**Noname manuscript No.**
(will be inserted by the editor)

# Technical Note: Algorithms for Optimal Dyadic Decision Trees

**Don Hush · Reid Porter**

Received: date / Accepted: date

**Abstract** A dynamic programming algorithm for constructing optimal dyadic decision trees was recently introduced, analyzed, and shown to be very effective for low dimensional data sets. This paper enhances and extends this algorithm by: introducing an adaptive grid search for the regularization parameter that guarantees optimal solutions for all relevant trees sizes, replacing the dynamic programming algorithm with a memoized recursive algorithm whose run time is substantially smaller for most regularization parameter values on the grid, and incorporating new data structures and data pre-processing steps that provide significant run time enhancement in practice.

**Keywords** Decision tree · Classification · Learning algorithm

## 1 Introduction

The most common algorithms for decision trees, e.g. CART [3] and C4.5 [15,16], use a greedy splitting algorithm to construct an initial tree followed by an "optimal" pruning algorithm to produce the final tree. However greedy approaches are generally not robust to the data distribution, and can therefore produce arbitrarily bad results ([5], Chapter 20). On the other hand, a simple structural risk minimization algorithm applied to cyclic dyadic decision trees (i.e. trees whose splits are determined by cycling through the coordinates and splitting at the interval mid-points) is guaranteed to be robust to distribution [21–23]. Recently it has been shown that allowing the dyadic splits to be performed in arbitrary order, and then designing the tree to minimize a regularized risk, also yields robust performance guarantees [23,2] and tends to give

The work was supported by the Laboratory Directed Research and Development (LDRD) office at Los Alamos National Laboratory.

D. Hush
MS B265, Group ISR-2, Los Alamos National Laboratory, Los Alamos, NM 87545
Tel.: 1-505-665-2722
Fax: 1-505-665-5220
E-mail: dhush@lanl.gov

R. Porter
MS D436, Group ISR-2, Los Alamos National Laboratory, Los Alamos, NM 87545

better results in practice [2]. The current best algorithm for designing these trees is the dynamic programming algorithm of Blanchard et al. [2] which was inspired by the "dyadic CART" algorithm of Donoho [6]. Blanchard et al. provide a thorough development of this algorithm and its properties, and demonstrate its utility through a series of empirical experiments. However the choices of class size and regularization parameter remain open issues for this method in practice. We develop an efficient algorithm that automatically chooses the class size to balance the trade-off between representational richness and computation. We show that the standard approach of searching a uniform grid for the regularization parameter can be flawed. Instead we describe a finite (unequally spaced) grid that can be computed ahead of time and is guaranteed to yield optimal solutions for all distinct (error, tree size) pairs that can be realized by minimizing the regularized risk. We also describe adaptive algorithms for efficiently searching this grid. Finally, to accelerate this grid search we replace the core dynamic programming algorithm with a memoized recursive algorithm that allows a type of lookahead pruning that yields significantly faster run times.

## 2 Definitions, Notation, and Background

Consider a rectangle $X \subset \Re^d$ defined by

$$X = [a_1, b_1] \times [a_2, b_2] \times ... \times [a_d, b_d]$$

where all $a_i$ and $b_i$ are finite and $a_i < b_i, \forall i$. A *dyadic split of $X$ along coordinate $i$* is a partition of $X$ into two *child* rectangles $L_i(X)$ and $R_i(X)$ which are formed by splitting at the midpoint of the $i^{th}$ coordinate interval,

$$L_i(X) := [a_1, b_1] \times ... \times [a_{i-1}, b_{i-1}] \times \left[a_i, \frac{a_i + b_i}{2}\right) \times [a_{i+1}, b_{i+1}] \times ... \times [a_d, b_d]$$

$$R_i(X) := [a_1, b_1] \times ... \times [a_{i-1}, b_{i-1}] \times \left[\frac{a_i + b_i}{2}, b_i\right] \times [a_{i+1}, b_{i+1}] \times ... \times [a_d, b_d] \; .$$

A *dyadic partition of $X$* is a partition $\pi(X) = \{X_1, X_2, X_3, ...\}$ whose members are terminal rectangles of a process that starts with $X$ and recursively applies dyadic splits to child rectangles[1]. This process can be represented by a *binary tree $T_X$* whose internal nodes correspond to the dyadic splits and whose leaves correspond to the terminal rectangles (as shown in Figure 1). Although a specific tree defines a unique partition, it is not necessarily the only tree that does so. Nevertheless, throughout this paper we use a binary tree $T_X$ to represent a partition, with the understanding that unless a specific tree is indicated, any valid tree will suffice. We can efficiently identify the terminal rectangle occupied by a point $x \in X$ by starting at the root node of the tree and, by comparing the component values of $x$ with the tree node split points, traversing the path to a leaf. By associating a value $y_j \in Y$ with each leaf rectangle $X_j \in \pi(X)$, and then assigning the value $y_j$ to all values of $x \in X_j$ we obtain a piecewise-constant function on $X$ that we denote by $T_X$. Throughout this paper we adopt an abuse of notation where we use $T_X$ to represent a tree, the partition formed by the tree, and the function derived from it, with the understanding that it will be clear from the

[1] We place no restrictions on the order in which coordinates are split or the relative number of splits for each coordinate. This is consistent with the definition in [2], but differs slightly from the definition in [21].

context which meaning is intended. When $Y$ is a finite set of labels associated with a pattern classification problem we call the function $T_X$ a *dyadic classification tree*. Let $k = (k_1, k_2, ..., k_d), k_i \geq 0$ and define $\mathcal{T}_k$ to be the class of dyadic classification trees with at most $k_i$ splits for dimension $i$. The members of $\mathcal{T}_k$ are formed by considering the partitions produced by all possible trees with at most $k_i$ splits for dimension $i$, and then considering all possible label assignments for each of these partitions. Now consider the following supervised classification problem.

**Definition 1** Let $P$ be an unknown probability distribution on $X \times Y$. Given a collection of data $D = ((x_1, y_1), ..., (x_n, y_n))$ sampled i.i.d. according to $P$, determine a value $k$ and a classifier $\hat{T}_X \in \mathcal{T}_k$ such that the classification error $e_P(\hat{T}_X) := P(\hat{T}_X(x) \neq y)$ is close to the optimal error $e_P^* := \inf_f e_P(f)$ (where $\inf_f$ is over all measurable functions).

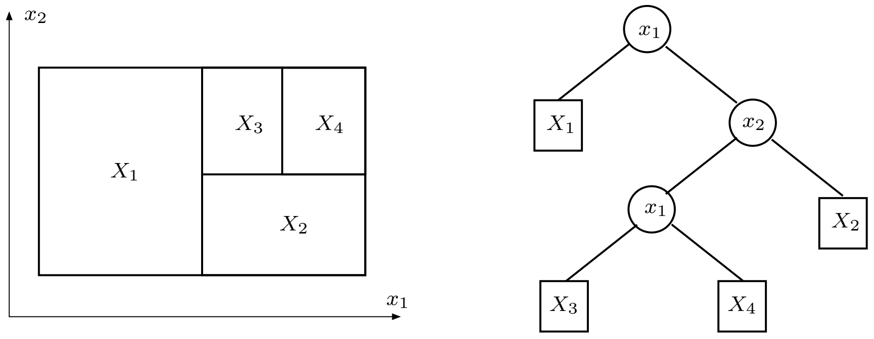

**Fig. 1** A dyadic partition (left) and a corresponding binary tree representation (right). Internal nodes of the tree are labeled by their split coordinate and external nodes by the partition rectangle they represent.

To solve this problem we follow the approach of Blanchard et al. [2] which chooses $\hat{T}_X$ to minimize a regularized empirical risk function. Our development requires that we decompose the problem into operations over dyadically constructed subsets of $X$. To this end we define $\sigma_k(X) = \{X_1, X_2, ...\}$ to be the set of all rectangles that appear in the dyadic partitions of $X$ represented by the trees in $\mathcal{T}_k$. For $\acute{X} \in \sigma_k(X)$ we define $\mathcal{T}_k(\acute{X})$ to be the class of dyadic classification trees that appear as a sub-tree of a tree in $\mathcal{T}_k$ with root rectangle $\acute{X}$. We adopt a slightly generalized definition of empirical classification error that allows us to specify a different loss value for each class assignment of each data sample. To this end we define a *weighted data set* to be a collection of pairs $\bar{D} = ((x_1, w_1), ..., (x_n, w_n))$ where $x_j \in X$, $w_j \in \Re_+^{|Y|}$ and $w_{jy}$ represents the loss incurred when label $y$ is assigned to $x_j$. We assume that the weight vectors are normalized so that

$$\sum_{j=1}^{n} w_{jy} = 1 - \hat{P}_y, \quad y \in Y$$

where $\hat{P}_y$ is the empirical class marginal probability for class $y$ (e.g. $\hat{P}_y$ is the fraction of samples that belong to class $y$). For example, if the classification problem in Definition 1 has two classes then the weight vector for a sample from class 1 would be $(0, 1/n)$.

In later sections we will replace samples with duplicate $x$ values by a single surrogate sample whose weight vector components take the form $\frac{k}{n}$ for some integer $k$.

