CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Erasmus University Digital Repository

Monotone Decision Trees

Rob Potharst, Jan C. Bioch and Thijs Petter

Department of Computer Science
Erasmus University Rotterdam
P.O. Box 1738, 3000 DR Rotterdam
The Netherlands
{bioch, potharst}@few.eur.nl

Abstract

In many classification problems the domains of the attributes and the classes are linearly ordered. Often, classification must preserve this ordering: this is called monotone classification. Since the known decision tree methods generate non-monotone trees, these methods are not suitable for monotone classification problems. In this report we provide a number of order-preserving tree-generation algorithms for multi-attribute classification problems with k linearly ordered classes.

1 Introduction

Ordinal classification refers to an important category of real-world problems, in which the attributes of the objects to be classified and the classes are ordered. For this class of problems classification rules often need to be order-preserving. In that case we have a monotone classification problem. In this paper we study the problem of generating decision-tree-classifiers for monotone classification problems: the attributes and the set of classes are linearly ordered. Ordinal classification for multi-attribute decision making has been studied recently by Ben-David [1, 2, 3] for discrete domains, and by Makino *et al.* [5] for the two-class problem with continuous attributes. However, although the tree-generation method of Ben-David accounts for the ordering of the attributes and of the classes, order-preserving is not guaranteed. Furthermore, the method of Makino *et al.* is restricted to the two-class problem. In this technical report we provide several algorithms for monotone classification problems with k-classes for discrete and continuous domains.

As an example of a monotone classification problem, suppose a bank wants to base its loan policy on a number of features of its clients, for instance on income, education level and criminal record. If a client is granted a loan, it can be one in three classes: low, intermediate and high. So, together with the loan option, we have four classes. Suppose further that the bank wants to base its loan policy on a number of credit worthiness decisions in the past. These past decisions are given in Table 1. A client with features at least as high as those of another client may expect to get at least as high a loan as the other client. So, finding a loan policy compatible with past decisions amounts to solving a monotone classification problem with the dataset of Table 1.

The organization of this Technical Report is as follows: in Section 2 we introduce monotone classification problems, develop some theory about those problems and end up

client	*income*	*education*	*crim.record*	*loan*
cl1	low	low	fair	no
cl2	low	low	excellent	low
cl3	average	intermediate	excellent	intermediate
cl4	high	low	excellent	intermediate
cl5	high	intermediate	excellent	high

Table 1: The bank loan dataset

to introduce the concept of a monotone decision tree. In sections 3 and 4 we propose and prove the correctness of different algorithms for the induction of monotone decision trees. The algorithms of Section 3 use local datasets and give only special decision trees, so-called minimal and maximal trees. The algorithms of Section 4 use a global dataset, that is updated during the algorithm. These algorithms produce general solutions to a monotone classification problem: monotone decision trees that are quite small, compared to the trees of Section 3. In Section 5 we report the results of some experiments with artificial datasets. Finally, Section 6 concludes this report.

Remark on this July 1997 edition of the report: in this edition the text of several subsections has not been filled in yet. These texts will be furnished in a Supplement to this report, to be published shortly.

2 Monotone Classification

Let $\mathcal{X}$ be a partially ordered space, called the *input space*, with partial ordering $<$, and let $\mathcal{C}$ be a finite linearly ordered set of *classes*, with linear ordering $<$. A *classification rule* or *class labeling* is a function

$$\lambda : \mathcal{X} \to \mathcal{C}$$

which assigns a class from $\mathcal{C}$ to every point in the input space $\mathcal{X}$. A *classification problem* is the problem of finding a class labeling λ that satisfies certain side conditions, to be specified in the problem description. One possible side condition is that the labeling λ be monotone: a *monotone* classification rule is a function $\lambda : \mathcal{X} \to \mathcal{C}$ for which

$$x \leq y \Rightarrow \lambda(x) \leq \lambda(y) \tag{1}$$

for all points $x, y \in \mathcal{X}$.

As an example, let $X_1, \ldots, X_n$ be a set of outcomes of academic and/or psychological tests that could be taken from an applicant to an educational institution. Each test X_i may take values x_i in, say, $\{0, 1, \ldots, 10\} = I$. So each applicant produces a vector $(x_1, x_2, \ldots, x_n) \in I^n = \mathcal{X}$. These vectors are indeed partially ordered, for instance the outcome $(5, 5, \ldots, 5)$ is smaller than the outcome $(6, 6, \ldots, 6)$, but $(5, 6, 5, 6, \ldots)$ is incomparable to $(6, 5, 6, 5, \ldots)$. The institution uses the test outcomes for its admission policy: some applicants are admitted to the institution, some are immediately rejected and finally some are conditionally admitted. So we have a set of three admissibility classes $\mathcal{C} = \{r, ca, a\}$. It obviously makes sense to order these classes as follows: $r < ca < a$. An admission policy of the institution can now be seen as a classification rule λ that assigns an admissibility class r, ca or a to every possible vector of test outcomes $(x_1, \ldots, x_n)$. If the institution cares for its public relations, it may want an admission policy that is

monotone: if one applicant has at least as good grades as another, normally it would be unwise to put him or her in a lower admissibility class. Thus, monotonicity would be a fair requirement to be met by a decent admission policy of the institution.

A very common classification problem occurs, when there is a dataset or set of examples available. The usual side condition to be met in such a situation is that the classification rule one is looking for should correctly classify all examples in the dataset. With this situation we will deal in the next subsection.

2.1 Monotone Datasets

A dataset is a finite collection of examples from the input space, together with a class labeling of all these examples. Formally, we define a dataset as follows:

Definition 1 A *dataset* $\mathcal{D}$ is a pair (D, λ) where $D \subset \mathcal{X}$ is a finite subset of the input space $\mathcal{X}$ and $\lambda : D \to \mathcal{C}$ is a class labeling of the elements of D. The elements of D will be called the *examples* of the dataset.

Note first of all that the class labeling λ of a dataset $\mathcal{D} = (D, \lambda)$ is *not* a classification rule: it is only defined on D, a subset of $\mathcal{X}$, while a classification rule must be defined on all elements of the input space $\mathcal{X}$. Secondly, we do not allow an example to have two or more different classes: all elements of the dataset must be consistently labeled.

Given a dataset $\mathcal{D} = (D, \lambda)$ we can try to solve the corresponding *monotone classification problem* of finding a monotone classification rule $\hat{\lambda} : \mathcal{X} \to \mathcal{C}$ that extends the class labeling λ of the dataset $\mathcal{D}$ to the entire input space $\mathcal{X}$. Thus, $\hat{\lambda}(x) = \lambda(x)$ for all $x \in D$. Obviously, if one wants to find a solution for such a monotone classification problem, the dataset itself has to be monotone:

Definition 2 A dataset $\mathcal{D} = (D, \lambda)$ is called *monotone* if the implication (1) holds for all $x, y \in D$.

The problem of checking whether a given dataset is monotone will be dealt with in Section 2.6. As an example of a monotone classification problem, consider the bank loan problem that was sketched in the Introduction. In order to save space we will often map the values of the attributes of a dataset to a set of numbers. For instance, Table 1 could be written as

X_1	X_2	X_3	C
0	0	1	0
0	0	2	1
1	1	2	2
2	0	2	2
2	1	2	3

when we use the mapping low $\to$ 0, average $\to$ 1, high $\to$ 2 for feature $X_1 = income$, etc. More often, we will write concisely

001	0
002	1
112	2
202	2
212	3

for the above dataset, while the first data element will be denoted as $001 : 0$, etc.

Finally, we will establish some notation to be used throughout this paper:

- The minimal and maximal elements of $\mathcal{C}$ will be denoted by $c_{\min}$ and $c_{\max}$ respectively.
- $[\mathcal{C}]$ denotes the set of intervals, based on elements of $\mathcal{C}$, thus
$$[\mathcal{C}] = \{[c, d] : c, d \in \mathcal{C}, c \leq d\}.$$
Note that $[\mathcal{C}]$ is partially ordered by the following order relation:
$$[c_1, d_1] \leq [c_2, d_2] \Leftrightarrow c_1 \leq c_2 \text{ and } d_1 \leq d_2.$$
- For all $x \in \mathcal{X}$, we define the *upset* generated by x as
$$\uparrow x = \{y \in \mathcal{X} : y \geq x\}$$
and, if D is a subset of $\mathcal{X}$ the upset generated by D is defined as
$$\uparrow D = \bigcup_{x \in D} \uparrow x.$$
- Similarly, for $x \in \mathcal{X}$, we define the *downset* generated by x as
$$\downarrow x = \{y \in \mathcal{X} : y \leq x\}$$
and the downset generated by a subset D of $\mathcal{X}$ is defined as
$$\downarrow D = \bigcup_{x \in D} \downarrow x.$$
Note, that $\downarrow\downarrow D = \downarrow D$ and $\uparrow\uparrow D = \uparrow D$.
- Finally, an element $x \in \mathcal{X}$ will be called *comparable* to at least one element of D if
$$x \in \uparrow D \cup \downarrow D.$$

2.2 Monotone Extensions of Datasets

As noted in Section 2.1 the problem of finding a solution to a monotone classification problem amounts to finding a monotone extension $\hat{\lambda}$ of the class labeling λ of a dataset $\mathcal{D} = (D, \lambda)$. Formally, a function $\hat{\lambda} : \mathcal{X} \to \mathcal{C}$ is an *extension* of $\lambda : \mathcal{X} \to \mathcal{C}$, if the restriction of $\hat{\lambda}$ to D i.e. $\hat{\lambda}|D = \lambda$. Or, if $\hat{\lambda}(x) = \lambda(x)$ for all $x \in D$. If $\mathcal{D} = (D, \lambda)$ is monotone, we denote the collection of all monotone extensions of λ with $\Lambda(\mathcal{D})$. Note that $\Lambda(\mathcal{D})$ is partially ordered by the order relation $\hat{\lambda} \leq \hat{\lambda}'$ iff $\hat{\lambda}(x) \leq \hat{\lambda}'(x)$ for all $x \in \mathcal{X}$. We will now define two special elements of this collection.

Definition 3 If $\mathcal{D} = (D, \lambda)$ is a monotone dataset, we define $\lambda_{\min}^{\mathcal{D}} : \mathcal{X} \to \mathcal{C}$, and $\lambda_{\max}^{\mathcal{D}} : \mathcal{X} \to \mathcal{C}$, as follows: for all $x \in \mathcal{X}$

$$\lambda_{\min}^{\mathcal{D}}(x) = \begin{cases} \max\{\lambda(y) : y \in D \cap \downarrow x\} & \text{if } x \in \uparrow D \\ c_{\min} & \text{otherwise} \end{cases}$$

and

$$\lambda_{\max}^{\mathcal{D}}(x) = \begin{cases} \min\{\lambda(y) : y \in D \cap \uparrow x\} & \text{if } x \in \downarrow D \\ c_{\max} & \text{otherwise.} \end{cases}$$

We will now show that the functions $\lambda_{\min}^{\mathcal{D}}$ and $\lambda_{\max}^{\mathcal{D}}$, as defined, are the minimal resp. maximal elements of $\Lambda(\mathcal{D})$.

Lemma 1 *If $\mathcal{D} = (D, \lambda)$ is a monotone dataset, for the functions $\lambda_{\min}^{\mathcal{D}}$ and $\lambda_{\max}^{\mathcal{D}}$ the following statements hold:*

(i) $\lambda_{\min}^{\mathcal{D}}, \lambda_{\max}^{\mathcal{D}} \in \Lambda(\mathcal{D})$

(ii) $\Lambda(\mathcal{D}) = \{\hat{\lambda} : \lambda_{\min}^{\mathcal{D}} \leq \hat{\lambda} \leq \lambda_{\max}^{\mathcal{D}} \text{ and } \hat{\lambda} \text{ monotone}\}$.

Proof: Part (i). For the first part of the lemma we need to prove that $\lambda_{\min}^{\mathcal{D}}$ and $\lambda_{\max}^{\mathcal{D}}$ 1) are extensions of λ and 2) that they are monotone. If $x \in D$, then because of the monotonicity of λ for all $y \in D \cap \downarrow x$ we have $\lambda(y) \leq \lambda(x)$ so that $\lambda_{\min}^{\mathcal{D}}(x) = \lambda(x)$. Thus, $\lambda_{\min}^{\mathcal{D}}$ (and $\lambda_{\max}^{\mathcal{D}}$ in the same way) is an extension of λ. To show that $\lambda_{\min}^{\mathcal{D}}$ is monotone, let $x \leq y$. Then, if $x \notin \uparrow D$, we have $\lambda_{\min}^{\mathcal{D}}(x) = c_{\min} \leq \lambda_{\min}^{\mathcal{D}}(y)$. So, suppose $x \in \uparrow D$. Then, since $x \leq y$ also $y \in \uparrow D$, and $D \cap \downarrow x \subset D \cap \downarrow y$. Thus, $\lambda_{\min}^{\mathcal{D}}(x) \leq \lambda_{\min}^{\mathcal{D}}(y)$. Similarly, we have $\lambda_{\max}^{\mathcal{D}}(x) \leq \lambda_{\max}^{\mathcal{D}}(y)$. So $\lambda_{\min}^{\mathcal{D}}$ and $\lambda_{\max}^{\mathcal{D}}$ are both monotone extensions of λ.

Part (ii). First we prove the $\subset$ part. Suppose $\hat{\lambda} \in \Lambda(\mathcal{D})$. If $x \notin \uparrow D$, then $\lambda_{\min}^{\mathcal{D}}(x) = c_{\min} \leq \hat{\lambda}(x)$. If, on the other hand $x \in \uparrow D$, then for any $y \in D \cap \downarrow x$ we have $\lambda(y) = \hat{\lambda}(y) \leq \hat{\lambda}(x)$, where the equality stems from the fact that $\hat{\lambda}$ is an extension of λ and the inequality follows from the monotonicity of $\hat{\lambda}$. So a maximum over those y must also be less than or equal to $\hat{\lambda}(x)$. Thus for all $x \in \mathcal{X}, \lambda_{\min}^{\mathcal{D}}(x) \leq \hat{\lambda}(x)$. Similarly, we can show that $\lambda_{\max}^{\mathcal{D}}(x) \geq \hat{\lambda}(x)$. Finally, we prove the $\supset$ part of (ii). If $\hat{\lambda}$ is monotone and satisfies the given inequalities, then to prove that $\hat{\lambda} \in \Lambda(\mathcal{D})$ amounts to showing that $\hat{\lambda}$ is an extension of λ. But if $x \in D$ then according to the above definition we have $\lambda_{\min}^{\mathcal{D}}(x) = \lambda_{\max}^{\mathcal{D}}(x) = \lambda(x)$. So $\lambda(x) \leq \hat{\lambda}(x) \leq \lambda(x)$, or $\hat{\lambda}(x) = \lambda(x)$. So $\hat{\lambda}$ is an extension of λ and the proof is complete. $\Box$

Remark 2.1 If $\mathcal{X}$ is finite, then $\Lambda(\mathcal{D})$ is a finite distributive lattice with universal bounds $\lambda_{\min}^{\mathcal{D}}$ and $\lambda_{\max}^{\mathcal{D}}$.

