent. Therefore, the overall running time of Algorithm 4 improved to $\mathrm{O}(|E|)$ from a brute force computation of $\mathrm{O}(m\hat{L})$, where $\hat{L} = \max_j \hat{\ell}_j$, if Kadanes's algorithm is implemented within in the rotation algorithm. Since Kadane's algorithm is applied to n attributes, the total running time to compute a potential incumbent in each subproblem is $\mathrm{O}(n\,|E|) \subseteq \mathrm{O}(mn)$.

Algorithm 5 Procedure to Compute Bounds and Incumbent with the rotation

```
1: $M \leftarrow \{i \in \{1, 2, \dots m\} \mid w_i \neq 0\}$
2: for $j = 1, \dots, n$ do $M \leftarrow$ bucketSortObs($\underline{a}_j, \overline{a}_j, \underline{b}_j, \overline{b}_j, M, x_j$)
3: $(E, \mathcal{E}) \leftarrow$ createInitEquivClass($\underline{a}, \overline{a}, \underline{b}, \overline{b}, M, X$)
4: (incumbent, $a_n, b_n$) $\leftarrow$ checkIncumbent($n, E, \mathcal{E}, x_j, w$, incumbent)
5: for $j = 1, \dots, n$ do
6:     bound $\leftarrow$ boundComputation($j, \underline{a}_j, \overline{a}_j, \underline{b}_j, \overline{b}_j, E, \mathcal{E}, X$)
7:     if $j = n$ then break
8:     if $\underline{a}_j = \overline{a}_j$ and $\underline{b}_j = \overline{b}_j$ then continue
9:     $E \leftarrow$ bucketSortE($\underline{a}_j, \overline{a}_j, \underline{b}_j, \overline{b}_j, E, \mathcal{E}, x_j$)
10:    (incumbent, $a_j, b_j$) $\leftarrow$ checkIncumbent($j, E, \mathcal{E}, x_j, w$, incumbent)
11: end for
```

Algorithm 5 summarizes the procedures to compute bounds for all potential children and an incumbent using the rotation algorithm. The function **checkIncumbent** in lines 4 and 10 is the incumbent computation using the minimum and maximum Kadane's algorithms. It is implemented when the the equivalence class indices, E, is sorted on attribute j. Leveraging the sorted equivalence class indices, E, the incumbent computation for each attribute reduces to O($|E|$). Therefore, the total time to compute bounds for all children and search an incumbent per each subproblem is bounded by the time to compute bounds for all children, so it is O($mn^2 + m^2n$).

2.5 Greedy RMA Heuristic

For several reasons, we supplemented our exact branch-and-bound method for the RMA problem by developing a heuristic solution procedure we call "greedy range search". Such a heuristic has several uses: first, it can be used to obtain a good starting value of the incumbent in the branch-and-bound search, pruning the search tree and thus saving memory, as well as obviating possible non-essential work in parallel implementations.

We run our heuristic once to attempt to find a box with maximum (positive) covered weight, and then run it again to try to find a box with minimum (negative) covered weight. In each case, the heuristic starts with the box $B = (a, b) = (\mathbf{0}, \ell - \mathbf{1})$, covering the entire (preprocessed) dataset. Each iteration of the heuristic then greedily selects an attribute j to constrain, modifying the box lower bound a_j and upper bound b_j. To select j, a_j, and b_j at each iteration of our greedy heuristic, we make use of Kadane's algorithm, as given in Algorithm 4.

Consider the case in which we are attempting to maximize the covered weight of the box. We evaluate each possible attribute $j = 1, \dots, n$ and use Kadane's algorithm on an array of length ℓ_j to identify the choice of a_j and b_j maximizing $w(B)$. We select the attribute j^* that maximizes the resulting covered weight and modify a_{j^*} and b_{j^*} accordingly.

We then iteratively repeat this procedure, considering at each iteration every attribute j except the j^* just selected in the prior iteration (changing the bounds on this attribute cannot improve the objective function unless we first change some other attribute bounds). At each iteration, we consider only narrowing the bounds on the attributes (that is, increasing a_j and/or decreasing b_j). We continue in this manner until the covered weight of the box cannot be increased by narrowing the bounds of any individual attribute j.

The case of attempting to find the most negative covered box weight is similar, substituting minimization for maximization at each step.

Figure 2.10 shows an example of this greedy range search method. For each attribute j in each iteration, the shaded portion of each rectangle shows the attribute values having the greatest weight (assuming maximization). In the first iteration, we initially have all $m_1 = m$ non-zero-weight observations. We apply Kadane's algorithm to each attribute $j = 1, \dots, n$, obtaining a set of candidate objective values $z_1^1, \dots, z_n^1$. In the example, we suppose that the largest of these values (in the case of maximization) is $z^*_{\max} = z_2^1$; we then modify (a_2, b_2), with the result that the box covers some smaller number of observations $m_2 < m_1 = m$, and repeat the procedure. In the second iteration, we consider further constraining each box dimension except for $j = 2$, which we just modified. Supposing for the case of the example that the next attribute we constrain is $j = 1$ (resulting in $m_3 < m_2$ covered observations), the third iteration considers every attribute except $j = 1$; it is once again possible to consider constraining attribute 2.

Algorithm 6 summarizes, with some simplifications, the maximization instance of

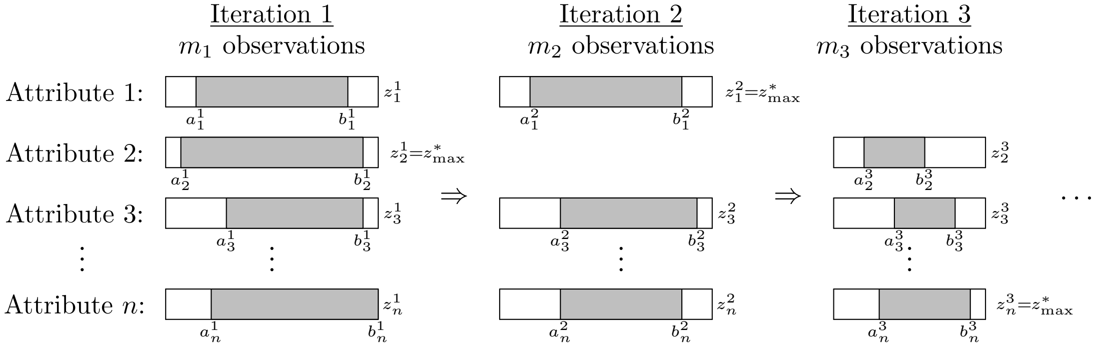


Figure 2.10: An example of the operation of our greedy heuristic.

our greedy range search procedure, assuming that the data have already been discretized as described in Section 2.1.1. The initial objective value is the sum of weights of all training observations, $\sum_{i=1}^{m} w_i$. The function call to **MaxKadane** on line 10 applies Kadane's algorithm to the portion of the working array W between elements a_j and b_j, returning the maximum contiguous subarray sum in z and extent (l, u) of the corresponding subarray, where $a_j \leq l \leq u \leq b_j$. The subroutine **dropObsNotCovered** invoked on line 15 removes from the list of indices M any observations that are excluded if the bounds of the box B are narrowed to (l^*, u^*) in attribute j^*. The actual implementation is somewhat more complicated because it effectively compresses the working array W to represent only values of each attribute j that remain represented in the set of observations M.

Examining the complexity of Algorithm 6, the highest-complexity steps in the body of the main **for** loop on lines 6-12 are lines 8-10; lines 8 and 10 has complexity $\mathrm{O}(b_j\text{–}a_j) \subseteq \mathrm{O}(\ell_j) \subseteq \mathrm{O}(m)$, and line 9 has complexity $\mathrm{O}(|M|) \subseteq \mathrm{O}(m)$. It follows that each iteration of the main **for** loop requires $\mathrm{O}(m)$ time. This **for** loop iterates over $\{1, \ldots, n\} \setminus j^*$, so each execution of the loop requires $\mathrm{O}(mn)$ time. Finally, each iteration of the **while** loop except the last must remove at least one observation from the covered set M, since otherwise there could no objective improvement and the **break** condition would trigger to terminate the loop. The call to **dropObsNotCovered** on line 15 can be implemented in $\mathrm{O}\big(|M|\big) \subseteq \mathrm{O}(m)$ time, so it follows that the **while** loop will execute

Algorithm 6 Maximum greedy range search

```
**Input:** $n \in \mathbb{R}$, $\ell \in \mathbb{N}^n$ (the number of distinct values in each attribute),
$X \in \mathbb{R}^{m \times n}$ (data), $w \in \mathbb{R}^m$ (observation weights)
**Output:** $z^*_{\max} \in \mathbb{R}, a, b \in \mathbb{N}^n$
**MaxGreedyRange**$(n, \ell, w, X)$:

$z^*_{\max} \leftarrow \sum_{i=1}^{m} w_i; j^* \leftarrow -1; (a, b) \leftarrow (\mathbf{0}, \ell - \mathbf{1}); M = \{1, \dots, m\}$; allocate $W \in \mathbb{R}^{\max\{\ell_1, \dots, \ell_n\}}$
**while** true **do**
  **for** $j \in \{1, \dots, n\} \setminus j^*$ **do**
    $z_{\max} \leftarrow z^*_{\max}$
    **for** $v = a_j, \dots, b_j$ **do** $W[v] = 0$
    **for** $i \in M$ **do** $W[x_{ij}] \leftarrow W[x_{ij}] + w_i$
    $(z, l, u) \leftarrow$ **MaxKadane**$(W, a_j, b_j)$
    **if** $z > z_{\max}$ **then** $(j^*, z_{\max}, l^*, u^*) \leftarrow (j, z, l, u)$
  **end for**
  **if** $z^*_{\max} \geq z_{\max}$ **then break**
  $a_{j^*} \leftarrow l^*; b_{j^*} \leftarrow u^*; z^*_{\max} \leftarrow z_{\max}$
  $M \leftarrow$ **dropObsNotCovered**$(j^*, l^*, u^*, M, x_{j^*})$
**end while**
**return** $(z^*_{\max}, a, b)$
```

at most m times and that the overall run time is $\mathrm{O}(m^2 n)$, although this is a very loose bound. This polynomial complexity contrasts with the potentially exponential run time of the branch-and-bound procedure.

2.6 Implementing the branch-and-bound algorithm

We implemented our branch-and-bound algorithm using PEBBL [13], an open-source C++ framework for branch and bound. PEBBL makes it relatively straightforward to move from a serial to a parallel implementation. Appendix A is a user guide for the RMA code.

2.6.1 Parallel RMA

In the initial phase of a parallel search, when the number of cutpoints exceeds the number of active search nodes of the branch-and-bound tree (the number of subproblems in the pool used by PEBBL) and the number of active search nodes is less than the

Algorithm 7 Selecting one cutpoint randomly among the optimal tied cutpoints in parallel Allreduce operation

```
Input: $(B_p, I_p, j_p, v_p)$, $(B_q, I_q, j_q, v_q)$
Output: $(B_q, I_q, j_q, v_q)$
ParallelRandomBestCutointSelection:

if $B_p < B_q$ then
    $(B_q, I_q, j_q, v_q) \leftarrow (B_p, I_p, j_p, v_p)$
else if $B_p = B_q$ then
    Generate $r$ (a uniform random number between 0 and 1)
    if $r < \frac{I_p}{I_p + I_q}$ then
        $(B_q, I_q, j_q, v_q) \leftarrow (B_p, I_p + I_q, j_p, v_p)$
    else
        $I_q \leftarrow I_p + I_q$
    end if
end if
return $(B_q, I_q, j_q, v_q)$
```

number of processors, we take advantage of PEBBL's ability to support special ramp-up search procedures. At the beginning of the search, PEBBL can exploit parallelism within each subproblem, rather than across the tree: all the processors synchronously search an identical sequence of search nodes. As our implementation explores each subproblem, it distributes a nearly equal number of cutpoints to each processor as shown Figure 2.11. The cutpoints are sorted by attribute, so the rotation algorithm can compute bounds efficiently. This tactic efficiently parallelizes the branching selection procedure, the most time-consuming aspect of branch-and-bound algorithm.

In the ramp-up procedure, if the random branching option is selected, the best cutpoint to branch is chosen randomly among ones with optimal tied bounds for the subproblem. This choice requires implementing the random tiebreaking procedure of Section 2.3.2.1 in a distributed manner. To this end, each processor first randomly chooses one cutpoint among the optimal tied ones encountered using the algorithm in Section 2.3.2.1. For each processor p, suppose B_p contains selected optimal bound values for 2 or 3 children created by a cutpoint (j_p, v_p) in processor p. Let I_p be the number of the optimal tied cutpoints B_p found in processor p. We combine the tuples (B_p, I_p, j_p, v_p) from each processor within an MPI_Allreduce operation using the

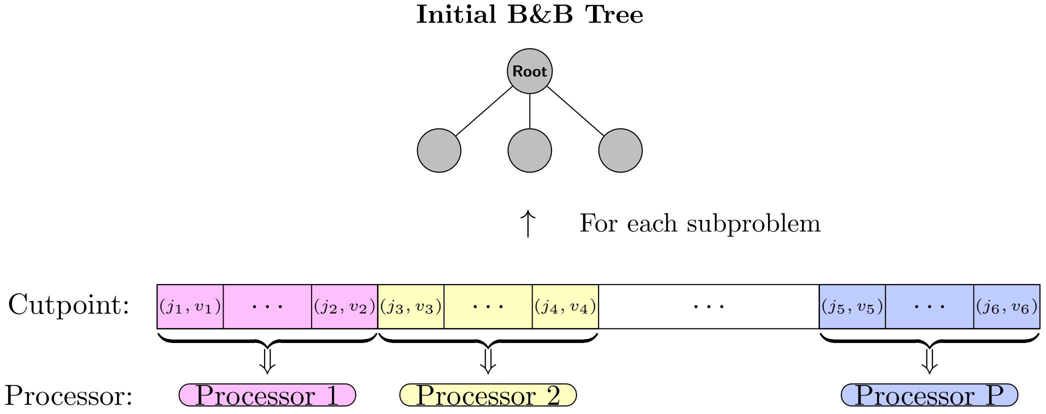


Figure 2.11: Ramp-up Search process

customized random operation described in Algorithm 3. The "<" symbol on line 4 of algorithm 7 denotes the lexicographic comparison of the vectors B_p and B_q, as explained in Section 2.3.2.1. If $B_p = B_q$, the probability such that the cutpoint from processor p is chosen is $\dfrac{I_p}{I_p + I_q}$. The MPI_Allreduce operation within Algorithm 7 takes O($\log P$) time, where P is the number of procedures.

We trigger PEBBL's "crossover" to its standard asynchronous search mode when the number of nodes in active subroblem pool becomes comparable to the number of processors or exceeds the number of possible cutpoints. After crossover, PEBBL asynchronously searches multiple nodes of the search tree in parallel, with individual bounding and branching operations being handled by a single processor. To implement cutpoint caching in asynchronously search phase, we developed a method to broadcast newly discovered cutpoints to other processors with relatively little redundancy. It utilizes a hashing procedure to assign an "owning" processor to each cutpoint pair (j, v). Whenever a processor finds an apparently new cutpoint, it sends it to the owning processor. If the owning processor has not encountered the pair before, it broadcasts it to all processors. In our current implementation with the cutpoint caching option, cutpoints already chosen for branching are stored in cache during the ramp-up process. However, only strong branching is implemented until the asynchronous search mode begins. If cutpoint caching were to be used during ramp-up, the number of available processors might be far greater than the number of applicable cutpoints in cache, and

(a) Binary datasets

Datasets	m	n
hungheart	294	72
cleveland	297	35
diabetes	768	33
cmc_bin	1,473	57
spam_bin	4,601	40
spam75_bin	4,601	73

(b) Integer datasets

Datasets	m	n	c
parkinson	195	22	1,028
climate	540	18	802
indian	583	10	628
breast	683	9	80
cmc_int	1,473	9	56
car	1,728	6	15
chess	3,196	36	37
EEG	14,980	14	203
credit_card	30,000	23	1,931
skin	245,057	3	765
poker	1,000,000	10	85

Table 2.5: Summary of datasets

Method	Description
RMA	RMA with the rotation method and strong branching
RMA_CC	RMA with the rotation method and cutpoint caching
RMA_BS	RMA with the rotation method binary cutpoint search
RMA_HB	RMA with the rotation method and hybrid branching
RMA_BF	RMA with a brute force algorithm to create equivalence classes and strong branching
MMA	Maximum Monomial Agreement solver by Goldberg [12]
MIP	The MIP formulation solved by Gurobi

Table 2.6: Method descriptions

many processors would be idle. A possible improvement to the current approach might be that when the number of applicable cutpoints is greater than the number of processors, cutpoint caching could be adopted during ramp-up. Moreover, binary search cutpoint is only applied during asynchronous search.

2.7 Computational results for RMA

2.7.1 Datasets and Methods

Table 2.5 summarizes both binarized and integerized datasets derived from the UCI data repository [**?**]. Table 2.5(a) shows binarized datasets from [12], and Boros *et al.* [4] explains the binarization process. Table 2.5(b) shows discretized datasets using the algorithm in Section 2.1.1. For the integerized datasets, the last column of this table shows c, the total number of cutpoints. For the binarized datasets, $c = n$.

Table 2.6 summarizes the methods to solve MMA and RMA problems.

2.7.2 Serial run time improvement from cutpoint caching

Tables 2.7 and 2.8 and figures 2.12 and 2.13 show the running time and the number of bounded subproblems for the RMA algorithm with the different cutpoint caching threshold levels and with the strong branching method. If the cutpoint caching threshold τ is 100%, the algorithm is same as the strong branching method. In tables 2.7 and 2.8, SB, CC, and SP respectively denote the number of bounded subproblems using the strong branching method, the number of bounded subproblems using cutpoints from the cache, and the total number of bounded subproblems, respectively, so SP = SB + CC. "%CC" indicates the percentage of the bounded subproblems using cutpoints from the cache, so %CC = CC / SP. Generally, the running time of the algorithm decreases for both binarized and integerized datasets as the cutpoint caching threshold decreases. Therefore, it is effective to compute bounds only for applicable cutpoints from the cache if there is at least one cached cutpoint applicable to the subproblem. Figures 2.12 and 2.13 show the number of subproblems using strong branching in blue and the number of subproblems using cutpoints from the cache in pink. The total number of the bounded subproblems is not sensitive to the cutpoint caching threshold level, except for the *indian* and *skin* datasets. For *indian* and *skin*, the running time and the total bounded subproblems significantly decrease as the cutpoint caching threshold level decreases. A partial explanation for this behavior may be that the global bound of the branch-and-bound algorithm may improve differently based on the choice of the cutpoint selected among the optimal tied cutpoints, and choosing from the applicable cached cutpoints may improve the global bound faster. Since the running time and the total bounded subproblems fluctuate remarkably for some datasets when randomly choosing the best cutpoint among the optimal tied ones, the results in this section are given using the tactic of selecting the first optimal cutpoint discovered as the best one to branch for more stable results.

(a) cleveland

τ(%)	Time	SB	CC	SP	%CC
100	0.5	165	0	165	0.0%
30	0.5	79	81	160	50.6%
20	0.5	47	104	151	68.9%
10	0.5	34	128	162	79.0%
5	0.5	28	136	164	82.9%
1	0.5	30	166	196	84.7%

(b) diabetes

τ(%)	Time	SB	CC	SP	%CC
100	1.1	175	0	175	0.0%
30	1.1	84	93	177	52.5%
20	1.1	55	128	183	69.9%
10	1.0	40	127	167	76.0%
5	1.0	37	156	193	80.8%
1	1.0	27	175	202	86.6%

(c) hungheart

τ(%)	Time	SB	CC	SP	%CC
100	3.5	446	0	446	0.0%
30	3.3	278	164	442	37.1%
20	2.7	184	205	389	52.7%
10	2.5	89	313	402	77.9%
5	2.1	64	302	366	82.5%
1	2.0	54	305	359	85.0%

(d) cmc_bin

τ(%)	Time	SB	CC	SP	%CC
100	6.2	339	0	339	0.0%
30	6.1	330	9	339	2.7%
20	5.8	293	46	339	13.6%
10	4.9	152	187	339	55.2%
5	4.4	76	266	342	77.8%
1	4.1	28	313	341	91.8%

(e) spam

τ(%)	Time	SB	CC	SP	%CC
100	5.8	177	0	177	0.0%
30	5.7	148	29	177	16.4%
20	5.4	92	84	176	47.7%
10	4.8	42	127	169	75.1%
5	4.6	31	134	165	81.2%
1	5.2	27	172	199	86.4%

(f) spam75

τ(%)	Time	SB	CC	SP	%CC
100	154.8	2,763	0	2,763	0.0%
30	145.6	2,052	711	2,763	25.7%
20	138.8	1,504	1,259	2,763	45.6%
10	135.5	612	2,103	2,715	77.5%
5	129.8	182	2,515	2,697	93.3%
1	130.2	55	2,777	2,832	98.1%

Table 2.7: Serial run time improvement using the cutpoint caching method for binary datasets

(a) breast

τ(%)	Time	SB	CC	SP	%CC
100	0.6	185	0	185	0.0%
30	0.6	177	8	185	4.3%
20	0.6	141	47	188	25.0%
10	0.5	90	97	187	51.9%
5	0.5	62	126	188	67.0%
1	0.4	42	137	179	76.5%

(b) chess

τ(%)	Time	SB	CC	SP	%CC
100	1.8	51	0	51	0.0%
30	1.7	47	4	51	7.8%
20	1.6	39	12	51	23.5%
10	1.5	35	18	53	34.0%
5	1.3	29	19	48	39.6%
1	1.3	25	30	55	54.5%

(c) cmc_int

τ(%)	Time	SB	CC	SP	%CC
100	1.0	203	0	203	0.0%
30	1.0	201	2	203	1.0%
20	1.0	195	8	203	3.9%
10	0.8	122	73	195	37.4%
5	0.7	73	130	203	64.0%
1	0.7	20	189	209	90.4%

(d) parkinson

τ(%)	Time	SB	CC	SP	%CC
100	71.0	3,124	0	3,124	0.0%
20	66.6	2,582	573	3,155	18.2%
10	50.7	1,510	1,300	2,810	46.3%
5	43.5	1,179	1,443	2,622	55.0%
1	41.5	1,027	1,569	2,596	60.4%
0.1	40.4	948	1,603	2,551	62.8%

