CLASSIFICATION TREES FOR PROBLEMS WITH MONOTONICITY CONSTRAINTS

R. POTHARST, A.J. FEELDERS

ERIM REPORT SERIES *RESEARCH IN MANAGEMENT*	
ERIM Report Series reference number	ERS-2002-45-LIS
Publication	April 2002
Number of pages	36
Email address corresponding author	potharst@few.eur.nl
Address	Erasmus Research Institute of Management (ERIM) Rotterdam School of Management / Faculteit Bedrijfskunde Erasmus Universiteit Rotterdam P.O. Box 1738 3000 DR Rotterdam, The Netherlands Phone: +31 10 408 1182 Fax: +31 10 408 9640 Email: info@erim.eur.nl Internet: www.erim.eur.nl

BIBLIOGRAPHIC DATA AND CLASSIFICATIONS

Abstract	For classification problems with ordinal attributes very often the class attribute should increase with each or some of the explaining attributes. These are called classification problems with monotonicity constraints. Classical decision tree algorithms such as CART or C4.5 generally do not produce monotone trees, even if the dataset is completely monotone. This paper surveys the methods that have so far been proposed for generating decision trees that satisfy monotonicity constraints. A distinction is made between methods that work only for monotone datasets and methods that work for monotone and non-monotone datasets alike.	
Library of Congress Classification (LCC)	5001-6182	Business
	5201-5982	Business Science
	HB 143	Mathematical Programming
Journal of Economic Literature (JEL)	M	Business Administration and Business Economics
	M 11 R 4	Production Management Transportation Systems
	C 6	Mathematical Methods and Programming
European Business Schools Library Group (EBSLG)	85 A	Business General
	260 K 240 B	Logistics Information Systems Management
	5 C	Logic
Gemeenschappelijke Onderwerpsontsluiting (GOO)		
Classification GOO	85.00	Bedrijfskunde, Organisatiekunde: algemeen
	85.34 85.20	Logistiek management Bestuurlijke informatie, informatieverzorging
	31.10	Logica
Keywords GOO	Bedrijfskunde / Bedrijfseconomie	
	Bedrijfsprocessen, logistiek, management informatiesystemen	
	Monotonie wiskunde, constraints, classificatietheorie, besliskunde, ordinale gegevens	
Free keywords	monotone, monotonicity constraint, classification, classification tree, decision tree, ordinal data	

Classification Trees for Problems with Monotonicity Constraints

R. Potharst
Erasmus University Rotterdam
P.O. Box 1738
3000 DR Rotterdam
Netherlands
potharst@few.eur.nl

A. J. Feelders
Utrecht University
P.O. Box 80089
3508 TB Utrecht
Netherlands
ad@cs.uu.nl

14 April 2002

Abstract

For classification problems with ordinal attributes very often the class attribute should increase with each or some of the explaining attributes. These are called classification problems with monotonicity constraints. Classical decision tree algorithms such as CART or C4.5 generally do not produce monotone trees, even if the dataset is completely monotone. This paper surveys the methods that have so far been proposed for generating decision trees that satisfy monotonicity constraints. A distinction is made between methods that work only for monotone datasets and methods that work for monotone and non-monotone datasets alike.

Keywords: monotone, monotonicity constraint, classification, classification tree, decision tree, ordinal data

1 Introduction

Even though data mining is often applied to domains where little theory is available, in many cases it is either known that the target function satisfies certain constraints, or it is simply required that the model constructed satisfies those constraints.

One type of constraint that is available in many applications states that the dependent variable (or its expected value) should be a monotonic function of the independent variables. Economic theory would state for example that people tend to buy less of a product if its price increases (ceteris paribus), so price elasticity of demand should be negative. The strength of this relationship and the precise functional form are however usually not dictated by economic theory. Other well-known examples are labor wages as a function of age and education (see e.g. [11]) or so-called hedonic price models where the price of a consumer good depends on a bundle of characteristics for which a valuation exists [9].

Another class of problems where monotonicity constraints often apply are so-called selection problems. Consider for example the selection of applicants for a job or a loan on the basis of their characteristics.