Theoretically, we now have at least two solutions for a monotone classification problem with dataset $\mathcal{D} = (D, \lambda)$: the minimal and maximal extension of λ. These two classification rules we will call the *minimal rule* and the *maximal rule* respectively. In addition we have for every point x in the input space bounds that any rule $\hat{\lambda}$ must satisfy:

$$\lambda_{\min}^{\mathcal{D}}(x) \leq \hat{\lambda}(x) \leq \lambda_{\max}^{\mathcal{D}}(x).$$

Any monotone classification rule that satisfies these bounds will be another solution to our problem.

We will now move one step ahead and require the representation of our classification rule to have a specific form, viz. the form of a classification tree or decision tree. This line of thought will be pursued in Section 2.5. First, we will see whether we need all the data in the dataset. It may be that leaving out certain elements of the dataset actually yields no change in the problem, since all the information needed is contained in the remaining data-elements. With this topic we will deal in the next subsection.

2.3 Redundancy

It may occur that not all elements of a dataset are needed to solve a monotone classification problem. In particular this is the case if there are elements in the dataset that, if removed, do not change the solution set of the problem. Such elements are called redundant. A formal definition runs as follows:

Definition 4 Let $\mathcal{D} = (D, \lambda)$ be a monotone dataset. A data element $x \in D$ is called *redundant* with respect to $\mathcal{D}$ if for the dataset $\mathcal{D}^- = (D^-, \lambda^-)$ with $D^- = D \setminus \{x\}$ and $\lambda^- = \lambda | D^-$ the following equality holds:

$$\Lambda(\mathcal{D}^-) = \Lambda(\mathcal{D}).$$

The following lemma expresses redundancy in less abstract terms. It gives a method to check whether a given data element is indeed redundant. For this we only need to calculate the minimal and maximal labeling of the suspected data element, based on the dataset with the element in question removed.

Lemma 2 *Let $\mathcal{D} = (D, \lambda)$ be a monotone dataset, let $x \in D$ be a datapoint in D and let $\mathcal{D}^- = (D^-, \lambda^-)$ with $D^- = D \setminus \{x\}$ and $\lambda^- = \lambda | D^-$. Then we have:*

$$x \text{ is redundant in } \mathcal{D} \Leftrightarrow \lambda_{\min}^{\mathcal{D}^-}(x) = \lambda(x) = \lambda_{\max}^{\mathcal{D}^-}(x). \tag{2}$$

Proof: First we prove the $\Rightarrow$ implication. If x is redundant in $\mathcal{D}$ then by definition $\Lambda(\mathcal{D}^-) = \Lambda(\mathcal{D})$, so $\min\{\hat{\lambda}(x) : \hat{\lambda} \in \Lambda(\mathcal{D}^-)\} = \min\{\hat{\lambda}(x) : \hat{\lambda} \in \Lambda(\mathcal{D})\}$ as well. The left side of this last equality can easily be seen to be equal to $\lambda_{\min}^{\mathcal{D}^-}(x)$, as follows from Lemma 1 applied to $\mathcal{D}^-$. The right side of the same equality is equal to $\lambda(x)$, since $x \in D$ and $\Lambda(\mathcal{D})$ contains only extensions of λ. Together, this yields $\lambda(x) = \lambda_{\min}^{\mathcal{D}^-}(x)$. In a similar way, using the max instead of the min function we can show that $\lambda(x) = \lambda_{\min}^{\mathcal{D}^-}(x)$, thus completing the proof of the first part. To prove the $\Leftarrow$ part, we first note that $\Lambda(\mathcal{D}) \subset \Lambda(\mathcal{D}^-)$ since D^- is a subset of D; so we only should prove $\Lambda(\mathcal{D}^-) \subset \Lambda(\mathcal{D})$ given the right hand side of (2). Let $\hat{\lambda}$ be any monotone extension of $\mathcal{D}^-$. To show that $\hat{\lambda}$ must also be a monotone extension of $\mathcal{D}$ it is enough to show that $\hat{\lambda}(x) = \lambda(x)$. From Lemma 1 it follows that $\hat{\lambda}(x)$ must be in between $\lambda_{\min}^{\mathcal{D}^-}(x)$ and $\lambda_{\max}^{\mathcal{D}^-}(x)$. So, together with the right hand side of (2) it follows that $\hat{\lambda}(x) = \lambda(x)$, proving $\hat{\lambda}$ to be a monotone extension of $\mathcal{D}$. This proves the lemma. $\Box$

In fact, the right hand side of (2) can only be true in one of the following three situations:

(i) $\exists x', x'' \in D$ such that $x' \leq x \leq x''$ and $\lambda(x') = \lambda(x) = \lambda(x'')$

(ii) $\exists x' \in D$ such that $x \leq x'$ and $c_{\min} = \lambda(x) = \lambda(x')$

(iii) $\exists x' \in D$ such that $x' \leq x$ and $\lambda(x') = \lambda(x) = c_{\max}$.

As an example, consider the following monotone datasets

002	0
101	1
102	1
112	1
021	2

001	0
002	0
112	1
202	2
212	3

001	0
002	1
112	2
202	3
212	3

each with 5 examples, three attributes, divided into three or four classes. In the first dataset the element 102 is clearly redundant. This can be seen by noting that 101 < 102 < 112, so if 101 and 112 have class 1, from the monotonicity of any solution it follows that 102 must also have class 1. The remaining datasets form an illustration to situations (ii) and (iii) respectively.

We conclude this section by noting that it is usually wise to remove all redundant elements from a dataset before performing any further calculations on it.

2.4 Generating Random Monotone Datasets

Note to the July 1997 edition: the text of this subsection will be published in the Supplement to this Technical report.

2.5 Monotone Decision Trees

Beginning with this section we will leave the very general viewpoint we had so far. From now on we will make a few assumptions about the concepts we are discussing. In the first place we will assume that our input space $\mathcal{X}$ is a coordinate space. Elements of $\mathcal{X}$ will be vectors $(x_1, \ldots, x_n)$ with coordinates x_i which will take their values from a linearly ordered space $\mathcal{X}_i$. So, formally our first assumption is:

Assumption 1 The input space $\mathcal{X}$ is of the form

$$\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_n$$

where $\mathcal{X}_i$ is a linearly ordered set for $i = 1, \ldots, n$. Here the order relation $\leq$ on $\mathcal{X}$ is defined as $x \leq y$ iff $x_i \leq y_i$ for all $i = 1, \ldots, n$.

Of course, this includes the very common situation that our examples are measurements on n variables $X_1, \ldots, X_n$, where the individual measurement on variable X_i yields a value x_i from an ordered set $\mathcal{X}_i$. So each of the variables may take its values from a different set, as long as all these coordinate sets are linearly ordered.

Our next assumption will concern the representation of the classification rules to be considered, viz. tree-like classification rules, also called classification trees or decision trees. Note, that this is no restriction since decision trees are universal approximators. A *decision tree classifier* is a classification rule that is constructed by splitting the input space $\mathcal{X}$ consecutively in a number of disjoint nonempty subsets, which in turn are splitted again, etc. Such a process can be pictured in a graph such as the one in Figure 1.

The tree consists of a number of *nodes* labeled t_0 to t_4, and a number of *leaves* labeled ℓ_1 to ℓ_7. At each node a subset of $\mathcal{X}$ is split into two or more nonempty subsets. For instance, at node t_0, the *root* of the tree, the input space $\mathcal{X}$ is split into three disjoint subsets T_1, T_2 and T_3; thus, $\mathcal{X} = T_1 \cup T_2 \cup T_3$. At node t_1, subset T_1 is again split into two disjoint subsets T_4 and T_5. Subset T_2 is not split any further, as is the case with all subsets in any of the leaves. In this way, the input space $\mathcal{X}$ is finally split up into as many disjoint nonempty subsets as there are leaves:

$$\mathcal{X} = T_2 \cup T_5 \cup T_6 \cup T_7 \cup T_8 \cup T_9 \cup T_{10}.$$

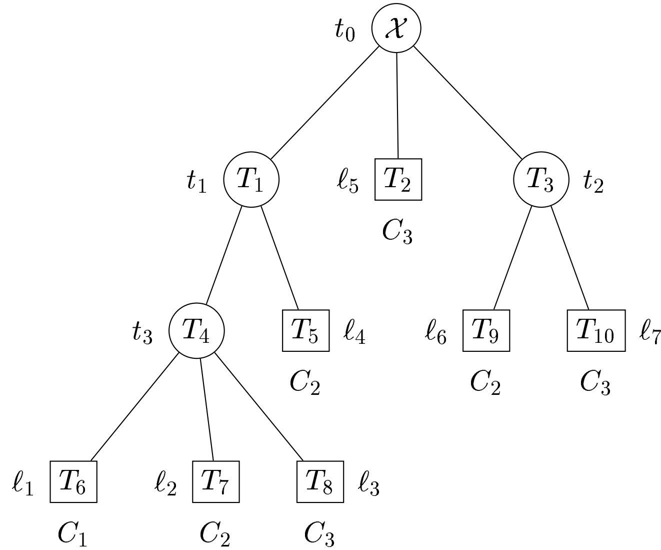

Figure 1: Decision Tree Classifier: Example

The classification rule arises from this splitting process by assigning a class to each leaf. In the above tree class C_1 is assigned to leaf ℓ_1, etc. as can be read from the figure. Thus the shown decision tree $\mathcal{T}$ defines the following classification rule:

$$\lambda_{\mathcal{T}}(x) = \begin{cases} C_3 & \text{if } x \in T_2 \\ C_2 & \text{if } x \in T_5 \\ C_1 & \text{if } x \in T_6 \\ C_2 & \text{if } x \in T_7 \\ C_3 & \text{if } x \in T_8 \\ C_2 & \text{if } x \in T_9 \\ C_3 & \text{if } x \in T_{10}. \end{cases}$$

In this way, a decision tree $\mathcal{T}$ induces a classification rule $\lambda_{\mathcal{T}} : \mathcal{X} \to \mathcal{C}$ as follows: if $x \in \mathcal{X}$ belongs to the subset of $\mathcal{X}$ associated with leaf ℓ_i then we define $\lambda_{\mathcal{T}}(x)$ to be the class assigned to ℓ_i. Finally, we remark that all nodes and leaves of a decision tree are associated with exactly one subset of the input space. In Figure 1 the names of these subsets have been printed inside the circles and boxes that represent the nodes and the leaves of the tree. By slight abuse of language we will often denote a node or leaf with its associated subset. Thus, we will write node T_4 when we mean node t_3 with associated subset T_4.

In this paper we will only consider so-called *univariate* decision trees: at each split the decision to which of the disjoint subsets an element belongs, is made using the information from one variable or coordinate only. Within this class of univariate decision trees, we will consider two types: *binary* and *n-ary* trees. For binary trees, at each node a split is made using a test of the form

$$X_i \leq c$$

for some $c \in \mathcal{X}_i, 1 \leq i \leq n$. Thus, for a binary tree, in each node the associated set $T \subset \mathcal{X}$ is split into the two subsets $T_L = \{x \in T : x_i \leq c\}$ and $T_R = \{x \in T : x_i > c\}$. An example of a univariate binary decision tree is the following:

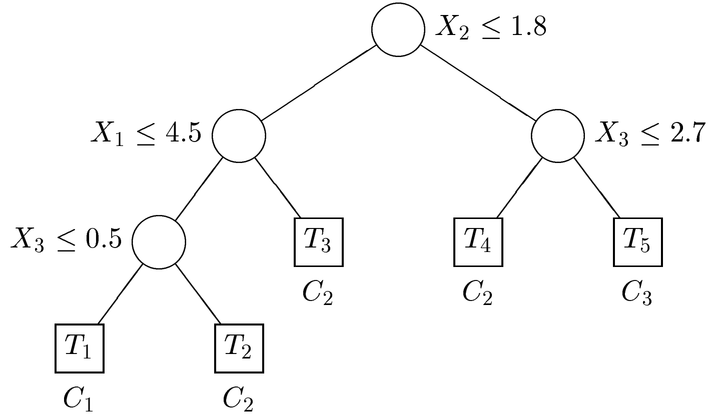

Figure 2: Univariate Binary Decision Tree: Example

This tree splits the input space $\mathcal{X} = \mathbb{R}^3$ into the five regions

$$\begin{aligned}
T_1 = &\quad \{x \in \mathbb{R}^3 : x_1 \leq 4.5, x_2 \leq 1.8, x_3 \leq 0.5\} \\
T_2 = &\quad \{x \in \mathbb{R}^3 : x_1 \leq 4.5, x_2 \leq 1.8, x_3 > 0.5\} \\
T_3 = &\quad \{x \in \mathbb{R}^3 : x_1 > 4.5, x_2 \leq 1.8\} \\
T_4 = &\quad \{x \in \mathbb{R}^3 : x_2 > 1.8, x_3 \leq 2.7\} \\
T_5 = &\quad \{x \in \mathbb{R}^3 : x_2 > 1.8, x_3 > 2.7\}
\end{aligned}$$

the first and the last of which are classified as C_1 and C_3 respectively, and the remaining regions as C_2.

For n-ary trees, we need an additional assumption on the properties of the input space $\mathcal{X}$:

Assumption 2 For the input space $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_n$ each $\mathcal{X}_i$ is a *finite* linearly ordered set, for $i = 1, \ldots, n$. Without loss of generality we may assume that for $1 \leq i \leq n$

$$\mathcal{X}_i = \{0, 1, \ldots, n_i\}$$

for some integer n_i.