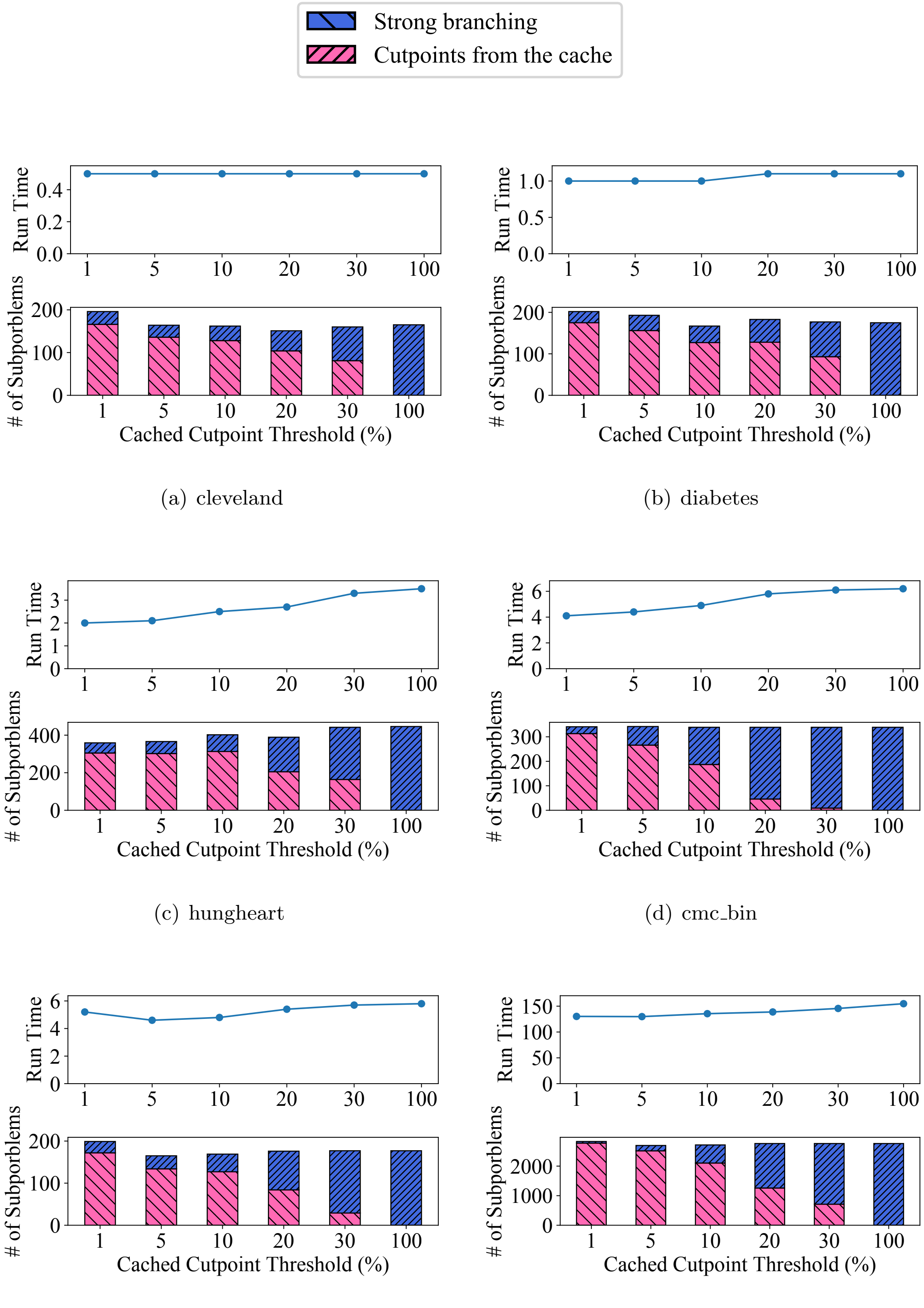
(e) indian

τ(%)	Time	SB	CC	SP	%CC
100	1,826.1	72,072	0	72,072	0.0%
20	1,946.5	59,911	12,244	72,155	17.0%
10	1,243.6	30,082	42,114	72,196	58.3%
5	671.0	8,527	63,698	72,225	88.2%
1	240.9	527	61,355	61,882	99.1%
0.1	135.3	224	40,152	40,376	99.4%

(f) skin

τ(%)	Time	SB	CC	SP	%CC
100	1,946.1	992	0	992	0.0%
10	1,602.4	831	137	968	14.2%
5	1,172.9	664	244	908	26.9%
1	393.5	264	451	715	63.1%
0.5	133.4	108	226	334	67.7%
0.1	66.6	27	248	275	90.2%

Table 2.8: Serial run time improvement using the cutpoint caching method for integer datasets



(a) cleveland

(b) diabetes

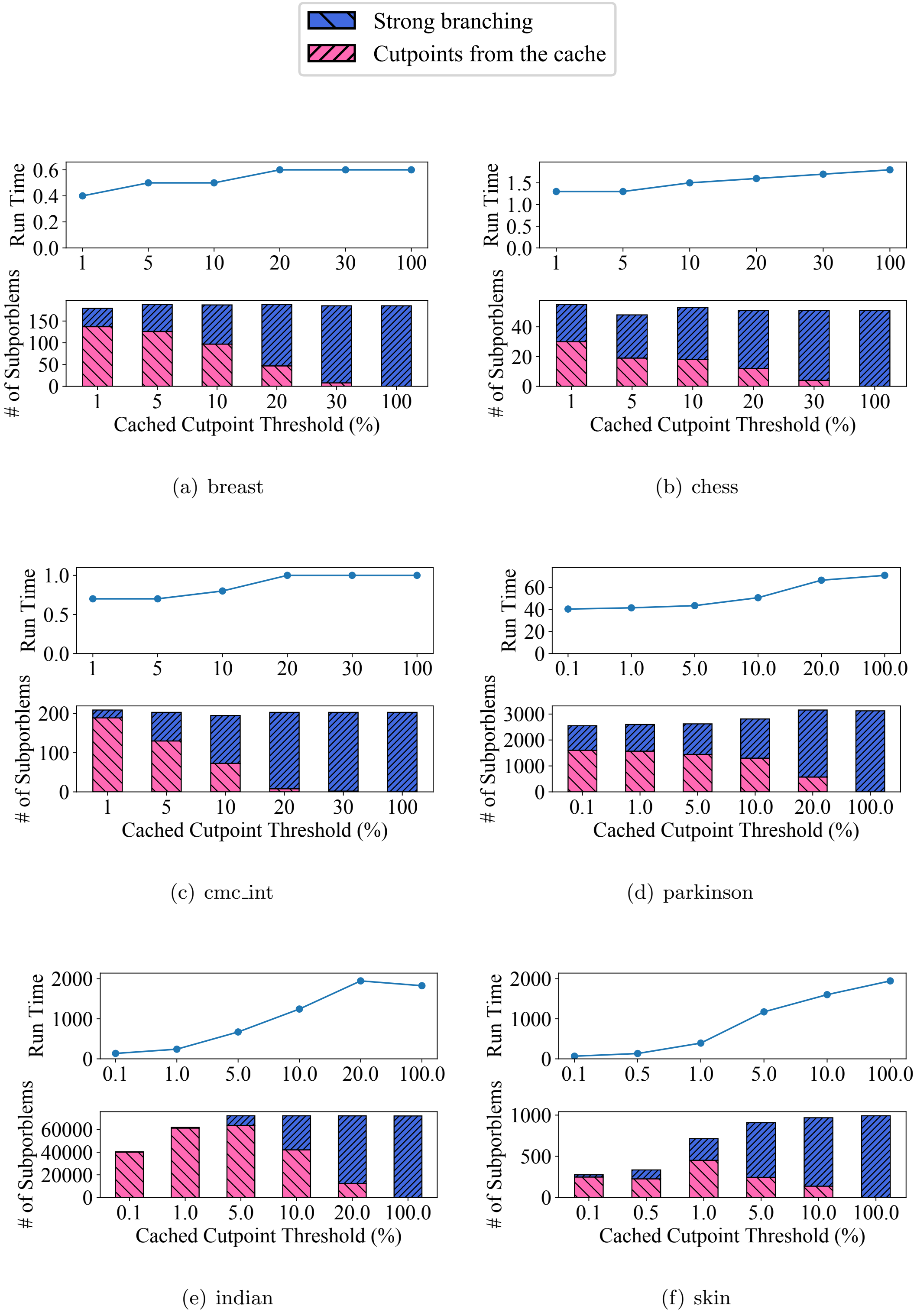
(c) hungheart

(d) cmc_bin

(e) spam

(f) spam75

Figure 2.12: Run time and the number of the bounded subproblems using different cutpoint caching thresholds, for the binarized datasets.



(a) breast

(b) chess

(c) cmc_int

(d) parkinson

(e) indian

(f) skin

Figure 2.13: Run time and the number of the bounded subproblemsusing different cutpoint caching thresholds, for the integerized datasets.

Method	*P*	Datasets					
		cleveland	diabetes	hungheart	cmc_bin	spam	spam75
RMA	1	**0.5**	1.1	3.5	6.2	5.9	154.8
RMA_CC	1	**0.5**	**1.0**	**2.0**	**4.1**	**5.2**	**130.2**
RMA_BF	1	11.2	70.5	310.0	1,104.8	2,053.9	—
MMA	1	20.5	43.6	147.3	909.6	554.4	18,966.0
MIP	1	8.5	14.8	11.5	199.8	276.6	829.4
RMA	16	**0.1**	0.3	**0.6**	1.0	2.0	13.6
RMA_CC	16	**0.1**	**0.2**	**0.6**	**0.8**	**1.2**	**11.6**
RMA_BF	16	1.7	6.9	23.1	146.7	220.4	—
MMA	16	2.7	5.0	14.2	86.4	56.2	1,533.0
MIP	16	4.7	14.3	10.0	86.5	218.5	584.4

Table 2.9: Run time in seconds for binary datasets

Method	*P*	Datasets					
		cleveland	diabetes	hungheart	cmc_bin	spam	spam75
RMA	1	**165**	175	446	339	177	2,763
RMA_CC	1	196	202	**359**	341	199	2,832
RMA_BF	1	**165**	175	446	339	177	—
MMA	1	218	141	404	**292**	**143**	2,450
MIP	1	446	**83**	418	943	151	**221**
RMA	16	203	185	421	371	177	2,809
RMA_CC	16	266	180	588	444	188	3,004
RMA_BF	16	**198**	182	**400**	378	177	—
MMA	16	263	**142**	417	**305**	**143**	2,493
MIP	16	573	151	525	985	253	**219**

Table 2.10: The number of bounded subproblems for binary datasets

2.7.3 Comparing algorithms to solve MMA problem

Tables 2.9 and 2.10 respectively show the running time in seconds and the total bounded subproblems for branch-and-bound methods to solve MMA problems, in which the explanatory variables are all binary. The *P* column shows the number of processors. The results of all variations of parallel RMA and MMA are the average of 5 runs. Due to the lack of significant runtime differences between the different choices of the best cutpoint to branch among the optimal tied ones for the binary datasets, tables 2.9 and 2.10 show the results selecting the first optimal cutpoint discovered. The MIP formulation was shown in Section 2.2. The Gurobi MIP solver with AMPL [16], an algebraic modeling language, was used to solve the MIP formulation. The results were obtained using a system with 2.10GHz Intel Xeon E5-2683 v4 CPUs. The running time of the serial RMA with the rotation algorithm is much faster, more than 100 times faster for some datasets, than serial MMA. Moreover, adopting the rotation algorithm for the RMA algorithm greatly improved the running time comparing to the RMA algorithm with brute-force construction of equivalence classes for each cutpoint in each subproblem. RMA_CC, that is RMA with the rotation algorithm and cutpoint caching, further improved the running time. The results for all the methods to solve the MMA problems using 16 cores for the binary datasets are also shown in tables 2.9 and 2.10. MMA and all variations of RMA are implemented using the PEBBL's built-in parallel search capabilities. For all binary datasets, RMA is always faster than Gurobi both in serial and parallel. The speedup of Gurobi utilizing multiple processors is far from the linear.

2.7.4 Comparing algorithms to solve RMA problem

Tables 2.11 and 2.12 show the running time and the total bounded subproblems to solve the RMA problems by Gurobi and all variations of RMA. Here, P denotes the number of processors. In tables 2.11 and 2.12, "_R", "_F", and "_L" extensions after each method indicate choosing the cutpoint randomly, selecting the first cutpoint, and choosing the last cutpoint among the optimal tied ones, respectively. The default setting is random choice, so if the procedure is not indicated, the results are for that method. For four out of seven datasets, Gurobi has a significantly smaller number of bounded subproblems than RMA due to its use of cutting plane methods. However, the speedup of Gurobi in parallel is not close to linear. Even though Gurobi can solve the problems faster than RMA for some datasets in serial, RMA utilizing 16 processors is faster than Gurobi using the same number of processors, for all datasets except *parkinson*. Adopting the cutpoint caching method speeds up the running time of RMA for the majority of the datasets. The results show that the cutpoint caching method does not always work well in parallel, and the binary search cutpoint method sometimes slows down the run time since it expands the branch-and-bound tree. The hybrid search branching method only worked well for the *climate* dataset when $\eta = 30$. For the *indian* dataset, the running time and the number of bounded subproblems are significantly different depending on the best cutpoint selection among the optimal tied ones, and the result varies with the randomization method. Gurobi was not able to solve *skin* instance,

		Datasets						
Method	P	car	breast	cmc_int	climate	indian	parkinson	skin
RMA_F	1	**0.0**	0.7	1.0	38.6	1,826.1	71.0	1,978.8
RMA_R	1	**0.0**	0.8	1.1	24.4	541.5	68.0	2,028.7
RMA_L	1	**0.0**	0.8	1.1	38.0	136.9	68.0	1,967.4
RMA_CC_F	1	**0.0**	**0.4**	**0.7**	34.5	120.9	40.4	65.9
RMA_CC_R	1	**0.0**	**0.4**	**0.7**	22.4	38.7	32.1	**59.6**
RMA_CC_L	1	**0.0**	**0.4**	**0.7**	34.8	20.8	33.8	**59.6**
RMA_BS	1	—	—	—	17.3	—	247.0	305.1
RMA_HB	1	—	—	—	12.8	—	—	—
MIP	1	12.9	0.5	63.5	**5.9**	**16.4**	**5.9**	—
RMA	16	**0.0**	0.2	**0.2**	**4.1**	35.8	8.1	221.4
RMA_CC	16	**0.0**	**0.1**	**0.2**	35.4	**3.6**	22.2	**26.7**
RMA_BS	16	—	—	—	4.2	213.7	8.9	36.0
MIP	16	4.5	0.3	12.1	16.0	15.5	**6.3**	—

Table 2.11: Run time in seconds for integer datasets

		Datasets						
Method	P	car	breast	cmc_int	climate	indian	parkinson	skin
RMA_F	1	**4**	185	**203**	943	72,072	3,124	992
RMA_R	1	**4**	243	205	605	19,903	3,273	992
RMA_L	1	**4**	215	**203**	883	4,385	2,869	991
RMA_CC_F	1	**4**	182	209	980	40,376	2,551	**275**
RMA_CC_R	1	**4**	179	212	652	14,427	3,165	278
RMA_CC_L	1	**4**	169	**203**	883	6,830	5,715	292
RMA_BS	1	—	—	—	743	—	12,961	1,379
RMA_HB	1	—	—	—	1,152	—	—	—
MIP	1	15	**1**	363	**1**	**453**	**5**	—
RMA	16	**5**	246	**304**	728	16,062	3,485	1,896
RMA_CC	16	**5**	276	330	1,107	10,504	8,103	**398**
RMA_BS	16	—	—	—	885	396,775	4,173	1,324
MIP	16	15	**1**	702	**1**	**348**	**4**	—

Table 2.12: The number of subproblems for integer datasets

possibly because of the large size of the corresponding MIP model, but all versions of RMA using tree rotation were able to solve it. Using cutpoint caching and binary search cutpoint methods significantly improved the running time for *skin*.

There are several reasons that directly solving RMA problem instances may be more efficient than binarizing datasets and the solving them as MMA instances. First, although the running time of binarization and integerization processes are not discussed here, the process of integerizing raw data is much faster than binarizing. Second, it appears to be significantly faster to solve integerized MMA instances than the binarized MMA instances obtained from the same data, in part possibly because the representation of data is much smaller with integerization than with binarization. For example, the binarized *cmc* dataset has 57 attributes but the integerized *cmc* dataset has only 9 attributes. While both RMA and MMA algorithms obtain the exact same objective value, the RMA algorithm is much faster for the integerized dataset than the binarized dataset.

2.7.5 Result for Parallel RMA

Tables 2.13-2.15 and figures 2.14 and 2.15 show the running time and the total bounded subproblems for RMA and RMA_CC. The abbreviation “SU” in tables 2.13-2.15 indicates the speedup calculated as $T_{\bar{P}}\bar{P}/T_P$, where $\bar{P}$ is the smallest number of processors on which the problem instance could be solved. When $\bar{P} \neq 1$, it is shown in parentheses in the “SU” column. In Figure 2.14, the left-side data are from an older version of the RMA algorithm storing all observations in each equivalence class, and with a minor difference in the bound computation process. The right-side data are for the RMA algorithm storing only one observation in each equivalence class. The second method improved running times, but its parallel speedup is degraded. Moreover, the number of bounded subproblems for RMA_CC fluctuates markedly as one varies the number of processors. One possible reason for this behavior is that the number of subproblems using strong branching during ramp-up varies with the number of processors, causing the cutpoint cache to be initialized differently. As shown in tables 2.13-2.15, the percentage of subproblems using only cutpoints from the cache decreases as the number of processors increases. Another possible reason is the variability of communication delay when distributing cutpoints between processors. Figure 2.15 shows that datasets with larger numbers of cutpoints and bounded subproblems have better parallel speedup. The reason that the number of bounded subproblems increases when using greater number of processors for the *poker* dataset is currently unknown and may require future investigation.

(a) climate (previous version of RMA_F)

	Strong Branching		
P	Time	SU	SB
$2^0 = 1$	91.5	—	943
$2^1 = 2$	47.4	1.9	940
$2^2 = 4$	24.6	3.7	907
$2^3 = 8$	13.1	7.0	837
$2^4 = 16$	6.4	14.3	725

(b) climate (RMA_F)

	Strong Branching		
P	Time	SU	SB
$2^0 = 1$	33.8	—	943
$2^1 = 2$	19.1	1.8	941
$2^2 = 4$	10.8	3.1	915
$2^3 = 8$	6.3	5.4	840
$2^4 = 16$	3.8	8.9	728

(c) climate (RMA_R)

	Strong Branching			Cutpoint Caching					
P	Time	SU	SP	Time	SU	SP	CC	SB	%CC
$2^0 = 1$	22.2	—	635	19.3	—	678	387	292	43.0%
$2^1 = 2$	14.8	1.5	745	18.4	1.0	651	411	240	36.8%
$2^2 = 4$	12.1	1.8	1,191	17.0	1.1	833	444	389	46.7%
$2^3 = 8$	12.1	1.8	2,149	19.0	1.0	1,078	527	551	51.1%
$2^4 = 16$	17.9	1.2	4,041	20.5	0.9	816	548	267	32.8%
$2^5 = 32$	14.6	1.5	4,115	19.0	1.0	804	555	249	31.0%

(d) parkinson (previous version of RMA_F)

	Strong Branching		
P	Time	SU	SB
$2^0 = 1$	174.0	—	4,311
$2^1 = 2$	75.2	2.3	4,100
$2^2 = 4$	40.1	4.3	4,360
$2^3 = 8$	25.2	6.9	4,743
$2^4 = 16$	18.4	9.4	6,401

(e) parkinson (RMA_F)

	Strong Branching		
P	Time	SU	SB
$2^0 = 1$	62.4	—	3,124
$2^1 = 2$	33.8	1.8	3,034
$2^2 = 4$	24.2	2.6	4,306
$2^3 = 8$	12.0	5.2	3,887
$2^4 = 16$	9.5	6.6	3,629

(f) parkinson (RMA_R)

	Strong Branching			Cutpoint Caching					
P	Time	SU	SP	Time	SU	SP	CC	SB	%CC
$2^0 = 1$	62.2	—	3,392	30.7	—	4,154	665	3,489	84.0%
$2^1 = 2$	38.2	1.6	3,715	18.0	1.7	2,901	767	2,134	73.6%
$2^2 = 4$	22.8	2.7	4,112	14.8	2.1	3,823	887	2,936	76.8%
$2^3 = 8$	19.0	3.3	5,435	14.0	2.2	4,133	939	3,194	77.3%
$2^4 = 16$	19.8	3.1	8,720	15.0	2.0	4,896	1,171	3,725	76.1%
$2^5 = 32$	16.4	3.8	10,910	5.8	5.3	22,468	2,969	19,499	86.8%

Table 2.13: Parallel speedup results for the *climate* and *parkinson* datasets

(a) indian (previous version of RMA_F)

	Strong Branching			Cutpoint Caching		
P	Time	SU	SP	Time	SU	SP
$2^0 = 1$	3,980.5	—	72,251	395.0	—	61,882
$2^2 = 4$	1,004.8	4.0	72,072	128.6	3.1	42,909
$2^3 = 8$	535.9	7.4	68,938	62.7	6.3	38,822
$2^4 = 16$	272.0	14.6	62,463	40.1	9.9	46,944
$2^5 = 32$	113.8	35.0	53,146	24.5	16.1	44,952
$2^6 = 64$	56.8	70.1	52,737	10.6	37.1	31,742
$2^7 = 128$	29.7	134.2	51,863	7.3	54.1	31,598
$2^8 = 256$	16.1	247.2	51,301	—	—	—
$2^9 = 512$	11.2	355.4	51,016	—	—	—

(b) indian (RMA_F)

	Strong Branching			Cutpoint Caching		
P	Time	SU	SP	Time	SU	SP
$2^0 = 1$	1,826.1	—	72,072	121.1	—	40,376
$2^1 = 2$	939.3	1.9	72,195	42.1	2.9	34,683
$2^2 = 4$	464.4	3.9	72,167	18.6	6.5	28,805
$2^3 = 8$	247.8	7.4	69,148	14.2	8.5	40,475
$2^4 = 16$	140.1	13.0	62,532	8.4	14.4	35,474
$2^5 = 32$	57.3	31.8	52,575	5.1	23.7	46,096
$2^6 = 64$	28.9	63.2	52,201	3.1	38.8	42,342
$2^7 = 128$	15.3	119.2	52,030	2.5	49.2	42,277
$2^8 = 256$	8.7	210.9	51,588	—	—	—
$2^9 = 512$	5.4	335.7	51,233	—	—	—

(c) indian (RMA_R)