Define the empirical error $e_{\bar{D}}$ of a tree $T_{\acute{X}} \in \mathcal{T}_k(\acute{X})$ to be the sum of losses over the samples from $\bar{D}$ that occupy the rectangle $\acute{X}$,

$$e_{\bar{D}}(T_{\acute{X}}) := \sum_{j:x_j \in \acute{X}} w_{jT_{\acute{X}}(x_j)}.$$

Define the corresponding regularized empirical risk

$$r_\lambda(T_{\acute{X}}) := e_{\bar{D}}(T_{\acute{X}}) \ + \ \lambda |T_{\acute{X}}|$$

where $\lambda \geq 0$ is the regularization parameter and $|T_{\acute{X}}|$ is the number of leaves in $T_{\acute{X}}$. When each sample $(x_j, y_j)$ in $D$ is converted to a sample $(x_j, w_j)$ in $\bar{D}$ using

$$w_{jy} = \begin{cases} 0, & y = y_j \\ 1/n, & y \neq y_j \end{cases}$$

then $e_{\bar{D}}$ represents the standard (unweighted) empirical classification error, and with $\acute{X} = X$ the corresponding risk $r_\lambda(T_X)$ represents the regularized empirical risk we seek to minimize.

Under some very general assumptions on $P$, Blanchard et al. show that with a suitable choice of $\lambda$, if $\hat{T}$ is a solution to

$$\min_{T_X \in \mathcal{T}_k(X)} r_\lambda(T_X) \tag{1}$$

then the excess error satisfies

$$E\left[e_P(\hat{T}) - e_P^*\right] \leq 2 \inf_{T \in \mathcal{T}_k} \left(e_P(T) - e_P^* + 2\lambda|T|\right) \ + \ \frac{c}{n}$$

where the expectation $E$ is over (i.i.d.) data collections $D$. This result tells us that on average, as $n$ goes to $\infty$, the excess error of a regularized empirical risk minimizer is bounded by the excess error of a distributional risk minimizer plus a corresponding regularization value (which we expect to be small). To implement this approach we must address the following issues: how to solve (1), how to choose $\lambda$, and how to choose $k$.

Blanchard et al. develop a dynamic programming algorithm for solving (1). With $k_i \propto \log n, \forall i$ they establish lower and upper bounds on the run time (and memory usage) that are dominated by the exponential term $(\log n)^d$. This prevents the algorithm from being practical for large data sets or dimensions larger than about 10. Nevertheless this algorithm has proven very effective, e.g. it typically produces smaller and more accurate decision trees than C4.5 [2]. We develop a simple recursive algorithm whose run time (and memory usage) obeys the same worst case upper bound, but possesses a much smaller (polynomial-time) lower bound. In practice the run time is often somewhere in-between and, unlike the previous dynamic programming algorithm, depends on the value of $\lambda$.

The most common approach for determining the regularization parameter $\lambda$ is to search a finite grid of values for one that minimizes a cross-validation (or hold-out) error. However solutions to (1) can be very sensitive to $\lambda$ and therefore easily overlooked with a uniform grid search. We describe a finite (unequally spaced) grid that can be computed ahead of time and is guaranteed to yield optimal solutions for all distinct (error, tree size) pairs that can be realized by minimizing the regularized risk. We also describe adaptive algorithms for efficiently searching this grid, and describe a computationally efficient way of integrating this search with our recursive algorithm for solving (1).

The choice of $k$ represents a trade-off between the richness of the function class $\mathcal{T}_k$ and the computational requirements for solving (1). We develop a computationally efficient algorithm that automatically chooses a value for $k$ that is reasonably small, but still includes trees in $\mathcal{T}_k$ that achieve the minimum possible empirical classification error.

The next four sections describe the recursive algorithm for solving (1), the adaptive grid search for $\lambda$, the algorithm for choosing $k$, and the specific data structures used to implement these algorithms. Appendix A provides a description of the data sets used in the experiments throughout this paper. All proofs for the lemmas in these sections can be found in Appendix B.

## 3 A Memoized Recursive (MR) Algorithm for Minimizing $r_\lambda$

When $\lambda$ is fixed and $k$ is finite the optimization problem in (1) can be solved, in principle, by exhaustive search. However, since $|\mathcal{T}_k(X)| = \Theta(2^{\sum_{i=1}^{d}(k_i+1)})$ (see below) this approach is only practical for small values of $d$ and $k$. To develop a more practical algorithm we must exploit the structural properties of this problem.

To begin we note that the empirical error $e$, number of leaves $|T|$, and risk $r$ are all values that can be decomposed into sums over sub-trees. For example, the risk of a tree is the sum of the risk of its left and right sub-trees, and by recursive application of this property the tree risk can also be expressed as the sum of the risks of its leaf nodes. Thus, for a fixed partition $T_X$, the minimum empirical error (and therefore the minimum risk) is obtained by assigning labels to the leaf nodes that correspond to the winner of a (weighted) majority vote over the data samples that occupy the leaf rectangle (where ties are resolved using a suitable tie-breaking strategy, e.g. choosing one of the tied labels at random). For convenience we also assign labels to internal nodes using the same strategy, i.e. using a (weighted) majority vote over the samples that occupy the node rectangle. Tree nodes that contain no data samples are assigned the label of the closest occupied ancestor node. All of the algorithms in this paper adopt this *majority vote rule* to determine the labels for tree nodes (both internal and external).

With $\lambda$ fixed we define an optimal tree over $\acute{X} \in \sigma_k(X)$

$$T^*_{\lambda,\acute{X}} \in \arg \min_{T_{\acute{X}} \in \mathcal{T}_k(\acute{X})} r_\lambda(T_{\acute{X}}) \tag{2}$$

and its corresponding optimal error and risk values

$$e^*_{\lambda,\acute{X}} := e_{\bar{D}}(T^*_{\lambda,\acute{X}}), \tag{3}$$

$$r^*_{\lambda,\acute{X}} := r_\lambda(T^*_{\lambda,\acute{X}}). \tag{4}$$

It is easy to prove that an optimal tree $T^*_{\lambda,\acute{X}}$ is composed of optimal sub-trees [23]. Combining this result, the additive property mentioned above, and the fact that there are only $d$ possible splits at any given node allows us to conclude that an optimal tree over $\acute{X}$ is either the single node tree $T^0_{\acute{X}}$ corresponding to the rectangle $\acute{X}$ with no splits, or one of the $d$ trees formed by making a dyadic split along one of the $d$ coordinates and attaching the corresponding optimal left and right sub-trees. This leads to the following recursive expression for the optimal risk

$$\begin{aligned} r^*_{\lambda,X} = \min\Big( & r_\lambda(T^0_X), \quad r^*_{\lambda,L_1(X)} + r^*_{\lambda,R_1(X)}, \\ & r^*_{\lambda,L_2(X)} + r^*_{\lambda,R_2(X)}, \quad \cdots \quad , \quad r^*_{\lambda,L_d(X)} + r^*_{\lambda,R_d(X)} \Big) \end{aligned} \tag{5}$$

and suggests the simple recursive algorithm in Procedure 1 (which is invoked with $\acute{X} = X$).

**Procedure 1** A simple recursive algorithm for designing an optimal dyadic decision tree.

```
1: ODDTDesign ($\acute{X}, \bar{D}$)
2:
3: {check for empty rectangle}
4: if ($\acute{X} \cap \bar{D} = \emptyset$) then
5:     $T_{\acute{X}*} \leftarrow$ empty leaf node
6:     Return($T_{\acute{X}*}$)
7: end if
8:
9: {initialize as a leaf node}
10: $T^*_{\acute{X}} \leftarrow T^0_{\acute{X}}$
11: $r^*_\lambda \leftarrow r_\lambda(T^0_{\acute{X}})$
12:
13: {search over coordinates for the best split}
14: for ($i = 1, 2, ..., d$) do
15:     if ((number of splits for coordinate $i$) $< k_i$) then
16:         split $\acute{X}$ into $L_i(\acute{X})$ and $R_i(\acute{X})$
17:         {determine optimal sub-trees}
18:         $T^*_{L_i(\acute{X})} \leftarrow$ ODDTDesign($L_i(\acute{X})$)
19:         $T^*_{R_i(\acute{X})} \leftarrow$ ODDTDesign($R_i(\acute{X})$)
20:         {save best split}
21:         if $\left(r_\lambda(T^*_{L_i(\acute{X})}) + r_\lambda(T^*_{R_i(\acute{X})}) < r^*_\lambda\right)$ then
22:             $r^*_\lambda \leftarrow r_\lambda(T^*_{L_i(\acute{X})}) + r_\lambda(T^*_{R_i(\acute{X})})$
23:             $T^*_{\acute{X}} \leftarrow$ tree formed by attaching $T^*_{L_i(\acute{X})}$ and $T^*_{R_i(\acute{X})}$ to $T^0_{\acute{X}}$
24:         end if
25:     end if
26: end for
27:
28: Return($T^*_{\acute{X}}$)
```

Procedure 1 considers all possible combinations of splits and therefore visits every occupied rectangle of every size in $\sigma_k(X)$. However it also considers all possible *sequences* of combinations, and consequently visits many rectangles multiple times, recomputing the corresponding optimal sub-trees each time. For example Procedure 1 would visit rectangle $X_4$ in Figure 1 via three different split sequences: $R_1L_2R_1$ (as suggested in the figure), $R_1R_1L_2$ and $L_2R_1R_1$. This happens because our recursive problem decomposition contains "overlapping sub-problems" which lead to redundant computation in the corresponding recursive algorithm. Blanchard et al. show how to eliminate this redundancy by using a *dynamic programming* (DP) approach which solves the problem from the bottom up rather than the top down. More specifically, the DP algorithm starts at the lowest possible tree level (corresponding to the smallest possible rectangles), and works its way up one level at a time until it reaches the root, computing optimal sub-trees at each level and storing them for use at the next highest level. The DP algorithm eliminates computational redundancy by storing intermediate results and retrieving them at a later point in the algorithm. This same thing can be accomplished using a so-called *memoized* recursive (MR) algorithm ([4], Section 16.2) which employs the top down recursion in Procedure 1, stores optimal sub-trees when they are first encountered, and then retrieves them at a later point (thereby avoiding the recursive call that would compute them again). This requires the following simple modifications to Procedure 1: add a command at line 27 that stores the optimal sub-tree that has just been computed, and add commands before the recursive calls on lines 18 and 19 that check to see if the optimal sub-trees have already been computed and can be retrieved from storage. In addition, rather than store an entire sub-tree each time it is sufficient to store the corresponding root rectangle along with pointers to its left and right child rectangles (which are root rectangles for other stored entries).