In each node of an n-ary tree, a split is made of the form

$$X_i = 0, X_i = 1, \ldots, X_i = n_i$$

for some $i \in \{1, \ldots, n\}$. Thus, for an n-ary tree, in each node the associated set $T \subset \mathcal{X}$ is split into $n_i + 1$ subsets $\{x \in T : x_i = 0\}, \{x \in T : x_i = 1\}, \ldots, \{x \in T : x_i = n_i\}$ for some $i \in \{1, \ldots, n\}$. As an example, consider an input space $\mathcal{X}$ with three variables, let $n_1 = 1, n_2 = 3, n_3 = 2$ so that X_1 can have values 0 and 1, X_2 can have values 0,1, 2, 3 and X_3 can have values 0,1,2. The following n-ary decision tree:

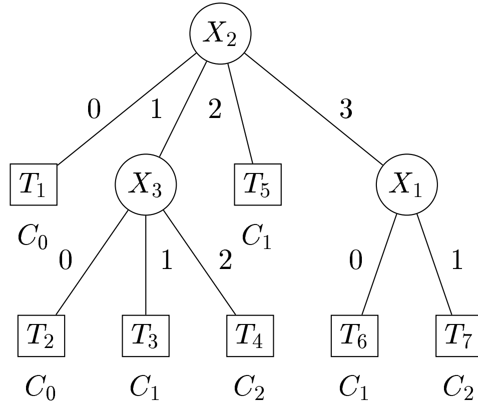

Figure 3: Univariate n-ary Decision Tree: Example

splits the input space $\mathcal{X} = \{0,1\} \times \{0,1,2,3\} \times \{0,1,2\}$ into seven regions $T_1, \dots, T_7$. For instance, region $T_4 = \{(x_1, x_2, x_3) \in \mathcal{X} : x_2 = 1, x_3 = 2\}$. Thus, T_4 consists of the vectors (0,1,2) and (1,1,2). A complete layout of the classification rule induced by the above tree can be seen in the following table:

vector	class	vector	class	leaf
000	C_0	100	C_0	
001	C_0	101	C_0	T_1
002	C_0	102	C_0	
010	C_0	110	C_0	T_2
011	C_1	111	C_1	T_3
012	C_2	112	C_2	T_4
020	C_1	120	C_1	
021	C_1	121	C_1	T_5
022	C_1	122	C_1	
030	C_1			
031	C_1			T_6
032	C_1			
		130	C_2	
		131	C_2	T_7
		132	C_2	

We conclude this section with two lemmas which give a characterization of the subsets associated with the nodes and leaves of a decision tree, for both binary and n-ary decision trees. In the first lemma we use the notation $\overline{\mathcal{X}}$ in the following sense: if $\mathcal{X}_i$ is a linearly ordered set we can always, if needed, add a minimal and a maximal element to get $\overline{\mathcal{X}}_i$. For instance, if $\mathcal{X}_i = \mathbb{R}$, then $\overline{\mathcal{X}_i} = \overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty\} \cup \{-\infty\}$. Next, with $\overline{\mathcal{X}}$ we mean $\overline{\mathcal{X}} = \overline{\mathcal{X}}_1 \times \overline{\mathcal{X}}_2 \times \ldots \times \overline{\mathcal{X}}_n$.

Lemma 3 *If $\mathcal{X}$ is an input space that satisfies Assumption 1 and $\mathcal{T}$ is a univariate binary decision tree on $\mathcal{X}$, then if $T \subset \mathcal{X}$ is the subset associated with an arbitrary node or leaf*

of $\mathcal{T}$,

$$T = \{x \in \mathcal{X} : a < x \leq b\} \tag{3}$$

for some $a, b \in \overline{\mathcal{X}}$ with $a < b$.

Proof: Let $\min_i$ (and $\max_i$) be the minimal (respectively maximal) element of $\overline{\mathcal{X}}$ for $i = 1, \ldots, n$. If T is associated with the root of tree $\mathcal{T}$, then T has the form (3) with $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$, $a_i = \min_i$ and $b_i = \max_i$, for $i = 1, \ldots, n$. Let us next assume that T for some node in the tree has form (3). We will then show that when T is split into $T_L = \{x \in T : x_i \leq c\}$ and $T_R = \{x \in T : x_i > c\}$ for some $i \in \{1, \ldots, n\}$ and $c \in \mathcal{X}_i$, also T_L and T_R have form (3). To prove this we first note that $c \leq a_i$ and $c \geq b_i$ are impossible, since either T_L or T_R would then be empty. So we may assume $a_i < c < b_i$. Now it is easy to see that $T_L = \{x \in \mathcal{X} : a < x \leq b'\}$ with $b'_j = b_j$ for $j \neq i$ and $b'_i = c$. By the same token $T_R = \{x \in \mathcal{X} : a' < x \leq b\}$ with $a'_j = a_j$ for $j \neq i$ and $a'_i = c$. Because each subset T associated with a node or leaf arises during the process of constructing the tree $\mathcal{T}$ as a T_L or a T_R, all must have form (3). □

Corollary 1 If $\mathcal{X}$ is an input space that satisfies both Assumption 1 and Assumption 2, then any subset T associated with a univariate binary decision tree $\mathcal{T}$ on $\mathcal{X}$ will satisfy

$$T = \{x \in \mathcal{X} : a \leq x \leq b\}$$

for some $a, b \in \mathcal{X}$, with $a \leq b$. As an abbreviation we will use the notation $T = [a, b]$ for a set of this form.

Lemma 4 *If $\mathcal{X}$ is an input space that satisfies the Assumptions 1 and 2, and $\mathcal{T}$ is a univariate n-ary decision tree on $\mathcal{X}$, then if $T \subset \mathcal{X}$ is any subset associated with a node or leaf of $\mathcal{T}$,*

$$T = \{(x_1, \ldots, x_n) \in \mathcal{X} : x_i = k_i \text{ for all } i \in I\} \tag{4}$$

for some $I \subset \{1, \ldots, n\}$. Here k_i are integers with $0 \leq k_i \leq n_i, i \in I$.

Proof: This follows directly from the way in which a univariate n-ary decision tree is constructed. □

For any node, the set I is the set of indices that have been encountered om the path from the root of the tree to that particular node.

Corollary 2 If $\mathcal{X}$ is an input space that satisfies both Assumption 1 and Assumption 2, then any subset T associated with a univariate n-ary decision tree $\mathcal{T}$ on $\mathcal{X}$ will also satisfy the same equation

$$T = \{x \in \mathcal{X} : a \leq x \leq b\}$$

for some $a, b \in \mathcal{X}$, with $a \leq b$. In this case, $a_i = b_i$ for all $i \in I$, and $a_i < b_i$ for $i \notin I$.

In the next section we show how to check whether a decision tree which we have constructed for a monotone classification problem, is itself also monotone or not.

2.6 Testing the Monotonicity of a Decision Tree

Note to the July 1997 edition: the text of this subsection will be published in the Supplement to this Technical report.

3 Induction of Monotone Decision Trees: Local Algorithms

We shall now show how we can generate from a data set $\mathcal{D}$ a decision tree $\mathcal{T}$. This process is also called *inducing* a decision tree $\mathcal{T}$ from a dataset $\mathcal{D}$. An algorithm for the induction of a decision tree $\mathcal{T}$ from a dataset $\mathcal{D}$ contains the following ingredients:

- a *splitting rule* $\mathcal{S}$: defines the way to generate a split in each node,
- a *stopping rule* $\mathcal{H}$: determines when to stop splitting and form a leaf,
- a *labeling rule* $\mathcal{L}$: assigns a class label to a leaf when it is decided to create one.

More precisely, a splitting rule is a function $\mathcal{S}$ that on the basis of a dataset $\mathcal{D}$ splits a subset $T \subset \mathcal{X}$ into a number of disjoint nonempty subsets of T. Thus,

$$\mathcal{S}(T, \mathcal{D}) = (T_1, \ldots, T_n)$$

with all T_i nonempty, $\bigcup T_i = T$ and $T_i \cap T_j = \emptyset$ for $i \neq j$. In the same vein, a stopping rule $\mathcal{H}$ is a Boolean function that returns **true** if according to dataset $\mathcal{D}$ we must stop splitting at subset T:

$$\mathcal{H}(T, \mathcal{D}) = \begin{cases} \textbf{true} & \text{if } T \text{ should not be split any further, according to } \mathcal{D}, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

In the first case the associated node of the tree becomes a leaf. Finally, a labeling rule determines which class must be associated with subset T, of course again on the basis of dataset $\mathcal{D}$:

$$\mathcal{L}(T, \mathcal{D}) = c$$

with $T \subset \mathcal{X}, c \in \mathcal{C}$. Notice, that all three functions $\mathcal{S}, \mathcal{H}$ and $\mathcal{L}$ depend on the argument $\mathcal{D}$.

If $\mathcal{S}, \mathcal{H}$ and $\mathcal{L}$ have been specified, then an *induction algorithm* according to these rules can be recursively described as in Figure 4.

```
tree(𝒳, 𝒟₀):
    split(𝒳, 𝒟₀)

split(T, 𝒟):
    if ℋ(T, 𝒟) then
        assign class label ℒ(T, 𝒟) to leaf T
    else
        begin
            (T₁, ... , Tₙ) := 𝒮(T, 𝒟);
            for  i := 1 to n do
                begin
                    𝒟_{Tᵢ} := update(𝒟, Tᵢ);
                    split(Tᵢ, 𝒟_{Tᵢ})
                end
        end
```

Figure 4: Monotone Tree Induction Algorithm: Local variant

In this algorithm outline there is one aspect that we have not mentioned yet: the *update rule*. In the algorithms we use, we shall allow the dataset to be updated at various moments during tree generation. During this process of updating we will incorporate in the dataset knowledge that is needed to guarantee the monotonicity of the resulting tree. In particular, we will consider two general kinds of update rules. In the first place, we will allow each node that is generated to have its own dataset, that is derived from the original dataset in a prescribed manner. We will call algorithms that use this technique algorithms with *local datasets* or *local algorithms*. In this case, the update rule will tell how to transform a dataset belonging to a parent node into the datasets for all the needed child nodes. The algorithm of Figure 4 is of this type: each child node T_i of parent node T gets its own dataset $\mathcal{D}_{T_i}$ derived from the parent dataset $\mathcal{D}$. In general, with these monotone decision tree algorithms, the union of the datasets of the child nodes will be larger than the dataset of the parent node. This is a contrast with classical, non-monotone trees, where the union of the child datasets is generally equal to the parent dataset.

Alternatively, during the whole process of tree generation, we use only one dataset, the *global dataset*. However, we will allow this global dataset to be adjusted during tree generation, to incorporate new information that is needed. Thus, a *global algorithm* will include a line such as $\mathcal{D}$:=update($\mathcal{D}, T$), which will update the global dataset $\mathcal{D}$ when a new node T is formed. The global variant of the tree induction algorithm outline is given in Figure 5:

tree($\mathcal{X}, \mathcal{D}_0$):
  split($\mathcal{X}, \mathcal{D}_0$)

split(T, **var** $\mathcal{D}$):
  $\mathcal{D}$:= update($\mathcal{D}, T$);
  if $\mathcal{H}(T, \mathcal{D})$ **then**
    assign class label $\mathcal{L}(T, \mathcal{D})$ to leaf T
  else
    begin
      $(T_1, \ldots, T_n) := \mathcal{S}(T, \mathcal{D})$;
      for $i := 1$ **to** n **do** split($T_i, \mathcal{D}$)
    end

Figure 5: Monotone Tree Induction Algorithm: Global variant

Note that $\mathcal{D}$ must now be passed to the split procedure as a variable parameter, since $\mathcal{D}$ is updated during execution of the procedure.

In this section we will present local algorithms for the induction of binary and n-ary decision trees. We will start with the n-ary case, since it is surprisingly easier than the binary case. The algorithms we derive in this section, although they produce guaranteedly monotone trees, will have the following drawback: they are implementations either of the minimal or of the maximal classification rule for a given monotone classification problem. In practice, these trees tend to be much larger than is needed. The global algorithms of the subsequent section do not suffer this drawback: the monotone trees produced by these global algorithms tend to be much smaller in size than their local counterparts.

3.1 Projection and Interval Datasets

In this subsection we will describe the way in which a dataset can be updated when a split is made during tree generation. First of all let us motivate the reader why this is not a trivial problem. It might be thought that to update $\mathcal{D} = (D, \lambda)$ when we move from node T to node $T_i \subset T$, we should just include all datapoints that belong to T_i and nothing more:

$$\mathcal{D}_i = (D_i, \lambda_i) \text{ with } D_i = D \cap T_i, \lambda_i = \lambda | D_i. \tag{5}$$

Why this is not enough, we will see in the next example. Let dataset $\mathcal{D} = (D, \lambda)$ be

001	0
002	1
112	2
202	2
212	3

Suppose, we take the first attribute as splitting variable, so that with the naive update rule (5) we would get the following three datasets $\mathcal{D}_0, \mathcal{D}_1$ and $\mathcal{D}_2$:

001	0
002	1

112	2

202	2
212	3

for $T_i = \{(i, x_2, x_3) : 0 \leq x_2, x_3 \leq 2\}, i = 0, 1, 2$. So $\mathcal{D}_1$ would contain just one element, suggesting this single piece of information is sufficient for the classification of all elements of T_1. However, this is clearly not so, for the class of $(1, 0, 2) \in T_1$ must be at least 1, since $(1, 0, 2) > (0, 0, 2)$. Thus, to classify the elements of T_1 properly we also need the information contained in data elements that do not belong to T_1.

How should we incorporate the information contained in a group of subdatasets into the subdataset at hand? In the course of this report we will have two answers to this question. The first answer is *projection* and the other answer is *cornering*. What we mean with this last concept, we will explain in Section 4. We will now show how the datapoints of the outside subsets can be projected on the subset at hand to give an extended updated dataset $\mathcal{D}_i$ that will do the needed work properly.

In the above example the projections of the data elements of T_0 and T_2 onto T_1 are 101, 102 and 112. Applying the monotonicity rule 1 to the original dataset $\mathcal{D}$ we see that $0 \leq \hat{\lambda}(101) \leq 2, 1 \leq \hat{\lambda}(102) \leq 2$ and $2 \leq \hat{\lambda}(112) \leq 2$, for any monotone extension $\hat{\lambda}$ of λ. Thus we see that the updated dataset $\mathcal{D}_1$ does not assign a fixed class to each data element, but an interval of admissible classes:

101	0,2
102	1,2
112	2,2

meaning that the class of data element 101 must be at least 0 and at most 2, etc. It appears that with these projected datasets we must work with *intervals of classes* rather than with single classes assigned to each data element. We will formalize these ideas as follows.

Definition 5 If $D \subset \mathcal{X}$ is any subset of the input space $\mathcal{X}$, then an *interval class labeling* $\overline{\lambda}$ of D is a function

$$\overline{\lambda} : D \to [\mathcal{C}]$$

where $[\mathcal{C}]$ is the set of intervals, based on elements of $\mathcal{C}$.

Remark Thus, for $x \in D$, $\overline{\lambda}(x)$ is an interval $[\overline{\lambda}_\ell(x), \overline{\lambda}_r(x)]$ with $\overline{\lambda}_\ell(x), \overline{\lambda}_r(x) \in \mathcal{C}$. Note that $[\mathcal{C}]$ is partially ordered by the order relation

$$[c, d] \leq [c', d'] \Leftrightarrow c \leq c' \text{ and } d \leq d'.$$

Definition 6 An *interval dataset* $\overline{\mathcal{D}}$ is a pair $(D, \overline{\lambda})$ where D is a finite subset of the input space $\mathcal{X}$ and $\overline{\lambda} : D \to [\mathcal{C}]$ is an interval class labeling of D.

Definition 7 An interval dataset $\overline{\mathcal{D}} = (D, \overline{\lambda})$ is *monotone* if

$$x \leq y \Rightarrow \overline{\lambda}(x) \leq \overline{\lambda}(y)$$

for all points $x, y \in D$.

Remark From this definition and the definition of the order relation on intervals it follows that an interval dataset $\overline{\mathcal{D}} = (D, \overline{\lambda})$ is monotone iff both $\overline{\lambda}_\ell$ and $\overline{\lambda}_r$ are monotone functions.

Definition 8 An *extension* $\hat{\lambda}$ of an interval dataset $\overline{\mathcal{D}} = (D, \overline{\lambda})$ is a function

$$\hat{\lambda} : \mathcal{X} \to \mathcal{C}$$

such that for all $x \in D$

$$\overline{\lambda}_\ell(x) \leq \hat{\lambda}(x) \leq \overline{\lambda}_r(x).$$

As before, $\Lambda(\overline{\mathcal{D}})$ will denote the set of all monotone extensions of dataset $\overline{\mathcal{D}}$.