	Strong Branching			Cutpoint Caching					
P	Time	SU	SP	Time	SU	SP	CC	SB	%CC
$2^0 = 1$	561.4	—	23,638	31.7	—	12,598	95	12,503	99.3%
$2^1 = 2$	453.2	1.2	36,690	14.1	2.3	9,645	154	9,491	98.4%
$2^2 = 4$	185.2	3.0	30,256	11.7	2.7	18,121	223	17,898	98.8%
$2^3 = 8$	70.5	8.0	21,532	4.9	6.4	10,738	349	10,390	96.8%
$2^4 = 16$	78.8	7.1	38,425	4.7	6.7	18,106	663	17,443	96.3%
$2^5 = 32$	47.1	11.9	47,373	3.1	10.2	19,730	1,129	18,601	94.3%

Table 2.14: Parallel speedup results for the *indian* dataset

(a) skin (RMA_F)

	Strong Branching			Cutpoint Caching		
P	Time	SU	SP	Time	SU	SP
$2^0 = 1$	1822.7	—	992	59.9	—	275
$2^1 = 2$	949.8	1.9	992	47.9	1.3	267
$2^2 = 4$	471.1	3.9	992	35.5	1.7	316
$2^3 = 8$	283.1	6.4	992	22.4	2.7	330
$2^4 = 16$	223.9	8.1	992	26.2	2.3	512.2
$2^5 = 32$	197.0	9.3	992	25.4	2.4	514

(b) credit_card (RMA_F)

	Strong Branching			Cutpoint Caching		
P	Time	SU	SP	Time	SU	SP
$2^0 = 1$	—	—	—	29,948.2	—	182,471
$2^1 = 2$	—	—	—	13,456.1	2.2	138,601
$2^2 = 4$	—	—	—	8,906.6	3.4	203,202
$2^3 = 8$	—	—	—	3,971.9	7.5	141,577
$2^4 = 16$	—	—	—	2,563.9	11.7	141,748
$2^5 = 32$	—	—	—	1,634.3	18.3	129,646
$2^6 = 64$	17,753.0	—	261,511	926.9	32.3	157,871
$2^7 = 128$	8,972.6	1.9 (2)	262,671	770.0	38.9	135,546
$2^8 = 256$	4,561.0	3.9 (4)	265,378	616.0	48.6	118,910
$2^9 = 512$	2,456.8	7.2 (8)	268,482	568.0	52.7	113,618
$2^{10} = 1024$	1,420.3	12.5 (16)	267,087	519.9	57.6	123,369

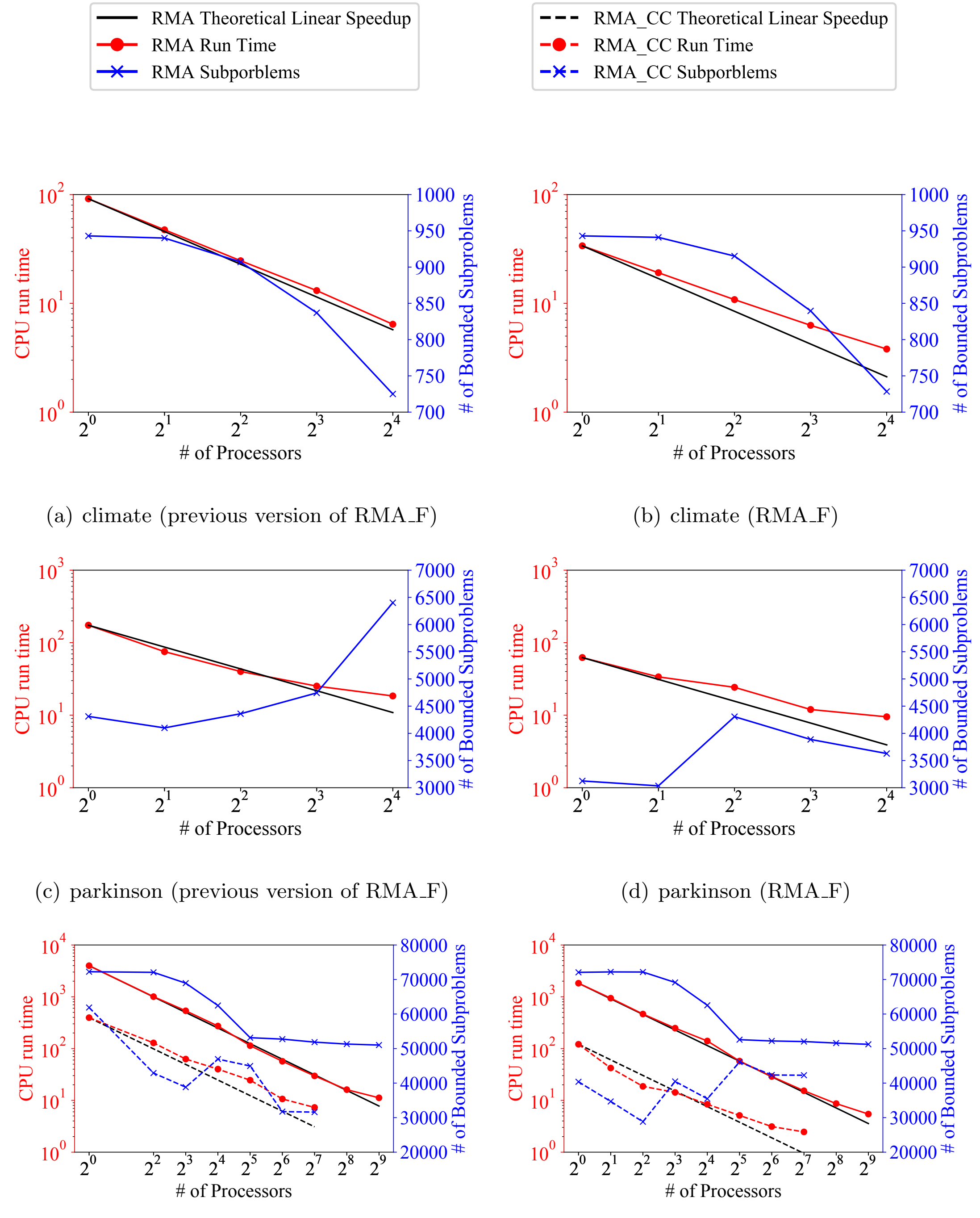
(c) poker (RMA_F)

	Strong Branching			Cutpoint Caching		
P	Time	SU	SP	Time	SU	SP
$2^6 = 64$	8,573.5	—	128,401	4,103.4	—	102,263
$2^7 = 128$	4,488.8	1.9 (2)	137,117	2,244.9	1.8 (2)	113,019
$2^8 = 256$	2,470.4	3.5 (4)	154,031	1,254.8	3.3 (4)	124,203
$2^9 = 512$	1,524.5	5.6 (8)	185,892	681.4	6.0 (8)	122,007
$2^{10} = 1024$	1,058.3	8.1 (16)	226,746	493.7	8.3 (16)	150,401
$2^{11} = 2048$	838.1	10.2 (32)	276,704	416.3	9.9 (32)	200,054

(d) EEG (RMA_R)

	Strong Branching			Cutpoint Caching					
P	Time	SU	SP	Time	SU	SP	CC	SB	%CC
$2^8 = 256$	9,527.8	—	13,262,648	1,555.5	—	7,741,421	7,730,211	11,210	99.9%
$2^9 = 512$	4,728.9	2.0 (2)	13,263,009	802.4	1.9 (2)	8,251,723	8,230,976	20,747	99.8%
$2^{10} = 1,024$	2,349.8	4.1 (4)	13,264,989	439.6	3.5 (4)	9,110,574	9,073,190	37,384	99.6%
$2^{11} = 2,048$	1,191.2	8.0 (8)	13,269,823	242.1	6.4 (8)	9,947,526	9,879,150	68,376	99.3%

Table 2.15: Parallel speedup results for larger datasets



(a) climate (previous version of RMA_F)

(b) climate (RMA_F)

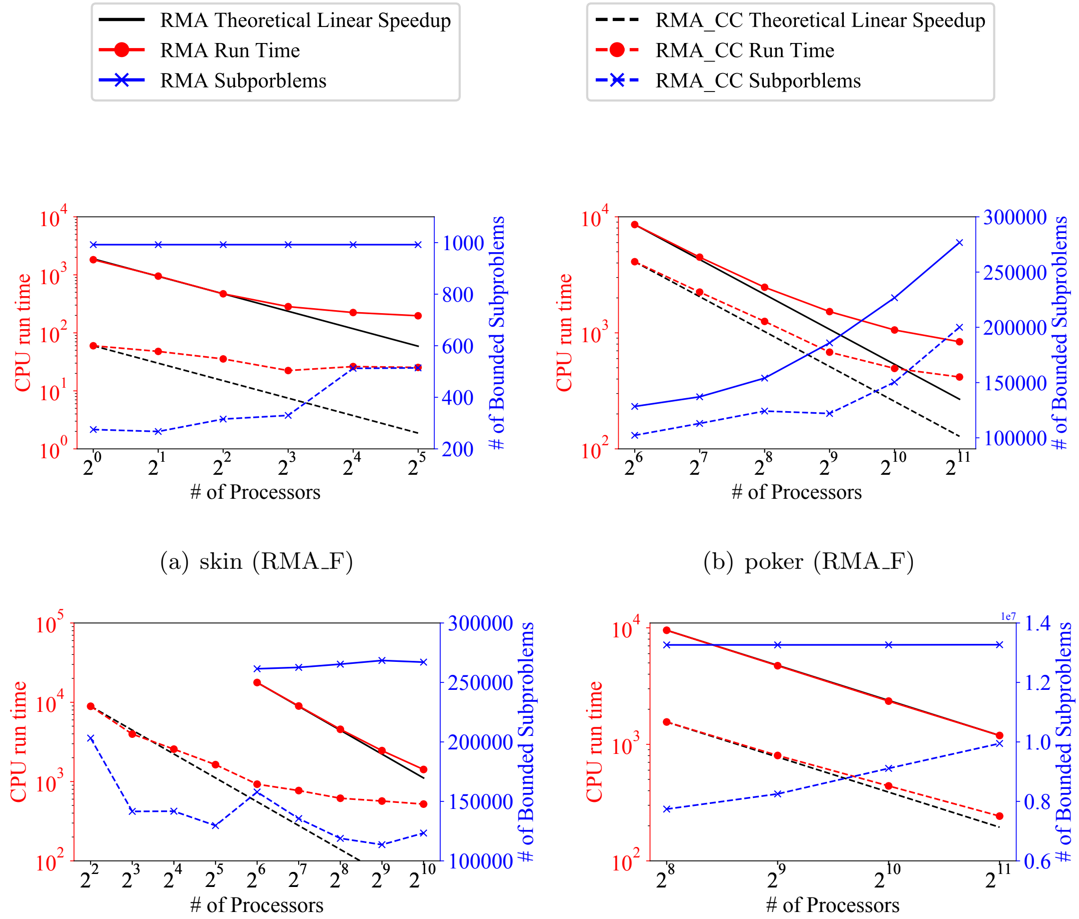
(c) parkinson (previous version of RMA_F)

(d) parkinson (RMA_F)

(e) indian (previous version of RMA_F)

(f) indian (RMA_F)

Figure 2.14: Parallel speedup results for smaller datasets



(a) skin (RMA_F)

(b) poker (RMA_F)

(c) credit_card (RMA_F)

(d) EEG (RMA_R)

Figure 2.15: Parallel speedup results for smaller datasets

Chapter 3

Classification Application

3.1 LPBoost Classification Model

The first application of the RMA problem constructs a classifier for a two-class classification problem from a linear combination of multidimensional "box"-based rules. As with the RMA problem, this classification problem assumes a set of training data having m observations, each with n attributes. Let $X \in \mathbb{R}^{m \times n}$ denote the explanatory matrix, and x_{ij} denote the (i, j)-th element of this matrix. The observations $X_1, \ldots, X_m \in \mathbb{R}^n$ are partitioned into "positive" and "negative" classes by two index sets $\Omega^+, \Omega^- \subset \{1, \ldots, m\}$. For each $i = 1, \ldots, m$, let $y_i = +1$ if $i \in \Omega^+$ and $y_i = -1$ if $i \in \Omega^-$. We will construct a classifier by a combination of base classifiers, identified by boxed-based rules. Suppose a rule function $r_{(a,b)} : \mathbb{R}^n \to \{0, 1\}$ is given by

$$r_{(a,b)}(X_i) = \begin{cases} 1, & \text{if } a_j \leq x_{ij} \leq b_j \ \forall\, j = 1, \ldots, n \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

Let a set K contain all possible pairs $(a, b) \in \mathbb{R}^n \times \mathbb{R}^n$ with $a \leq b$, constituting a catalog of all the possible rules of the form (3.1) that we wish to be available to our classification model:

$$f(X_i) = \gamma_0 + \sum_{k \in K} \gamma_k r_k(X_i), \tag{3.2}$$

where γ_0 is a constant term, $\gamma_k \in \mathbb{R}$, $k \in K$, can be positive or negative, so $\gamma_k > 0$ indicates that a observation covered by the box-based rule k votes to classify it as positive, and *vice versa.* The classifier operates by predicting a new point $X_i \in \mathbb{R}^n$ to be in the positive class if $f(X_i) > 0$, and in the negative class if $f(X_i) < 0$. When

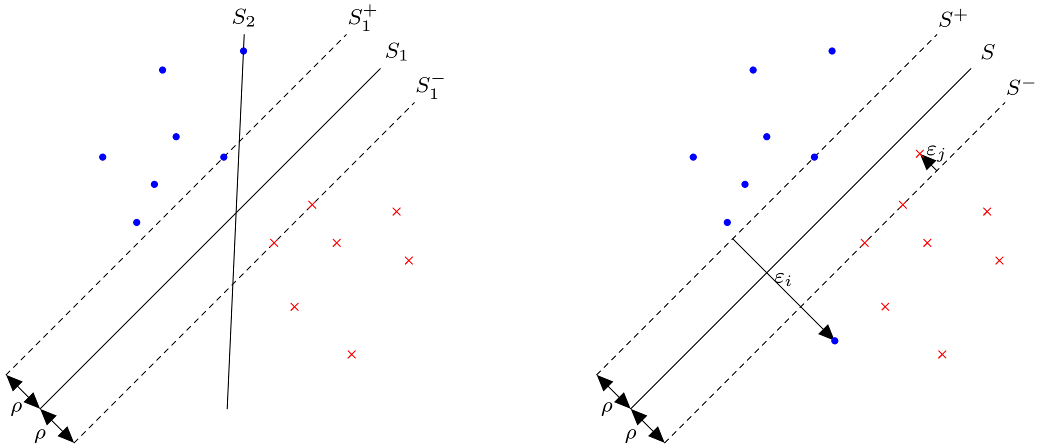
$f(X_i) = 0$, if the training data contain more positive class samples, then the classifier predicts positive; otherwise, negative. The objective is to construct a strong classifier, $\text{sign}(f) : \mathbb{R}^n \rightarrow \{-1, 0, +1\}$, that accurately classifies unseen observations $(X_{i'}, y_{i'})$.

The set K will typically be extremely large: restricting each a_j and b_j to values that appear as x_{ij} for some i, which is sufficient to describe all possible distinct behaviors of rules of the form (3.1) on the dataset X, there are still $\prod_{j=1}^{n} \ell_j(\ell_j + 1)/2 \geq 3^n$ possible choices for (a, b), where $\ell_j = \left|\bigcup_{i=1}^{m}\{x_{ij}\}\right|$ is the number of distinct values for x_{ij} in attribute j, as proved below.

Lemma 3.1.1. *The total number of possible combinations $(a, b) \in \mathbb{R}^n \times \mathbb{R}^n$ with values drawn from the dataset X with $a \leq b$ is $\prod_{j=1}^{n} \ell_j(\ell_j + 1)/2 \geq 3^n$ where $\ell_j = \left|\bigcup_{i=1}^{m}\{x_{ij}\}\right|$ is the number of distinct values for x_{ij} in attribute j.*

Proof. We show that there are $\dfrac{\ell_j(\ell_j + 1)}{2}$ distinct choices of (a_j, b_j) such that $a_j \leq b_j$ on the dataset X. There are ℓ_j combinations of (a, b) such that $a = b$, and $\binom{\ell_j}{2} = \dfrac{\ell_j(\ell_j - 1)}{2}$ combinations of (a, b) such that $a < b$. Therefore, the total number of such combinations for each attribute is $\ell_j + \dfrac{\ell_j(\ell_j - 1)}{2} = \dfrac{2\ell_j + \ell_j(\ell_j - 1)}{2} = \dfrac{\ell_j^2 + \ell_j}{2} = \dfrac{\ell_j(\ell_j + 1)}{2}$. Since there are n attributes, $K = \prod_{j=1}^{n} \dfrac{\ell_j(\ell_j + 1)}{2}$. □

Due to the potentially huge number of possible rules, we will dynamically generate a set of rules to construct a strong classifier. LPBoost [11] is a column-generation procedure for constructing a weighted voting classifier. Similarly to Support Vector Machines (SVM) [3, 2], the objective of LPBoost is maximizing a margin between positive and negative classifier boundaries with minimizing the sum of misclassification errors by the margin classifier. Two common two-class classification models are *soft margin* and *hard margin* classifiers. The soft margin classifier tolerates misclassification by the soft margin, and minimizes the sum of the misclassification errors. The hard margin classifier enforces that all observations are correctly classified. The soft margin classification model is generally more practical than the hard margin classification model since the soft margin can tolerate outliers. As shown in Figure 3.1(a), two hard margin separators S_1 and S_2 classify all observations correctly. However, S_1 is generally a better separator since S_1 has greater hard margin, the distance between boundaries S_1^+



(a) S_1 separator is better than S_2 separator since S_1 has the hard margin separators S_1^+ and S_1^- with maximum margin length ρ, while S_2 has zero hard margin.

(b) S is a separator with soft margin separators S^+ and S^-, separating boundaries with the soft margin length ρ. This formulation violations $\epsilon_i > 0$ of the soft margin.

Figure 3.1: Illustration of LPBoost: dots and x's represent positive and negative training observations, respectively.

and S_1^-. Figure 3.1(b) shows a soft margin classifier, with some observations allowed to be misclassified by the soft margin classifier, and ϵ_i being the classification error by the soft margin, called soft margin violation. The LP-Boost separator is a hyperplane in the space containing the r_k^+ and r_k^- features created by column generation. In the space of the original observations, it usually cannot be depicted as a hyperplane. The decision rule of the margin classifier is: for any $\rho \in \mathbb{R}$, if $f(X_i) \geq \rho$, the observation is classified positive; else if $f(X_i) \leq \rho$, the observation is classified negative.

We split $r_k(X_i)$ into $r_k^+(X_i)$ and $r_k^-(X_i)$, respectively called a positive and negative box. Hence, $r_k^+(X_i) = r_k(X_i)$ and $r_k^-(X_i) = -r_k(X_i)$ with the respective coefficients $\gamma_k^+ \in \mathbb{R}_+$ and $\gamma_k^- \in \mathbb{R}_+$. The classifier (3.2) reduces to:

$$f(X_i) = \gamma_0 + \sum_{k \in K} \left(\gamma_k^+ r_k^+(X_i) + \gamma_k^- r_k^-(X_i) \right). \qquad (3.3)$$

The linear programming model solved by LPBoost is:

$$\min_{\gamma,\epsilon,\rho} \quad -\rho + D\sum_{i=1}^{m} \epsilon_i^p \tag{3.4a}$$

$$\text{s.t.} \quad y_i\left(\gamma_0 + \sum_{k\in K}\left(\gamma_k^+ r_k^+(X_i) + \gamma_k^- r_k^-(X_i)\right)\right) + \epsilon_i \geq \rho, \qquad i = 1\ldots m \tag{3.4b}$$

$$\sum_{k\in K}(\gamma_k^+ + \gamma_k^-) = 1 \tag{3.4c}$$

$$\epsilon_i \geq 0, \qquad i = 1,\ldots,m \tag{3.4d}$$

$$\gamma_k^+, \gamma_k^- \geq 0, \qquad k \in K. \tag{3.4e}$$

The variables γ_k^+ and γ_k^- are non-negative voting classifier weights, ρ is a soft margin between the two class boundaries, and the variables ϵ_i represent soft margin violation. p is either 1 or 2. A non-negative parameter D represents the tradeoff between the sum of misclassification errors and margin maximization. This model has $2+m+2\,|K|$ variables and $1+m$ constraints except for nonegativity.

The LPBoost column generation procedure starts by including only a small subset of the possible weak classifiers variables γ_j^+ and/or γ_j^- in (3.4), resulting in a "restricted primal problem". LPBoost then uses an auxilliary optimization procedure to find new variables γ_j^+ and/or γ_j^- that "price out" properly to enter the basis, adjoins these variables to the restricted primal, and re-solves it. It repeats this procedure until no more columns price out properly to enter the basis or the smallest reduced cost is very close to 0. We initially choose $K' = \emptyset$, so there are no rules to construct the initial classifier (3.3). Hence, the column generation procedure starts by solving the pricing problem with each observation having weight y_i/m. After solving the first pricing problem, an initial restricted master problem is formulated. Solving the restricted master problem yields optimal Lagrange multipliers $\mu \in \mathbb{R}_+^m$ and $\alpha \in \mathbb{R}$ for constrants (3.4b) and (3.4c) respectively. From the structure of the (3.4), the reduced costs of γ_k^+ and γ_k^- are respectively:

$$\mathrm{rc}[\gamma_k^+] = -\alpha - \sum_{i=1}^{m} r_k^+(X_i) y_i \mu_i \tag{3.5a}$$

$$\mathrm{rc}[\gamma_k^-] = -\alpha - \sum_{i=1}^{m} r_k^-(X_i) y_i \mu_i, \tag{3.5b}$$

and hence, for each $k \in K$, that

$$\begin{aligned}\min_{k\in K}\left\{\mathrm{rc}[\gamma_k^+], \mathrm{rc}[\gamma_k^-]\right\} &= \min_{k\in K}\left\{-\sum_{i=1}^{m} r_k^+(X_i) y_i \mu_i, -\sum_{i=1}^{m} r_k^-(X_i) y_i \mu_i\right\} - \alpha \\ &= -\max_{k\in K}\left\{\sum_{i=1}^{m} r_k^+(X_i) y_i \mu_i, \sum_{i=1}^{m} r_k^-(X_i) y_i \mu_i\right\} - \alpha\end{aligned}$$

Therefore, the column with the most negative reduced cost may be found by solving

$$z^* = \max_{k\in K}\left\{\sum_{i=1}^{m} r_k^+(X_i) y_i \mu_i, \sum_{i=1}^{m} r_k^-(X_i) y_i \mu_i\right\}, \tag{3.6}$$

and the natural stopping condition for column generation is $z^* \le -\alpha$. Since $y_i = -1$ or $+1$ respectively represents a training sample marked as negative or positive, let the observation weight

$$w_i = y_i \mu_i = \begin{cases} \mu_i & \text{if } i \in \Omega^+ \\ -\mu_i & \text{if } i \in \Omega^- \end{cases} \qquad i = 1, \ldots, m.$$

By substituting $y_i \mu_i$ for w_i, the pricing problem (3.6) becomes:

$$z^* = \max_{k\in K}\left\{\sum_{i=1}^{m} r_k^+(X_i) w_i, \sum_{i=1}^{m} r_k^-(X_i) w_i\right\}, \tag{3.7}$$

The master problem constraints (3.4b) reduce to the decision rule of the soft margin classifier with soft margin violation ϵ_i: $y_i f(X_i) + \epsilon_i \ge \rho$. If the soft margin classifier misclassifies the observation i, then this constraint (3.4b) is active with $\epsilon_i > 0$ and $\mu_i > 0$; otherwise, $\epsilon_i = 0$ and $\mu_i = 0$. Hence, μ_i indicates the soft margin misclassification weight for observation i. Maximizing the first and second terms are summarized below.