Both the DP and MR algorithms consider all possible combinations of splits, and therefore visit the exact same number of rectangles, and store (and retrieve) the exact same optimal sub-trees. Furthermore, the operations performed within each recursive call are very similar to the operations performed each time the DP algorithm visits a rectangle. Thus, the computational requirements of these two algorithms are essentially the same (both memory and run time). The primary advantage of the memoized recursive algorithm is that, because it works from the top down, it allows a *lookahead pruning* scheme (described next) that can substantially reduce the computational requirements.

The idea behind lookahead pruning is to avoid computing optimal sub-trees that cannot improve the current solution. One of the simplest ways to do this is to avoid the search over individual coordinate splits in lines 14-26 of Procedure 1 when the initial "leaf node" risk $r_\lambda(T^0_{\acute{X}})$ cannot be improved. The next lemma provides a simple test that identifies this case.

**Lemma 1** ***(sub-tree improvement bound)*** *Let $\acute{X} \in \sigma_k(X)$ and define $T^0_{\acute{X}}$ to be the single node classifier that assigns the same label to all points in $\acute{X}$ using a majority vote rule over samples from $\bar{D}$. Then for any $\lambda > e_{\bar{D}}(T^0_{\acute{X}})$ the solution to $\arg\min_{T_{\acute{X}} \in \mathcal{T}_k(\acute{X})} r_\lambda(T_{\acute{X}})$ is $T^0_{\acute{X}}$.*

Given this lemma, the lookahead pruning strategy can be implemented by inserting a test for $e_{\bar{D}}(T^0_{\acute{X}}) < \lambda$ at line 12 in Procedure 1 and skipping the search over coordinates if the test is true. The computational savings obtained using this simple strategy can be quantified by the *improvement ratio*, which is the ratio of the number of rectangles visited by the DP algorithm (or equivalently the MR algorithm without pruning) to the number visited by the MR-with-pruning (MRP) algorithm. Figure 2 plots the number of rectangles versus $\lambda$ for the thyroid data set[2]. For larger $\lambda$ the improvement ratio is dramatic (i.e. several orders of magnitude), while at smaller $\lambda$ the improvement ratio shrinks to 1. At the rule of thumb $\lambda = 2/n$ the improvement ratio is approximately 4.8. The average improvement ratio across all $\lambda$ values is 3.3. Here we have used a complete $\lambda$

[2] The data sets used throughout this paper are described in detail in Appendix A.

grid that is guaranteed to produce all "distinct solutions" realizable with this approach (see Section 4), which includes numerous solutions produced by smaller $\lambda$ values that overfit the data. Grid searches that are able to avoid these smaller $\lambda$ values, e.g. the so-called early stopping methods, will have a much higher average improvement ratio. Table 1 shows average improvement ratios for several data sets. The fourth column

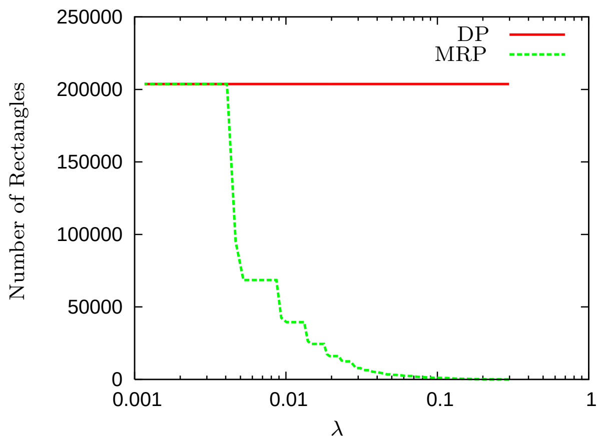

**Fig. 2** The results in this plot are obtained using the thyroid data set with all $n = 215$ samples. This plot shows the number of rectangles visited by the DP and MRP algorithms as $\lambda$ varies over a grid of values determined by the algorithm in Section 4. At $\lambda = 2/n = .0093$ pruning reduces the number of rectangles by approximately 4.8.

shows the improvement ratio averaged across $\lambda$ grid values (where a "complete" $\lambda$ grid has been used in all cases), and the sixth column shows improvement ratio statistics computed across resampled training sets with $\lambda = n/2$. The results in this table show remarkable improvement for such a simple pruning strategy.

**Table 1** For each data set, $d$ is the dimension, $n_{design}$ is the total number of data samples available for design, and $n$ the number of samples in the resampled data sets. The fourth column shows the average improvement ratio across all $\lambda$ grid values when the MRP algorithm is applied to the $n_{design}$ samples. The sixth column shows the minimum, mean, and maximum improvement ratios across the resampled data sets with $\lambda = n/2$ (the rule of thumb).

| Data Set | $d$ | $n_{design}$ | Improvement Ratio for $\lambda$ grid search | $n$ | Improvement Ratio for $\lambda = 2/n$ (min mean max) |
|---|---|---|---|---|---|
| banana | 2 | 5300 | 3.5 | 400 | (1.6 2.8 4.4) |
| breast cancer | 9 | 277 | 1.7 | 200 | (2.4 3.6 6.4) |
| diabetes | 8 | 768 | 1.6 | 468 | (2.3 2.5 2.8) |
| flare-solar | 9 | 1066 | 2.4 | 666 | (1.7 2.0 2.7) |
| thyroid | 5 | 215 | 3.3 | 140 | (1.9 3.9 7.7) |
| titanic | 3 | 2201 | 1.2 | 150 | (1.0 1.3 2.9) |
| Spambase | 4 | 3601 | 2.2 | 3301 | (3.9 4.1 4.3) |
| Intrusion | 4 | 49403 | 7.5 | 44463 | (3.3 5.1 7.5) |
| Mushroom | 10 | 6124 | 4.2 | 4624 | (1.2 1.2 1.2) |

## 4 Algorithms for $\lambda$

Blanchard et al. [2] suggest that the regularization parameter take the form $\lambda = \kappa/n$, and they show good empirical results for $1 \le \kappa \le 4$. They also suggest a rule of thumb that chooses $\kappa = 2$ for a 2-class classification problem [2]. A more robust approach, and a common method for determining tuning parameters in practice, is to choose $\lambda$ (or equivalently $\kappa$) to minimize a cross-validation (or hold-out) error. This usually involves searching a finite grid of parameter values which is determined heuristically. In this section we develop a principled approach to this search. In particular we develop tight upper and lower bounds on the interval of search, describe a finite (unequally spaced) grid of values that can be computed ahead of time and is guaranteed to yield optimal solutions for all solution values $(e, |T_X|)$ that are realizable by minimizing $r_\lambda$, describe adaptive algorithms for efficiently searching this grid, and describe a computationally efficient way to integrate this search with the MRP algorithm in Section 3.

We note that the approach taken here is directly analogous to the optimal *pruning* procedure in CART [3]. Both cases seek a suitable value of $\lambda$ for the criterion $r_\lambda$, but the class of trees considered by the CART pruning algorithm obeys a very convenient nesting property, while the (typically) much larger class $\mathcal{T}_k(X)$ considered here does not. Consequently the development here is a bit more general and the algorithms are a bit more technical.

We start by formalizing our intuition that minimizing $r_\lambda$ over a sequence of decreasing $\lambda$ values will generate a sequence of decision trees with decreasing error and increasing size. To this end we state the following monotonicity Lemma.

**Lemma 2** *(monotonicity in $\lambda$) Using the definitions in (2-4) the following inequalities hold for any* $0 \le \lambda_1 < \lambda_2$

$$\begin{aligned} & r^*_{\lambda_1,X} \le r^*_{\lambda_2,X} \\ & e^*_{\lambda_1,X} \le e^*_{\lambda_2,X} \\ & |T^*_{\lambda_1,X}| \ge |T^*_{\lambda_2,X}| \end{aligned}$$

Having established monotonicity for different $\lambda$ values we now give a rule for choosing a unique-valued solution for a fixed $\lambda$ value.

**Definition 2** ***(selection criterion)*** Let $\mathcal{T}^*_{k,\lambda}(X) = \arg\min_{T_X \in \mathcal{T}_k(X)} r_\lambda(T_X)$ be the solution set for regularization value $\lambda$. Select the solution $T^*_{\lambda,X}$ to be any member of $\mathcal{T}^*_{k,\lambda}(X)$ with the largest number of leaves[3].

This definition allows us to define a unique sequence of solution values as $\lambda$ is varied continuously from 1 to 0 (later in this section we will show that the range $0 \le \lambda \le 1$ is sufficient to generate all possible solutions).