Lemma 1 of subsection 2.2 also holds for interval datasets if we define the minimal and maximal extension of an interval dataset as follows:

Definition 9 If $\overline{\mathcal{D}} = (D, \overline{\lambda})$ is a monotone interval dataset, we define $\lambda^{\overline{\mathcal{D}}}_{\text{min}} : \mathcal{X} \to \mathcal{C}$ and $\lambda^{\overline{\mathcal{D}}}_{\text{max}} : \mathcal{X} \to \mathcal{C}$ as follows: for all $x \in \mathcal{X}$

$$\lambda^{\overline{\mathcal{D}}}_{\text{min}}(x) = \begin{cases} \max\{\overline{\lambda}_\ell(y) : y \in D \cap \downarrow x\} & \text{if } x \in \uparrow D \\ c_{\text{min}} & \text{otherwise} \end{cases}$$

and

$$\lambda^{\overline{\mathcal{D}}}_{\text{max}}(x) = \begin{cases} \min\{\overline{\lambda}_r(y) : y \in D \cap \uparrow x\} & \text{if } x \in \downarrow D \\ c_{\text{max}} & \text{otherwise.} \end{cases}$$

With this definition, the following extension of Lemma 1 to monotone interval datasets holds:

Lemma 5 *If $\overline{\mathcal{D}} = (D, \overline{\lambda})$ is a monotone interval dataset, then for the functions $\lambda^{\overline{\mathcal{D}}}_{\text{min}}$ and $\lambda^{\overline{\mathcal{D}}}_{\text{max}}$ the following statements hold:*

(i) $\lambda^{\overline{\mathcal{D}}}_{\text{min}}, \lambda^{\overline{\mathcal{D}}}_{\text{max}} \in \Lambda(\overline{\mathcal{D}})$

(ii) $\Lambda(\overline{\mathcal{D}}) = \{\hat{\lambda} : \lambda_{\min}^{\overline{\mathcal{D}}} \leq \hat{\lambda} \leq \lambda_{\max}^{\overline{\mathcal{D}}}$ and $\hat{\lambda}$ monotone$\}$.

Proof: The proof can almost be copied from the proof of Lemma 1 and it is left to the reader. □

Remark 3.1 Note that Definition 9 is actually a special case of Definition 3, if we take as interval class labeling $\overline{\lambda} : \overline{\lambda}_\ell(x) = \overline{\lambda}_r(x) = \lambda(x)$, for $x \in D$. With this choice of $\overline{\lambda}$, Lemma 5 also becomes a corollary of Lemma 1.

3.2 Redundancy for interval datasets

Note to the July 1997 edition: the text of this subsection will be published in the Supplement to this Technical report.

3.3 Projection: n-ary splits

As noted in subsection 2.5, an n-ary decision tree is formed by successive splits using tests t of the form

$$t = \{X_i = a\} \tag{6}$$

Thus, if we move from node T to node $T_a \subset T$ using a test of the form (6), T_a will be defined by $T_a = \{x \in T : x_i = a\}$. The resemblance of this process to the projection of a space $\mathcal{X}$ to a subspace $\mathcal{X}_t = \{x \in \mathcal{X} : x$ satisfies $t\}$ motivates the following definitions.

Definition 10 If t is a test of the form (6) for some $i \in \{1, \ldots, n\}, a \in \{0, \ldots, n_i\}$, and $D \subset \mathcal{X}$ is any subset of the input space $\mathcal{X}$, then the *projection* $\pi_t(D)$ of D is defined as

$$\pi_t(D) = \{(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n) : (x_1, \ldots, x_n) \in D\}.$$

Definition 11 If $\overline{\mathcal{D}} = (D, \overline{\lambda})$ is a monotone interval dataset and t is a test of the form (6) for some $i \in \{1, \ldots, n\}$, $a \in \{0, \ldots, n_i\}$, then the *projection* $\pi_t(\overline{\mathcal{D}})$ of this interval dataset $\overline{\mathcal{D}}$ is an interval dataset $\overline{\mathcal{D}}' = (D', \lambda')$ with

$$\begin{cases} D' = \pi_t(D) \\ \overline{\lambda}'_\ell(x) = \lambda_{\min}^{\overline{\mathcal{D}}}(x) & \text{for all } x \in D' \\ \overline{\lambda}'_r(x) = \lambda_{\max}^{\overline{\mathcal{D}}}(x) & \text{for all } x \in D'. \end{cases}$$

Remark 3.2 For points $x \in D'$ with $x \in D$ (i.e. points that were already in the subspace before projection), we have automatically: $\overline{\lambda}'_\ell(x) = \overline{\lambda}_\ell(x)$ and $\overline{\lambda}'_r(x) = \overline{\lambda}_r(x)$, because of the monotonicity of dataset $\overline{\mathcal{D}}$.

For example, take the dataset of the beginning of Section 3.1 written as an interval dataset $\overline{\mathcal{D}}$:

001	0,0
002	1,1
112	2,2
202	2,2
212	3,3

If the test $t = \{X_1 = 1\}$, the projected dataset $\pi_t(\overline{\mathcal{D}})$ becomes

101	0,2
102	1,2
112	2,2

In fact, $\pi_t(\{001, 002, 112, 202, 212\}) = \{101, 102, 112\}$ and $\lambda^{\overline{\mathcal{D}}}_{\min}(101) = 0$, since 001 is the only element in D smaller than 101 and $\overline{\lambda}_\ell(001) = 0$. In the same vein, $\lambda^{\overline{\mathcal{D}}}_{\max}(101) = 2$, since the smallest element in D larger than 101 is 112, which has $\overline{\lambda}_r(112) = 2$. Proceeding in the same manner, one finds the interval of the data element 102. The interval of 112 does not have to be calculated, since it is also a member of $\overline{\mathcal{D}}$, and thus keeps its interval.

Now, let $\Lambda(\overline{\mathcal{D}})$ be the set of monotone extensions $\hat{\lambda} : \mathcal{X} \to \mathcal{C}$ of dataset $\overline{\mathcal{D}}$. Let $\pi_t(\Lambda(\overline{\mathcal{D}}))$ be the set of all restrictions $\hat{\lambda}|\mathcal{X}_t$ to the subspace $\mathcal{X}_t$ of functions $\hat{\lambda} \in \Lambda(\overline{\mathcal{D}})$. Furthermore, let $\Lambda(\pi_t(\overline{\mathcal{D}}))$ be the set of all monotone extensions of the projected dataset $\pi_t(\overline{\mathcal{D}})$. Then we have the following projection theorem:

Theorem 1 *If $\overline{\mathcal{D}} = (D, \overline{\lambda})$ is a monotone interval dataset, t is a test of the form (6) for some $a \in 1, \dots, n_i, i \in 1, \dots, n$, and $\pi_t(\overline{\mathcal{D}}) = (D', \overline{\lambda}')$ as in Definition 9 is its projection, then*

(i) *for $x \in D \cap D' : \overline{\lambda}'_\ell(x) = \overline{\lambda}_\ell(x)$ and $\overline{\lambda}'_r(x) = \overline{\lambda}_r(x)$*

(ii) *$\pi_t(\overline{\mathcal{D}})$ is a monotone interval dataset*

(iii) *$\Lambda(\pi_t(\overline{\mathcal{D}})) = \pi_t(\Lambda(\overline{\mathcal{D}}))$.*

Proof: Part (i). If $x \in D$, then $\overline{\lambda}'_\ell(x) = \lambda^{\overline{\mathcal{D}}}_{\min}(x) = \overline{\lambda}_\ell(x)$ and $\overline{\lambda}'_r(x) = \lambda^{\overline{\mathcal{D}}}_{\max}(x) = \overline{\lambda}_r(x)$ see the Remark following Definition 11.

Part (ii). Both $\overline{\lambda}'_\ell$ and $\overline{\lambda}'_r$ are restrictions to D' of $\lambda^{\overline{\mathcal{D}}}_{\min}$ and $\lambda^{\overline{\mathcal{D}}}_{\max}$. These last two are monotone according to Lemma 5. So $\overline{\lambda}'_\ell$ and $\overline{\lambda}'_r$ are monotone as well, and the assertion to be proved follows from the remark following Definition 7.

Part (iii). First we prove the $\supset$ part of the assertion. Let $\hat{\lambda} : X_t \to \mathcal{C}$ be a restriction to X_t of a function $\hat{\lambda} \in \Lambda(\overline{\mathcal{D}})$. Then $\overline{\lambda}$ is monotone on X_t and $\lambda^{\overline{\mathcal{D}}}_{\min} \le \hat{\lambda} \le \lambda^{\overline{\mathcal{D}}}_{\max}$. For $x \in D'$ we have $\overline{\lambda}'_\ell(x) = \lambda^{\overline{\mathcal{D}}}_{\min}(x)$ and $\overline{\lambda}'_r(x) = \lambda^{\overline{\mathcal{D}}}_{\max}(x)$, so on D' we have $\overline{\lambda}'_\ell \le \hat{\lambda} \le \overline{\lambda}_r$. Thus $\hat{\lambda}$ is a monotone extension of $\pi_t(\overline{\mathcal{D}})$.

Next, we prove the $\subset$ part of the assertion. Let $\hat{\lambda} : X_t \to \mathcal{C}$ be a monotone extension of $\pi_t(\overline{\mathcal{D}}) = (D', \overline{\lambda}')$; of course, we then have

$$\overline{\lambda}'_\ell(x) \le \hat{\lambda}(x) \le \overline{\lambda}'_r(x), \text{ for } x \in D'. \tag{7}$$

Now, we must extend $\hat{\lambda}$ from $\mathcal{X}_t$ to the whole space $\mathcal{X}$. To accomplish this, we take the following detour. We define a new non-interval dataset $\tilde{\mathcal{D}} = (\tilde{D}, \tilde{\lambda})$. Next, we show that this dataset $\tilde{\mathcal{D}}$ is monotone, and that $\tilde{\lambda} = \hat{\lambda}$ on X_t. So any monotone extension of $\tilde{\mathcal{D}}$ will be an extension of $\hat{\lambda}$ to the whole space $\mathcal{X}$. Such an extension will then be proved to be a member of $\Lambda(\overline{\mathcal{D}})$, which will finish the proof.

Let $\tilde{\mathcal{D}} = (\tilde{D}, \tilde{\lambda})$ be defined as follows: $\tilde{D} = \mathcal{X}_t \cup D$ and $\tilde{\lambda} : \tilde{D} \to \mathcal{C}$ is defined as

$$\tilde{\lambda}(x) = \begin{cases} \overline{\lambda}_\ell(x) & \text{for } x \in D \setminus \mathcal{X}_t \text{ and } x_i < a \\ \hat{\lambda}(x) & \text{for } x \in \mathcal{X}_t \\ \overline{\lambda}_r(x) & \text{for } x \in D \setminus \mathcal{X}_t \text{ and } x_i > a \end{cases}$$

Note that by this definition $\tilde{D}$ is devided into three subsets, which we will call A, B and C respectively. We will now show that $\tilde{\lambda}$ as defined is monotone on $\tilde{D}$. Let x and y be arbitrary elements of $\tilde{D}$, with $x \leq y$. We then have six cases:

1) x and y both in A. In this case $\tilde{\lambda}(x) \leq \tilde{\lambda}(y)$ follows from the monotonicity of $\overline{\lambda}_\ell$.

2) x and y both in B. The monotonicity of $\hat{\lambda}$ does the job.

3) x and y both in C. Since $\overline{\lambda}_r$ is monotone, we are done.

4) Suppose $x \in A$ and $y \in B$. The other way around is impossible since $x \leq y$. Now, let x' be the projection of x on $\mathcal{X}_t$, then we also have $x' \leq y$, since $x_i \leq y_i$, and the other components are implied by $x \leq y$. So we can write

$$\tilde{\lambda}(x) = \overline{\lambda}_\ell(x) \leq \overline{\lambda}'_\ell(x') \leq \hat{\lambda}(x') \leq \hat{\lambda}(y) = \tilde{\lambda}(y).$$

If we number these (in)equalities from (1) to (5), we can say that (1) follows from $x \in A$, (2) follows from the definition of $\lambda^{\overline{\mathcal{D}}}_{\text{min}}$ and the fact that $x \leq x'$, (3) follows from equation (7), (4) from the monotonicity of $\hat{\lambda}$ and (5) from $y \in B$.

5) Suppose $x \in A$ and $y \in C$. Again, the other way around is impossible since $x \leq y$. We then have

$$\tilde{\lambda}(x) = \overline{\lambda}_\ell(x) \leq \overline{\lambda}_r(x) \leq \overline{\lambda}_r(y) = \tilde{\lambda}(y)$$

where (1) follows from $x \in A$, (2) from Definition... of an interval dataset, (3) from the monotonicity of $\overline{\lambda}_r$ on D and (4) from $y \in C$.

6) Suppose $x \in B$ and $y \in C$. Let $y' = \pi_t(y)$, so $y' \in D'$ and $y \leq y'$. Again, we have $x \leq y'$, and we can write

$$\tilde{\lambda}(x) = \hat{\lambda}(x) \leq \hat{\lambda}(y') \leq \overline{\lambda}'_r(y') \leq \overline{\lambda}_r(y) = \tilde{\lambda}(y)$$

with arguments similar to those of 4).

This proves the monotonicity of $\tilde{\lambda}$ on $\tilde{D}$. The definition of $\tilde{\lambda}$ implies, that $\overline{\lambda}_\ell \leq \tilde{\lambda} \leq \overline{\lambda}_r$ on D, so any monotone extension of $\tilde{\mathcal{D}}$ will also be a monotone extension of $\overline{\mathcal{D}}$. This shows that any monotone extension of $\tilde{\mathcal{D}}$ can be taken as extension of $\hat{\lambda}$ to the whole space X. □

Remark 3.3 From part (i) of this Theorem, it follows that for points x, that already belonged to subspace $\mathcal{X}_t$ before projection, the interval $[\overline{\lambda}_\ell(x), \overline{\lambda}_r(x)]$ does not change by projection. From part (ii) it follows that the projected dataset $\pi_t(\overline{\mathcal{D}})$ may in turn serve as the dataset for the construction of a new tree beginning at the new mode, on subspace $\mathcal{X}_t$. From part (iii) it follows, that during projection, no information gets lost, nor that spurious information gets introduced: each solution of the refined problem corresponds with a solution of the whole problem, and vice versa.

A consequence of the commutativity of the projection- and the extension operator, as proved in Theorem 1, is the following

Corollary 3 If $t_1 = \{X_i = a\}$ and $t_2 = \{X_j = b\}$ are two tests of the form (6), then

$$\Lambda(\pi_{t_2}(\pi_{t_1}(\overline{\mathcal{D}}))) = \pi_{t_2}(\Lambda(\pi_{t_1}(\overline{\mathcal{D}}))) = \pi_{t_2}(\pi_{t_1}(\Lambda(\overline{\mathcal{D}}))).$$

In words: the dataset which you get after two (or more) projection steps, has the same solution set as the original dataset, provided these solutions are restricted to the subspace at hand.