1. Maximizing the first term of (3.7) is finding the positive box r_k^+ such that the sum of the misclassification weights for covered positive observations most greatly exceeds the sum of the misclassification weights for covered negative observations.

2. Maximizing the second term of (3.7) is finding the negative box r_k^- such that the sum of the misclassification weights for covered negative observations most greatly exceeds the sum of the misclassification weights for covered positive observations.

Problem (3.7) finds either the optimal positive or negative box to enter the restricted master problem, and it reduces to the RMA problem derived by a positive or negative box, as explained below. The following problem is the RMA problem:

$$\max_{k \in K} \left| \sum_{i}^{m} w_i r_k(X_i) \right| = \max_{k \in K} \left\{ \sum_{i}^{m} w_i r_k(X_i), -\sum_{i}^{m} w_i r_k(X_i) \right\}. \tag{3.8}$$

Maximizing the first term of the right hand side of (3.8) is the same as maximizing the first term of (3.7), and maximizing second term of he right hand side of (3.8) is the same as maximizing the second term of (3.7). Therefore, if the RMA solution arises from the first term, the algorithm selects the positive box r_k^+; else if the RMA solution arises from the second term, it selects the negative box r_k^- to add to the restricted master problem.

With the respective Lagrange multipliers $\mu \in \mathbb{R}_+^m$ and $\alpha \in \mathbb{R}$ of the master problem constraints of (3.4b) and (3.4c), the Lagrangian function of the LPBoost formulation 3.4 is:

$$g(\mu, \alpha)_{\mu \geq 0, \alpha} \tag{3.9a}$$

$$= \min_{\epsilon, \gamma^+, \gamma^- \geq 0, \gamma_0, \rho} - \rho + D \sum_{i=1}^{m} \epsilon_i^p + \alpha \left(1 - \sum_{k \in K} \left(\gamma_k^+ + \gamma_k^- \right) \right) \tag{3.9b}$$

$$+ \sum_{i=1}^{m} \left(\rho - y_i \left(\gamma_0 + \sum_{k \in K} \left(\gamma_k^+ r_k^+(X_i) - \gamma_k^- r_k^-(X_i) \right) \right) - \epsilon_i \right) \mu_i. \tag{3.9c}$$

By rearranging the variables,

$$
\begin{aligned}
g(\mu,\alpha)_{\mu\geq 0,\alpha} = \alpha + \min_{\rho}\left(-\rho + \sum_{i=1}^{m} \rho\mu_i\right) - \min_{\gamma_0} \sum_{i=1}^{m} y_i\mu_i\gamma_0 & \qquad (3.10a)\\
+ \min_{\epsilon\geq 0}\left(D\sum_{i=1}^{m}\epsilon_i^p - \sum_{i=1}^{m}\epsilon_i\mu_i\right) & \qquad (3.10b)\\
+ \min_{\gamma^+\geq 0}\left(-\sum_{i=1}^{m} y_i\mu_i \sum_{k\in K}\gamma_k + r_k^+(X_i) - \alpha\sum_{k\in K}\gamma_k^+\right) & \qquad (3.10c)\\
+ \min_{\gamma^-\geq 0}\left(-\sum_{i=1}^{m} y_i\mu_i \sum_{k\in K}\gamma_k - r_k^-(X_i) - \alpha\sum_{k\in K}\gamma_k^-\right). & \qquad (3.10d)
\end{aligned}
$$

Factoring by each primal variable, $\rho \in \mathbb{R}, \epsilon_i, \gamma_k^+, \gamma_k^- \in \mathbb{R}_+$,

$$
\begin{aligned}
g(\mu,\alpha)_{\mu\geq 0,\alpha} = \alpha + \min_{\rho}(-1 + \sum_{i=1}^{m}\mu_i)\rho - \min_{\gamma_0}\sum_{i=1}^{m} y_i\mu_i\gamma_0 & \qquad (3.11a)\\
+ \sum_{i=1}^{m}\min_{\epsilon_i\geq 0}\left(D\epsilon_i^{p-1} - \mu_i\right)\epsilon_i & \qquad (3.11b)\\
+ \sum_{k\in K}\min_{\gamma_k^+\geq 0}\left(-\sum_{i=1}^{m} y_i\mu_i r_k^+(X_i) - \alpha\right)\gamma_k^+ & \qquad (3.11c)\\
+ \sum_{k\in K}\min_{\gamma_k^-\geq 0}\left(-\sum_{i=1}^{m} y_i\mu_i r_k^-(X_i) - \alpha\right)\gamma_k^-. & \qquad (3.11d)
\end{aligned}
$$

Since the Lagrangian function $g(\mu,\alpha)$ must be less than equal to z^*, the objective value of the primal LPBoost problem, $\max g(\mu,\alpha) \leq z^*$. Since each $\rho, \gamma_0 \in \mathbb{R}$ is unrestricted, their coefficients must be 0. Since each $\epsilon_i, \gamma_k^+, \gamma_k^- \in \mathbb{R}_+$ are non-negative, their coefficients have to be non-negative. Therefore, the dual of (3.4) for $p = 1$ is:

$$
\begin{aligned}
&\max_{\mu\geq 0,\alpha} && \alpha && && \text{(3.12a)}\\
&\text{ST} && -\sum_{i=1}^{m}\mu_i y_i r_k^{+}(X_i) \geq \alpha, && k \in K && \text{(3.12b)}\\
& && -\sum_{i=1}^{m}\mu_i y_i r_k^{-}(X_i) \geq \alpha, && k \in K && \text{(3.12c)}\\
& && \sum_{i=1}^{m} y_i \mu_i = 0 && && \text{(3.12d)}\\
& && \sum_{i=1}^{m} \mu_i = 1 && && \text{(3.12e)}\\
& && 0 \leq \mu_i \leq D, && i = 1, \ldots, m. && \text{(3.12f)}
\end{aligned}
$$

The dual formulation for $p = 1$ has $1+m$ variables and $2+m+2\,|K|$ constraints except the non-negative constraints.

When $p = 2$, (3.11b) is: $\min_{\epsilon\geq 0} \sum_{i=1}^{m} \left(D\epsilon_i^2 - \mu_i\epsilon_i\right)$. Since $D\epsilon_i^2 - \mu_i\epsilon_i$ is a convex function, it attains its minimum value when $(\partial/\partial\epsilon_i)[D\epsilon_i^2 - \mu_i\epsilon_i] = 2D\epsilon_i - \mu_i = 0$. Solving this equation, $\epsilon_i = \dfrac{\mu_i}{2D}$. By substituting ϵ_i,

$$
\begin{aligned}
D\epsilon_i^2 - \mu_i\epsilon_i &= D\left(\frac{\mu_i}{2D}\right)^2 - \mu_i\left(\frac{\mu_i}{2D}\right)\\
&= D\frac{\mu_i^2}{4D^2} - \frac{\mu_i^2}{2D}\\
&= \frac{\mu_i^2 - 2\mu_i^2}{4D}\\
&= -\frac{\mu_i^2}{4D}.
\end{aligned}
$$

Therefore, we can substitute in (3.11b) using:

$$
\min_{\epsilon\geq 0} \sum_{i=1}^{m} D\epsilon_i^2 - \mu_i\epsilon_i = -\sum_{i=1}^{m}\frac{\mu_i^2}{4D}.
$$

Therefore, the dual formulation for $p = 2$ is:

$$\max_{\mu \geq 0, \alpha} \qquad \alpha - \sum_{i=1}^{m} \frac{\mu_i^2}{4D} \tag{3.15a}$$

$$\text{ST} \qquad -\sum_{i=1}^{m} \mu_i y_i r_k^+(X_i) \geq \alpha, \qquad k \in K \tag{3.15b}$$

$$-\sum_{i=1}^{m} \mu_i y_i r_k^-(X_i) \geq \alpha, \qquad k \in K \tag{3.15c}$$

$$\sum_{i=1}^{m} y_i \mu_i = 0 \tag{3.15d}$$

$$\sum_{i=1}^{m} \mu_i = 1. \tag{3.15e}$$

Constraints (3.12e) and (3.15e) correspond to the primal variable ρ, constraints (3.12d) and (3.15d) correspond to the primal variable γ_0, and dual constraints (3.12b), (3.12c), (3.15b), and (3.15c) correspond to the primal variables γ_k^+ and γ_k^-. The primal variable ϵ_i corresponds to constraint (3.12f) in the $p = 1$ case and to the quadratic objective function term in the $p = 2$ case.

3.1.1 Translating discretized box-based rules back to the original scale

We discretize the data X using the process described in Section 2.1.1 for some (small) parameter value δ, and solve the RMA problem. It is necessary to translate the resulting boxes back to the original, pre-integerized coordinate system to process unseen datasets. We perform this translation by expanding box boundaries to lie halfway between the boundaries of the clusters of points created by the discretization procedure, except when the lower boundary of the box has the lowest possible value or the upper boundary has the largest possible value. In these cases, we expand the box boundaries to $-\infty$ or $+\infty$, respectively. More precisely, for each observation variable j and $v \in \{0, \ldots, \ell_j - 1\}$, let $x_{j,v}^{\min}$ be the smallest value of x_{ij} assigned to the integer value v by Algorithm 1, and $x_{j,v}^{\max}$ be the largest. If $\hat{a}, \hat{b} \in \mathbb{N}^n$, $\hat{a} \leq \hat{b}$ describes a discretized box arising from the solution of the preprocessed RMA problem, we choose the corresponding box boundaries $a, b \in \mathbb{R}^n$

in the original coordinate system to be given by, for $j = 1, \ldots, n$,

$$a_j = \begin{cases} -\infty, & \text{if } \hat{a}_j = 0 \\ \frac{1}{2}(x^{\max}_{j,\hat{a}_j-1} + x^{\min}_{j,\hat{a}_j}), & \text{otherwise} \end{cases} \qquad b_j = \begin{cases} +\infty, & \text{if } \hat{b}_j = \ell_j - 1 \\ \frac{1}{2}(x^{\max}_{j,\hat{b}_j} + x^{\min}_{j,\hat{b}_j+1}), & \text{otherwise.} \end{cases}$$

Overall, our procedure is equivalent to solving the pricing problem (3.7) over some set of boxes $K = \mathcal{K}_\delta(X)$. For $\delta = 0$, the resulting set of boxes $\mathcal{K}_0(X)$ is such that the corresponding set of rules $\{r_k \mid k \in \mathcal{K}_0(X)\}$ comprises every box-based rule distinguishable on the dataset X. For small positive values of δ, the set of boxes $\mathcal{K}_\delta(X)$ excludes those corresponding to rules that "cut" between closely spaced observations.

3.2 Full Algorithm Implementation

Algorithm 8 details the exact LPBoost procedure using RMA, called *LPBR*. In each column generation iteration, it can find t different RMA solutions and enter them into the restricted dual problem. To this end, it uses PEBBL's ability to enumerate multiple near-optimal solutions.

Gurobi [22], a commercial linear programming solver, was used to solve the restricted master problem in each iteration. The weight of each observation i in the RMA subproblem is the value of dual variable in y_i times the restricted primal LP problem.

The choice of the parameter D is essential for LPBoost. If D is relatively large, then the model penalizes the sum of soft margin violation $\sum_{i=1}^m \epsilon_i$ more, and the soft margin ρ tends to be small; else if D is relatively small, then the model attempts to increase the soft margin ρ and tolerates larger sum of soft margin violation $\sum_{i=1}^m \epsilon_i$. The general guideline to choose the parameter D is shown below.

Lemma 3.2.1. *For the master problem in Algorithm 8 to have a solution that has a finite value and can change with the iteration count s, the parameter $D \in \mathbb{R}_+$ in* (3.4) *should be $D = \dfrac{1}{\nu m}$ where m is the number of training samples and* $0 < \nu < 1$ *[11, 26].*

Proof. Consider the following cases.

Algorithm 8 LPBR algorithm

```
1: **Input:** data $X \in \mathbb{R}^{m \times n}, y \in \{-1, +1\}^m$, penalty parameter $D \geq 0$, column generation tolerance $\theta \geq 0$, integer $t \geq 1$, aggregation tolerance $\delta \geq 0$, and iteration limit $S$
2: **Output:** $K' \subset \mathcal{K}_\delta(X)$, $\gamma \in \mathbb{R}^{|K'|}$
3: **LPBoost:**

4: Assign weights: $w_i = \frac{y_i}{m}, i = 1, \ldots, m$
5: $\alpha \leftarrow -\infty$
6: $K' \leftarrow \emptyset$
7: **for** $s = 1, \ldots, S$ **do**
8:     Use the RMA branch-and-bound algorithm, with preprocessing as in Section 2.1.1, to identify a set of the $t$ best possible positive or negative box solutions of $k_1, \ldots, k_t$ to
           $$\max_{k \in \mathcal{K}_\delta(X)} \left| \sum_{i=1}^{m} w_i r_k(x_i) \right|, \qquad (3.16)$$
       where $w_i = y_i \mu_i$, with objective values $z_1 \geq z_2 \geq \cdots \geq z_t$
9:     **if** $z_1 \leq -\alpha + \theta$ **break**
10:    **for each** $l \in \{1, \ldots, t\}$ **with** $z_l > -\alpha + \theta$ **do**
11:        $K' \leftarrow K' \cup \{k_l\}$ where the box $k_l$ is translated back to the original scale as in Section 3.1.1
12:    **end for**
13:    Solve the restricted master problem to obtain optimal primal variables $(\gamma^+ - \gamma^-)$ and dual variables $(\mu, \alpha)$
14: **end for**
15: **return** $(K', \gamma := \gamma^+ - \gamma^-)$
```

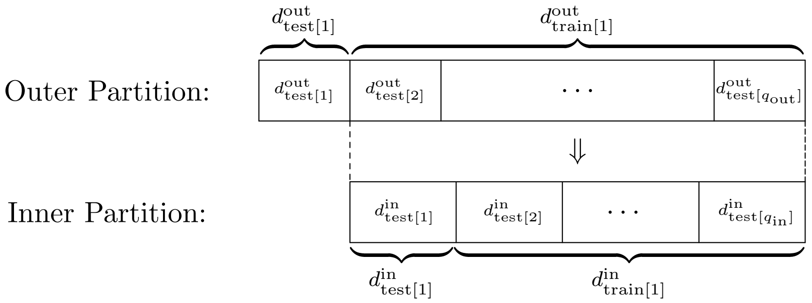
Case 1: $D < \dfrac{1}{m}$ $(\nu > 1)$: Constraints (3.12e) and (3.12f) cannot be satisfied. Therefore the dual 3.12 of the master problem is infeasible, so the master problem will be either unbounded or infeasible.

Case 2: $D = \dfrac{1}{m}$ $(\nu = 1)$: Constraints (3.12e) and (3.12f) enforce that $\mu_i = \dfrac{1}{m}$ for $i = 1, \ldots, m$. Thus, the pricing problem of each column generation iteration is always same as the initial pricing problem, with each observation having weight $w_i = \frac{y_i}{m}$. Hence, the result of the subproblem in step 8 will be identical in every iteration and the classifier cannot improve after the first iteration.

Case 3: $\dfrac{1}{m} < D$ $(0 < \nu < 1)$: Constraints (3.12e) and (3.12f) can be satisfied.

Therefore, Case 3 should be satisfied. □

The cross-validation-based experimental results below indicate that the optimal choice



$d^{\text{out}}_{\text{test}[1]}$
$d^{\text{out}}_{\text{train}[1]}$
Outer Partition:
$d^{\text{out}}_{\text{test}[1]}$
$d^{\text{out}}_{\text{test}[2]}$
$\cdots$
$d^{\text{out}}_{\text{test}[q_{\text{out}}]}$
$\Downarrow$
Inner Partition:
$d^{\text{in}}_{\text{test}[1]}$
$d^{\text{in}}_{\text{test}[2]}$
$\cdots$
$d^{\text{in}}_{\text{test}[q_{\text{in}}]}$
$d^{\text{in}}_{\text{test}[1]}$
$d^{\text{in}}_{\text{train}[1]}$

Figure 3.2: Training and testing datasets in the bilevel cross-validation procedure, in the case $i = 1, j = 1$

of $D = \dfrac{1}{\nu m}$ is generally discovered in the range $0.1 \leq \nu \leq 0.5$.

3.3 Computational Testing

3.3.1 Setting a Parameter by Cross-Validation

To evaluate the performance of our LPBR classification algorithm, we need to set its parameters, in particular the tradeoff parameter between misclassification error and margin maximization. As is standard in the machine learning community, we use a cross-validation procedure to determine suitable parameters for each dataset. In our computational tests, we embed this procedure within another cross-validation procedure whose main purpose is to evaluate the performance of our classification method.

Algorithm 9 outlines our full bilevel cross-validation procedure, with q_{out} folds of outer cross validation and q_{in} folds of inner cross validation. The set Λ holds the various parameter combinations to be tested. The subroutine **RandomPartition**(d, q) randomly partitions an input dataset $d = (X, y)$ into q smaller datasets of nearly equal size. The first call to **RandomPartition** produces a q_{out}-way partition that we use to evaluate the overall performance of the algorithm and its competitors. We evaluate each algorithm by its performance averaged over the q_{out} folds of this partition, using one element of the partition as the testing set and combining the remaining elements into the training set. Within each of these q_{out} folds, we use a q_{in}-way partition of the resulting training set to decide on the best choice of algorithm parameters λ from the

Algorithm 9 Bilevel cross validation procedure to evaluate LPBR and set its parameters

```
Input: $q_{\text{out}}, q_{\text{in}} \in \mathbb{N}, d = (X, y) \in (\mathbb{R}^{m \times n}, \mathbb{R}^m), \Lambda \subset \mathbb{R}^T$ (parameter combinations to test)
Output: $\bar{\varepsilon}_{\text{our}}, \bar{\varepsilon}_{\text{other}}$
BilevelCrossValidation:

$(d^{\text{out}}_{\text{test}[1]}, \ldots, d^{\text{out}}_{\text{test}[p]}) \leftarrow$ RandomPartition$(d, q_{\text{out}})$
for $i = 1, \ldots, q_{\text{out}}$ do
  $\varepsilon^* = \infty$
  $d^{\text{out}}_{\text{train}[i]} \leftarrow d \setminus d^{\text{out}}_{\text{test}[i]}$
  $(d^{\text{in}}_{\text{test}[1]}, \ldots, d^{\text{in}}_{\text{test}[q]}) \leftarrow$ RandomPartition$(d^{\text{out}}_{\text{train}[i]}, q_{\in})$
  for each $\lambda \in \Lambda$ do
    for $j = 1, \ldots, q_{\text{in}}$ do
      $d^{\text{in}}_{\text{train}[j]} \leftarrow d^{\text{out}}_{\text{train}[i]} \setminus d^{\text{in}}_{\text{test}[j]}$
      $\mathcal{M}_{\text{our}} \leftarrow$ TrainOurModel$(d^{\text{in}}_{\text{train}[j]}, \lambda)$
      $\varepsilon_{\text{our}}[j] \leftarrow$ EvaluteModel$(\mathcal{M}_{\text{our}}, d^{\text{in}}_{\text{test}[j]})$
    end for
    $\bar{\varepsilon}_{\text{our}} = \text{avg}(\epsilon_{\text{our}})$
    if $\bar{\varepsilon}_{\text{our}} < \varepsilon^*$ then $\lambda^* \leftarrow \lambda$; $\varepsilon^* \leftarrow \bar{\varepsilon}_{\text{our}}$
  end for
  $\mathcal{M}_{\text{our}} \leftarrow$ TrainOurModel$(d^{\text{out}}_{\text{train}[i]}, \lambda^*)$
  $\varepsilon_{\text{our}}[i] \leftarrow$ EvaluteModel$(\mathcal{M}_{\text{our}}, d^{\text{out}}_{\text{test}[i]})$
  $\mathcal{M}_{\text{other}} \leftarrow$ TrainOtherModel$(d^{\text{out}}_{\text{train}[i]})$
  $\varepsilon_{\text{other}}[i] \leftarrow$ EvaluateModel$(\mathcal{M}_{\text{other}}, d^{\text{out}}_{\text{test}[i]})$
end for
$\bar{\varepsilon}_{\text{our}} = \text{avg}(\varepsilon_{\text{our}})$
$\bar{\varepsilon}_{\text{other}} = \text{avg}(\varepsilon_{\text{other}})$
return $(\bar{\varepsilon}_{\text{our}}, \bar{\varepsilon}_{\text{other}})$
```

set of candidates Λ.

Figure 3.2 depicts the situation when the outer loop index i is 1 (we are testing the first of the q_{out} outer folds) and the inner loop index j is also 1 (we are testing the first of the q_{in} inner folds). The outer testing set is $d^{\text{out}}_{\text{test}[1]}$, the first element returned from the initial call to **RandomPartition**, and the outer training set is the remainder of the input dataset. To choose the best parameters $\lambda \in \Lambda$, we partition this training set into q parts. In the first iteration of the inner loop (over j) the inner testing set is the first element of this inner partition, with the remaining elements making up the inner testing set. We select the parameter setting λ^* that has the best performance averaged over the folds of the inner partition.

	Dataset						
Method	parkinson	hungheart	cleveland	breast	cmc	car	chess
LPBR	10.00%	**18.72%**	**17.52%**	**3.07%**	**30.69%**	**1.07%**	0.53%
AdaBoost	**7.69%**	22.46%	17.84%	3.59%	33.13%	1.27%	**0.48%**
Random Forests	9.49%	19.05%	18.53%	**3.07%**	30.96%	1.88%	1.49%

Table 3.1: Average testing classification error rate over the small datasets. The smallest value in each column is bolded.