**Definition 3** ***(solution-value-sequence)*** Define the two-tuple $(e^*_{\lambda,X}, |T^*_{\lambda,X}|)$ to be the *solution value* for a solution $T^*_{\lambda,X}$ given by Definition 2. Define the *solution-value-sequence* $(e_1, l_1), (e_2, l_2), \ldots$ to be the sequence of distinct solution values generated as $\lambda$ is varied continuously from 1 to 0.

[3] Alternatively we could select the member with the smallest number of leaves, but this would introduce a slight change in Lemma 3 below: the strict and non-strict inequalities in (6) would change places.

We now develop a characterization of this sequence and its members. First recall that the sample weight vector components are either 0, $\frac{1}{n}$, or $\frac{k}{n}$, where $\frac{k}{n}$ corresponds to cases where duplicate samples are replaced by a surrogate. Because the number of distinct error values is no more than $n$ this sequence will contain no more than $n$ values. In practice the number of values is often much smaller than $n$ as illustrated in Table 2. Next, although the number of occupied leaves in any given tree can be no larger than $n$, the total number of leaves (i.e. the sum of occupied and unoccupied leaves) can exceed $n$. Furthermore it is difficult to obtain a non-trivial bound on the number of unoccupied leaves a priori, since several dyadic splits may be required before a split is located in a region where the data are concentrated. This can happen, for example, when the data samples are concentrated near corners of the original rectangle. A crude bound on the total number of leaves is $2^{\sum_i k_i}$, which corresponds to the number of leaves in the largest trees in $\mathcal{T}_k$. However once we have the data it is easy to compute a tighter bound that we call $l_{bound}$. Our approach is to set $l_{bound}$ equal to the number of leaves in the minimum error tree built by a simple greedy algorithm. Because the greedy algorithm is sub-optimal $l_{bound}$ is an obvious upper bound on the number of leaves in the minimum error solutions obtained by solving $[\min r_\lambda|_{\lambda=0}]$, and therefore, by the monotonicity result in Lemma 2, $l_{bound}$ is an upper bound on the number of leaves in solutions to $\left[\min r_\lambda|_{\lambda\geq 0}\right]$ (i.e. solutions for all $\lambda$). In this paper we compute $l_{bound}$ using a simple greedy algorithm that builds a tree by recursively performing the dyadic splits that maximize the single split error reduction until the minimum error is achieved. The experimental results in Table 3 suggest that this value of $l_{bound}$ is often much smaller than $n$, and typically no more than 5 times the number of leaves $l_{max}$ in the largest solutions to $[\min r_\lambda]$.

**Table 2** For each data set, $n$ is the total number of samples in the data set. This table shows the number of distinct solutions to (1) as $\lambda$ is varied from 1 to 0.

| Data Set | $n$ | number of distinct solutions |
|---|---|---|
| banana | 5300 | 71 |
| breast cancer | 277 | 16 |
| diabetes | 768 | 21 |
| flare-solar | 1066 | 9 |
| thyroid | 215 | 11 |
| titanic | 2201 | 4 |
| Spambase | 3601 | 41 |
| Intrusion | 49403 | 32 |
| Mushroom | 6124 | 8 |

This next lemma provides a complete characterization of $\lambda$ in terms of the solution value sequence and is the key result in our development of a grid search algorithm for $\lambda$.

**Lemma 3** *Given a solution-value-sequence* $(e_1, l_1), (e_2, l_2), ..., (e_m, l_m)$*, define* $(e_0, l_0) := (2e_1, 0)$ *and* $(e_{m+1}, l_{m+1}) := (0, \infty)$*. Then, for* $i \in \{1, 2, ..., m\}$*, a solution with solution value* $(e_i, l_i)$ *can be obtained by solving* $\min_{T_X \in \mathcal{T}_k(X)} r_{\lambda_i}(T_X)$ *with any value of* $\lambda_i$ *satisfying*

$$\frac{e_i - e_{i+1}}{l_{i+1} - l_i} < \lambda_i \leq \frac{e_{i-1} - e_i}{l_i - l_{i-1}}. \tag{6}$$

**Corollary 1** *All values in the solution-value-sequence can be realized by minimizing $r_\lambda$ with $\lambda$ values in the interval $0 < \lambda \leq e_1 < 1$.*

This lemma reveals several important properties. First, $\frac{e_{i-1}-e_i}{l_i-l_{i-1}}$ represents a *progress-to-cost* ratio, where progress is measured in terms of error reduction and cost in terms of the additional number of leaves required to achieve this reduction. The lemma above tells us that $\lambda$ is a lower bound on this ratio. Basic guidelines from learning theory suggest that small values of this ratio may lead to overfitting. More specifically, the general notions that "learning is compression" and "memorizing the data can lead to poor generalization" suggest that this ratio should be no smaller than approximately $1/n$ for good generalization. This is consistent with the empirical results in [2] that show the best generalization for $\lambda \geq 1/n$. This lemma also tells us that this progress-to-cost ratio is monotonic in the solution sequence. This implies for example that if the data is distributed so that only a large tree can give a large error reduction, e.g. a checkerboard data distribution, then this large tree will be the first non-trivial solution in the sequence. Finally this lemma shows how the $\lambda$-value intervals can vary dramatically in size. Often we expect larger error reductions from the initial tree size increases, suggesting larger interval sizes for larger $\lambda$ values. Figure 3 plots the number of leaves versus $\lambda$ for solutions to (1) with the diabetes data set (note that the $\lambda$ axis is plotted on a log scale). This plot confirms that the $\lambda$-interval sizes vary dramatically, and that the largest coincide with large $\lambda$ values. These observations suggest that a $\lambda$-search that uses a uniform grid (even with log-$\lambda$ scaling) is likely to either be inefficient (large number of grid points), overlook specific solution values (small number of grid points), or both. Next we develop a $\lambda$-search that uses an adaptive grid derived from the result in (6).

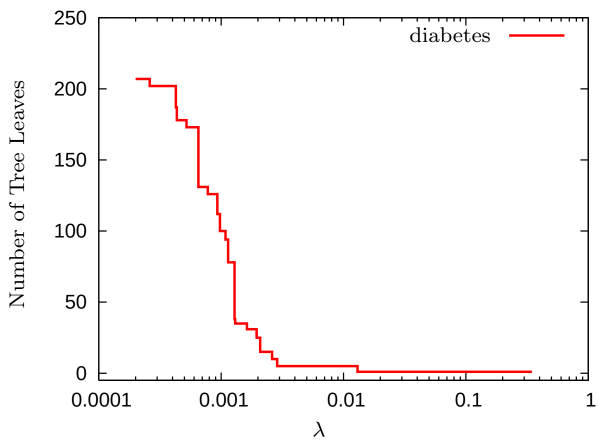

**Fig. 3** The results in this plot are obtained using the diabetes data set with all $n = 768$ samples. We plot the number of leaves versus $\lambda$ for solutions to (1). Note that the $\lambda$ axis is plotted on a log scale and that $2/n = .0026$ for this data set.

If we knew the solution-value-sequence ahead of time then Lemma 3 would give us a way to compute a complete set of optimal trees by solving (1) with one $\lambda$ value selected from each interval in (6). For example it would be sufficient to choose

$$\lambda_i = \frac{e_{i-1} - e_i}{l_i - l_{i-1}}, \quad i = 1, 2, ..., m.$$

However we don't know the solution values ahead of time. Nevertheless, they can take only a finite number of values. Indeed, $\Delta_e = (e_{i-1} - e_i)$ can take at no more that $n$ values (often much less) and $\Delta_l = (l_i - l_{i-1})$ can take no more than $l_{bound}$ values. Thus, one approach is to compute the ratio $\Delta_e/\Delta_l$ for all $O(nl_{bound})$ combinations of $\Delta_e$ and $\Delta_l$, remove any duplicates, and then step through the values in this $\lambda$-*list* from largest to smallest, computing solutions to (1) as we go. This is guaranteed to find solutions for all distinct solution values. However it will also recompute the same solutions numerous times (because it will visit numerous $\lambda$ values of from each interval in (6)). Some of this redundancy can be avoided by recomputing a smaller $\lambda$-list each time a new solution value is encountered, since every new solution reduces the number of possibilities for the remaining values of $\Delta_e$ and $\Delta_l$. The experimental results in columns 5 and 6 of Table 3 show that this adaptive approach yields a substantial reduction (often several orders of magnitude) in the total number of $\lambda$ values that are visited. Furthermore, the larger $\lambda$-lists (with smaller steps) occur in the beginning where (1) can be solved with less computation (because of the more aggressive lookahead pruning for larger $\lambda$ values). In this sense the $\lambda$ search and lookahead pruning algorithms are complementary. Next we develop a simple heuristic that provides an additional reduction in the number of $\lambda$ values in the search.