We get another consequence of this theorem by comparing the minimal and maximal extension of the original and the projected datasets. If on $\mathcal{X}_t$ the whole set of monotone extensions is the same before and after projection, then also the minimal and maximal extensions of both datasets must be the same:

Corollary 4 If t is a test of the form (6), then on $\mathcal{X}_t$

$$\lambda_{\min}^{\overline{\mathcal{D}}}|\mathcal{X}_t = \lambda_{\min}^{\pi_t(\overline{\mathcal{D}})}$$

and

$$\lambda_{\max}^{\overline{\mathcal{D}}}|\mathcal{X}_t = \lambda_{\max}^{\pi_t(\overline{\mathcal{D}})}.$$

Of course, this also holds for two projection steps $t_1 = \{X_i = a\}$ and $t_2 = \{X_j = b\}$. If $t_1 \wedge t_2$ is the conjunction of the two tests then, on subspace $\mathcal{X}_{t_1 \wedge t_2}$

$$\lambda_{\min}^{\overline{\mathcal{D}}}|\mathcal{X}_{t_1 \wedge t_2} = \lambda_{\min}^{\pi_{t_2}(\pi_{t_1}(\overline{\mathcal{D}}))}$$

and

$$\lambda_{\max}^{\overline{\mathcal{D}}}|\mathcal{X}_{t_1 \wedge t_2} = \lambda_{\max}^{\pi_{t_2}(\pi_{t_1}(\overline{\mathcal{D}}))}.$$

As a direct consequence of these last equations, let us consider the following situation: from dataset $\overline{\mathcal{D}}$ we construct after one projection step t_1, dataset $\pi_{t_1}(\overline{\mathcal{D}})$. If subsequently, we project this dataset one more time using test t_2, it might be questioned which dataset must be referred to for calculating the intervals of this new dataset: $\overline{\mathcal{D}}$ or $\pi_{t_1}(\overline{\mathcal{D}})$? The answer is, of course, that this does not matter: both give the same intervals for $\pi_{t_2}(\pi_{t_1}(\overline{\mathcal{D}}))$. Formally, this could be written down as follows:

$$\pi_{t_1 \wedge t_2}(\overline{\mathcal{D}}) = \pi_{t_2}(\pi_{t_1}(\overline{\mathcal{D}}))$$

where the equality sign means that both the set of datapoints and all associated intervals of the mentioned datasets are the same.

As an illustration of this point, consider again the dataset $\overline{\mathcal{D}}$:

001	0,0
002	1,1
112	2,2
202	2,2
212	3,3

After one projection step with $t_1 = \{X_1 = 1\}$ this dataset becomes $\overline{\mathcal{D}}'$

112	2,2
101	0,2
102	1,2

as we have seen before. After a second projection step with $t_2 = \{X_3 = 0\}$ we get the following dataset $\overline{\mathcal{D}}''$

100	0,2
110	0,2

as can be seen, since $\lambda^{\overline{\mathcal{D}}'}_{\text{min}}(100) = \lambda^{\overline{\mathcal{D}}'}_{\text{min}}(110) = 0$ and $\lambda^{\overline{\mathcal{D}}'}_{\text{max}}(100) = \lambda^{\overline{\mathcal{D}}'}_{\text{max}}(110) = 2$. However, we also have $\lambda^{\overline{\mathcal{D}}}_{\text{min}}(100) = \lambda^{\overline{\mathcal{D}}}_{\text{min}}(110) = 0$ and $\lambda^{\overline{\mathcal{D}}}_{\text{max}}(100) = \lambda^{\overline{\mathcal{D}}}_{\text{min}}(110) = 2$. Thus, the intervals of 100 and 110 can be calculated with either $\overline{\mathcal{D}}'$ or $\overline{\mathcal{D}}$.

3.4 Projection: binary splits

For the induction of binary trees, we need projections π_t based on a test t of the form

$$t = \{X_i \leq a\} \tag{8}$$

or

$$t = \{X_i \geq a\} \tag{9}$$

A subspace corresponding with a test t of this form we shall again call $\mathcal{X}_t$, p.e. $\mathcal{X}_t = \{x = (x_1, \ldots, x_n) \in \mathcal{X} : x_i \leq a\}$, if t is of form (8) or (9).

For projections using tests of this form, we shall use a definition slightly different from the one in the preceding subsection:

Definition 12 If t is a test of the form (8) or (9), we define

$$\pi_t(x) = \begin{cases} x & \text{for } x \in \mathcal{X}_t \\ (x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n) & \text{for } x \notin \mathcal{X}_t. \end{cases}$$

If $D \subset \mathcal{X}$ is any subset of $\mathcal{X}$, then $\pi_t(D) = \{\pi_t(x) : x \in D\}$.

Note that any points outside the subspace are only projected to the edge of the subspace, not to the interior! With this definition of the projection of a subset of $\mathcal{X}$, we can leave the definition of the projection of an interval dataset (Definition 11) unchanged for the binary case. Again, consider the example of dataset $\overline{\mathcal{D}}$

001	0,0
002	1,1
112	2,2
202	2,2
212	3,3

If a split into $t_1 = \{X_1 \leq 0\}$ and $t_2 = \{X_1 \geq 1\}$ is affected, we get the following datasets $\pi_{t_1}(\overline{\mathcal{D}})$ and $\pi_{t_2}(\overline{\mathcal{D}})$:

001	0,0
002	1,1
012	1,2

101	0,2
102	1,2
112	2,2
202	2,2
212	3,3

Note, that the second dataset contains no element 201, although it is a projection of 001 into the set $\{X_1 \geq 1\}$.

We shall now prove that in this case the equivalent of Theorem 1 remains valid.

Theorem 2 *If $\overline{\mathcal{D}} = (D, \overline{\lambda})$ is a monotone interval dataset, and t is a test of the form (8) or (9) for some $a \in \mathcal{X}_i$, $i \in \{1, \ldots, n\}$ and $\pi_t(\overline{\mathcal{D}}) = (D', \overline{\lambda}')$ defined according to Definition..., then we have*

(i) *for $x \in D \cap D' : \overline{\lambda}'(x) = \overline{\lambda}(x)$*

(ii) *$\pi_t(\overline{\mathcal{D}})$ is a monotone interval dataset*

(iii) *$\Lambda(\pi_t(\overline{\mathcal{D}})) = \pi_t(\Lambda(\overline{\mathcal{D}}))$.*

Proof: Part (i). For points $x \in D'$ with $x \in D$ we have $\pi_t(x) = x$, so $\overline{\lambda}'_\ell(x) = \lambda^{\overline{\mathcal{D}}}_{\mathrm{min}}(x) = \overline{\lambda}_\ell(x)$ and $\overline{\lambda}'_r(x) = \lambda^{\overline{\mathcal{D}}}_{\mathrm{max}}(x) = \overline{\lambda}_r(x)$, which proves this part.

Part (ii). Both $\overline{\lambda}'_\ell$ and $\overline{\lambda}'_r$ are monotone on $\mathcal{X}_t$, since on $\mathcal{X}_t$ they are equal to the monotone functions $\lambda^{\overline{\mathcal{D}}}_{\mathrm{min}}$ and $\lambda^{\overline{\mathcal{D}}}_{\mathrm{max}}$ respectively.

Part (iii). The $\supset$ part is exactly equal to the $\supset$ part of Theorem 1. So let us proceed with the $\subset$ part. First, we will give an argument that deals with the case $t = \{X_i \leq a\}$. At the end of the proof we will show how to deal with the case $t = \{X_i \geq a\}$. Let $\hat{\lambda} : \mathcal{X}_t \to \mathcal{C}$ be a monotone extension of $\pi_t(\overline{\mathcal{D}}) = (D', \overline{\lambda}')$, so that

$$\overline{\lambda}'_\ell(x) \leq \hat{\lambda}(x) \leq \overline{\lambda}'_r(x) \text{ for } x \in D'. \tag{10}$$

Our task is to extend $\hat{\lambda}$ to all of $\mathcal{X}$. We shall go about in a way similar to the proof of Theorem 1:

Let $\tilde{\mathcal{D}} = (\tilde{D}, \tilde{\lambda})$ be a non-interval dataset, defined as follows: $\tilde{D} = \mathcal{X}_t \cup D$ and let $\tilde{\lambda} : \tilde{D} \to \mathcal{C}$ be defined as

$$\tilde{\lambda}(x) = \begin{cases} \hat{\lambda}(x) & \text{for } x \in \mathcal{X}_t \\ \overline{\lambda}_r(x) & \text{for } x \in D \setminus \mathcal{X}_t \end{cases}$$

Note that by this definition $\tilde{D}$ is devided into two subsets, which we will call A and B respectively. Note further, that for $x \in B$ we have $x_i > a$. We must now show that $\tilde{\lambda}$ as defined is monotone on $\tilde{D}$. So, let $x, y \in \tilde{D}$ be anyelements from $\tilde{D}$, with $x \leq y$.

Case 1. If x and y are both members of A, or if both are members of B, then clearly $\tilde{\lambda}(x) \leq \tilde{\lambda}(y)$, for both $\hat{\lambda}$ and $\overline{\lambda}$ are monotone.

Case 2. Let $x \in A$ and $y \in B$; the other way around is impossible, since $x \leq y$. Furthermore, let $y' = \pi_t(y)$, so $y' \in D'$, and $y' \leq y$. Then we also have $x \leq y'$, for $x_i \leq a = y'_i$ and all other components components are implied by $x \leq y$. So, we have

$$\tilde{\lambda}(x) = \hat{\lambda}(x) \leq \hat{\lambda}(y') \leq \overline{\lambda}'_r(y') \leq \overline{\lambda}_r(y) = \tilde{\lambda}(y)$$

where the second (in)equality follows from the monotonicity of $\hat{\lambda}$ and the third (in)equality follows from (10). So $\tilde{\lambda}$ is monotone on $\tilde{D}$ in all cases. Now, as an extension of $\hat{\lambda}$ we take an arbitrary monotone extension of dataset $\tilde{\mathcal{D}}$. For such an extension $\hat{\lambda}$ we have $\hat{\lambda} \in \Lambda(\overline{\mathcal{D}})$, so part (iii) has been proven for the case $t = \{X_i \leq a\}$. To cover the case $t = \{X_i \geq a\}$ it is sufficient to define $\tilde{\mathcal{D}} = (\tilde{D}, \tilde{\lambda})$ as follows: $\tilde{D} = \mathcal{X}_t \cup D$ and

$$\tilde{\lambda}(x) = \begin{cases} \overline{\lambda}_\ell(x) & \text{for } x \in D \setminus \mathcal{X}_t \text{ (so: } x_i < a) \\ \hat{\lambda}(x) & \text{for } x \in \mathcal{X}_t. \end{cases}$$

With this choice of $\tilde{\mathcal{D}}$ we can almost duplicate the above proof. We leave this to the reader. □

We end this subsection with two remarks.

Remark 3.4 Although it might seem to be necessary for the case $t = \{X_i \leq a\}$ to use as the projected dataset the set

$$\pi_t(D) = \bigcup_{b \leq a} \pi_{t_b}(D), \text{ with } t_b = \{X_i = b\}$$

it appears from Theorem 2 that all datapoints from $\bigcup_{b<a} \pi_{t_b}(D)$ that were not in D before, are redundant in $\pi_t(\overline{\mathcal{D}})$.

As an example, the data element 201 with interval $[0, 2]$ would be clearly redundant in the dataset $\pi_{t_2}(\overline{\mathcal{D}})$ following Definition 12, as can be seen from the theorem of Section 3.2.

Remark 3.5 Let $|\overline{\mathcal{D}}|$ be the number of datapoints in a dataset $\overline{\mathcal{D}} = (D, \overline{\lambda})$. So $|\overline{\mathcal{D}}| = |D|$. Then from Theorem 2 it follows that for $t = \{X_i \leq a\}$ and $t = \{X_i \geq a\}$

$$|\pi_t(\overline{\mathcal{D}})| \leq |\overline{\mathcal{D}}|.$$

This will be very helpful while building a tree: it means that the deeper we go into the tree, the smaller the associated datasets become.

3.5 Algorithms for the minimal and maximal trees

In this subsection we will propose some complete algorithms for inducing decision trees. We will treat both the binary and the n-ary case, and we will use the technique of projection that was explained in the preceding paragraphs. The trees produced by the proposed algorithms will be implementations of the minimal resp. maximal monotone extensions of a monotone dataset.

We will start with input spaces that satisfy both Assumption 1 and Assumption 2 (Section 2.5). In Section 3.6 we will show, that the same result can be reached with Assumption 1 only.

We will now proceed to specify the update rule, the splitting rule, the stopping rule and the labeling rule of the proposed algorithms. Once these have been specified the algorithm for the induction will be known: see Figure 4 of the beginnig of this section. Note that in this subsection we will only treat algorithms that use local datasets.

We start with the update rule: when a new node T is formed, it must be one of the form (8) or (9) and the new dataset $\overline{\mathcal{D}}_T$ will be formed from the dataset $\overline{\mathcal{D}}$ of the parent node as follows

$$\text{(U):} \qquad \overline{\mathcal{D}}_T = \pi_t(\overline{\mathcal{D}}) \cup \mathcal{M}_T$$

where π_t is defined in Definition... and $\mathcal{M}_T = (M_T, \overline{\lambda})$ with

$$M_T = \{a, b\}$$

and

$$\overline{\lambda}_\ell(a) = \lambda^{\overline{\mathcal{D}}}_{\min}(a), \overline{\lambda}_r(a) = \lambda^{\overline{\mathcal{D}}}_{\max}(a)$$
$$\overline{\lambda}_\ell(b) = \lambda^{\overline{\mathcal{D}}}_{\min}(b), \overline{\lambda}_r(b) = \lambda^{\overline{\mathcal{D}}}_{\max}(b).$$

Here, a and b stem from the fact that T must be of the form

$$T = \{x \in \mathcal{X} : a \leq x \leq b\}.$$

Thus, a is the minimal element of T and b is the maximal element of T.

Next, we consider the splitting rule $\mathcal{S}(T, \overline{\mathcal{D}})$. We assume that each split of a node T will be of the form

$$\text{(S1):} \qquad T = T_L \cup T_R$$

with

$$T_L = \{x \in T : x_i \leq c\}$$

and

$$T_R = \{x \in T : x_i > c\}$$

for some $c \in \mathcal{X}_i$. Note that, because of Assumption 2, T_R can also be written as $T_R = \{x \in T : x_i \geq c'\}$ for some $c' \in \mathcal{X}_i$.

The stopping rule, that will be used, will have the following form: we will stop at node T when for its associated interval dataset $\overline{\mathcal{D}} = (D, \overline{\lambda})$ we have either

$$\text{(H1):} \qquad \forall x, y \in D : \overline{\lambda}_\ell(x) = \overline{\lambda}_\ell(y)$$

or

$$\text{(H2):} \qquad \forall x, y \in D : \overline{\lambda}_r(x) = \overline{\lambda}_r(y)$$

Thus, we will stop in a node when either all left points or all right points of the intervals of the datapoints in this node will be equal. During the course of an algorithm we will use either (H1) or (H2), not both alternatingly.