Because the monotonicity constraint is quite common in practice, many data analysis techniques have been adapted to be able to handle such constraints.

Isotonic regression, for example, deals with regression problems with monotonicity constraints. The traditional method used in isotonic regression is the *pool-adjacent violaters algorithm* [15]. This method however only works in the one-dimensional case. A versatile non-parametric method is given in [11].

Monotonicity constraints have also been investigated in the neural network literature. In [16] the monotonicity of the neural network is guaranteed by enforcing constraints on the weights during the training process. Daniels and Kamp [8] present a class of neural network that are monotonic by construction. This class is obtained by considering multilayer neural networks with non-negative weights.

Various methods have also been proposed for classification problems with monotonicity constraints, such as decision lists [4], logical analysis of data [5], rough sets [6] and instance-based learning [3, 1].

Classification or decision trees are among the most popular algorithms for classification problems in data mining and machine learning. Therefore we consider in this paper methods to build monotone classification trees.

In Section 2 we define monotone classification and other important concepts that are used throughout the paper. We also provide a motivating example concerning applicants for a bank loan, that is used to illustrate many of the algorithms presented.

The paper then divides into algorithms that work on monotone datasets (Section 3) and algorithms that also work on non-monotone data sets (Section 4).

In Section 3.2 we present an algorithm that forces the construction of a monotone tree by adding, if required, the corner elements of a node with an appropriate class label to the dataset. A somewhat more efficient algorithm that first builds a quasi-monotone tree, and then repairs, if required, any minor local non-monotonicities is presented in Section 3.3.

In Section 4 we present two algorithms that work on non-monotone data. The first is due to Ben-David [2], and adapts the well-known entropy splitting criterion by including a measure for the non-monotonicity of the tree that results after the split. In Section 4.2 we present a straightforward generate-and-test approach that constructs many different trees by resampling the training data, and selects a monotonic tree.

Finally, in Section 5 we end with a discussion, and some ideas for further research.

2 Monotone Classification

Let $\mathcal{X}$ be a partially ordered set of instances, called the *instance space*, and let $\mathcal{C}$ be a finite linearly ordered set of *classes*. The order relations of $\mathcal{X}$ and $\mathcal{C}$ will both be denoted by $\leq$. An *allocation rule* is a function

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

which assigns a class from $\mathcal{C}$ to every instance in the instance space $\mathcal{X}$. A *classification problem* is the problem of finding a class labeling f that satisfies certain constraints, to be specified in the problem description. One possible constraint is that the labeling f be monotone: a *monotone* allocation rule is a function $f : \mathcal{X} \to \mathcal{C}$ for which

$$\mathbf{x} \leq \mathbf{x}' \Rightarrow f(\mathbf{x}) \leq f(\mathbf{x}') \tag{1}$$

for all instances $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$. In this paper, $\mathcal{X}$ will always be a *feature space* $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_p$ consisting of vectors $\mathbf{x} = (x_1, x_2, \ldots, x_p)$ of values on p features or attributes. Here we assume that each feature takes values x_i in a linearly ordered set $\mathcal{X}_i$. The partial ordering $\leq$ on $\mathcal{X}$ will be the ordering induced by the order relations of its coordinates $\mathcal{X}_i$: $\mathbf{x} = (x_1, x_2, \ldots, x_p) \leq \mathbf{x}' = (x'_1, x'_2, \ldots, x'_p)$ if and only if $x_i \leq x'_i$ for all i. It is easy to see that a classification rule on a feature space is monotone if and only if it is non-decreasing in each of its features, when the remaining features are held fixed.

As an example, consider a selection procedure for applicants to a job based on the outcomes of a series of academic and/or psychological tests. If each of the test outcomes x_i is scored from low (bad performance) to high (good performance) and the classes are taken to be 0 = not selected and 1 = selected, then it would be very natural to demand the selection rule to be monotone. In fact, the requirement of monotonicity would be equivalent to excluding all situations in which applicant A scores better or at least as good on all tests as applicant B, whereas B gets selected and A does not.