**Definition 4** ***(error reduction and split ratio)*** For a rectangle $\acute{X} \in \sigma_k(X)$ and a tree $T_{\acute{X}} \in \mathcal{T}_k(\acute{X})$ define the *error reduction* to be

$$\delta(T_{\acute{X}}) := e_{\bar{D}}(T^0_{\acute{X}}) - e_{\bar{D}}(T_{\acute{X}})$$

and the *split ratio* to be

$$g(T_{\acute{X}}) := \frac{\delta(T_{\acute{X}})}{|T_{\acute{X}}|}.$$

**Lemma 4** ***(split ratio bound)*** *Let $T^*_X(i-1)$ and $T^*_X(i)$ be optimal dyadic decision trees corresponding to solution values $(e_{i-1}, l_{i-1})$ and $(e_i, l_i)$ respectively. Let $\bar{T}_X(i)$ be a minimum error sub-tree of $T^*_X(i)$ of size $|\bar{T}_X(i)| = |T^*_X(i-1)|$, and let $T^*_{X_1}(i), T^*_{X_2}(i), ..., T^*_{X_m}(i)$ be the sub-trees that are pruned from $T^*_X(i)$ to obtain $\bar{T}_X(i)$. Then*

$$\frac{e_{i-1} - e_i}{l_i - l_{i-1}} \leq \frac{\sum_{j=1}^m \delta(T^*_{X_j}(i))}{\sum_j |T^*_{X_j}(i)|} \leq \max_{j \in \{1,...,m\}} g(T_{X^*_j}(i)).$$

Although the MRP instance that determines the current solution $T^*_X(i-1)$ has no obvious way to determine the sub-trees $T^*_{X_j}(i)$ of the next solution, it may visit them during its search. Indeed, it visits numerous sub-trees that are sub-optimal for the current value of $\lambda$, but are candidates for inclusion in optimal trees for smaller values of $\lambda$. This motivates the following heuristic for adaptively choosing the next $\lambda$ value during the $\lambda$ grid search. Compute the split ratio of every non-optimal sub-tree visited during the current MRP instance and choose the next $\lambda$ value to be the largest split ratio that is less than the current $\lambda$ value. If there is no split value satisfying this criterion then choose the next value in the adaptive $\lambda$-list as before. It is easy to add this heuristic to the MRP algorithm since the visited non-optimal sub-trees are simply the sub-trees not chosen in the local search over the $d+1$ candidates in each recursive call. The experimental results in columns 6 and 7 of Table 3 show that this *adaptive+heuristic* method yields a substantial reduction (often several orders of magnitude over the adaptive method alone) in the total number of $\lambda$ values visited.

This heuristic is somewhat conservative in that it is very likely to choose a next $\lambda$ value that is greater than or equal to the desired value as long as the current instance of MRP explores a sufficiently rich collection of non-optimal sub-trees. Thus, it will only skip over distinct-solution $\lambda$ intervals if the current instance of MRP fails to visit an appropriate collection of non-optimal sub-trees. The parenthetic results in the last two columns of Table 3 suggest that this is a rare case. It only happens with three of the nine data sets, and in these cases skips only 1, 2, and 4 intervals.

**Table 3** For each data set, $d$ is the dimension, $n$ is the total number of samples in the data set, $l_{bound}$ is the number of leaves in a minimum error tree built by a simple greedy algorithm, and $l_{max}$ is the number of leaves in the largest solution produced by the MRP algorithm. The "original $\lambda$-list" column shows the number of distinct $\lambda$ values determined by computing the ratio $\Delta_e/\Delta_l$ for all $O(nl_{bound})$ combinations of $\Delta_e$ and $\Delta_l$. The "adaptive search" column shows the number of $\lambda$ values visited by the adaptive method that recomputes a smaller $\lambda$-list each time a new solution value is encountered. The last column shows the number of $\lambda$ values visited by the adaptive+heuristic algorithm described in the text. The parenthetic values in the last two columns represent the total number of distinct solutions found with these searches.

| | | | | | Number of $\lambda$ values | | |
|---|---|---|---|---|---|---|---|
| Data Set | $d$ | $n$ | $l_{bound}$ | $l_{max}$ | original $\lambda$-list | adaptive search | adaptive+ heuristic search |
| banana | 2 | 5300 | 1493 | 1078 | 2,152,240 | 141,130 (71) | 309 (71) |
| breast cancer | 9 | 277 | 210 | 75 | 9391 | 1027 (16) | 88 (14) |
| diabetes | 8 | 768 | 472 | 207 | 75,740 | 11,656 (21) | 354 (20) |
| flare-solar | 9 | 1066 | 150 | 35 | 2569 | 106 (9) | 56 (9) |
| thyroid | 5 | 215 | 59 | 26 | 2319 | 167 (11) | 74 (11) |
| titanic | 3 | 2201 | 9 | 6 | 28 | 9 (4) | 6 (4) |
| Spambase | 4 | 3601 | 2701 | 581 | 543,242 | 38,479 (41) | 758 (37) |
| Intrusion | 4 | 49403 | 645 | 159 | 360,717 | 3425 (32) | 653 (32) |
| Mushroom | 10 | 6124 | 33 | 9 | 12804 | 197 (8) | 95 (8) |

## 5 Algorithms for $k$ and Data Compression

It may appear that the top down nature of the MRP algorithm eliminates the need to specify $k$ (the maximum number of splits per dimension). However it is easy to verify that if $D$ contains duplicate $x$ samples with different $y$ values then without $k$ the MRP algorithm could end up exploring an infinitely long tree branch. This situation is easily resolved by terminating the recursive calls when a node's $x$ values are all equal. However a similar behavior (i.e. exploring a very long tree branch) can occur when $D$ contains a group of $x$ values that are sufficiently close together. If we allow this type of behavior then we can eliminate the need for $k$ altogether. However by retaining $k$ we can prevent this type of behavior, and at the same time create an extremely efficient supporting data structure for MRP (see Section 6), and incorporate a data compression pre-processing step that can significantly improve the run time.

Our choice of $k$ represents a heuristic trade-off between the richness of the function class $\mathcal{T}_k$ and the computational requirements for solving (1). In this section we develop a computationally efficient algorithm that automatically chooses a value for $k$ that is reasonably small, but still includes trees in $\mathcal{T}_k$ that achieve the minimum possible empirical classification error.

We start by replacing duplicate samples, i.e. groups of samples that have the same $x$ value, by a surrogate sample that has the same $x$ value and a surrogate weight vector. The surrogate weight vector is formed by summing of the duplicate sample weight vectors and then translating, by subtracting the minimum component value from each component, so that the resulting vector has a minimum component value of zero. This gives a minimum loss of zero for every data sample, which shifts the risk value but does not change the minimum risk solution set. The minimum achievable empirical classification error is zero after this step. An efficient implementation of this step can be obtained by filtering the data samples one coordinate at a time. Starting with the first coordinate we sort, scan, group duplicate values, and discard non-duplicate values. Then, using only the samples with duplicate first coordinate values, we repeat the process with the second coordinate values, and so on with the other coordinate values until we have filtered the samples along all $d$ coordinates. The samples in the final collection of "duplicate groups" are replaced by surrogates as described above. This step performs a total of $d$ sorts and scans, each with no more than $n$ values, and so has run time $O(dn \log n)$.

Once we've replaced duplicates a conservative way to guarantee that $\mathcal{T}_k$ contains trees that achieve zero empirical classification error is to choose each $k_i$ separately to be the minimum number of dyadic splits that partition the $i^{th}$ coordinate into zero-loss intervals (i.e. intervals where labels can be assigned so that the loss is zero for all samples in each interval) [2]. If the distribution of $X$ has a bounded density with respect to the Lebesgue measure then this should yield $k_i$ on the order of $\log n$ [2], but smaller values of $k$ can be obtained by considering splits along *all $d$ coordinates simultaneously.* We describe a simple two step approach. The first step finds the smallest value of $k_0$ such that the partition produced by splitting all $d$ coordinates into $2^{k_0}$ equal size intervals contains only zero-loss rectangles. For each value of $k_0 = 1, 2, ...$ this zero-loss test requires $O(ndk_0)$ computation (i.e. $dk_0$ computations to determine the leaf rectangle occupied by each of the $n$ samples, and then no more than $n$ additional computations to determine of the loss values for each occupied leaf), and so if $k_0^*$ is the final value the total run time is $O(ndk_0^{*2})$. Thus, when $k_0^* = O(\log n)$ the run time is $O(nd(\log n)^2)$. The second step then selects $k_i$ for each coordinate $i$ separately by keeping the smallest value from $\{0, 1, ..., k_0^*\}$ that realizes the same partition of the data samples as the partition formed by $k_i = k_0^*$. This second step obeys the same run time bound as the first.

Table 4 shows the results of designing $k$ with this method where column 3 shows the range of $k$ values obtained over the resampled data sets. The values of $k$ appear to be reasonably stable across data sets, and in many cases are surprisingly small. Column 3 also compares these values with the range of component values reported in [2] (shown in the square brackets). The values obtained with our automated method were generally the same or smaller.

Once $k$ has been determined the data can be compressed as follows. Consider the zero-loss rectangles produced by splitting each coordinate $i$ into $2^{k_i}$ equal size intervals.

All data samples that occupy the same zero-loss rectangle are replaced by a surrogate sample whose $x$ value is the mean of the individual $x$ values, and whose weight vector is the sum of the individual weight vectors. This yields a compressed data set that can be used as input to the MRP algorithm without changing the minimum risk solution set.

The last column in Table 4 shows statistics for the compressed data set sizes computed over the resampled data sets. These sizes represent a combination of the compression obtained by replacing duplicates and replacing samples that occupy the same zero-loss rectangles. In some data sets there is little effect, but with others there is a dramatic reduction in the number of samples (e.g. an order of magnitude), which in turn leads to significant computational savings in the MRP algorithm. Perhaps the most intriguing results are obtained with the titanic data set which is compressed to only 9 samples an average.