Next we come to the labeling rule $\lambda(T, \overline{\mathcal{D}})$. This rule will only be fired when either (H1) or (H2) is true, so all leftpoints or all rightpoints of the intervals of all datapoints will be the same. Now we define

$$\text{(L):} \qquad \lambda(T, \overline{\mathcal{D}}) = \begin{cases} \overline{\lambda}_\ell(x) & \text{for any } x \in D, \text{ if (H1) holds} \\ \overline{\lambda}_r(x) & \text{for any } x \in D, \text{ if (H2) holds.} \end{cases}$$

Finally we will have to add the following refinement to the splitting rule $\mathcal{S}(T, \overline{\mathcal{D}})$: when we work with the stopping rule (H1), at each splitting of the form $T_L = \{X_i \leq c\}$, $T_R = \{X_i > c\}$, we will have to pick $c \in \mathcal{X}_i$ such that

$$\text{(S2):} \qquad \exists x, y \in D \text{ with } \overline{\lambda}_\ell(x) \neq \overline{\lambda}_\ell(y), x \in T_L \text{ and } y \in T_R.$$

If we work with stopping rule (H2), at each splitting we will have to pick $c \in \mathcal{X}_i$ such that

$$\text{(S3):} \qquad \exists x, y \in D \text{ with } \overline{\lambda}_r(x) \neq \overline{\lambda}_r(y), x \in T_L \text{ and } y \in T_R.$$

Now, we are in a position to formulate the main theorem of this section.

Theorem 3 *If $\mathcal{X}$ is an input space that satisfies Assumption 1 and 2, if $\mathcal{D} = (D, \lambda)$ is a monotone dataset on $\mathcal{X}$, if the functions update, $\mathcal{S}$, $\mathcal{H}$ and $\mathcal{L}$ satisfy either*

$$\text{(i):} \qquad \text{(U), (S1), (S2), (H1) and (L)}$$

or

$$\text{(ii):} \qquad \text{(U), (S1), (S3), (H2) and (L)}$$

then the algorithm described in Figure 4 of this section will generate a monotone binary decision tree $\mathcal{T}$ with associated class labeling $\lambda_{\mathcal{T}}$. Furthermore we have in case (i)*: $\lambda_{\mathcal{T}} = \lambda^{\mathcal{D}}_{\min}$ and in case* (ii)*: $\lambda_{\mathcal{T}} = \lambda^{\mathcal{D}}_{\max}$, the minimal, respectively maximal monotone extension of $\mathcal{D}$.*

Proof: We prove the theorem for case (i). Case (ii) is similar, and will be left to the reader.
a) First we will prove that for each leaf T where the stopping rule fires, we have

$$\lambda_{\mathcal{T}}(x) = \lambda^{\mathcal{D}}_{\min}(x), \text{ for all } x \in T. \tag{11}$$

By definition, (10) holds for $x \in D_T$, viz. on D_T $\lambda_{\mathcal{T}}(x)$ is equal to $\overline{\lambda}_\ell(x)$ which is equal to $\lambda^{\mathcal{D}}_{\min}(x)$, see the Projection Theorem and the definition of $\pi_t(\mathcal{D})$ in Definition 11. Now, let c_T be the value of $\lambda_{\mathcal{T}}(x)$ for $x \in T$. That this value is constant follows from (4) and (6). We must now show that

$$\lambda^{\mathcal{D}}_{\min}(x) = c_T, \text{ for all } x \in T. \tag{12}$$

If a and b are resp. the minimal and the maximal element of T, then $a, b \in D_T$ because of (2), so we also have

$$\lambda^{\mathcal{D}}_{\min}(a) = c_T \text{ and } \lambda^{\mathcal{D}}_{\min}(b) = c_T.$$

But, since $\lambda^{\mathcal{D}}_{\min}$ is monotone, we have for all $x \in T = \{x : a \leq x \leq b\}$

$$c_T = \lambda^{\mathcal{D}}_{\min}(a) \leq \lambda^{\mathcal{D}}_{\min}(x) \leq \lambda^{\mathcal{D}}_{\min}(b) = c_T,$$

which proves (12) and consequently (11).
b) Since (11) holds for all leaves, the assertion about $\lambda_{\mathcal{T}}$ in the theorem is now proved as well. The monotonicity of $\mathcal{T}$ now follows from the monotonicity of $\lambda^{\mathcal{D}}_{\min}$.
c) Finally, we must show that the tree is finite. This follows from the fact that for each new T_L and T_R, we always have $|T_L| < |T|$ and $|T_R| < |T|$, see Corollary 2 of the last subsection. Eventually, the size of the dataset with a node must diminish until it consists of a single element. Then (H1) automatically holds, and the stopping rule fires. Thus the tree never becomes deeper than the number of elements in the original dataset. □

As an example of the operation of the above algorithm, let us look at the bank loan dataset $\mathcal{D}$ of Table 1, which is first transformed into the following interval dataset $\overline{\mathcal{D}}$:

001	0,0
002	1,1
112	2,2
202	2,2
212	3,3

Suppose for the first split we use the test $\{X_1 \leq 0\}$. As shown in the example under Definition 12, we get the following projected datasets

001	0,0
002	1,1
012	1,2

101	0,2
102	1,2
112	2,2
202	2,2
212	3,3

However, by the update rule these datasets are supplemented with the minimal and maximal elements of their nodes: 000 with interval $[0,0]$ and 022 with interval $[1,3]$ are added to the left dataset; 100 with interval $[0,2]$ and 222 with interval $[3,3]$ are added to the right dataset, ending up with

000	0,0
001	0,0
002	1,1
012	1,2
022	1,3

100	0,2
101	0,2
102	1,2
112	2,2
202	2,2
212	3,3
222	3,3

Proceeding with the left dataset we see that the left endpoints of the intervals are not all equal so we can not stop yet. Thus, the node must be split again. Suppose we split it using the test $\{X_3 \leq 1\}$ to give the following projected datasets:

000	0,0
001	0,0
011	0,2
021	**0,3**

002	1,1
012	1,2
022	1,3

To the left dataset the maximal element 021 with interval $[0,3]$ is added by the update rule. We now see, that all the left endpoints of the intervals in the left dataset are equal to 0. Thus, we can stop, make a leaf, and assign class 0 to it. Proceeding in this way, we end up with the decision tree of Figure 6. When we use stopping rule (H2) we get the decision tree of Figure 7.

Having shown in Theorem 3 how binary decision trees can be constructed, we now turn our attention to n-ary trees. It turns out that the n-ary case is only a slight variation of the binary case: only the splitting rule must be changed, the other rules can remain what they are. In fact, the splitting rule $\mathcal{S}(T,\overline{\mathcal{D}})$ must have the following form. At each node, the associated subset T is split into n_i+1 subsets, if the variable X_i is used as a splitting variable with values $0,1,\ldots,n_i$. Thus, the splitting rule will have the form

$$(S1*): \qquad T = T_0 \cup T_1 \cup \ldots \cup T_{n_i}$$

where

$$T_j = \{x \in T : x_i = j\}$$

for each $j \in \{0,1,\ldots,n_i\}$. In addition, the variable X_i must be chosen such that

$$(S2*): \qquad \exists x,y \in D : x_i \neq y_i$$

whether or not stopping rule (H1) or (H2) is used.

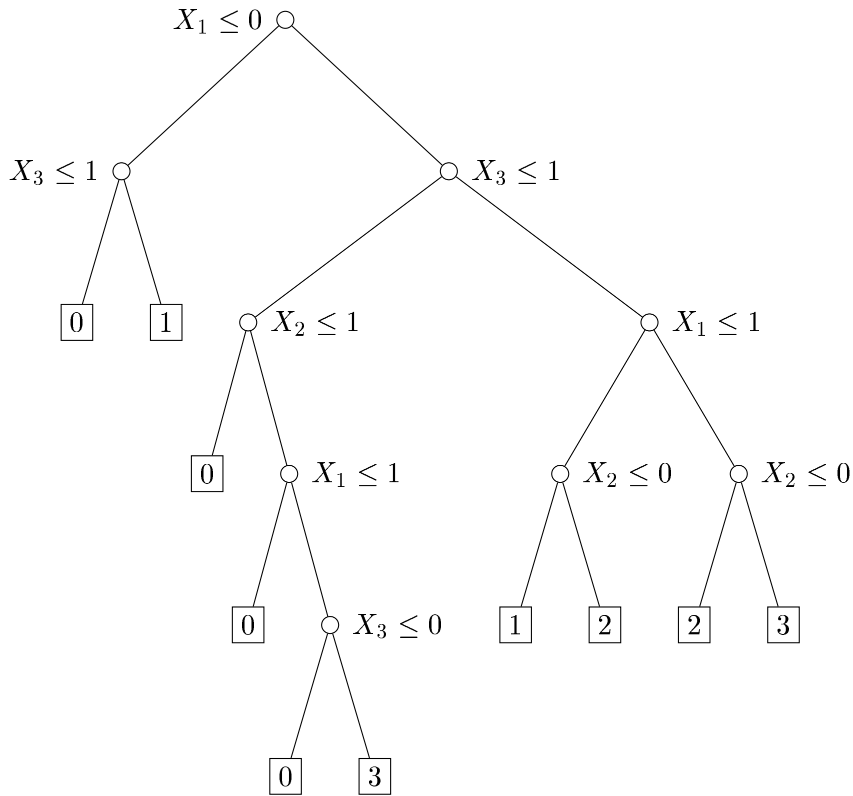

Figure 6: Decision Tree for the Minimal Extension: binary case

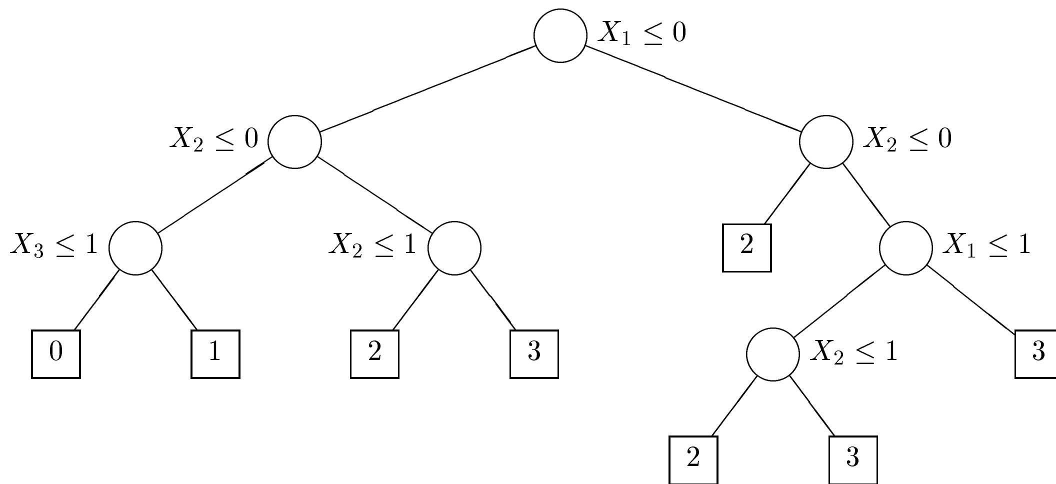

Figure 7: Decision Tree for the Maximal Extension: binary case

Using these changes in the splitting rule, we can now formulate the analog of Theorem 3 for the n-ary case.

Theorem 4 *If $\mathcal{X}$ is an input space that satisfies Assumption 1 and 2, if $\mathcal{D} = (D, \lambda)$ is a monotone dataset on $\mathcal{X}$, if the functions update, $\mathcal{S}$, $\mathcal{H}$ and $\mathcal{L}$ satisfy either*

(i): (U), (S1*), (S2*), (H1) and (L)

or

(ii): (U), (S1*), (S2*), (H2) and (L)

then the algorithm described in Figure 4 of this section will generate a monotone n-ary decision tree $\mathcal{T}$ with associated class labeling $\lambda_{\mathcal{T}}$. Furthermore we have in case (i)*: $\lambda_{\mathcal{T}} = \lambda_{\min}^{\mathcal{D}}$ and in case* (ii)*: $\lambda_{\mathcal{T}} = \lambda_{\max}^{\mathcal{D}}$, the minimal, respectively maximal monotone extension of $\mathcal{D}$.*

Proof: The proof remains the same, due to the fact that a set T associated with a node in an n-ary tree, just as in the binary case, has the form

$$T = \{x \in \mathcal{X} : a \leq x \leq b\}$$

for some $a, b \in \mathcal{X}$, see Corollary 2 of Section 2.5. Thus, the proof of Theorem 3 will work here as well. □

As an example of the operation of the described algorithms, let us look at our running example dataset $\mathcal{D}$

001	0
002	1
112	2
202	2
212	3

which becomes after the projection step $\{X_1 = 1\}$

112	2,2
101	0,2
102	1,2

However, the update rule adds two elements to this last set, the minimal element 100 with interval $[0, 2]$ and the maximal element 122 with interval $[2, 3]$, yielding

100	0,2
101	0,2
102	1,2
112	2,2
122	2,3

Now, even if we use stopping rule (H2) we cannot stop yet, since not all right-ends of the intervals are equal.

Running the algorithm of Theorem 4 for stopping rule (H1) gives the decision tree of Figure 8

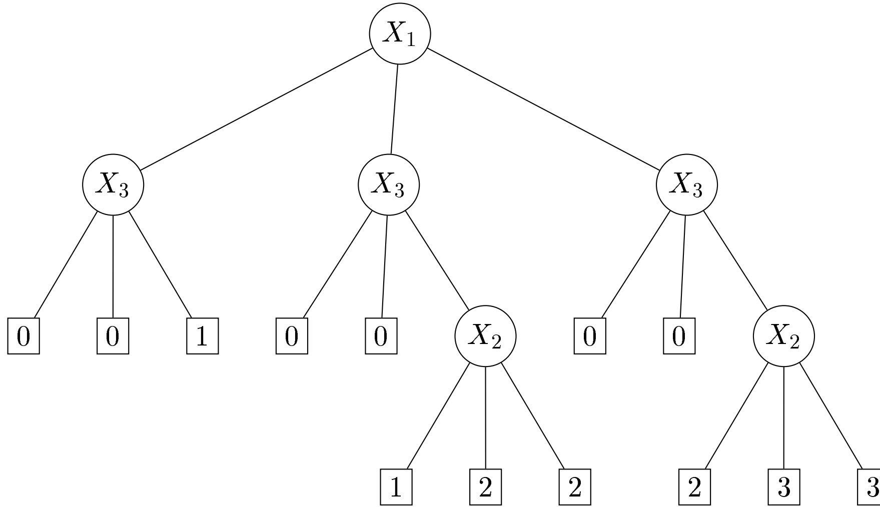

Figure 8: Decision Tree for the Minimal Extension: n-ary case

and running it with stopping rule (H2) yields the tree of Figure 9.