The end of Algorithm 9 invokes the subroutines **TrainOtherModel** and **EvaluateModel** to evaluate the performance of a competing model.

3.3.2 Numerical Results for Small Datasets

We tested the LPBR procedure on some small datasets from the UCI data repository [25], as described in Table 2.5. We compared its performance to random forests and AdaBoost.

We used our bilevel cross-validation procedure twice with $q_{\text{out}} = 5$ outer folds and $q_{\text{in}} = 3$ inner folds. This procedure gave us a performance sample of size 10 for each dataset, but each with an 80-20 data split between training and testing data. We selected the LPBR's parameters as follows:

- After some initial experimentation, we used $\delta = 0.005$ and $p = 2$.
- We used our inner cross-validation procedure to select the parameters D. We experimented $D = \dfrac{1}{\nu m}$, selecting the possible values from $\nu = \{0.0001, 0.001, 0.005, 0.01\}$.
- The inner iteration limit was $S = 20$.
- The iteration limit for the outer cross-validation was $S = 50$ for datasets with less than 500 observations, and $S = 100$ otherwise.
- We set $t = 1$, so we only added one box rule per iteration of column generation.

We initially create one-dimensional greedy box rules for all attributes, and then solve the initial restricted master problem. To improve the running time to choose the parameter D, we only used our greedy RMA heuristic to solve the subproblems in the

	Dataset						
Method	parkinson	hungheart	cleveland	breast	cmc	car	chess
LPBR	**0.00%**	**0.00%**	**0.00%**	**0.00%**	17.40%	0.48%	**0.01%**
AdaBoost	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**3.00%**	**0.46%**	0.02%
Random Forests	**0.00%**	0.09%	0.08%	**0.00%**	8.40%	0.59%	0.35%

Table 3.2: Average training classification error rate over the small datasets. The smallest value in each column is bolded.

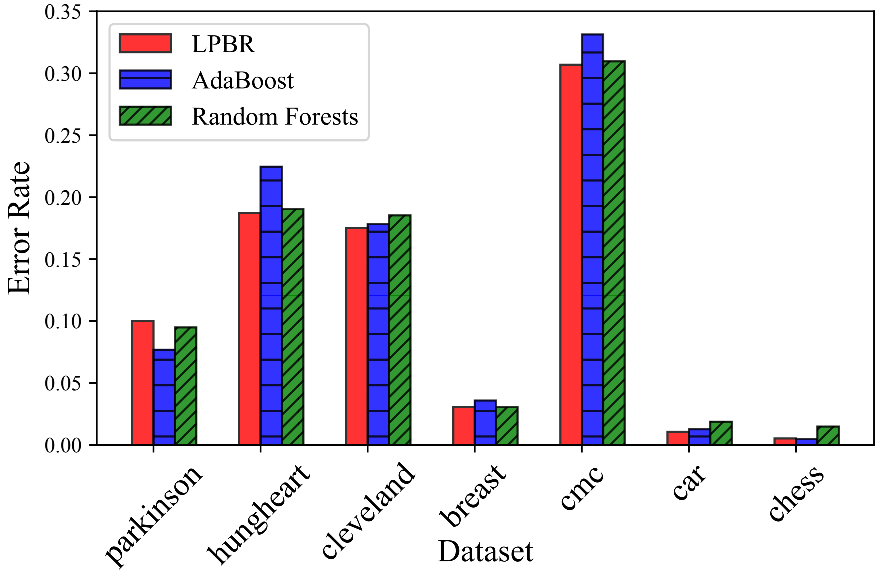
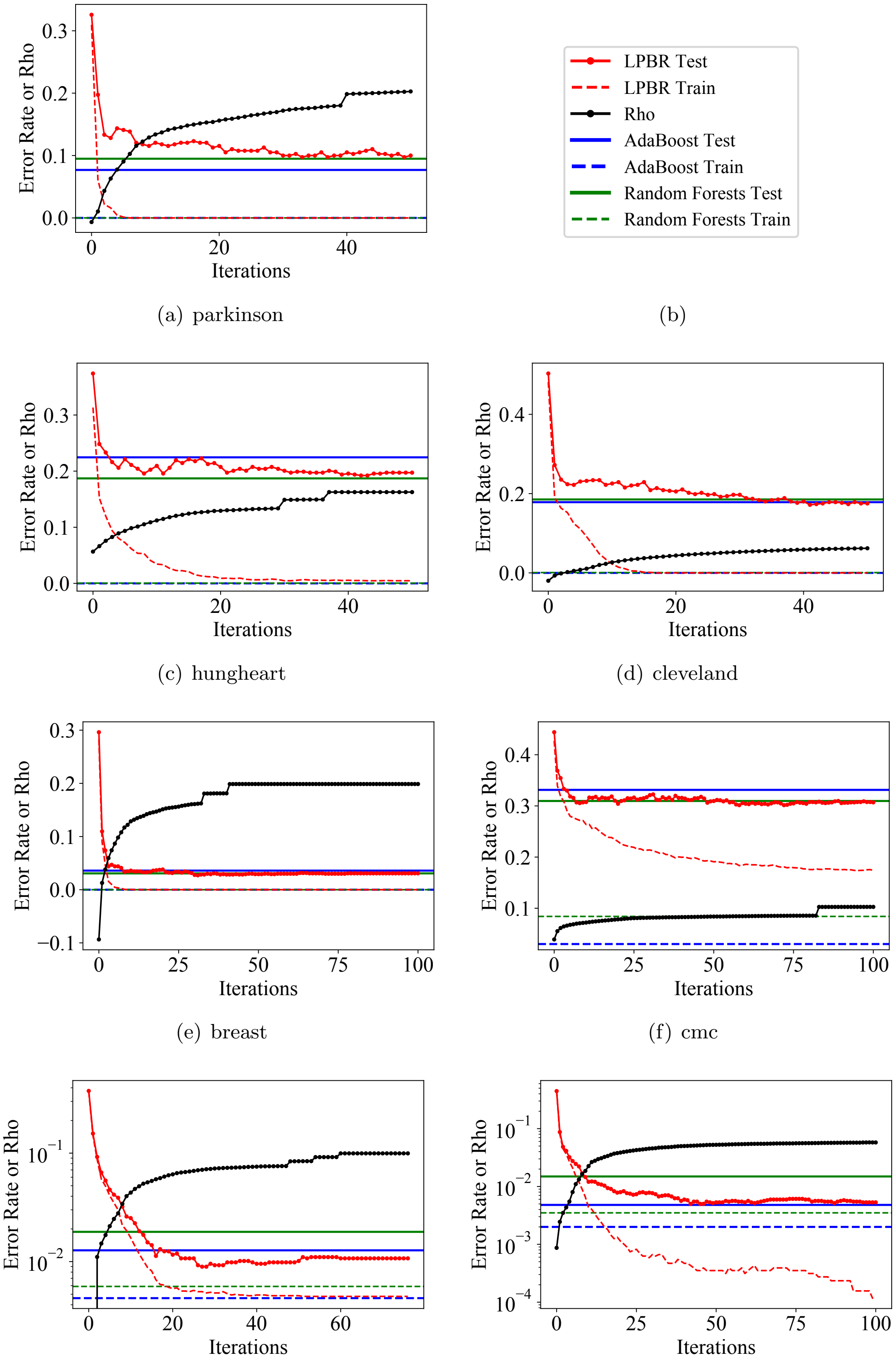


Figure 3.3: Testing classification error rates of three different classification methods.

inner cross-validation. The subproblem in the outer cross-validation was initially solved by greedy RMA. Once the greedy RMA objective satisfies the stopping condition or the current greedy RMA solution is same as the previous RMA solution, the subproblem is solved exactly using our parallel branch-and-bound algorithm.

The classification error of each rate run was calculated by the number of observations misclassified by the final decision rule, divided by the number of training or testing observations. Tables 3.1 and 3.2 respectively show the average testing and training classification error rates. LPBR has the lowest testing classification error rates for 5 out of 7 datasets shown in Table 3.1 and Figure 3.3. However, all three methods of LPBR, AdaBoost, and random forests are very competitive. Figure 3.4 shows that the average testing classification error rate generally decreases and ρ increases as the number of box-based rules of the classification model increases in each iteration. Even though the training error is 0, the testing error can be improved as the margin ρ increases.



(a) parkinson

(b)

(c) hungheart

(d) cleveland

(e) breast

(f) cmc

(g) car

(h) chess

Figure 3.4: Testing and training classification error rates and ρ as a function of iteration count for the datasets.

Chapter 4

Regression Application

4.1 A Penalized Regression Model with Rules

The second application of the RMA problem concerns generalized regression constructing a prediction function from a linear combination of the standard linear regression variables and box-based rules. To make our notation below more concise, we let X denote the matrix whose rows are $X_1^\top, \ldots, X_m^\top$, and also let $y = (y_1, \ldots, y_m) \in \mathbb{R}^m$. We may then express a problem instance by the pair (X, y). We also let x_{ij} denote the $(i, j)^{\text{th}}$ element of this matrix, that is, the value of attribute j in observation i.

Let K be some set of pairs $(a, b) \in \mathbb{R}^n \times \mathbb{R}^n$ with $a \leq b$, constituting a catalog of all the possible rules of the form (1.22) that we wish to be available to our regression model. The set K will typically be extremely large: restricting each a_j and b_j to values that appear as x_{ij} for some i, which is sufficient to describe all possible distinct behaviors of rules of the form (1.22) on the dataset X, there are still $\prod_{j=1}^n \ell_j(\ell_j + 1)/2 \geq 3^n$ possible choices for (a, b), where $\ell_j = \left|\bigcup_{i=1}^m \{x_{ij}\}\right|$ is the number of distinct values for x_{ij}.

The predictors $\hat{f}$ that our method constructs are of the form

$$\hat{f}(x) = \beta_0 + \sum_{j=1}^{n} \beta_j x_j + \sum_{k \in K} \gamma_k r_k(x) \tag{4.1}$$

for some $\beta_0, \beta_1, \ldots, \beta_m, (\gamma_k)_{k \in K} \in \mathbb{R}$. Finding an $\hat{f}$ of this form is a matter of linear regression, but with the regression coefficients in a space with the potentially very high dimension of $1+n+|K|$. As is now customary in regression models in which the number of explanatory variables potentially outnumbers the number of observations, we employ a LASSO-class model in which all explanatory variables except the constant term have L_1 penalties. Letting $\beta = (\beta_1, \ldots, \beta_n) \in \mathbb{R}^n$ and $\gamma \in \mathbb{R}^{|K|}$, let $f_{\beta_0,\beta,\gamma}(\,\cdot\,)$ denote the predictor function in (4.1). We then propose to estimate β_0, β, γ by (approximately) solving

$$\min_{\beta_0,\beta,\gamma} \left\{ \sum_{i=1}^{m} |f_{\beta_0,\beta,\gamma}(X_i) - y_i|^p + C \|\beta\|_1 + E \|\gamma\|_1 \right\}, \tag{4.2}$$

where $p \in \{1,2\}$ and $C, E \geq 0$ are scalar parameters. For $p = 2$, this model is essentially the classic LASSO as originally proposed by [28]. The decision variables are $\epsilon \in \mathbb{R}^m$, $\beta_0 \in \mathbb{R}$, $\beta^+, \beta^- \in \mathbb{R}^n$, and $\gamma^+, \gamma^- \in \mathbb{R}^{|K|}$. To put (4.2) into a form more suitable for column generation, we split the regression coefficient vectors into positive and negative parts, so that $\beta = \beta^+ - \beta^-$ and $\gamma = \gamma^+ - \gamma^-$, with $\beta^+, \beta^- \in \mathbb{R}^n_+$ and $\gamma^+, \gamma^- \in \mathbb{R}^{|K|}_+$. Introducing one more vector of variables $\epsilon \in \mathbb{R}^m_+$, we obtain the equivalent problem formulation

$$\min_{\substack{\beta_0\in\mathbb{R},\ \beta^+,\beta^-\geq 0\\ \gamma^+,\gamma^-\geq 0,\ \epsilon\geq 0}} \quad \sum_{i=1}^{m} \epsilon_i^p + C \sum_{j=1}^{n} (\beta_j^+ + \beta_j^-) + E \sum_{k\in K} (\gamma_k^+ + \gamma_k^-) \tag{4.3a}$$

$$\text{ST} \qquad \beta_0 + X_i^\top(\beta^+ - \beta^-) + \sum_{k\in K} r_k(X_i)(\gamma_k^+ - \gamma_k^-) - \epsilon_i \leq \quad y_i, \quad i = 1,\ldots,m \tag{4.3b}$$

$$-\beta_0 - X_i^\top(\beta^+ - \beta^-) - \sum_{k\in K} r_k(X_i)(\gamma_k^+ - \gamma_k^-) - \epsilon_i \leq -y_i, \quad i = 1,\ldots,m \tag{4.3c}$$

This model is constructed so that in any optimal solution, $\epsilon_i = |f_{\beta_0,\beta,\gamma}(X_i) - y_i|$ for $i = 1,\ldots,m$. For each $i = 1,\ldots,m$, constraints (4.3b) and (4.3c) respectively tie ϵ_i to the overestimation or underestimation of y_i since they respectively reduce to

$$f_{\beta_0,\beta^+,\beta^-,\gamma^+,\gamma^-}(X_i) - \epsilon_i \leq \quad y_i \qquad \Leftrightarrow \qquad f_{\beta_0,\beta^+,\beta^-,\gamma^+,\gamma^-}(X_i) - y_i \leq \quad \epsilon_i, \tag{4.4a}$$

$$-f_{\beta_0,\beta^+,\beta^-,\gamma^+,\gamma^-}(X_i) - \epsilon_i \leq -y_i \qquad \Leftrightarrow \qquad f_{\beta_0,\beta^+,\beta^-,\gamma^+,\gamma^-}(X_i) - y_i \geq -\epsilon_i. \tag{4.4b}$$

If $p = 1$, problem (4.3) is a linear program, and if $p = 2$ it is a convex, linearly constrained quadratic program. In either case, there are $2m$ constraints (other than nonnegativity), but the number of variables is $1 + m + 2n + 2|K|$. Because of this potentially unwieldy number of variables, we propose to solve (4.3) by using the classical technique of column generation. As usual, our column generation algorithm cycles between solving two optimization problems, the restricted master problem and the pricing problem. In our case, the restricted master problem is the same as (4.3), but with K replaced by some (presumably far smaller) $K' \subseteq K$; we initially choose $K' = \emptyset$. Solving the restricted master problem yields optimal Lagrange multipliers $\mu \in \mathbb{R}^m_+$ and $\nu \in \mathbb{R}^m_+$ for constrants (4.3b) and (4.3c), respectively. From the structure of the (4.3), the reduced costs of γ_k^+ and γ_k^- are respectively

$$\mathrm{rc}[\gamma_k^+] = E - \sum_{i=1}^m r_k(x_i)\nu_i + \sum_{i=1}^m r_k(x_i)\mu_i \tag{4.5a}$$

$$\mathrm{rc}[\gamma_k^-] = E + \sum_{i=1}^m r_k(x_i)\nu_i - \sum_{i=1}^m r_k(x_i)\mu_i, \tag{4.5b}$$

and hence, for each $k \in K$, we have that

$$\begin{aligned} \min\left\{\mathrm{rc}[\gamma_k^+], \mathrm{rc}[\gamma_k^-]\right\} &= \min\left\{E - \left|\sum_{i=1}^m r_k(x_i)(\nu_i - \mu_i)\right|\right\} \\ &= E - \max\left|\sum_{i=1}^m r_k(x_i)(\nu_i - \mu_i)\right| \end{aligned}$$

Therefore, the column with the most negative reduced cost may be found by solving

$$z^* = \max_{k \in K}\left|\sum_{i=1}^m r_k(x_i)(\nu_i - \mu_i)\right| \tag{4.6}$$

and the natural stopping condition for column generation is $z^* \leq E$. Problem (4.6) is an instance of the RMA problem, as shown in Section 2.1.

If the optimal solution of (4.6) corresponds to a positive value of $\sum_{i=1}^m r_k(x_i)(\nu_i - \mu_i)$, then the corresponding entering variable is γ_k^+, whereas a negative value means that the entering variable is γ_k^-. In a slight departure from classical column generation, we add both of the variables γ_k^+ and γ_k^- to the restricted master problem whenever either one of them is selected to enter the basis by the pricing problem (4.6); this is equivalent to adjoining the value k to the set K'.

In terms of the respective Lagrange multipliers $\mu, \nu \in \mathbb{R}^m_+$ of the master problem constraints (4.3b) and (4.3c), the Lagrangian function of (4.3) is:

$$q(\mu,\nu)_{\mu,\nu\geq 0} = \tag{4.7a}$$

$$\min_{\substack{\beta_0\in\mathbb{R},\, \beta^+,\beta^-\geq 0\\ \gamma^+,\gamma^-\geq 0,\, \epsilon\geq 0}} \quad \sum_{i=1}^{m}\epsilon_i^p + C\sum_{j=1}^{n}(\beta_j^+ + \beta_j^-) + E\sum_{k\in K}(\gamma_k^+ + \gamma_k^-) \tag{4.7b}$$

$$+\sum_{i=1}^{m}\mu_i\left[\beta_0 + {X_i}^T(\beta^+ - \beta^-) + \sum_{k\in K} r_j(X_i)(\gamma_j^+ - \gamma_j^-) - \epsilon_i - y_i\right] \tag{4.7c}$$

$$+\sum_{i=1}^{m}\nu_i\left[-\beta_0 - {X_i}^T(\beta^+ - \beta^-) - \sum_{k\in K} r_j(X_i)(\gamma_j^+ - \gamma_j^-) - \epsilon_i + y_i\right]. \tag{4.7d}$$

The Lagrangian function (4.7) may be written as:

$$q(\mu,\nu)_{\mu,\nu\geq 0} = \sum_{i=1}^{m}\nu_i y_i - \sum_{i=1}^{m}\mu_i y_i \tag{4.8a}$$

$$+\min_{\epsilon\geq 0}\sum_{i=1}^{m}\epsilon_i^p - \sum_{i=1}^{m}\mu_i\epsilon_i - \sum_{i=1}^{m}\nu_i\epsilon_i \tag{4.8b}$$

$$+\min_{\beta_0}\beta_0\sum_{i=1}^{m}\mu_i - \beta_0\sum_{i=1}^{m}\nu_i \tag{4.8c}$$

$$+\min_{\beta^+\geq 0} C\sum_{j=1}^{n}\beta_j^+ + \sum_{i=1}^{m}\mu_i {X_i}^T\beta^+ - \sum_{i=1}^{m}\nu_i {X_i}^T\beta^+ \tag{4.8d}$$

$$+\min_{\beta^-\geq 0} C\sum_{j=1}^{n}\beta_j^- + \sum_{i=1}^{m}\nu_i {X_i}^T\beta^- - \sum_{i=1}^{m}\mu_i {X_i}^T\beta^- \tag{4.8e}$$

$$+\min_{\gamma^+\geq 0} E\sum_{k\in K}\gamma_k^+ + \sum_{i=1}^{m}\mu_i\sum_{k\in K} r_j(X_i)\gamma_k^+ - \sum_{i=1}^{m}\nu_i\sum_{k\in K} r_j(X_i)\gamma_k^+ \tag{4.8f}$$

$$+\min_{\gamma^-\geq 0} E\sum_{k\in K}\gamma_k^- + \sum_{i=1}^{m}\nu_i\sum_{k\in K} r_k(X_i)\gamma_k^- - \sum_{i=1}^{m}\mu_i\sum_{k\in K} r_k(X_i)\gamma_k^-. \tag{4.8g}$$

Factoring by each response value $y_i \in \mathbb{R}$ and by each primal variable, $\beta_0 \in \mathbb{R}_+$, $\beta_j^+, \beta_j^-, \gamma_k^+, \gamma_k^- \in \mathbb{R}_+$ except $\epsilon_i \in \mathbb{R}$, we obtain

$$
\begin{aligned}
q(\mu,\nu)_{\mu,\nu\geq 0} =& \sum_{i=1}^{m} y_i(\nu_i - \mu_i) && \text{(4.9a)}\\
&+ \min_{\epsilon \geq 0} \sum_{i=1}^{m} \epsilon_i^p - (\mu_i + \nu_i)\epsilon_i && \text{(4.9b)}\\
&+ \min_{\beta_0} \beta_0 \sum_{i=1}^{m} (\mu_i - \nu_i) && \text{(4.9c)}\\
&+ \min_{\beta^+ \geq 0} \sum_{j=1}^{n} \beta_j^+ \left(C + \sum_{i=1}^{m} \nu_i x_{ij} - \sum_{i=1}^{m} \mu_i x_{ij} \right) && \text{(4.9d)}\\
&+ \min_{\beta^- \geq 0} \sum_{j=1}^{n} \beta_j^- \left(C + \sum_{i=1}^{m} \mu_i x_{ij} - \sum_{i=1}^{m} \nu_i x_{ij} \right) && \text{(4.9e)}\\
&+ \min_{\gamma^+ \geq 0} \sum_{k \in K} \gamma_k^+ \left(E + \sum_{i=1}^{m} \mu_i r_k(X_i) - \sum_{i=1}^{m} \nu_i r_k(X_i) \right) && \text{(4.9f)}\\
&+ \min_{\gamma^- \geq 0} \sum_{k \in K} \gamma_k^- \left(E + \sum_{i=1}^{m} \nu_i r_k(X_i) - \sum_{i=1}^{m} \mu_i r_k(X_i) \right). && \text{(4.9g)}
\end{aligned}
$$

When $p = 1$, (4.9b) is: $\min_{\epsilon \geq 0} \sum_{i=1}^{m} \epsilon_i - (\mu_i + \nu_i)\epsilon_i = \min_{\epsilon \geq 0} \sum_{i=1}^{m} \epsilon_i(1 - \mu_i - \nu_i)$. The Lagrangian function $q(\mu, \nu)$ must be less than equal to the optimal objective value of the primal REPR problem. Since a primal variable $\beta_0 \in \mathbb{R}$ is unrestricted, its coefficient has to be 0. Since each primal variable $\epsilon_i, \beta_j^+, \beta_j^-, \gamma_k^+, \gamma_k^- \in \mathbb{R}_+$ must be non-negative, its coefficient has to be non-negative. Hence, the dual formulation of the REPR master problem for $p = 1$ is:

$$
\begin{aligned}
&\max_{\mu,\nu\geq 0} && \sum_{i=1}^{m} y_i(\nu_i - \mu_i) && && (4.10a)\\
&\text{ST} && \sum_{i=1}^{m}(\mu_i - \nu_i) = 0 && && (4.10b)\\
& && \mu_i + \nu_i = \epsilon_i, && i = 1, \ldots, m && (4.10c)\\
& && C + \sum_{i=1}^{m} \nu_i x_{ij} - \sum_{i=1}^{m} \mu_i x_{ij} \geq 0, && j = 1, \ldots n && (4.10d)\\
& && C + \sum_{i=1}^{m} \mu_i x_{ij} - \sum_{i=1}^{m} \nu_i x_{ij} \geq 0, && j = 1, \ldots n && (4.10e)\\
& && E + \sum_{i=1}^{m} \mu_i r_k(X_i) - \sum_{i=1}^{m} \nu_i r_k(X_i) \geq 0, && k \in K && (4.10f)\\
& && E + \sum_{i=1}^{m} \nu_i r_k(X_i) - \sum_{i=1}^{m} \mu_i r_k(X_i) \geq 0, && k \in K. && (4.10g)
\end{aligned}
$$