**Table 4** For each data set, $d$ is the dimension and $n$ the number of samples in the resampled training sets. This table summarizes results computed across the resampled data sets. Column 3 shows the results of the automatic method for determining $k$. The square brackets show the range of component values reported in [2]. *For the diabetes data set the automatic method gave $k = (4, 4, 4, 4, 4, 4, 4, 4)$, but we were forced to reduce to $k = (3, 3, 3, 3, 3, 3, 3, 3)$ because of the excessive memory requirements. Column 4 shows the results of the two step compression process that starts by replacing duplicates and then, after $k$ has been determined, replacing samples that occupy the same zero-loss rectangles.

| | | | | | $\acute{n}$ | | |
|---|---|---|---|---|---|---|---|
| Data Set | $d$ | $n$ | $k$ | | (min | mean | max) |
| banana | 2 | 400 | (6-10, 6-10) | [14] | 372 | 395 | 400 |
| breast cancer | 9 | 200 | (3, 2, 3-4, 3-4, 1, 2, 1, 3, 1) | [1-4] | 178 | 186 | 194 |
| diabetes | 8 | 468 | (3, 3, 3, 3, 3, 3, 3, 3)* | [3] | 445 | 454 | 463 |
| flare-solar | 9 | 666 | (1, 2, 2, 1, 1, 1, 3-4, 2-3, 1-2) | [1-3] | 72 | 81 | 91 |
| thyroid | 5 | 140 | (3-5, 3-5, 3-5, 3-5, 3-5) | [5-6] | 62 | 123 | 140 |
| titanic | 3 | 150 | (1-2, 1, 1) | [1-2] | 5 | 9 | 14 |
| Spambase | 4 | 3301 | (10, 11, 15, 13) | | 1651 | 1664 | 1674 |
| Intrusion | 4 | 44463 | (10-11, 10-11, 10-11, 10-11) | | 7640 | 7719 | 7764 |
| Mushroom | 10 | 4624 | (2, 2, 2, 2, 2, 2, 2, 2, 2, 2) | | 728 | 735 | 742 |

## 6 Data Structures

Both the dynamic programming and memoized recursive algorithms require a database $S$ for insertion and retrieval of optimal sub-trees. Rather than store an entire sub-tree in each database entry it is sufficient to store the corresponding root rectangle along with pointers to its left and right child rectangles (which are root rectangles for other database entries). Blanchard et al. use a *dynamic dictionary* database which performs insertions and retrievals in $O(\log |S|)$ time [2]. However these rectangles are more naturally stored in a *hash table* database. This is largely because they possess a simple integer representation that can be used either to index the hash table directly, or to develop a natural class of hash functions that are *universal* [4] and therefore guarantee $O(1)$ expected run time for insertions and retrievals.

This integer representation can be developed as follows. The complete set of dyadic intervals for coordinate $i$ can be represented by the nodes of a complete binary tree with $2^{k_i}$ leaves. Therefore the exact number of such intervals is $2^{k_i+1}-1$ and so they can be uniquely represented by a $(k_i+1)$-bit integer. This gives exactly $\prod_{i=1}^{d}(2^{k_i+1}-1)$ total rectangles in $\sigma_k(X)$ which can be uniquely represented by a $\left(\sum_{i=1}^{d}(k_i+1)\right)$-bit integer. If $\sum_{i=1}^{d}(k_i+1)$ is sufficiently small then this integer can be used to index hash table entries directly so that insertions and retrievals are performed in $O(1)$ time. This is what happens with the banana and titanic data sets. On the other hand, if $\sum_{i=1}^{d}(k_i+1)$ is too large we can still use this integer representation to develop a natural class of hash functions that are *universal* [4], and therefore guarantee an *expected* run time of $O(1)$ for insertions and retrievals.

Given the rectangle $X$ and the vector $k$, a rectangle $\acute{X} \in \sigma_k(X)$ can be uniquely represented by the integer pairs $(p_1, q_1), ..., (p_d, q_d)$ where $p_i, q_i \in \{0, ..., 2^{k_i}\}$ are *indices* of the left and right endpoints of the $i^{th}$ coordinate interval for $\acute{X}$. For example, with $k = (2,1)$ the rectangle $X_3$ in Figure 1 can be uniquely represented by the pairs $(2,3),(1,2)$, and with $k = (3,3)$ this same rectangle would be represented by the pairs $(4,6),(4,8)$. A more compact representation can be obtained by replacing each "endpoint index pair" with an "endpoint index sum", giving rise to the endpoint index sum vector $(s_1, ..., s_d)$, $s_i = p_i + q_i$. For example, with $k = (2,1)$ and $k = (3,3)$ the rectangle $X_3$ is uniquely represented by the sum vectors $(5,3)$ and $(10,12)$ respectively. The uniqueness of this representation is a consequence of the dyadic splits[4] . The values $s_i$ are examples of the $(k_i+1)$-bit integer representations for dyadic intervals mentioned above, and their concatenation is an example of a unique integer representation for rectangles. Now consider a hash table $S$ where $|S|$ is a prime number larger than the number of rectangles to be stored in the table. Then if the integer values $u_i, i = 1, ..., d$ are chosen randomly according to a uniform distribution over $\{0, ..., |S|-1\}$ the hash function

$$h(\acute{X}) := \left(\sum_{i=1}^{d} u_i s_i\right) \mod |S|$$

is a member of a *universal class of hash functions*, and therefore the expected number of times the hash value of any two members of $\sigma_k(X)$ will be the same is less than 1 [4]. Thus the expected run time for insertions and retrievals, over random samplings of $(u_1, ..., u_d)$, is $O(1)$.

In our case it is easy to determine a suitable value for the hash table size $|S|$ ahead of time. First we know that it will never be required to store more than $|\mathcal{T}_k| = |\sigma_k(X)| = \prod_{i=1}^{d}(2^{k_i+1}-1)$ rectangles. However we can often compute a tighter upper bound on the storage as follows. Blanchard et al. prove that when $k_i = k_0, \forall i$ the number of rectangles occupied by a specific data sample is exactly $\rho = (k_0+1)^d$. The same proof technique can be used to establish the more general result $\rho = \prod_{i=1}^{d}(k_i+1)$ when the values of $k_i$ are possibly different. Thus $\acute{n}\rho$ (where $\acute{n}$ is the size of the compressed data set) is a simple (and often tight) bound on the total number of rectangles. With this it is sufficient to choose $|S|$ to be the smallest prime number that is larger than $\min(\acute{n}\rho, |\mathcal{T}_k|)$.

Table 5 shows statistics for $|\mathcal{T}_k|$ and $\acute{n}\rho$ computed across the resampled data sets. It also shows the statistics for the actual number of rectangles stored by the MRP instance

[4] To verify that the endpoint index sum vector is a *unique* representation of the rectangle we show how it can be used to uniquely reconstruct the two endpoint indices: $p_i = (s_i - \Delta)/2$ and $q_i = (s_i + \Delta)/2$ where $\Delta$ is the smallest nonzero term $b_j 2^j$ from the binary expansion $s_i = \sum_{j=0}^{k_i} b_j 2^j, \quad b_j \in \{0,1\}$.

that is invoked with the smallest $\lambda$ value, i.e. the instance that visits the maximum number of rectangles and therefore places the highest demand on the storage data structure. For every data set except *titanic* the value $\acute{n}\rho$ is a much tighter bound on storage than $|\mathcal{T}_k|$, and in all cases the value $\min(\acute{n}\rho, |\mathcal{T}_k|)$ leads to a reasonable and practical choice for the hash table size.

**Table 5** This table shows statistics for $|\mathcal{T}_k|$, $\acute{n}\rho$, and the actual number of rectangles stored in the hash table. In all cases the MRP algorithm was invoked with $\lambda = 1/nl_{bound}$ to maximize the number of stored rectangles. The symbol **B** is used to represent a number that is greater than 6 orders of magnitude larger than the largest $\acute{n}\rho$ row entry.

| Data Set | $\|\mathcal{T}_k\|$ (min mean max) | $\acute{n}\rho$ (min mean max) | Number of Rects. (min mean max) | |
|---|---|---|---|---|
| banana | (16 680 4190) | (18 32 48) | (2.1 4.3 6.4) | $\times 10^3$ |
| breast cancer | (67 156 286) | (3.3 4.3 5.6) | (1.5 2.1 3.0) | $\times 10^6$ |
| diabetes | (256 256 256) | (2.9 3.0 3.0) | (1.1 1.2 1.4) | $\times 10^7$ |
| flare-solar | (13 59 129) | (2.7 5.2 7.5) | (1.2 2.3 3.7) | $\times 10^5$ |
| thyroid | (7.6 2500 9900) | (.6 5.3 10.9) | (.15 .84 1.8) | $\times 10^5$ |
| titanic | (27 42 63) | (40 86 168) | (19 37 59) | $\times 10^0$ |
| Spambase | (**B** **B** **B**) | (49 49 49) | (8.1 8.3 8.7) | $\times 10^6$ |
| Intrusion | (**B** **B** **B**) | (112 146 161) | (2.3 2.8 3.0) | $\times 10^6$ |
| Mushroom | (282 282 282) | (43 43 44) | (5.1 5.1 5.2) | $\times 10^6$ |

The hash table implementation of $S$ requires a method for collision resolution. We adopt a method that stores all rectangles that hash to the same location in a linked list, and then resolves collisions by searching the list. Table 6 provides statistics on the length of these lists computed across all occupied hash table locations and all resampled data sets. As expected the average list size is always less than 2, confirming the *universal* hash property mentioned above. Since $S$ typically has a large number of entries, and the MRP algorithm requires repeated access to most of them, the ability to quickly access a vast majority of these entries has a significant affect on the overall run time.