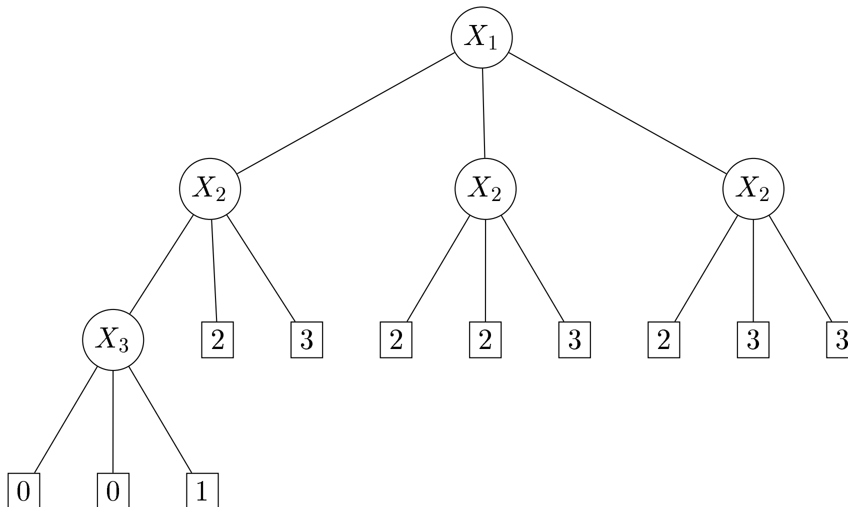

Figure 9: Decision Tree for the Maximal Extension: n-ary case

We close this section with a number of remarks.

Remark 3.6 Note that these theorems actually prove a whole class of algorithms to be correct: the requirements set by the theorem to the splitting rule are quite general. Almost nothing is said in the requirements about how to select the attribute X_i. Obvious

candidates for attribute-selection are the well-known impurity measures like entropy, Gini or the twoing rule, see Breiman *et al.* [4].

Remark 3.7 It is possible to simplify the algorithms described in Theorem 3 and 4 in the following way. One can leave out the $\mathcal{M}_T$-part in the update rule (U), provided the original dataset $\mathcal{D}$ is supplemented with two data elements, viz. $x_{\min}$, the minimal element of the whole input space $\mathcal{X}$, and $x_{\max}$, the maximal element of $\mathcal{X}$. Of course, one assigns to these data elements the intervals $[\lambda^{\mathcal{D}}_{\min}(x_{\min}), \lambda^{\mathcal{D}}_{\max}(x_{\min})]$ and $[\lambda^{\mathcal{D}}_{\min}(x_{\max}), \lambda^{\mathcal{D}}_{\max}(x_{\max})]$ respectively. It is easy to show that the projections of the points $x_{\min}$ and $x_{\max}$ on a node T, are equal to the minimal, resp. maximal elements of that node. This situation renders the $\mathcal{M}_T$-part superfluous, in case $x_{\min}$ and $x_{\max}$ are added to the original dataset $\mathcal{D}$.

Remark 3.8 Another simplification of the algorithms of Theorem 3 and 4 is possible, if one is only interested in either a tree for the minimal extension or a tree for the maximal extension, not in both. In that situation, it is not needed to work with the interval datasets: for instance, if one only wants a tree for the minimal extension, it is sufficient to work with the left endpoints of all the intervals, as can be seen when one scrutinizes the described algorithm. Thus, in that situation one can refrain from calculating the right endpoints of the intervals altogether: they are not needed. A similar remark can be made if one is only interested in a tree for the maximal extension.

3.6 Changes needed for Continuous Attributes

Note to the July 1997 edition: the text of this subsection will be published in the Supplement to this Technical report.

4 Induction of Monotone Decision Trees: Global Algorithms

In this section we will describe another class of algorithms for the induction of both binary and n-ary decision trees. These algorithms will only make use of a global dataset, as explained in the beginning of Section 3. Such a global dataset will be a non-interval dataset, so we return to our original concept of a dataset of Definition 1. The decision trees generated by the algorithms of this section will not be necessarily representations of the minimal or maximal extension of the dataset at hand. In general, the labeling rules associated with the trees of this section will be somewhere in between the minimal and maximal extension. As will be shown in Section 5, the trees of this section tend to be much smaller than the trees generated by the algorithms of Section 3.

4.1 Algorithms for Discrete Attributes

We will start with the description of the global algorithm for the binary tree case. To start with, we will assume the input space $\mathcal{X}$ to satisfy the Assumptions 1 and 2. In Section 4.2, we will explain how we can dispense with Assumption 2. As noted in the beginning of Section 3, we only need to specify a splitting rule, a stopping rule, a labeling rule and an update rule. Together these are then plugged into the algorithm of Figure 5 to give a complete description of the algorithm under consideration. Note that by Corollary 1 of Section 2.5 each node to be split or to be made into a leaf has the form

$$T = \{x \in \mathcal{X} : a \leq x \leq b\} \tag{13}$$

for some $a, b \in \mathcal{X}$.

We start with describing the update rule. When this rule fires, the dataset $\mathcal{D} = (D, \lambda)$ will be updated. In our algorithm at most two elements will be added to the dataset, each time the update rule fires. Recall, that because T is of the form (13), a is the minimal element of T and b is the maximal element of T. Now, either a or b, or both will be added to $\mathcal{D}$, provided with a well-chosen labeling. If a and b both already belong to D, nothing changes. Here is the complete update rule:

update (**var** $\mathcal{D}, T$):
if $a \notin D$ **then**
begin
$D := D \cup \{a\}$;
$\lambda(a) := \lambda^{\mathcal{D}}_{\max}(a)$
end;
if $b \notin D$ **then**
begin
$D := D \cup \{b\}$;
$\lambda(b) := \lambda^{\mathcal{D}}_{\min}(b)$
end

Figure 10: The Standard Update Rule

The splitting rule $\mathcal{S}(T, \mathcal{D})$ must be such that at each node the associated subset T is split into two nonempty subsets

$$\mathcal{S}(T, \mathcal{D}) = (T_L, T_R) \text{ with } T_L = \{x \in T : x_i \leq c\} \text{ and } T_R = \{x \in T : x_i > c\} \tag{14}$$

for some $i \in \{1, \ldots, n\}$, and some $c \in \mathcal{X}_i$. Note, that because of the assumption (see Section 2), T_R can also be written as $T_R = \{x \in T : x_i \geq c'\}$ for some $c' \in X_i$. Furthermore, the splitting rule must satisfy the following requirement: i and c must be chosen such that

$$\exists x, y \in D \cap T \text{ with } \lambda(x) \neq \lambda(y), x \in T_L \text{ and } y \in T_R. \tag{15}$$

Next, we consider the stopping rule $\mathcal{H}(T, \mathcal{D})$. As a result of the actions of the update rule, both the minimal element a and the maximal element b of T belong to D. Now, as a stopping rule we will use:

$$\mathcal{H}(T, \mathcal{D}) = \begin{cases} \textbf{true} & \text{if } \lambda(a) = \lambda(b), \\ \textbf{false} & \text{otherwise.} \end{cases} \tag{16}$$

Finally, the labeling rule $\mathcal{L}(T, \mathcal{D})$ will be simply:

$$\mathcal{L}(T, \mathcal{D}) = \lambda(a) = \lambda(b). \tag{17}$$

For the proof that this algorithm works we will need two lemma's.

Lemma 6 *Let $\mathcal{D} = (D, \lambda)$ be a monotone dataset with $D \subset \mathcal{X}$ and $\lambda : D \to \mathcal{C}$. Let x' be an arbitrary element of $\mathcal{X} \setminus D$, and let $c' \in \mathcal{C}$ be such that*

$$\lambda^{\mathcal{D}}_{\min}(x') \leq c' \leq \lambda^{\mathcal{D}}_{\max}(x').$$

If $\mathcal{D}' = (D', \lambda')$ *is defined as follows:*

$$\left\{ \begin{array}{l} D' = D \cup \{x'\} \\ \lambda'(x) = \left\{ \begin{array}{ll} \lambda(x) & \text{for } x \in D \\ c' & \text{for } x = x' \end{array} \right. \end{array} \right.$$

then the following assertions are true:

(i) $\mathcal{D}$ *is a monotone dataset*

(ii) $\lambda_{\min}^{\mathcal{D}} \leq \lambda_{\min}^{\mathcal{D}'} \leq \lambda_{\max}^{\mathcal{D}'} \leq \lambda_{\max}^{\mathcal{D}}$

(iii) $\Lambda(\mathcal{D}') \subset \Lambda(\mathcal{D})$.

Proof: Part (i). To prove that $\mathcal{D}'$ is a monotone dataset we only need to prove that for any $x \in D$:

$$x \leq x' \Rightarrow \lambda(x) \leq \lambda'(x')$$

and

$$x' \leq x \Rightarrow \lambda'(x') \leq \lambda(x).$$

But, if $x \leq x'$ and $x \in D$, then $\lambda(x) \leq \lambda_{\min}^{\mathcal{D}}(x')$ according to the definition of $\lambda_{\min}^{\mathcal{D}}$, so $\lambda(x) \leq \lambda_{\min}^{\mathcal{D}}(x') \leq c' = \lambda'(x')$. The case $x' \leq x$ can be treated similarly.

Part (ii). Follows from the definition of $\lambda_{\min}^{\mathcal{D}}$ and $\lambda_{\max}^{\mathcal{D}}$ and the fact, that $D \subset D'$.

Part (iii). Let $\hat{\lambda}$ be an arbitrary element of $\lambda(\mathcal{D}')$. So $\hat{\lambda}$ is monotone and $\lambda_{\min}^{\mathcal{D}'} \leq \hat{\lambda} \leq \lambda_{\max}^{\mathcal{D}'}$. According to part (ii) of this lemma, we also have $\lambda_{\min}^{\mathcal{D}} \leq \hat{\lambda} \leq \lambda_{\max}^{\mathcal{D}}$. Thus, $\hat{\lambda}$ is also a monotone extension of $\mathcal{D}$. □

Lemma 7 *If* $\mathcal{D} = (D, \lambda)$ *is a monotone dataset and* $a, b \in D$, *such that* $a \leq b$ *and* $\lambda(a) = \lambda(b) = c \in \mathcal{C}$, *then for all* $\hat{\lambda} \in \Lambda(\mathcal{D})$ *we have for all* $x \in T = \{x \in \mathcal{X} : a \leq x \leq b\}$

$$\lambda(x) = c.$$

Proof: From the monotonicity of λ it follows that for $x \in T$:

$$c = \lambda(a) \leq \lambda(x) \leq \lambda(b) \leq \lambda(b) = c. \ \square$$

Now we can formulate and prove the main theorem of this section.

Theorem 5 *If* $\mathcal{X}$ *is an input space that satisfies Assumption 1 and 2, if* $\mathcal{D} = (D, \lambda)$ *is a monotone dataset on* $\mathcal{X}$, *if the functions* $\mathcal{S}, \mathcal{H}, \mathcal{L}$ *satisfy* (14), (15), (16) *and* (17), *then the algorithm of Figure 5 of Section 3 together with the update rule of Figure 6 will generate a monotone decision tree* $\mathcal{T}$ *with* $\lambda_{\mathcal{T}} \in \Lambda(\mathcal{D})$.

Proof: The update rule of the algorithm generates a finite sequence of datasets $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k$, with $\mathcal{D}_i = (D_i, \lambda_i), D_i \in \mathcal{X}, \lambda_i : D_i \to \mathcal{C}, 1 \leq i \leq k$, such that, according to Lemma 1, each $\mathcal{D}_i$ is monotone, $D \subset D_1 \subset D_2 \subset \ldots \subset D_k$,

$$\lambda_{\min}^{\mathcal{D}} \leq \lambda_{\min}^{\mathcal{D}_1} \leq \ldots \leq \lambda_{\min}^{\mathcal{D}_k} \leq \lambda_{\max}^{\mathcal{D}_k} \leq \ldots \leq \lambda_{\max}^{\mathcal{D}_1} \leq \lambda_{\max}^{\mathcal{D}},$$

and

$$\Lambda(\mathcal{D}_k) \subset \ldots \subset \Lambda(\mathcal{D}_1) \subset \Lambda(\mathcal{D}).$$

The update rule guarantees, that the minimal and maximal element of each node, where the stopping rule fires, are members of the dataset. So for such a node, Lemma 2 asserts there is just one labeling rule for this node: $\lambda_{\mathcal{T}}$. For the last dataset $\mathcal{D}_k$ we must have: all minimal and maximal elements of all leaves are members of $\mathcal{D}_k$, so $\Lambda(\mathcal{D}_k)$ will consist of just one member: $\lambda_{\mathcal{T}}$. The process must be finite since we have a finite input space $\mathcal{X}$, and each D_i must be a subset of $\mathcal{X}$. □

Again, just like the comparable theorems of Section 3, note that this theorem actually proves a whole class of algorithms to be correct: the requirements set by the theorem to the splitting rule are quite general. Nothing is said in the requirements about how to select the attribute X_i and how to calculate the cut-off point c for a test of the form $t = \{X_i \leq c\}$. Obvious candidates for attribute-selection and cut-off point calculation are the well-known impurity measures like entropy, Gini or the twoing rule, see Breiman *et al.* [4].

As an illustration of the operation of the presented algorithm we will use it to generate a monotone decision tree for the dataset of Table 1. As an impurity criterium we will use entropy, see [6]. Starting in the root, we have $T = \mathcal{X}$, so $a = 000$ and $b = 222$. Now, $\lambda^{\mathcal{D}}_{\max}(000) = 0$ and $\lambda^{\mathcal{D}}_{\min}(222) = 3$, so the elements 000:0 and 222:3 are added to the dataset, which then consists of 7 examples. Next, six possible splits are considered: $X_1 \leq 0, X_1 \leq 1, X_2 \leq 0, X_2 \leq 1, X_3 \leq 0$ and $X_3 \leq 1$. For each of these possible splits we calculate the decrease in entropy as follows. For the test $X_1 \leq 0$, the space $\mathcal{X} = [000, 222]$ is split into the subset $T_L = [000, 022]$ and $T_R = [100, 222]$. Since T_L contains three data elements and T_R contains the remaining four, the average entropy of the split is $\frac{3}{7} \times 0.92 + \frac{4}{7} \times 1 = 0.97$. Thus, the decrease in entropy for this split is $1.92 \Leftrightarrow 0.97 = 0.95$. When calculated for all six splits, the split $X_1 \leq 0$ gives the largest decrease in entropy, so it is used as the first split in the tree. Proceeding with the left node $T = [000, 022]$ we start by calculating $\lambda^{\mathcal{D}}_{\min}(022) = 1$ and adding the element 022:1 to the dataset $\mathcal{D}$, which will then have eight elements. We then consider the four possible splits $X_2 \leq 0, X_2 \leq 1, X_3 \leq 0$ and $X_3 \leq 1$, of which the last one gives the largest decrease in entropy, and leads to the nodes $T_L = [000, 021]$ and $T_R = [002, 022]$. Since $\lambda^{\mathcal{D}}_{\min}(021) = 0 = \lambda(000)$, T_L is made into a leaf with class 0. Proceeding in this manner we end up with the decision tree of Figure 11 which is easily checked to be monotone.