When $p = 2$, (4.9b) is: $\min_{\epsilon\geq 0} \sum_{i=1}^{m} \epsilon_i^2 - (\mu_i + \nu_i)\epsilon_i$. Since $\epsilon_i^2 - (\mu_i + \nu_i)\epsilon_i$ is a convex function, it attains its minimum value when $(\partial/\partial\epsilon_i)[\epsilon_i^2 - \mu_i\epsilon_i - \nu_i\epsilon_i] = 2\epsilon_i - \mu_i - \nu_i = 0$. Solving this equation, $\epsilon_i = \dfrac{\mu_i + \nu_i}{2}$. By substituting ϵ_i,

$$
\begin{aligned}
\epsilon_i^2 - (\mu_i + \nu_i)\epsilon_i &= (\frac{\mu_i + \nu_i}{2})^2 - (\mu_i + \nu_i)\frac{\mu_i + \nu_i}{2}\\
&= \frac{(\mu_i + \nu_i)^2}{4} - \frac{(\mu_i + \nu_i)^2}{2}\\
&= \frac{(\mu_i + \nu_i)^2 - 2(\mu_i + \nu_i)^2}{4}\\
&= -\frac{(\mu_i + \nu_i)^2}{4}.
\end{aligned}
$$

Therefore, we can substitute for (4.9b) using:

$$
\min_{\epsilon\geq 0} \sum_{i=1}^{m} \epsilon_i^2 - (\mu_i + \nu_i)\epsilon_i = -\sum_{i=1}^{m} \frac{(\mu_i + \nu_i)^2}{4}.
$$

Therefore, the dual formulation for $p = 2$ is:

$$
\begin{aligned}
\max_{\mu,\nu\geq 0} \quad & \sum_{i=1}^{m} y_i(\nu_i - \mu_i) - \frac{1}{4}\sum_{i=1}^{m}(\nu_i + \mu_i)^2 && && \text{(4.13a)}\\
\text{ST} \quad & \sum_{i=1}^{m}(\mu_i - \nu_i) = 0 && && \text{(4.13b)}\\
& C + \sum_{i=1}^{m} \nu_i x_{ij} - \sum_{i=1}^{m} \mu_i x_{ij} \geq 0, && j = 1, \dots n && \text{(4.13c)}\\
& C + \sum_{i=1}^{m} \mu_i x_{ij} - \sum_{i=1}^{m} \nu_i x_{ij} \geq 0, && j = 1, \dots n && \text{(4.13d)}\\
& E + \sum_{i=1}^{m} \mu_i r_k(X_i) - \sum_{i=1}^{m} \nu_i r_k(X_i) \geq 0, && k \in K && \text{(4.13e)}\\
& E + \sum_{i=1}^{m} \nu_i r_k(X_i) - \sum_{i=1}^{m} \mu_i r_k(X_i) \geq 0, && k \in K. && \text{(4.13f)}
\end{aligned}
$$

Constraints (4.10b) and (4.13b) correspond to the primal variable β_0, dual constraints (4.10d), (4.10e), (4.13c), and (4.13d) correspond to the primal variables β_j^+ and β_j^-, and the dual constraints (4.10f), (4.10g), (4.13e), and (4.13f) correspond to the primal variables γ_k^+ and γ_i^-. The primal variable ϵ_i corresponds to constraint (4.10c) in the $p = 1$ case and to the quadratic objective function term in the $p = 2$ case.

Since the every restricted master problem contains the variables $\beta_0 \in \mathbb{R}$ and $\epsilon_i, \beta_j^+, \beta_j^- \in \mathbb{R}_+$, the dual constraints other than (4.10f), (4.10g)/(4.13e), (4.13f) are satisfied at every column generation iteration. However, the constraints (4.10f), (4.10g)/(4.13e), (4.13f) are not necessarily satisfied because not all the possible variables γ_k^+, γ_k^- are present in the restricted master. As is customary in column generation, the pricing problem may be viewed as finding the most violated dual constraint of the form (4.10f), (4.10g)/(4.13e), (4.13f).

If we take K to be the set of all possible boxes on $\mathbb{R}^n$, the pricing problem (4.6) may be reduced to RMA by setting $w_i = \mu_i - \nu_i$ for each $i = 1, \dots, m$, where $\mu, \nu \in \mathbb{R}_+^m$ are the respective dual variables for constraints (4.3b) and (4.3c) of the restricted master problem, as above.

The RMA subproblems generated by our column generation procedure have some special structure: for any given $i = 1, \dots, m$, only one of the constraints (4.3b) or (4.3c) can be binding, except in the situation that $f_{\beta_0,\beta^+,\beta^-,\gamma^+,\gamma^-}(X_i) = y_i$ and $\epsilon_i = 0$, that is, the model exactly fits the response y_i. Otherwise, either the current model overestimates y_i, in which case constraint (4.3c) is slack and $\nu_i = 0$, or conversely y_i is underestimated, and constraint (4.3b) is slack and $\mu_i = 0$.

Consider now the case $p = 1$. The dual constraints (4.10c) correspond to the primal variables ϵ_i, so for any observation y_i not being estimated exactly, complementary slackness requires that constraint (4.10c) be binding. Thus we deduce that

$$\begin{aligned} w_i &= \mu_i = 1 && \text{when observation } i \text{ is overestimated} \\ w_i &= -\nu_i = -1 && \text{when observation } i \text{ is underestimated.} \end{aligned}$$

Next consider the case $p = 2$. If $(\epsilon, \beta^+, \beta^-, \gamma^+, \gamma^-, \beta_0)$ is an optimal primal solution and (μ, ν) is an optimal dual solution, then for any $i = 1, \ldots, m$, we should have from the optimality conditions of the minimization within (4.7) that $(\partial/\partial\epsilon_i)[\epsilon_i^2 - \mu_i\epsilon_i - \nu_i\epsilon_i]$, which reduces to $2\epsilon_i = \mu_i + \nu_i$. We therefore conclude that

$$\begin{aligned} w_i &= \mu_i = 2\epsilon_i && \text{when observation } i \text{ is overestimated} \\ w_i &= -\nu_i = -2\epsilon_i && \text{when observation } i \text{ is underestimated.} \end{aligned}$$

Combining all these observations and supposing that no observations are fitted exactly, we conclude that in the $p = 1$ case the pricing problem is to locate a box in which overestimated observations most outnumber underestimated ones or *vice versa.* If $p = 2$, the objective of the pricing problem is instead to find a box in which the sum of overestimation errors most greatly exceeds the sum of underestimation errors or *vice versa.*

Within the context of our REPR regression method, we set the RMA weight vector to $w = \mu - \nu$, discretize the data X using using the process described in Section 2.1.1, and solve the RMA problem. For unseen datasets, the resulting boxes are translated back to the original, pre-integerized coordinate system, as shown in Section 3.1.1.

4.2 Full Algorithm and Implementation

Algorithm 10 REPR: Rule-enhanced penalized regression

```
Input: data $X \in \mathbb{R}^{m \times n}, y \in \mathbb{R}^m$, penalty parameters $C, E \geq 0$, column generation tolerance $\theta \geq 0$, integer $t \geq 1$, aggregation tolerance $\delta \geq 0$, and iteration limit $S$
Output: $\beta_0 \in \mathbb{R}, \beta \in \mathbb{R}^n, K' \subset \mathcal{K}_\delta(X)$, $\gamma \in \mathbb{R}^{|K'|}$
REPR:

$K' \leftarrow \emptyset$
for $s = 1, \ldots, S$ do
  Solve the restricted master problem to obtain optimal primal variables $(\beta_0, \beta^+, \beta^-, \gamma^+, \gamma^-)$ and dual variables $(\nu, \mu)$
  Use the RMA branch-and-bound algorithm, with preprocessing as in Section 2.1.1,
  to identify a set of $t$ of the best possible solutions $k_1, \ldots, k_t$ to
      $\max_{k \in \mathcal{K}_\delta(X)} \left| \sum_{i=1}^{m} r_k(x_i)(\nu_i - \mu_i) \right|,$      (4.14)
  with objective values $z_1 \geq z_2 \geq \cdots \geq z_t$
  if $z_1 \leq E + \theta$ break
  for each $l \in \{1, \ldots, t\}$ with $z_l > E + \theta$ do
    $K' \leftarrow K' \cup \{k_l\}$ where the box $k_l$ is translated back to the original scale as in Section 3.1.1
  end for
end for
return $(\beta_0, \beta := \beta^+ - \beta^-, K', \gamma := \gamma^+ - \gamma^-)$
```

The pseudocode in Algorithm 10 describes our full REPR column generation procedure for solving (4.3), preprocessing the input data using the procedure described in Section 2.1.1 and then solving the pricing problem using the branch-and-bound methods described in Sections 2.3-2.5. Several points bear mentioning: first, the nonnegative scalar parameter θ allows us to incorporate a tolerance into the column generation stopping criterion, so that we terminate when all reduced costs exceed $-\theta$ instead of when all reduced costs are nonnegative. This kind of tolerance is customary in column generation methods. The tolerance δ, on the other hand, controls the space of columns searched over. Furthermore, using some features already available in PEBBL, our implementation of the RMA branch-and-bound algorithm can identify any desired number $t \geq 1$ of the best possible RMA solutions, as opposed to just one value of k attaining the maximum in (4.14). This t is also a parameter to our procedure, so at each iteration of Algorithm 10 we may adjoin up to t new rules to K'. Adding multiple columns per iteration is a common technique in column generation methods. Finally, the algorithm has a parameter S specifying a limit on the number of column generation iterations, meaning that at the output model will contain at most St rules.

In addition to implementing the RMA branch-and-bound algorithm using C++ and PEBBL as mentioned above, we implemented the remainder of Algorithm 10 in C++, using the Gurobi commercial optimizer [22] to solve the restricted master problems, employing its LP or QP simplex method and "warm starting" from the optimal solution basis of the previous restricted master problem for $s > 1$. If one is careful to "warm-start" the solution of the restricted master problem in this way, solving the RMA pricing problem tends to be the most time-consuming part of Algorithm 10, so we used true parallel computing only in that portion of the algorithm. The remainder of the algorithm, including solving the restricted master problems, was executed in serial and redundantly on all processors.

4.3 Computational Testing

4.3.1 Setting Parameters by Cross Validation

To evaluate the performance REPR, we need to set its parameters, in particular the L_1 penalty coefficients C and E. Small values of C and E (along with a high iteration limit S) can lead to overfitting — good apparent performance on training data but relatively poor performance on testing data. Our computational test utilizes the bilevel cross-validation method, as shown in Section 3.3.2.

The evaluated competing models include RuleFit, random forests, gradient boosting, and simple linear regression. Some of these methods may perform their own internal parameter optimizations.

4.3.2 Small Test Datasets and Data Standardization

We tested the REPR procedure on some small datasets from the UCI data repository [25], as described in Table 4.1. The column of this table shows $|K_0(X)|$, the total number of boxes $B(a, b)$ that have distinguishable coverage on each dataset.

Dataset	m	n	$\lvert K_0(X)\rvert$
SERVO	167	10	9.8×10^{5}
CONCRETE	103	9	2.7×10^{29}
MACHINE	209	6	2.5×10^{15}
YACHT	308	6	2.6×10^{10}
MPG	392	7	3.3×10^{19}
COOL	768	8	1.1×10^{10}
HEAT	768	8	1.1×10^{10}
AIRFOIL	1503	5	1.0×10^{11}

Table 4.1: Summary of the small experimental datasets.

Before applying REPR, we standardized the problem input data so that the response vector y and all columns x_j of X have mean zero and standard deviation 1, as follows: given any non-standardized problem input instance $(\hat{X}, \hat{y})$, we set

$$y_i = \frac{\hat{y}_i - \mu(\hat{y})}{\sigma(\hat{y})} \quad \forall\, i = 1, \dots, m \quad x_{ij} = \frac{\hat{x}_{ij} - \mu(\hat{x}_j)}{\sigma(\hat{x}_j)} \quad \forall\, i = 1, \dots, m \quad \forall\, j = 1, \dots, n,$$

where $\mu(\,\cdot\,)$ and $\sigma(\,\cdot\,)$ respectively denote computing the mean and (sample) standard deviation of a vector, and $\hat{x}_j$ denotes the j^{th} column of $\hat{X}$. For input data containing categorical attributes as well as numerical ones, we also converted each c-way categorical attribute into $c - 1$ binary attributes.

4.3.3 Numerical Results for Small Datasets

We used our bilevel cross-validation procedure twice with $q_{\text{out}} = 5$ outer folds and $q_{\text{in}} = 3$ inner folds. This procedure gave us a performance sample of size 10 for each dataset, but each with an 80-20 data split between training and testing data.We selected the REPR's parameters as follows:

- We used our inner cross-validation procedure to select the parameters C and E. We experimented only with cases in which $C = E$, selecting the possible values from $\{0, 0.5, 1, 1.5, 2\}$.

- We did complete tests for both $p = 1$ (absolute deviation loss) and $p = 2$ (quadratic loss).

- After some initial experimentation, we used $\delta = 0.005$.

- The iteration limit was $S = 150$.
- We set $t = 1$, so we only added one box rule per iteration of column generation.

In addition to testing the REPR procedure shown in Algorithm 10, we also tested a variant we call GREPR, for "greedy REPR". In this variant, we replace step 7 of Algorithm 10 with a single invocation to the greedy method of Section 2.5. Thus, GREPR does not attempt to solve the pricing problem exactly, substituting a greedy heuristic instead. In this case, we are not able to use the termination test in line 8 of Algorithm 10, because the greedy method solution may not necessarily correspond to the column with the most negative reduced cost. Instead, we simply terminate GREPR whenever the greedy method returns the same box rule on two successive calls (which would otherwise cause the method to enter an infinite loop). So far, we have only experimented with GREPR for $p = 2$.

We compared the performance of REPR and GREPR to RuleFit, random forests, gradient boosting and classical linear regression. To implement RuleFit, random forests, gradient boosting and classical linear regression, we used their publicly available R packages with default parameter settings; the variant of gradient boosting implemented is described in [27].

Table 4.2 summarizes the mean squared error (MSE) performance of the various methods, averaged over the 10 runs in our two invocations of the cross-validation procedure and scaled by $(1/m)\sum_{i=1}^{m} y_i^2$. The smallest average MSE for each dataset is shown in bold. Figure 4.1 displays the same information as a bar chart, normalized so that the $p = 1$ REPR value has height 1 for each dataset. In each case, the smallest average MSE is attained by either the $p = 1$ or $p = 2$ case of REPR. GREPR's prediction accuracy is comparable to but slightly worse overall than the two REPR variants.

Table 4.3 summarizes the standard deviation of the sample of 10 calculated MSE values, which we take as a measure of prediction stability. The values shown are scaled by $(1/m)\sum_{i=1}^{m} |y_i|$. Together, the $p = 2$ REPR and GREPR methods have the lowest standard deviation for every dataset except CONCRETE and HEAT. For HEAT, the $p = 1$ REPR variant is lowest, and linear regression has the lowest standard deviation for

	Dataset			
Method	SERVO	CONCRETE	MACHINE	YACHT
REPR ($p = 1$)	0.14377	**0.00346**	**0.23200**	0.00846
REPR ($p = 2$)	**0.11180**	0.00436	0.27860	**0.00282**
GREPR ($p = 2$)	0.15030	0.00513	0.24725	0.01296
RuleFit	0.14275	0.00730	0.51282	0.00728
Random forests	0.27276	0.01303	0.27119	0.14695
Gradient boosting	0.31585	0.00896	0.60671	0.02152
Linear regression	0.68122	0.00562	0.48644	0.77343

	Dataset			
Method	MPG	COOL	HEAT	AIRFOIL
REPR ($p = 1$)	**0.01228**	0.00038	0.00152	0.00020
REPR ($p = 2$)	0.01531	**0.00035**	**0.00086**	**0.00018**
GREPR ($p = 2$)	0.01488	0.00066	0.00103	0.00021
RuleFit	0.01376	0.00049	0.00459	0.00026
Random forests	0.01438	0.00239	0.00569	0.00084
Gradient boosting	0.02085	0.00144	0.00503	0.00074
Linear regression	1.02085	0.01763	0.01722	0.00150

Table 4.2: Average MSE over the small datasets, scaled by $(1/m)\sum_{i=1}^{m} y_i^2$. The smallest value in each column is bolded.

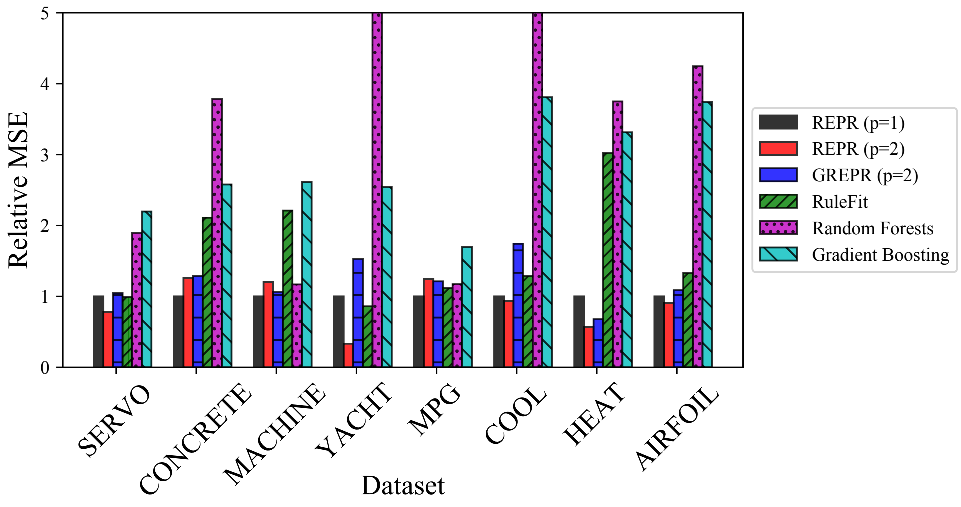


Figure 4.1: Visual comparison of average MSE values, with the REPR value normalized to 1.

	Dataset							
Method	SERVO	CONCRETE	MACHINE	YACHT	MPG	COOL	HEAT	AIRFOIL
REPR ($p = 1$)	0.21450	0.10030	60.03165	0.08461	0.09040	0.00603	**0.00498**	0.00125
REPR ($p = 2$)	**0.15365**	0.09101	**55.37055**	0.02701	0.08716	0.00133	0.00694	**0.00371**
GREPR ($p = 2$)	0.20063	0.08711	63.92032	**0.02070**	**0.07597**	**0.00065**	0.00891	0.00429
RuleFit	0.16335	0.09818	89.88322	0.03146	0.11868	0.00175	0.02073	0.00865
Random forests	0.19868	0.20743	101.31404	0.53098	0.12293	0.00954	0.01840	0.01454
Gradient boosting	0.20947	0.12787	96.74113	0.28814	0.09804	0.00514	0.01844	0.00704
Linear regression	0.35761	**0.06793**	53.50609	1.43197	0.08256	0.06962	0.03962	0.00649

Table 4.3: Standard deviation of MSE over the small datasets, scaled by $(1/m)\sum_{i=1}^{m}|y_i|$. The smallest value in each column is bolded.

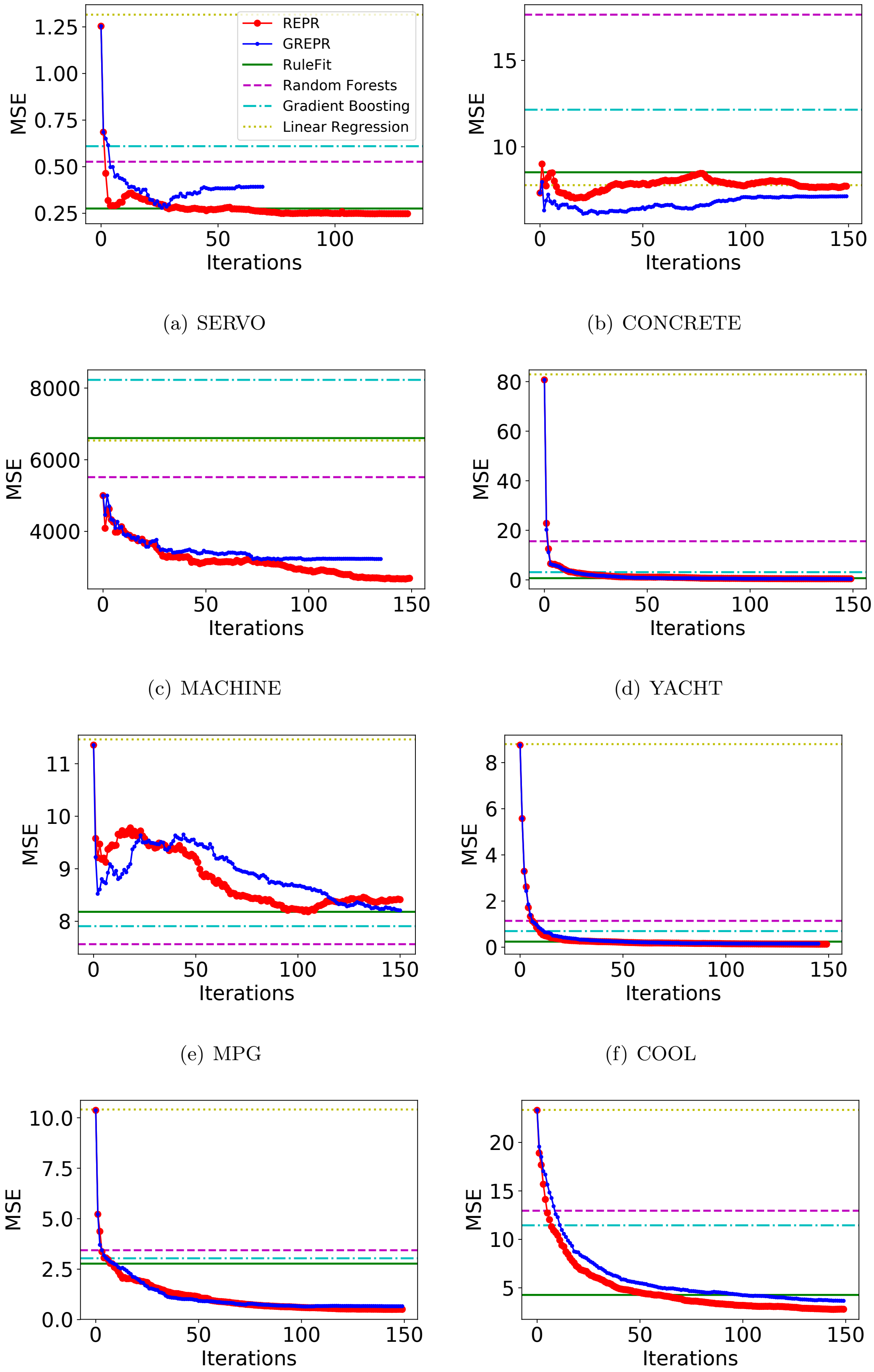
	Dataset							
Method	SERVO	CONCRETE	MACHINE	YACHT	MPG	COOL	HEAT	AIRFOIL
REPR ($p = 1$)	1.9	74.7	46.7	3.3	1261.1	9.5	9.8	42.9
REPR ($p = 2$)	1.6	115.7	41.7	5.6	6921.9	14.3	16.9	87.1
GREPR ($p = 2$)	0.2	0.2	0.7	0.4	3.0	5.0	4.2	29.9

Table 4.4: Average REPR and GREPR run times in seconds over the small datasets.