## 7 Related Work and Conclusions

Decision trees have applicability in diverse areas such as decision table programming, switching theory, boolean expression evaluation, machine fault location, taxonomy representation, database management, and pattern recognition. The evolution of decision tree design methods can be traced through a collection of survey articles that span nearly four decades of work [14,10,19,11]. Decision tree design problems are roughly characterized by the following ingredients:

– the way the tree is used (e.g. to *represent* or *summarize* an existing finite data set, or to make *predictions* about future data drawn from a potentially infinite set),
– the criterion used to assess the tree quality (e.g. the size of the tree, the depth of the tree, the cost of individual variable components, or the regularized risk)
– the characteristics and constraints of the tree class (e.g. the number of children per node (binary or $m$-ary), the number of components used in the split rule (univariate or multivariate), and possible constraints on the split position (unconstrained, dyadic))

**Table 6** For each data set, $d$ is the dimension and $n$ the number of samples in the resampled training sets. This table shows the average hash table linked list size computed across all occupied hash table locations and all resampled data sets. In all cases the MRP algorithm was invoked with $\lambda = 1/nl_{bound}$ to maximize the number of stored rectangles. *In the banana and titanic data sets the total number of potential splits was small enough that exact hashing (with no collisions) was possible.

| Data Set | $d$ | $n$ | Linked list size for occupied hash locations (min mean max) |
|---|---|---|---|
| banana | 2 | 400 | 1 * |
| breast cancer | 9 | 200 | 1.45 1.66 2.30 |
| diabetes | 8 | 468 | 1.33 1.53 1.79 |
| flare-solar | 9 | 666 | 1.00 1.07 1.83 |
| thyroid | 5 | 140 | 1.00 1.04 1.23 |
| titanic | 3 | 150 | 1 * |
| Spambase | 4 | 3301 | 1.03 1.04 1.05 |
| Intrusion | 4 | 44463 | 1.03 1.04 1.05 |
| Mushroom | 10 | 4624 | 1.13 1.18 1.31 |

The greatest emphasis has been placed on binary trees with univariate split rules and unconstrained split positions. In most cases it is NP-Hard to find an optimal tree. Thus, algorithms that produce optimal trees are only practical for small problem instances, or instances where the data has special structure. Nevertheless, it is often possible to derive optimal tree algorithms that are much faster than brute force search, and therefore turn out to have practical utility. Early work in this direction used a branch-and-bound (BB) approach to build decision table representations with minimum storage (smallest tree) or minimum average access time (balanced tree) [17,18]. Subsequent work showed how to improve efficiency, extend to prediction problems, and incorporate a broader class of optimization criteria through the use of dynamic programming methods [7,9, 20,13,8].

The computational requirements of pattern recognition problems have focused most efforts on non-optimal methods. For example the most common design algorithms, e.g. CART [3] and C4.5 [15,16], use a greedy splitting algorithm to construct an initial tree followed by a *pruning* algorithm to produce the final tree. These algorithms are fast, scale to large problem sizes, and work well in some important applications, but are generally not robust to the data distribution, and can therefore produce arbitrarily bad results ([5], Chapter 20).

Recent work with *dyadic* decision trees has provided a potential way to address this weakness. The dyadic restriction on the split rule allows the construction of a simple structural risk minimization algorithm that is guaranteed to be robust to distribution [21–23]. Alternatively, minimizing $r_\lambda$ over a general class of dyadic trees also yields robust performance guarantees [23,2] and tends to give better results in practice [2]. Blanchard et al. [2] developed a DP algorithm for this problem with upper and lower run time bounds dominated by $(\log n)^d$. This is analogous to the $O((m+1)^d)$ run time of a much earlier DP algorithm for decision tables [8] (where $m$ is the maximum number of values taken by an individual variable).

Although DP algorithms have a significant run time advantage over BB, they require that the deepest tree level be known ahead of time (since they build the tree from the bottom up). However the DP algorithm can sometimes be replaced by a memoized recursive algorithm that retains the computational advantages of DP but builds the tree from the top down (e.g. see [4], Section 16.2). For example Nijssen et al. use an MR algorithm to build "optimal decision tree answers" to constraint-based queries over finite data sets [12]. In this paper we have developed an MR algorithm for dyadic decision trees, and added a simple look-ahead pruning method that yields a significantly smaller lower run time bound $\Omega(n)$ and provides substantial speed-ups for most regularization parameter values.

Determining an appropriate value for the regularization parameter $\lambda$ is a critical part of the tree design process, and plays a central role in most tree pruning algorithms. In particular the choice of $\lambda$ in the ODT method is directly analogous to the choice of $\lambda$ in the CART pruning method. Indeed the CART pruning method uses the same regularized risk $r_\lambda$ but explores a smaller and simpler class of trees (i.e. nested subtrees of the greedy tree) which greatly simplifies the search over $\lambda$. For the ODT method we have provided a complete characterization of $\lambda$ in terms of the solution value sequence, and exploited this result to develop algorithms for searching a finite grid of regularization parameter values that is guaranteed to yield optimal solutions for all relevant trees sizes. Finally, we have developed an efficient hash table data structure to support the MR-with-pruning algorithm, and have developed a collection of data pre-processing algorithms that determine the hash table size, provide run time enhancement through data compression, and provide additional "look-ahead pruning" by using the components of $k$ to limit the number of splits per dimension.

## References

1. A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.
2. G. Blanchard, C. Schäfer, Y. Rozenholc, and K.-R. Müller. Optimal dyadic decision trees. *Mach. Learn.*, 66(2-3):209–241, 2007.
3. L. Breiman, J. Friedman, J. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth, 1984.
4. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.
5. L. Devroye, L. Györfi, and G. Lugosi. *A Probabilistic Theory of Pattern Recognition.* Springer, New York, NY, 1996.
6. D. Donoho. Cart and best-ortho-basis: a connection. *Annals of Statistics*, 25:1870–1911, 1997.
7. M.R. Garey. Optimal binary identification procedures. *SIAM Journal on Applied Mathematics*, 23(2):173–186, September 1972.
8. A. Lew. Optimal conversion of extended-entry decision tables with general cost criteria. *Communications of the ACM*, 21(4):269–279, April 1978.
9. W.S. Meisel and D.A. Michalopoulos. A partitioning algorithm with application in pattern classification and the optimization of decision trees. *IEEE Transactions on Computers*, 22(1):93–103, January 1973.
10. B.M.E. Moret. Decision trees and diagrams. *Computing Surveys*, 14(4):593–623, December 1982.
11. S.K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2:345–389, 1998.
12. S. Nijssen and E. Fromont. Mining optimal decision trees from itemset lattices. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 530–539, 2007.
13. H.J. Payne and W.S. Meidel. An algorithm for constructing optimal binary decision trees. *IEEE Transactions on Computers*, 26(9):905–916, September 1977.
14. U.W. Pooch. Translation of decision tables. *Computing Surveys*, 6(2):125–151, June 1974.
15. J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.
16. J.R. Quinlan. Improved use of continuous attributes in c4.5. *Journal of Artificial Intelligence Research*, 4:77–90, 1996.

17. L.T. Reinwald and R.M. Soland. Conversion of limited-entry decision tables to optimal computer programs I: minimum average processing time. *Journal of the ACM*, 13(3):339–358, July 1966.
18. L.T. Reinwald and R.M. Soland. Conversion of limited-entry decision tables to optimal computer programs II: minimum storage requirement. *Journal of the ACM*, 14(4):742–755, October 1967.
19. S.R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, May 1991.
20. H. Schumacher and K.C. Sevcik. The synthetic approach to decision table conversion. *Communications of the ACM*, 19(6):343–351, June 1976.
21. C. Scott and R. Nowak. Dyadic classification trees via structural risk minimization. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 359–366. MIT Press, Cambridge, MA, 2003.
22. C. Scott and R. Nowak. Near-minimax optimal classification with dyadic classification trees. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.
23. C. Scott and R. Nowak. Minimax optimal classification with dyadic decision trees. *IEEE Transactions on Information Theory*, 52(4):1335–1353, 2006.

## A Data Set Descriptions

The first six data sets, banana through titanic, are the same data sets used in [2]. They are pre-processed versions of the corresponding UCI repository [1] data sets[5]. Each data set has been partitioned into "training" and "test" sets a total of 100 times by random sampling. The "training" sets represent the 100 resampled data sets used in the experiments throughout this paper.

The three additional data sets, Spambase, intrusion and mushroom, were included because they are considerably larger than the previous data sets. Each of these data sets was randomly sampled a total of 10 times to obtain the resampled data sets used in the experiments throughout this paper.

The mushroom data set was derived from the corresponding data set in the UCI repository [1]. All 21 features of the original data are symbolic and were converted to numerical values by assigning the values 1,2,3,... to the first, second, third, ... symbol. Although this is arguably not the best representation, it turns out to be sufficient to give perfect (zero error) generalization estimates for tree classifiers. It also turns out that perfect results can be obtained with only a subset of these 21 features. In this paper we kept 10 features that appeared to work best when applied on their own.

The Spambase data set was also derived from the corresponding data set in the UCI repository [1]. In this paper we kept 4 of the 57 original features that appeared to work best when applied on their own. We restricted to 4 features because the memory requirements of our algorithm were too large otherwise (i.e. experiments with this data set were constrained by memory requirements, not run time). Nevertheless we achieved classification errors around 11%, which is only about 4% larger than the error achievable using all 57 features. In some applications it is worth considering a small sacrifice in performance for such a large reduction in the number of features.