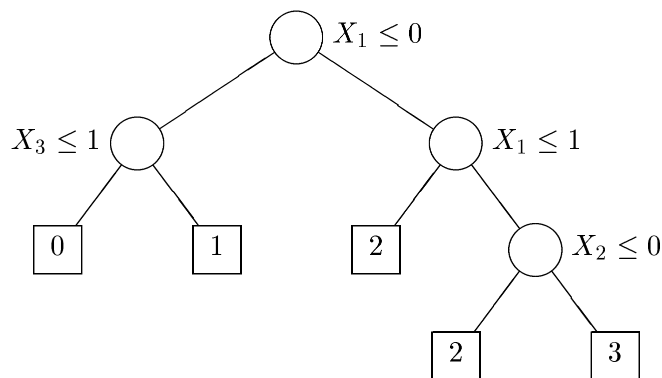

Figure 11: Decision Tree for the Bank Loan Dataset produced by the Standard Algorithm

Remark 4.1 A slight but sometimes useful variation of the above algorithm is the following. We change the update rule to

```
update (var 𝒟, T):
    if T is homogeneous then
        begin
            if a ∉ D then
                begin
                    D := D ∪ {a};
                    λ(a) := λ^𝒟_max(a)
                end;
            if b ∉ D then
                begin
                    D := D ∪ {b};
                    λ(b) := λ^𝒟_min(b)
                end
        end
```

Figure 12: Update Rule: a variation

thus, only adding the corner-elements to the dataset if the node T is *homogeneous*, i.e. if

$$\forall x, y \in D \cap T : \lambda(x) = \lambda(y).$$

If T is homogeneous, we will use the notation λ_T for the common value $\lambda(x)$ of all $x \in D \cap T$. The stopping rule becomes:

$$\mathcal{H}(T, \mathcal{D}) = \begin{cases} \textbf{true} & \text{if } T \text{ is homogeneous and } \lambda(a) = \lambda(b), \\ \textbf{false} & \text{otherwise} \end{cases}$$

and the labeling rule:

$$\mathcal{L}(T, \mathcal{D}) = \lambda(a) = \lambda(b) = \lambda_T.$$

With these changes the theorem remains true as can be easily seen. However, whereas with the standard algorithm from the beginning one works at 'monotonizing' the tree, this algorithm starts adding corner elements only when it has found a homogeneous node. For instance, if one uses maximal decrease of entropy as a measure of the performance of a test-split $t = \{X_i \leq c\}$, this new algorithm is equal to Quinlan's ID3-algorithm, until one hits upon a homogeneous node; from then on our algorithm starts adding the corner elements a and b to the dataset, enlarging the tree somewhat, but making it monotone. We call this process *cornering*. Thus, our algorithm can be seen as a method that first builds a traditional (non-monotone) tree with a method such as ID3, C4.5 or CART, and next makes it monotone by adding corner elements to the dataset. This observation yields also the possible use of this variant: if one has an arbitrary (non-monotone) tree for a monotone classification problem, it can be 'repaired' i.e. made monotone by adding corner elements to the leaves and growing some more branches where necessary.

As an example of the use of this remark, suppose we have the following monotone dataset $\mathcal{D}$:

000	0
001	1
100	0
110	1

Suppose further, that someone hands us the following decision tree for classifying the above dataset:

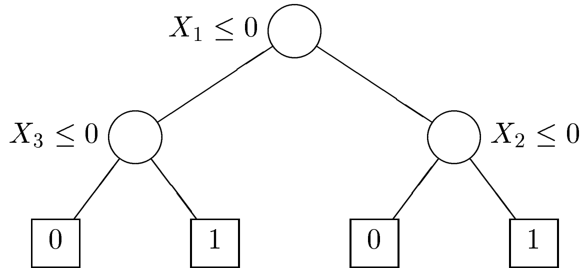

Figure 13: Non-monotone Decision Tree

This tree indeed classifies $\mathcal{D}$ correctly, but although $\mathcal{D}$ is monotone, the tree is not. In fact, it classifies data element 001 as belonging to class 1 and 101 as 0. Clearly, this is against monotonicity rule (1). To correct the above tree, we apply the algorithm of Remark 4.1 to it. We add the maximal element of the third leaf 101 to the dataset with the value $\lambda_{\min}^{\mathcal{D}}(101) = 1$. The leaf is subsequently split and the resulting tree is easily found to be monotone:

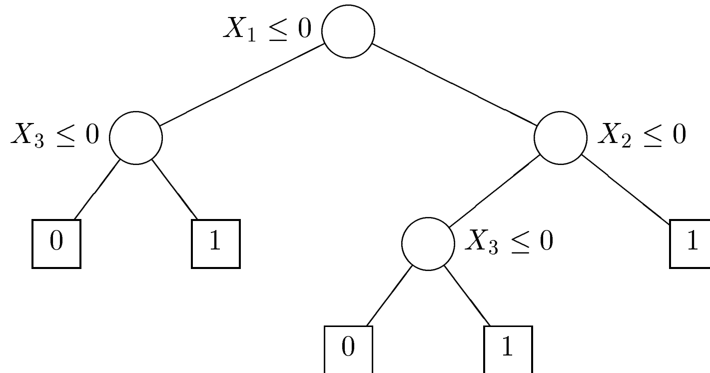

Figure 14: The above tree, but repaired

Of course, if we would have grown a tree directly with the above dataset $\mathcal{D}$ with the standard algorithm we would have ended up with a smaller tree, which is equally correct and monotone:

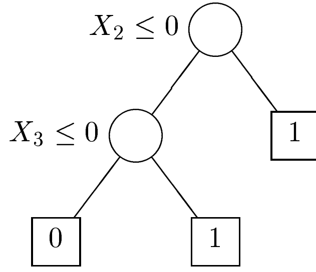

Figure 15: Monotone Tree produced by the Standard Algorithm

Nevertheless, it helps to know that we can make an arbitrary tree monotone by splitting up some of the leaves and adding a few more branches.

Remark 4.2 The main algorithm of this section further suggests the following selection criterion. First note, that for each $T = \{x \in \mathcal{X} : a \leq x \leq b\}$ with $T \cap D \neq \emptyset$ we have

$$\lambda^{\mathcal{D}}_{\max}(a) \leq \lambda^{\mathcal{D}}_{\min}(b).$$

This can be seen as follows: let x_0 be an element of $T \cap D$, then

$$\lambda^{\mathcal{D}}_{\max}(a) \leq \lambda(x_0) \leq \lambda^{\mathcal{D}}_{\min}(b).$$

We now define the variation of the dataset on T as follows:

$$\text{var }(T) = |[\lambda^{\mathcal{D}}_{\max}(a), \lambda^{\mathcal{D}}_{\min}(b)]| \Leftrightarrow 1$$

It is clear that $\text{var}(T) = 0$ iff $\lambda^{\mathcal{D}}_{\max}(a) = \lambda^{\mathcal{D}}_{\min}(b)$. Clearly, this measure can be used as an impurity measure, and the decrease in variation can be taken as an attribute selection criterium. However, some experiments have shown that it is inferior to entropy or Gini: trees grown with this impurity measure tend to be somewhat larger than those grown with entropy or the Gini-index.

4.2 Changes Needed for Continuous Attributes

Note to the July 1997 edition: the text of this subsection will be published in the Supplement to this Technical report.

5 Experiments

We did some experiments to get an idea of the usefulness of our methods and to compare them with those of Ben-David[1, 2, 3] and Makino *et al.*[5]. First we did some experiments to investigate the size of the trees that our methods would generate, also in comparison with other methods.

We generated random monotone datasets with 10, 20, 30, etc. examples and built trees with each of those datasets, using four different methods: ID3 as a general method, which does not generate monotone trees, and three methods presented in this report:

MinEx, which is the method for inducing bivariate Minimal Extension trees introduced in Section 3, MT1 is the variation method of Remark 4.1, MT2 is the main method of Section 4 which we called the Standard Algorithm. As an aside, we use the abbreviation MT for Monotone Tree. For each number of examples we generated four different datasets, each from a universe with 5 attributes, each having 3 possible values, while all data elements where evenly divided over 4 classes. The results for the number of leaves of the generated trees are shown in Table 2.

examples	ID3	MinEx	MT1	MT2
10	7.3	17.8	16.0	8.3
20	12.5	37.3	30.0	19.8
30	17.0	46.5	44.8	32.0
40	21.5	49.0	43.8	36.8
50	30.8	62.3	48.8	40.3
60	31.0	67.5	53.3	45.8
70	38.3	68.3	57.5	48.8
80	43.3	79.3	67.5	62.7
90	47.5	80.8	68.0	63.3
100	57.8	88.5	79.0	66.0

Table 2: Size of trees: Number of Leaves

The size of a tree can also be measured by looking at the depth of a tree. One way to measure this depth is the average path length: the average length of a path from the root of the tree to a leaf. For instance, the average path length of the tree of Figure 15 is 1.67 since there are two paths of length 2 and one of length 1. In Table 3 you will find the results of our measurements, where the size of the generated trees is measured in average path length.

examples	ID3	MinEx	MT1	MT2
10	2.6	3.6	3.2	2.8
20	3.6	4.9	4.5	4.2
30	3.9	5.2	5.0	4.8
40	4.3	5.4	5.0	4.8
50	4.9	5.8	5.3	5.2
60	4.8	6.0	5.4	5.2
70	5.1	5.8	5.5	5.4
80	5.4	6.2	5.9	5.8
90	5.5	6.3	5.9	5.8
100	5.8	6.4	6.2	6.0

Table 3: Size of Trees: Average Path Length

Another measure of the depth of a tree is the expected number of comparisons needed to classify an arbitrary new example presented to the tree. If $T_1, \ldots, T_k$ are the leaves of a tree, this measure can be calculated as

$$\text{Expected Number of Comparisons Needed} = \sum_{i=1}^{k} \ell_i \frac{|T_i|}{|\mathcal{X}|}$$

where ℓ_i is the length of the path from the root to the leaf T_i. One advantage of this method of measuring the size of a tree is, that it can also be applied to a non-tree method such as OLM [1], where a new example also must be compared with a number of elements of the OLM-database. Thus, this measure is also a measure of the efficiency of the generated classifiers. The results are shown in Table 4.

examples	ID3	OLM	MinEx	MT1	MT2
10	3.3	6.8	5.2	4.8	3.5
20	4.0	11.9	6.3	5.6	4.8
30	4.6	15.1	6.4	6.0	5.6
40	4.8	18.4	6.6	6.0	6.0
50	5.6	22.5	6.9	6.3	6.0
60	5.6	24.6	6.9	6.6	6.5
70	6.1	27.0	7.0	6.8	6.4
80	6.1	26.0	7.2	6.8	6.7
90	6.2	27.7	7.1	6.8	6.7
100	6.4	34.0	7.1	6.9	6.7

Table 4: Expected Number of Comparisons Needed

Thus, it seems that as a classifying tool, decision trees are much more efficient than a method such as OLM, although OLM produces genuinely monotone classification rules.

As a second experiment we did an attempt to investigate the generalizing power of the proposed methods. Again, we generated random monotone datasets of size 10, 20, etc. But now we used these datasets for 3-fold cross validation experiments: we build a tree on two thirds of a dataset, and tested the tree on the remaining one third. Each cross validation experiment was repeated four times. The average percentage of correctly classified examples will be found in Table 5 for each of the five methods we tested.

examples	ID3	OLM	MinEx	MT1	MT2
10	40.1	29.9	37.5	55.6	55.6
20	27.4	19.0	32.0	37.9	38.9
30	37.0	29.0	43.0	48.3	45.8
40	52.6	34.8	49.8	57.5	55.1
50	32.5	25.1	36.0	47.1	46.6
60	46.7	28.3	42.1	54.0	56.3
70	49.0	26.0	45.3	56.0	55.3
80	35.0	22.5	41.0	51.5	48.7
90	55.3	28.9	51.1	64.5	63.6
100	47.0	29.3	46.3	61.2	59.0
Average	42.3	27.3	42.3	53.4	52.5

Table 5: Percentage Correctly Classified in 3-fold Cross Validation

As a tentative result, it seems that our methods of Section 4 are better in predicting a class for a new example than the other methods for these monotone problems.

As a third and final experiment we wanted to compare our main methods with those of Makino *et al.* To do this we could only consider two class problems, since their method

works only in that situation. Thus, we generated monotone datasets for two class problems with size 10, 20, etc., we generated trees with Makino and our Standard method MT2, and we measured the size of the resulting trees, with the above three criteria. In addition, we measured the speed of the algorithm for generating the trees in seconds on our computer. The results are shown in Table 6.

	# leaves		average depth		# comparisons		speed	
examples	Makino	MT2	Makino	MT2	Makino	MT2	Makino	MT2
10	4.6	6.4	2.6	2.7	2.0	2.2	24.2	1.2
20	7.8	8.0	3.5	3.5	2.7	2.6	33.8	1.8
30	11.0	11.6	4.0	4.2	3.2	3.1	38.8	2.6
40	15.6	15.8	4.8	4.9	3.5	3.5	51.4	2.8
50	16.0	15.8	5.1	4.8	3.4	3.4	62.6	3.2
60	23.2	24.6	5.6	5.5	4.2	4.2	98.8	5.0
70	23.3	24.7	5.5	5.4	4.2	4.3	94.3	5.3

Table 6: Comparison with Makino *et al.*

It appears that our algorithm MT2 in the 2-class situation generates trees of comparable size, but it is 10 to 20 times as fast as the method of Makino *et al.*

6 Conclusion and further remarks

We have provided a number of tree generation algorithms for monotone classification problems with discrete and continuous domains and k classes. This improves and extends results of Ben-David [1] and Makino *et al.* [5]. In this report we discussed some further experiments with the two-class problem that show that our algorithm is 10 to 20 times as fast as that of [5].

Acknowledgement

We thank René van Dordregt for many discussions and for help with the experiments.

References

[1] Ben-David, A., Sterling, L., Pao, Y.H., (1989). Learning and classification of monotonic ordinal concepts. In Computational Intelligence vol. 5, 45-49.

[2] Ben-David, A., (1992). Automatic generation of symbolic multiattribute ordinal knowledge-based DSSs: methodology and applications. In Decision Sciences, vol. 23, 1357-1372.

[3] Ben-David, A., (1995). Monotonicity maintenance in information-theoretic machine learning algorithms. In Machine Learning, vol. 19, 29-43.

[4] Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J., (1984). Classification and regression trees. Chapman and Hall, NewYork. Second Edition 1993.

[5] Makino, K., Suda, T., Yano, K., Ibaraki, T., (1996). Data analysis by positive decision trees. In International symposium on cooperative database systems for advanced applications (CODAS), Kyoto, 282-289.

[6] Quinlan, J.R., (1986). Induction of decision trees. In Machine Learning, vol. 1, 81-106.