CONCRETE; however the prediction performance of linear regression on CONCRETE is poor, so stability of prediction is not of great benefit.

Table 4.4 summarizes the run time performance of REPR and GREPR, in average seconds per run. Each run was performed on a single 2.1GHz Xeon E5-2695 v4 node of one of our university computing resource clusters. The REPR tests ran their pricing problems in parallel on the 36 processor cores available on each node, whereas GREPR is fully serial. REPR is much slower than the competing methods we tested; the competing methods typically ran in less than a second on the datasets in Table 4.1 and no case took more than 2.3 seconds. However, REPR's running time is not prohibitive for the small datasets of Table 4.1, except for the REPR methods on the MPG dataset, for which the RMA branch-and-bound trees averaged millions of search nodes. The MPG runs could therefore be sped up by using more processor cores to accelerate the branch-and-bound search. Another way to improve run times is to simply use GREPR, which runs much faster for most models. Sometimes, however, as in the case of AIRFOIL, the restricted master problems can take significant time to solve, in which case the run time advantage of GREPR is less pronounced.

The charts in Figure 4.2 give more detailed information for the small datasets. For the C and E parameter values selected by the inner cross validation step, these charts show how the $p = 2$ REPR and GREPR prediction MSEs (in the original, non-standardized coordinate system) evolve with each iteration, with each data point averaged over the two 5-fold cross-validations; the horizontal lines indicate the average MSE level for the competing procedures. MSE generally declines as REPR adds rules, although some diminishing returns are evident for CONCRETE. The first iteration of the REPR models is equivalent to a simple LASSO model, which has similar performance to linear regression. We observed that GREPR tends to stop earlier than the exact REPR algorithm. Interestingly, the charts show only limited evidence of overfitting by REPR and GREPR, even when large numbers of rules are incorporated into the model.



(a) SERVO

(b) CONCRETE

(c) MACHINE

(d) YACHT

(e) MPG

(f) COOL

(g) HEART

(h) AIRFOIL

Figure 4.2: MSE as a function of iteration count for the smaller datasets.

Dataset	m	n	$\lvert K_0(X)\rvert$
SML_dining	4127	12	1.4×10^{67}
SML_room	4127	12	1.4×10^{67}
parkinsons_total	5875	20	2.0×10^{117}
parkinsons_motor	5875	20	2.0×10^{117}

Table 4.5: Summary of larger datasets.

	Dataset			
Method	SML_dining	SML_room	parkinsons_total	parkinsons_motor
GREPR	$\mathbf{1.763 \times 10^{-6}}$	$\mathbf{1.456 \times 10^{-3}}$	6.937×10^{-3}	9.429×10^{-3}
RuleFit	6.741×10^{-6}	3.675×10^{-3}	6.509×10^{-3}	8.906×10^{-3}
Random forests	2.269×10^{-6}	1.501×10^{-3}	$\mathbf{4.491 \times 10^{-3}}$	$\mathbf{5.425 \times 10^{-3}}$
Gradient boosting	1.228×10^{-5}	4.720×10^{-3}	2.595×10^{-2}	2.592×10^{-2}
Linear regression	1.606×10^{-5}	6.933×10^{-3}	1.023×10^{-1}	1.151×10^{-1}

Table 4.6: Average MSE over the larger datasets. The smallest value in each column is bolded

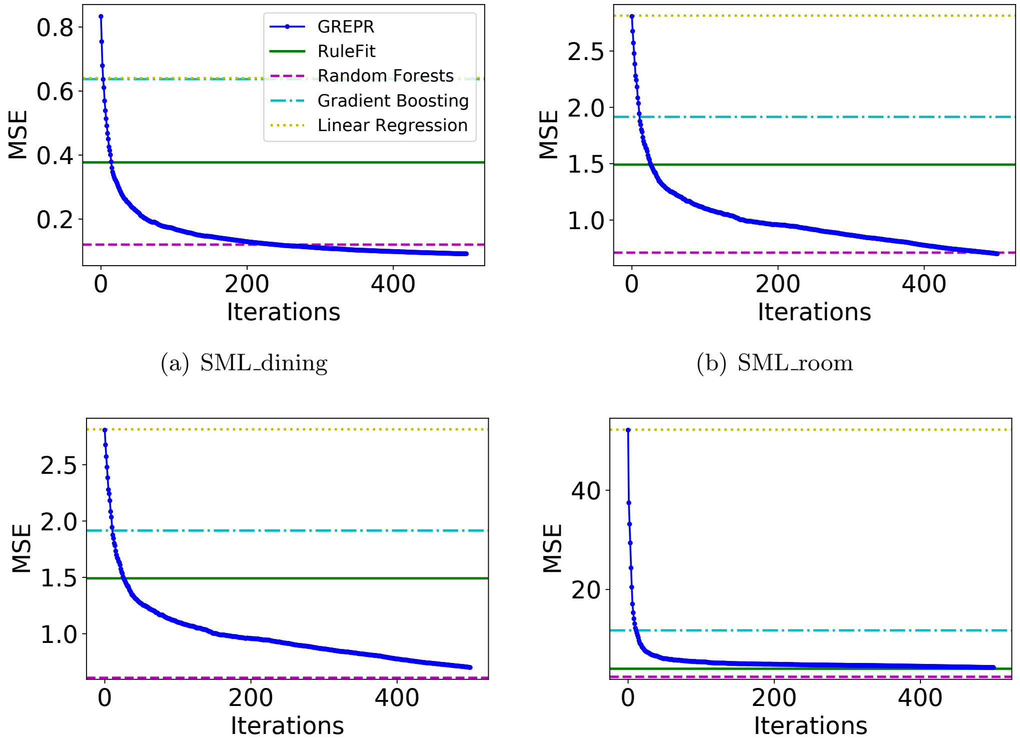
4.3.4 Preliminary Numerical Results with Larger Datasets

We also made some preliminary experiments with the $p = 2$ GREPR method and the non-REPR competitors on four UCI datasets having similarly small numbers of attributes n, but considerably larger numbers of observations m, as summarized in Table 4.5. We used the same testing methodology as in the previous section, except that we let GREPR run for up to $S = 500$ iterations and set $\delta = 0$ in the discretization procedure.

The average MSE levels produced by the various methods are presented in Table 4.6. Here, the results appear less favorable to our class of methods. GREPR obtains the lowest MSE values for the two "SML" datasets, but the random forest method obtains the lowest MSE on the Parkinsons datasets. Table 4.7 presents the standard deviation of MSE values obtained on the larger datasets. Again, GREPR obtains the smallest values on the two SML datasets, but random forests obtain the lowest value for the Parkinsons datasets. Finally, Figure 4.3 shows the evolution of the average MSE with the number of GREPR iterations, similarly to Figure 4.2. Interestingly, there is no evidence of overfitting through 500 iterations.

	Dataset			
Method	SML_dining	SML_room	parkinsons_total	parkinsons_motor
GREPR	$\mathbf{3.586 \times 10^{-5}}$	$\mathbf{2.045 \times 10^{-4}}$	3.671×10^{-4}	1.269×10^{-3}
RuleFit	3.925×10^{-5}	2.882×10^{-4}	7.370×10^{-4}	4.775×10^{-4}
Random forests	6.511×10^{-5}	2.528×10^{-4}	$\mathbf{3.192 \times 10^{-4}}$	$\mathbf{3.375 \times 10^{-4}}$
Gradient boosting	1.519×10^{-4}	3.389×10^{-4}	1.709×10^{-3}	7.396×10^{-4}
Linear regression	1.976×10^{-4}	7.375×10^{-4}	3.673×10^{-3}	2.829×10^{-3}

Table 4.7: Standard deviation of MSE over the larger datasets. The smallest value in each column is bolded



(a) SML_dining

(b) SML_room

(c) parkinsons_total

(d) parkinsons_motor

Figure 4.3: MSE as a function of iteration count for the larger datassets.

Chapter 5

Conclusions and Possible Improvements

This dissertation described an efficient parallel branch-and-bound algorithm for the $\mathcal{NP}$-hard RMA problem, with non-strong branching. Using parallel computing further improved the runtime and capacity to solve larger RMA problems exactly. The two-class classification and regression applications of the RMA problem outperformed some existing algorithms in prediction accuracy and stability.

The following list contains possible improvements and extensions the work in to this dissertation. These could be explored in further research.

- Use of dimensional reduction techniques to reduces the level of difficulty, such as removing some attributes highly related to the other attributes, for the RMA problem with maintaining an optimal or near-optimal solution

- Removing infeasible regions by generating cutting planes before our parallel enumeration method using PEBBL

- An optimization-based bound to improve the bounding process in the branch-and-bound algorithm

- An efficient parallel implementation to solve the LPBR and REPR restricted master problems, instead of using a commercial MIP solver, Gurobi

- A mini-batch method to find multiple boxed-based rules in parallel in each column-generation iteration

- An efficient choice of parameters for LPBR and REPR master problems

References

[1] Timo Aho, Bernard Ženko, Sašo Džeroski, and Tapio Elomaa. Multi-target regression with rule ensembles. *J. Mach. Learn. Res.*, 13(Aug):2367–2407, 2012.

[2] Kristin P. Bennett and Erin J. Bredensteiner. Duality and geometry in svm classifiers. In *In Proc. 17th International Conf. on Machine Learning*, pages 57–64. Morgan Kaufmann, 2000.

[3] Kristin P. Bennett and Colin Campbell. Support vector machines: Hype or hallelujah? *SIGKDD Explor. Newsl.*, 2(2):1–13, December 2000.

[4] Endre Boros, Peter L. Hammer, Toshihide Ibaraki, and Alexander Kogan. Logical analysis of numerical data. *Mathematical Programming*, 79(1):163–190, Oct 1997.

[5] Leo Breiman. Random forests. *Mach. Learn.*, 45:5–32, 2001.

[6] Jens Clausen. Branch and bound algorithms – principles and examples, 1999.

[7] William W. Cohen and Yoram Singer. A simple, fast, and effective rule learner. In *Proc. of the 16th Nat. Conf. on Artificial Intelligence*, pages 335–342, 1999.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[9] Krzysztof Dembczyński, Wojciech Kotłowski, and Roman Słowiński. Solving regression by learning an ensemble of decision rules. In *International Conference on Artificial Intelligence and Soft Computing, 2008*, volume 5097 of *Lecture Notes in Artificial Intelligence*, pages 533–544. Springer-Verlag, 2008a.

[10] Krzysztof Dembczyński, Wojciech Kotłowski, and Roman Słowiński. Maximum likelihood rule ensembles. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 224–231, New York, NY, USA, 2008b. ACM.

[11] Ayhan Demiriz, Kristin P. Bennett, and John Shawe-Taylor. Linear programming boosting via column generation. *Mach. Learn.*, 46(1-3):225–254, March 2002.

[12] Jonathan Eckstein and Noam Goldberg. An improved branch-and-bound method for maximum monomial agreement. *INFORMS J. Comput.*, 24(2):328–341, 2012.

[13] Jonathan Eckstein, William E. Hart, and Cynthia A. Phillips. PEBBL: an object-oriented framework for scalable parallel branch and bound. *Math. Program. Comput.*, 7(4):429–469, 2015.

[14] Lester R. Ford, Jr. and David R. Fulkerson. A suggested computation for maximal multi-commodity network flows. *Manage. Sci.*, 5:97–101, 1958.

[15] Scott Fortmann-Roe. Understanding the bias-variance tradeoff, 2012. `http://scott.fortmann-roe.com/docs/BiasVariance.html`.

[16] Robert Fourer, David M. Gay, and Brian W. Kernighan. Ampl: A mathematical programing language. In Stein W. Wallace, editor, *Algorithms and Model Formulations in Mathematical Programming*, pages 150–151, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[17] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, EuroCOLT '95, pages 23–37, London, UK, UK, 1995. Springer-Verlag.

[18] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Ann. of Stat.*, pages 1189–1232, 2001.

[19] Jerome H. Friedman and Bogdan E. Popescu. Predictive learning via rule ensembles. *Ann. Appl. Stat.*, 2(3):916–954, 2008.

[20] Paul C. Gilmore and Ralph E. Gomory. A linear programming approach to the cutting-stock problem. *Oper. Res.*, 9:849–859, 1961.

[21] Igor Griva, Stephen G. Nash, and Ariela Sofer. *Linear and Nonlinear Optimization.* SIAM, second edition, 2009.

[22] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016.

[23] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R.* Springer, 2013.

[24] Miroslav Kubat. *Introduction to machine learning.* Springer, 2015.

[25] Moshe Lichman. UCI machine learning repository, 2013. University of California, Irvine, School of Information and Computer Sciences, `http://archive.ics.uci.edu/ml`.

[26] Gunnar Rätsch, Bernhard Schölkopf, Alex J. Smola, Klaus-Robert Müller, Takashi Onoda, and Sebastian Mika. v-arc: Ensemble learning in the presence of outliers. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 561–567. MIT Press, 2000.

[27] Greg Ridgeway. Generalized boosted models: a guide to the gbm package, 2007. `http://www.saedsayad.com/docs/gbm2.pdf`.

[28] R. Tibshirani. Regression shrinkage and selection via the lasso. *J. R. Statist. Soc. B*, 58(1):267–288, 1996.

[29] Jerome Friedman Trevor Hastie, Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2007.

[30] Sholom M. Weiss and Nitin Indurkhya. Optimized rule induction. *IEEE Expert*, 8(6):61–69, 1993.

[31] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B*, 67:301–320, 2005.

Appendix A

RMA Code Descriptions

RMA is implemented using PEBBL, and [13] explains how to use PEBBL. This chapter additionally explains specific parameters and some main procedures in the RMA solver. An example command to run the RMA solver with a debugging option is:

```
./rma --delta=10 datafile.txt
```

A.1 Parameters

`delta`

Layer:	Serial and Parallel
Datatype:	`double`
Default value:	`-1.0`
Constraints:	`[0, 1.0)` or `-1.0`

This parameter is δ, a relative tolerance to aggregate close consecutive values, as shown in Section 2.1.1. It is scaled by the 95% central confidence interval of the original data distribution.

`binSize`

Layer:	Serial and Parallel
Datatype:	`integer`
Default value:	`-1`
Constraints:	`a positive integer` or `-1`

If this value is set to $L > 0$, the original data are discretized into partitions of L equal intervals.

`limitInterval`

Layer:	Serial and Parallel
Datatype:	`double`
Default value:	`0.1`
Constraints:	`(0, 1.0)`

This parameter limits the maximum fraction of the entire data range in the original non-discretized data values, that can have the same discretized value. It is ρ in Section 2.1.1, and is scaled by the 95% central confidence interval of the original data distribution. If an interval, generated by the current δ, violates this limit, the violated range is recursively discretized by shrinking δ until this limit is no longer violated.

`shrinkDelta`

Layer:	Serial and Parallel
Datatype:	`double`
Default value:	`0.95`
Constraints:	`(0, 1.0)`

This parameter specifies the level of shrinking δ when `limitInterval` caught violated.

`limitDistVals`

Layer:	Serial and Parallel
Datatype:	`integer`
Default value:	`-1`
Constraints:	`a nonnegative integer` or `-1`

If this value is less than the distinct value ℓ_j in attribute j after the recursive discretization and `delta > 0`, then data in attribute j is furthermore discretized by partitioning into L equal intervals.

`removeDuplicateObs`

Layer: Serial and Parallel

Datatype: `bool`

Default value: `true`

If `true`, RMA removes duplicated observations based on the discretized explanatory matrix X, and adds the weight of removed observation to the weight of merging observation.

`getInitialGuess`

Layer: Serial and Parallel

Datatype: `bool`

Default value: `true`

If `true`, RMA implements the greedy heuristic in Section 2.5 to obtain an initial incumbent before the branch-and-bound procedure.

`perCachedCutPts`

Layer: Serial and Parallel

Datatype: `double`

Default value: `0.000001`

Constraints: `(0.0, 1.0]`

If this parameter value is less than `1.0`, RMA implements cutpoint cashing. The default value is `0.000001`. Thus, RMA generally considers only the cached cutpoints if there is at least one applicable cached cutpoint.

`binarySearchCutVal`

Layer: Serial and Parallel

Datatype: `integer`

Default value: `-1`

If this value is greater than 3 and less than the number of distinct values ℓ_j in an attribute j, RMA implements binary cupoint search for attribute j.

`branchSelection`

Layer: Serial

Datatype: `integer`

Default value: `0`

Constraints: {`0,1,2`}

When there are optimal tied solutions, the RMA solver lets the option “0” to randomly chooses one, option “1” to select the first one, and option “2” to select the last one among them.

`countingSort`

Layer: Serial and Parallel

Datatype: `bool`

Default value: `false`

The default sorting algorithm is bucket sort. If `true`, it is replaced by counting sort.

`testWt`

Layer: Serial and Parallel

Datatype: `bool`

Default value: `false`

RMA starts with that each observation has a weight, y_i divided by the number of training observations, in default setting. If this parameter is `true`, RMA can set different weights by using a weight data file, i.e. “testWt.txt”, which contains different weights for a current dataset. It is specified after the current dataset in command line. The number of weights in the file has to be equal to the number of observations in the current data file. An example command to use this feature is:

```
./rma --testWt=true datafile.txt testW.txt
```

`bruteForceEC`

Layer: Serial and Parallel

Datatype: `bool`

Default value: `false`

If `true`, RMA constructs equivalence classes without using the rotation algorithm. This option was created to demonstrate the benefits of using the rotation algorithm.

`bruteIncumbent`

Layer: Serial and Parallel

Datatype: `bool`

Default value: `false`

If `true`, RMA computes an incumbent by a brute force algorithm. It still utilizes Kadane's algorithm to find an incumbent, but it scans all observations for each distinct value of each attribute. This option is a validation method for the current incumbent computation. If the `bruteForceEC` parameter is `true`, then this parameter is automatically set to be `true` as well.

`perLimitAttrib`

Layer: Serial and Parallel

Datatype: `double`

Default value: `1.0`

Constraints: `(0.0, 1.0]`

This parameter limits the percentage of all attributes that RMA inspects. If the first x percentage of attributes are inspected already, RMA does not inspect the other attributes anymore. Therefore, an optimal solution may not be obtained.

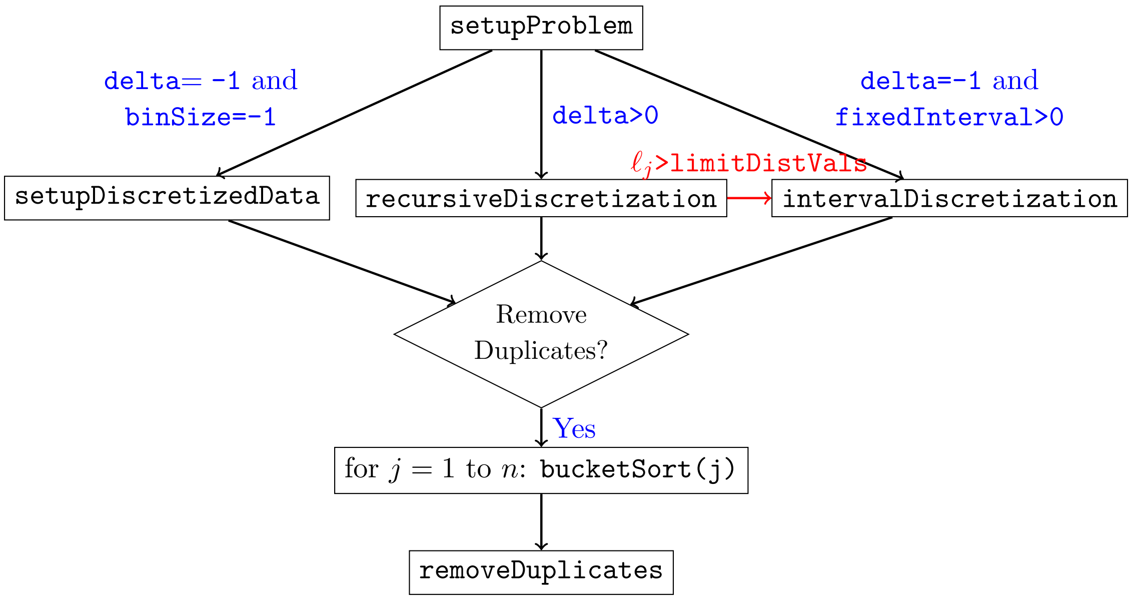


Figure A.1: Procedures for reading a data file and preprocessing data

`writeCutPts`

Layer: Serial

Datatype: `bool`

Default value: `false`

If `true`, RMA writes each cutpoint (j, v) chosen in order into a solution file. This information shows which cutpoints were chosen and in which order they are chosen in each branch of the tree.

A.2 Serial RMA procedures

The user-defined branching and sub-branching classes, `serRMA` and `serRMASub`, are created for the serial branch-and-bound procedures using PEBBL. They are respectively derived from the `branching` and `branchSub` classes of PEBBL.