The intrusion data set was derived from the KDD Cup 1999 Intrusion Detection Competition data[6]. In this paper we kept 4 of the 31 original features that appeared to work best when applied on their own. Once again we restricted to 4 features because the memory requirements were too large otherwise, and once again we noticed only a small degradation in performance due to this restriction.

Performance comparisons for the first six data sets are shown in Blanchard et al. [2] and they indicate that the ODT methods tends to produce smaller and (slightly) more accurate decision trees than C4.5. Table 7 compares the the ODT method described in this paper to the standard CART method for the three additional data sets. The results in the table suggest that

[5] The pre-processed data sets are available at `http://ida.first.fraunhofer.de/projects/bench/benchmarks.htm`.

[6] See `http://www.sigkdd.org/kddcup/index.php?section=1999&method=task`

the ODT method tends to produce smaller trees with approximately the same classification error as CART.

**Table 7** This table compares the tree size and generalization error estimate of ODT and CART on the three large data sets. These results represent statistics computed over 10 random partitions of the data into "design" and "test" sets, where the design/test set sizes were 3601/1000 for Spambase, 49,403/444,618 for intrusion, and 6124/2000 for mushroom. The CART algorithm, with the Gini split criterion and the standard pruning method (based on 10-fold cross-validation), was applied to each of the 10 design sets. The error estimates for each of these 10 trees were computed on the corresponding test sets and the results summarized here. For the ODT method the design sets were partitioned into "training" and "validation" sets (approximately 90% train and 10% validation), the adaptive+heuristic grid search was applied to the training sets and solutions that minimized the empirical classification error on the validation set were selected. Final error estimates for the 10 selected trees were again computed on the corresponding test sets and summarized here.

| | Error Estimate | |
|---|---|---|
| | ODT | CART |
| Data Set | (min mean max) | (min mean max) |
| Spambase | 0.092 0.109 0.125 | 0.093 0.113 0.143 |
| Intrusion | 0.0012 0.0015 0.0019 | 0.0014 0.0015 0.0017 |
| Mushroom | 0 0 0 | 0.0 0.00015 0.0015 |
| | Number of Leaves | |
| | ODT | CART |
| Data Set | (min mean max) | (min mean max) |
| Spambase | 23 50.7 77 | 19 66.4 233 |
| Intrusion | 44 82 156 | 55 92 161 |
| Mushroom | 9 9 9 | 29 30.6 33 |

## B Proofs

*Proof (Lemma 1)* First note that if the optimal tree size is 1 then $T^0_{\bar{X}}$ is the optimal tree. We complete the proof by assuming there is an optimal solution for $\lambda > e_{\bar{D}}(T^0_{\bar{X}})$ with number of leaves $l^*_\lambda \geq 2$, and then show that this leads to a contradiction. Let $r^*_\lambda$ and $e^*_\lambda$ be the risk and the error of the assumed optimal solution. The risk satisfies

$$\begin{aligned} r^*_\lambda &= e^*_\lambda + \lambda l^*_\lambda \\ &\geq \lambda l^*_\lambda \\ &= \lambda(l^*_\lambda - 1) + \lambda \\ &> e_{\bar{D}}(T^0_{\bar{X}})(l^*_\lambda - 1) + \lambda \\ &\geq e_{\bar{D}}(T^0_{\bar{X}}) + \lambda \end{aligned}$$

where the fourth line follows from the assumption that $\lambda > e_{\bar{D}}(T^0_{\bar{X}})$ and the last line follows from the assumption that $l^*_\lambda \geq 2$. Since the risk $e_{\bar{D}}(T^0_{\bar{X}}) + \lambda$ is a realizable risk value, the inequality above contradicts the optimality assumption for $r^*_\lambda$ implying that the optimal tree size must be 1. □

*Proof (Lemma 2)* The following abbreviated notation will be used in this proof.

$$\begin{aligned} r_i &:= r^*_{\lambda_i,X} \\ e_i &:= e^*_{\lambda_i,X} \\ l_i &:= |T^*_{\lambda_i,X}| \end{aligned}$$

The optimality of $T^*_{\lambda_i,X}$ for $\lambda_i$ gives the following two inequalities,

$$r_1 = e_1 + \lambda_1 l_1 \leq e_2 + \lambda_1 l_2 \ , \tag{7}$$

$$r_2 = e_2 + \lambda_2 l_2 \leq e_1 + \lambda_2 l_1 \ . \tag{8}$$

Combining (7) with $\lambda_1 < \lambda_2$ yields

$$r_1 \leq e_2 + \lambda_1 l_2 \leq e_2 + \lambda_2 l_2 = r_2$$

which proves the first inequality in the theorem. Solving for $(e_1 - e_2)$ in (7) and (8), and combining the results gives

$$\lambda_2(l_2 - l_1) \quad \leq \quad e_1 - e_2 \quad \leq \quad \lambda_1(l_2 - l_1).$$

The combination of $\lambda_2(l_2 - l_1) \leq \lambda_1(l_2 - l_1)$ and $0 \leq \lambda_1 < \lambda_2$ imply that $l_2 \leq l_1$ which proves the third inequality in the theorem. Finally, the combination of $e_1 - e_2 \quad \leq \quad \lambda_1(l_2 - l_1)$ with $\lambda_1 \geq 0$ and $l_2 - l_1 \leq 0$ imply $e_1 \leq e_2$ which completes the proof. □

*Proof (Lemma 3)* The optimality of $T^*_{\lambda_i,X}$ for $\lambda_i$ gives

$$e_i + \lambda_i l_i \leq e_{i+1} + \lambda_i l_{i+1}. \tag{9}$$

Solving for $\lambda_i$ gives

$$\lambda_i \geq \frac{e_i - e_{i+1}}{l_{i+1} - l_i}.$$

Next we show that this inequality must be strict, i.e.

$$\lambda_i > \frac{e_i - e_{i+1}}{l_{i+1} - l_i}. \tag{10}$$

Assume equality in (9) so that $\lambda_i = \frac{e_i - e_{i+1}}{l_{i+1} - l_i}$. This implies that both $(e_i, l_i)$ and $(e_{i+1}, l_{i+1})$ are optimal solution values for $\lambda_i$ and since $l_{i+1} > l_i$ the selection criterion in Definition 3 will not choose $(e_i, l_i)$ as the solution value corresponding to $\lambda_i$. Thus, if $(e_i, l_i)$ appears in the solution-value-sequence then it must be obtained using a value $\lambda_i$ that satisfies the strict inequality in (10). Next, the optimality of $T^*_{\lambda_i,X}$ for $\lambda_i$ gives

$$e_i + \lambda_i l_i \leq e_{i-1} + \lambda_i l_{i-1}.$$

Solving for $\lambda_i$ and combining with the above result gives

$$\frac{e_i - e_{i+1}}{l_{i+1} - l_i} < \lambda_i \leq \frac{e_{i-1} - e_i}{l_i - l_{i-1}} \tag{11}$$

For $i = 1, 2, ..., m$ this equation defines $m$ disjoint intervals whose union is the interval $(0, e_1]$. Since all values of $\lambda$ in $(0, e_1]$ produce a solution whose value appears in the solution value sequence, and since the $m$ intervals defined by (11) are disjoint, it must be true that *all* values of $\lambda$ in the $i^{th}$ interval yield a solution value $(e_i, l_i)$. □

*Proof (Lemma 4)* The additivity of error and tree size over sub-trees allows us to write

$$\begin{aligned} e_{i-1} - e_i &= e(T^*_X(i-1)) - e(T^*_X(i)) \\ &= e(T^*_X(i-1)) - e(\bar{T}_X(i)) + \sum_{j=1}^{m} \delta(T^*_{X_j}(i)) \end{aligned}$$

and

$$\begin{aligned} l_i - l_{i-1} &= |T^*_X(i)| - |T^*_X(i-1)| \\ &= |\bar{T}_X(i)| + \sum_{j=1}^{m} |T^*_{X_j}(i)| - |T^*_X(i-1)| \\ &= \sum_{j=1}^{m} |T^*_{X_j}(i)|. \end{aligned}$$

Since $e(T^*_X(i-1))$ is minimal for trees of size $|T^*_X(i-1)|$ we have $e(T^*_X(i-1)) - e(\bar{T}_X(i)) \geq 0$ and therefore

$$e_{i-1} - e_i \leq \sum_{j=1}^{m} \delta(T^*_{X_j}(i)) \tag{12}$$

and

$$\frac{e_{i-1} - e_i}{l_i - l_{i-1}} \leq \frac{\sum_{j=1}^{m} \delta(T^*_{X_j}(i))}{\sum_j |T^*_{X_j}(i)|} \leq \max_{j \in \{1,\dots,m\}} \left( \frac{\delta(T^*_{X_j}(i))}{|T^*_{X_j}(i)|} \right) = \max_{j \in \{1,\dots,m\}} g(T_{\acute{X}^*_j}(i))$$

where the last inequality follows from the bound

$$\frac{\sum_{j=1}^{m} a_j}{\sum_{j=1}^{m} b_j} \leq \max_{j \in \{1,\dots,m\}} \left( \frac{a_j}{b_j} \right).$$

which holds for any $a_j > 0$, $b_j > 0$, $j = 1, \dots, m$. □