A.2.1 Methods in `serRMA` class

Figure A.1 shows procedures to read a dataset and to preprocess data in the `serRMA` class.

```
bool serRMA::setupProblem(int& argc, char**& argv)
```

This method reads a data file and stores the dataset information. If explanatory variables are not discretized yet, it can be discretized by the recursive discretization method or the discretization method by fixed intervals.

```
bool serRMA::setupDiscretizedData()
```

If a dataset is already discretized, this method is invoked to setup the problem set.

```
bool serRMA::recursiveDiscretization()
```

If the `delta` parameter is greater than `0`, this function recursively discretizes explanatory variables as shown in Section 2.1.1.

```
bool serRMA::intervalDiscretization()
```

If the `binSize` parameter is not `-1`, this function is invoked to discretize the original data into partitions of L equal intervals.

```
solution* serRMA::initialGuess()
```

This method is invoked after preprocessing data and before the branch-and-bound procedure. If the `getInitialGuess` parameter is `true` (default), it computes an initial greedy solution, as shown in Section 2.5.

A.2.2 Methods in `serRMASub` class

Figure A.2 graphically shows the order of methods invoked to compute the bounds and incumbents in the `serRMASub` class.

```
void serRMASub::boundComputation()
```

The method computes bounds for given applicable cutpoints by the branching option and selects the best one.

```
void serRMASub::createInitialEquivClass()
```

The method creates the initial equivalence classes based on given $(\underline{a}, \overline{a}, \underline{b}, \overline{b})$ in each subproblem.

```
void serRMASub::computeIncumbent(const int& j)
```

This method computes an incumbent using maximization and minimization Kadane's algorithms for attribute j.

```
void serRMASub::strongBranching()
```

If the parameters, `perCachedCutpts = 1` and `binarySearchCutVals = -1`, RMA invokes this method to perform strong branching.

```
void serRMASub::cachedBranching()
```

If the parameters, `perCachedCutpts < 1` and `binarySearchCutVals` = `-1`, as in the default settings, RMA invokes this method to implement cutpoint caching.

```
void serRMASub::binaryBranching()
```

If the parameters `perCachedCutpts = 1` and `binarySearchCutVals` > `0`, RMA invokes this method to perform binary cutpoint search for the attribute if its applicable cutvalues greater than the `binarySearchCutVals` parameter; otherwise, strong branching.

```
void serRMASub::hybrindBranching()
```

If the parameters, `perCachedCutpts < 1` and `binarySearchCutVals > 0` , RMA invokes this method to implement hybrid branching. RMA performs binary cutpoint search for the attribute if its applicable cutvalues greater than the `binarySearchCutVals` parameter; otherwise, cutpoint caching.

```
void serRMASub::splitSubproblem(const int& j, const int& v)
```

This method splits the current subproblem into two or three children by branching at cutpoint (j, v).

```
void serRMASub::dropEquivClass()
```

For a up or down child, RMA invokes this method to drop equivalence classes no longer covered.

```
void serRMASub::mergeEquivClass()
```

For a middle child, RMA invokes this method to merge some equivalence classes.

```
void serRMASub::computeBound()
```

This method computes the bounds of two or three children.

A.3 Parallel RMA procedures

The `parRMA` and `parRMASub` classes contains methods required for the parallel implementation using PEBBL.

A.3.1 Methods in `parRMASub` class

`void parRMA::pack()` `void parRMA::unpack()`

These methods in the `parRMA` class pack or unpack, respectively, the problem instance information such as the number of observation, the number of attributes, explanatory variables, response values, weights, the number of distinct features in each attribute, the number of total cutpoints, and applicable parameters for the parallel implementation.

A.3.2 Methods in `parRMASub` class

`void parRMASub::pack()` `void parRMASub::unpack()`

These methods in the `parRMASub` class pack or unpack, respectively, the four vectors of $(\underline{a}, \overline{a}, \underline{b}, \overline{b})$ for each subproblem and the best local cutpoint (j^*, v^*) to share with the other processors in asynchronous search.

`void parRMASub::boundComputation()`

This method in the `parRMASub` class only computes bounds for assigned cutpoints for each processor in the ramp-up process. In this process, RMA selects the best cutpoint, and broadcasts it to the other processor. Once the ramp-up process ends and PEBBL enters its asynchrous phase, each processor is assigned one subproblem and computes bounds using the `boundComputation` method in the serial class of `RMASub`.

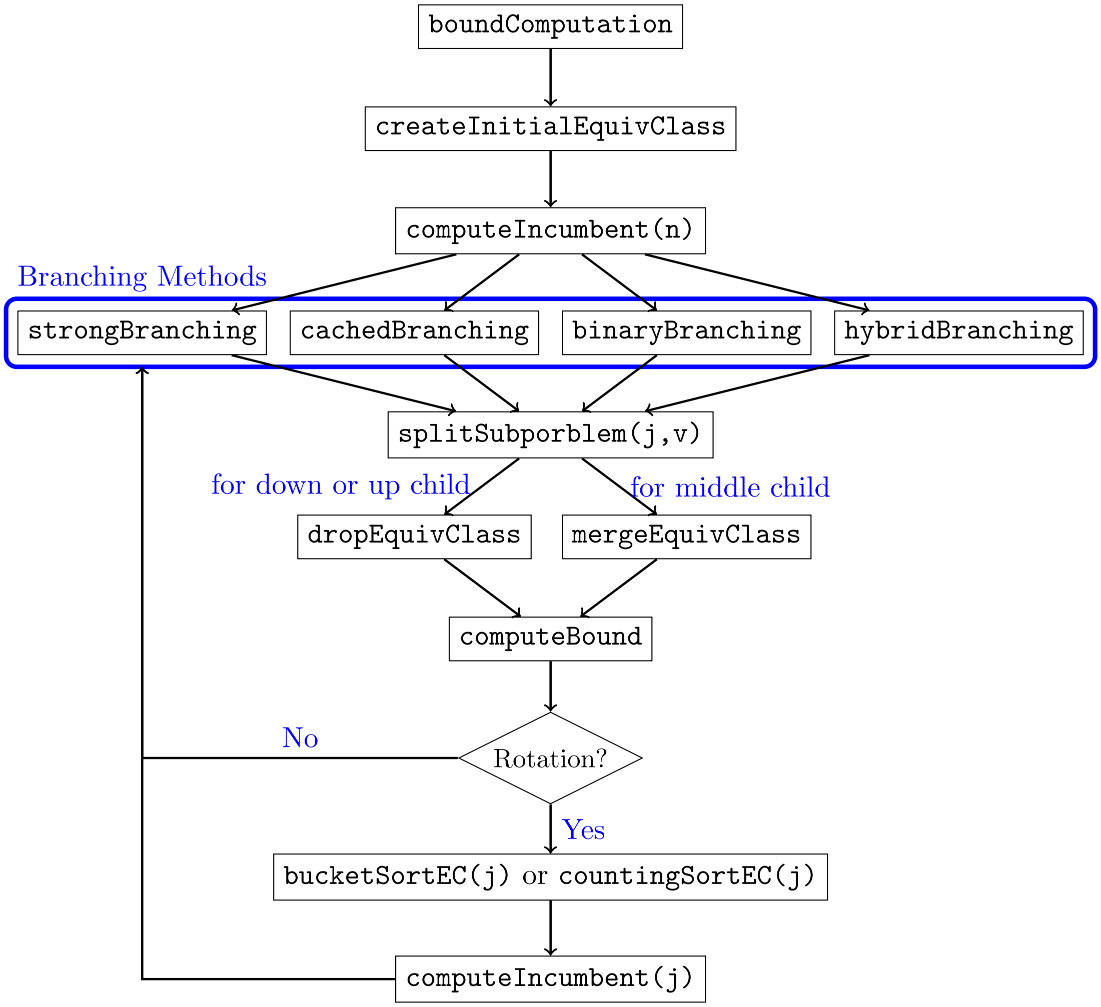


Figure A.2: Procedures to compute bounds for child subproblems induced by the current subproblem: the selected branching method provides each cutpoint (j, v) from attribute $j = 1$ to n in this order. Once RMA finishes checking cutpoints in attribute j, it rotates the sorted observation list. Incumbent computations are embedded within the rotation algorithm.

Appendix B

LPBR and REPR Code Descriptions

This appendix explains the class structure, the parameters, and the methods for LPBR and REPR. These two implementations follow similar procedures with minor differences. Example commands to run the LPBR and REPR solvers respectively with user-selected penalty parameters are:

```
./boosting --lpboost=true --nu=0.5 datafile.txt
./boosting --c==0.5 --e=0.5 datafile.txt
```

B.1 Class Structures

For code portability, the class structure to implement our two boosting algorithms, LPBR and REPR, is organized as shown in Figure B.1. The `allParams` and `rmaParams` classes contain parameters for the boosting procedures and the RMA algorithm, respectively. The `Base` class contains the methods to read a data file name and parameters defined in command line. It is derived from the `allParams` and `rmaParams` classes for easy access to the parameters. The `Data` class, inherited from the `Base` class, contains data from an input data file, the methods related to preprocessing datasets, and the preprocessed data. The `CrossValidation` class, derived from the `Data` class, implements cross-validation procedures by containing the `CompLPBR` and `LPBR` classes for LPBR and the `CompREPR` and `REPR` classes for REPR. The `CompLPBR` and `CompREPR` classes respectively implement the competing models of LPBR and REPR. The `CompLPBR` and `CompREPR` classes are inherited from the `CompModels` class that contains common methods and data for the `CompLPBR` and `CompREPR` classes. The `LPBR` and `REPR` classes respectively implement the column generation procedures of LPBR and REPR. The `LPBR` and `REPR` classes are inherited from the `Boosting` class that contains common

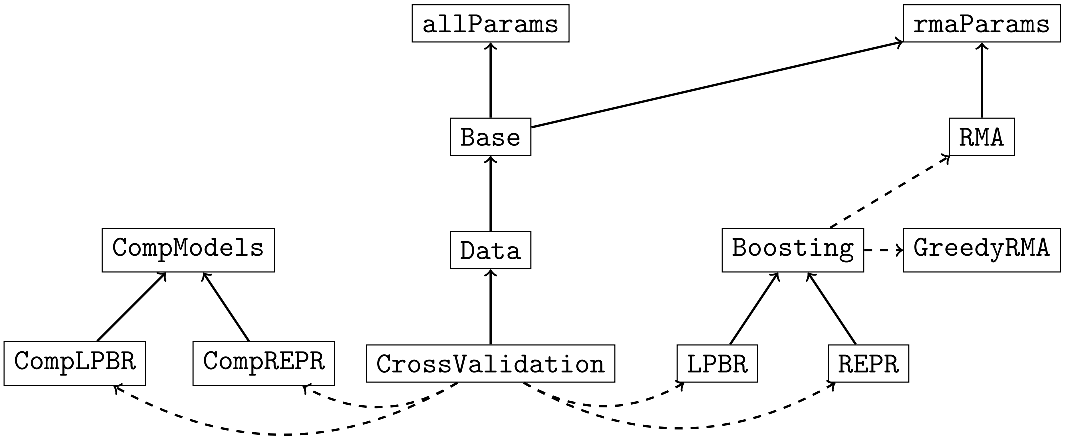


Figure B.1: Inheritance and containment class structure for LPBR and REPR: each class name is specified inside a box. Straight and dashed arrows respectively indicate inheritance and containment.

methods and data for the `LPBR` and `REPR` classes. The `RMA` class implements our parallel branch-and-bound by containing the `serRMA` and `parRMA` classes for serial and parallel layers, respectively, as described in Appendix A. The `RMA` class is derived from the `rmaParams` class. The `GreedyRMA` class contains data and methods required for our greedy heuristic.

B.2 Parameters

The `allParams` class stores parameters for our boosting algorithms. The parameters for RMA, as shown in Appendix A, are still available for LPBR and REPR solvers, and they are contained in the `rmaParams` class, except that the `delta`, `limitInterval`, `shrinkDelta`, `binSize`, `limitDistVals`, and `getInitialGuess` parameters are stored in the `allParams` class. The `testWt` parameter is no longer available for these boosting implementations.

`outerCV`

Application:	LPBR and REPR
Datatype:	`bool`
Default value:	`false`

If `true`, 5-fold cross-validation is implemented.

`innerCV`

Application: LPBR and REPR

Datatype: `bool`

Default value: `false`

If `true`, 3-fold inner cross-validation is implemented after the outer partitions are created.

`shuffleObs`

Application: LPBR and REPR

Datatype: `bool`

Default value: `true`

If `true`, observation indices are shuffled before creating outer partitions.

`readShuffledObs`

Application: LPBR and REPR

Datatype: `bool`

Default value: `false`

If `true`, a pre-shuffled observation-index list is obtained from a text file specified after the current datafile, instead of creating a new shuffled observation index list. This list is used before creating outer partitions. An example command to use this feature is:

```
./boosting --readShuffledObs=true datafile.txt randObservation.txt
```

`writeShuffledObs`

Application: LPBR and REPR

Datatype: `bool`

Default value: `true`

If `true`, a shuffled observation-index list is saved in a text file.

`iterations`

Application:	LPBR and REPR
Datatype:	`integer`
Default value:	`1`
Constraints:	`a positive integer`

This parameter specifies an upper limit on the number of column generation iterations.

`d`

Application:	LPBR
Datatype:	`double`
Default value:	`2/m` (m is the number of training samples)
Constraints:	`a positive double`

This parameter is the non-negative penalty parameter D for LPBR.

`nu`

Application:	LPBR
Datatype:	`double`
Default value:	`0.5`
Constraints:	`(0,1)`

This parameter specifies ν such that $D = \dfrac{1}{\nu m}$ where m is the number of training samples. If the parameter `d` is specified, this parameter is ignored.

`p`

Application:	REPR
Datatype:	`integer`
Default value:	`1`
Constraints:	`1` or `2`

This parameter specifies p in the REPR formulation.

`c`

Application:	REPR
Datatype:	`double`
Default value:	`1`
Constraints:	`a non-negative double`

This parameter is the non-negative L1 penalty parameter C for REPR.

`e`

Application:	REPR
Datatype:	`double`
Default value:	`1`
Constraints:	`a non-negative double`

This parameter is the non-negative L1 penalty parameter E for REPR.

`lpboost`

Application:	LPBR and REPR
Datatype:	`bool`
Default value:	`false`

If `true`, the LPBR solver is invoked; else, the REPR solver is invoked.

`exactRMA`

Application:	LPBR and REPR
Datatype:	`bool`
Default value:	`true`

If `true`, the subproblem, RMA, is solved exactly using PEBBEL; else, it is solved by our greedy heuristic.

`compareModels`

Application:	LPBR and REPR
Datatype:	`bool`
Default value:	`false`

If `true`, competing models are implemented to evaluate our boosting algorithms.

`evalEachIter`

Application: LPBR and REPR

Datatype: `bool`

Default value: `false`

If `true`, our current boosting model is evaluated in each column generation iteration.

`evalFinalIter`

Application: LPBR and REPR

Datatype: `bool`

Default value: `false`

If `true`, our boosting model is evaluated at the end of the column generation process.

`writeNodeTime`

Application: LPBR and REPR

Datatype: `bool`

Default value: `false`

If `true`, the number of bounded subproblems and CPU run time to solve each RMA problem in each column generation iteration are saved in a text file named "BBNode_CPUTime_*datafile_name*".

B.3 Methods for LPBR and REPR

Figure B.2 shows methods called for the column generation procedures of LPBR or REPR. Even though the details of each method differ between the `LPBR` and `REPR` classes, they follow similar procedures except that the `LPBR` class does not have to call the `solveInitMasterProblem` function.

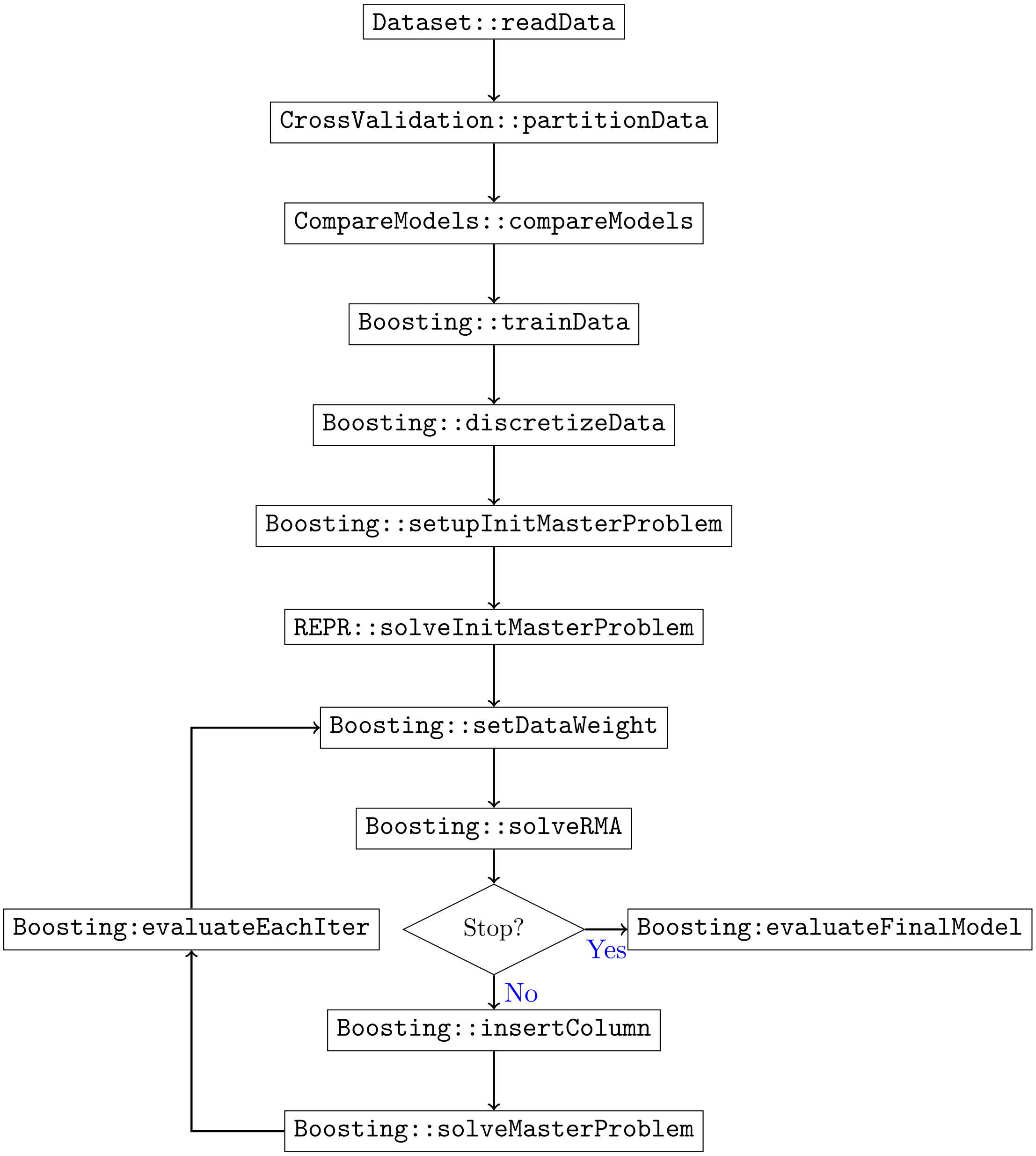

Dataset::readData
CrossValidation::partitionData
CompareModels::compareModels
Boosting::trainData
Boosting::discretizeData
Boosting::setupInitMasterProblem
REPR::solveInitMasterProblem
Boosting::setDataWeight
Boosting::solveRMA
Boosting:evaluateEachIter
Stop?
Yes
Boosting:evaluateFinalModel
No
Boosting::insertColumn
Boosting::solveMasterProblem

Figure B.2: Procedures for our boosting algorithms of LPBR and REPR

B.3.1 `Data` class

`void Data::readData()`

This method reads a data file and saves explanatory variables $X \in \mathbb{R}^{m \times n}$ and a response variable $y \in \mathbb{R}^n$ for REPR or $y \in \{0,1\}^n$ or $\{-1,1\}^n$ for LPBoost.

`void Data::readRandObs()`

If the `readShuffledObs` parameter is `true` and a text file, containing randomized observations indices, is specified in a command line, this method uses the pre-randomized observation-index list in the file for cross-validation or bootstrapping.

`void Data::integerizeDataRecursively()`

If the parameter `delta > 0`, this method is invoked to discretize a explanatory matrix recursively.

`void Data::integerizeDataByFixedBin()`

If the parameter `fixedBinSize > 0` and the parameter `delta = -1`, then explanatory variables are discretized into ℓ equal size bins in each attribute.

`void Data::setStandData()`

This method is invoked to standardize a dataset (X, y) for REPR not LPBR.

B.3.2 `CrossValidation` class

`void CrossValidation::setOuterPartition()`

This method randomly partitions observations into 5 sets.

`void CrossValidation::setInnerPartition()`

This methods randomly partitions observations in a given outer training set into 3 sets.

B.3.3 `CompareModels` class

`void CompareModels::runCompModels()`

If `compModels` parameter is `true`, then this method is invoked and several competing models of LPBR or REPR are implemented to compare with our boosting methods.

B.3.4 `Boosting` class

The `Boosting` class contains both non-virtual functions and pure virtual functions. The non-virtual functions are methods exactly same for both LPBR and REPR procedures, and the pure virtual functions are methods utilized by both `LPBR` and `REPR` classes but their details are different.

`void Boosting::setInitialMaster()`

This method sets up an initial restricted master problem.

`void Boosting::setDataWts()`

This method updates a weight of each training observation after solving the restricted master problem in each column generation iteration.

`void Boosting::solveRMA()`

This method solves the RMA problem after updating the training observation weights.

`void Boosting::setIntegerBounds()`

This method stores the box-based rule solutions discovered by solving the RMA problems.

`void Boosting::setOriginalBounds()`

This method translates the box-based rule solutions stored in the `setIntegerBounds` function back to the original variable scaling (before discretization).

`void Boosting::insertColumns()`

This method inserts one or more columns using one or more solutions given by solving the RMA problem.

`void Boosting::solveMaster()`

This method solves the restricted master problem after one or more new columns are inserted.

`void Boosting::evaluateEachIter()`

If the `evalEachIter` parameter is `true`, this method is invoked to evaluate our current classifier or predictor in each column generation iteration.

`void Boosting::evaluateFinalModel()`

If the `evalEachIter` parameter is `false` and the `evalFinalIter` parameter is `true`, this method is invoked to evaluate our final classifier or predictor.

B.3.5 `REPR` class

`void REPR::solveInitialMaster()`

This method solves the initial restricted master problem.

B.3.6 `GreedyRMA` class

```
void GreedyRMA::runGreedyRangeSearch()
```

This method implements the minimum and maximum Kadane's algorithms to obtain a greedy box-based rule.