A very common classification problem occurs, when the allocation rule should be induced from an available dataset or set of examples: for a finite number of instances a corresponding class is given; an allocation rule should be constructed that 'fits' these data. Formally, a *dataset* is a series $(\mathbf{x}_1, c_1), (\mathbf{x}_2, c_2), \ldots, (\mathbf{x}_n, c_n)$ of n examples $(\mathbf{x}_i, c_i)$ where each $\mathbf{x}_i$ is an element of the instance space $\mathcal{X}$ and c_i is a class label from $\mathcal{C}$. The presence of noise may lead to inconsistencies in the dataset that might disturb the faultless operation of our algorithms. We call a dataset *consistent* if for all i, j we have $\mathbf{x}_i = \mathbf{x}_j \Rightarrow c_i = c_j$. That is, each instance in the dataset has a unique associated class. For such a dataset it makes sense to speak of the class $\lambda(\mathbf{x})$ associated with an instance $\mathbf{x}$. Another important distinction we make in this paper is between monotone and non-monotone datasets. In fact, the methods of Section 3 work only for monotone datasets whereas those of Section 4 can be used also for non-monotone datasets. We call a dataset *monotone* if for all i, j we have $\mathbf{x}_i \leq \mathbf{x}_j \Rightarrow c_i \leq c_j$. It is easy to see that a monotone dataset is necessarily consistent. In fact, if $\mathbf{x}_i = \mathbf{x}_j$ then we have $\mathbf{x}_i \leq \mathbf{x}_j$ and $\mathbf{x}_j \leq \mathbf{x}_i$, so $c_i \leq c_j$ and $c_j \leq c_i$, and consequently, $c_i = c_j$. This discussion leads to the following formal definitions.

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

Note that the class labeling λ of a consistent dataset $\mathcal{D} = (D, \lambda)$ is *not* an allocation rule: it is only defined on D, a subset of $\mathcal{X}$, while an allocation rule must be defined on all elements of the instance space $\mathcal{X}$. In fact, a classification problem for a consistent dataset consists of finding an allocation rule f that is an extension of the class labeling λ of the dataset to the whole instance space $\mathcal{X}$.

Definition 2 A *monotone dataset* is a consistent dataset $\mathcal{D} = (D, \lambda)$ for which the implication (1) holds for all $\mathbf{x}, \mathbf{x}' \in D$ with f replaced by λ.

We will now give 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:

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	high
cl5	high	intermediate	excellent	high

Table 1: The bank loan dataset

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.

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	3
2	1	2	3

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

001	0
002	1
112	2
202	3
212	3

for the above dataset.

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.
- $[a, b]$ denotes the interval $\{\mathbf{x} \in \mathcal{X} : a \leq \mathbf{x} \leq b\}$, where both a and b are instance vectors from $\mathcal{X}$.
- $(a, b]$ denotes the interval $\{\mathbf{x} \in \mathcal{X} : a < \mathbf{x} \leq b\}$, where both a and b are instance vectors from $\mathcal{X}$.
- For all $\mathbf{x} \in \mathcal{X}$, we define the *upset* generated by $\mathbf{x}$ as
$$\uparrow\mathbf{x} = \{\mathbf{y} \in \mathcal{X} : \mathbf{y} \geq \mathbf{x}\}$$
and, if D is a subset of $\mathcal{X}$ the upset generated by D is defined as
$$\uparrow D = \bigcup_{\mathbf{x} \in D} \uparrow\mathbf{x}.$$
- Similarly, for $\mathbf{x} \in \mathcal{X}$, we define the *downset* generated by $\mathbf{x}$ as
$$\downarrow\mathbf{x} = \{\mathbf{y} \in \mathcal{X} : \mathbf{y} \leq \mathbf{x}\}$$
and the downset generated by a subset D of $\mathcal{X}$ is defined as
$$\downarrow D = \bigcup_{\mathbf{x} \in D} \downarrow\mathbf{x}.$$

2.1 Monotone Extensions of Datasets

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

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

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

and

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

We will now show[1] that the functions $\lambda^{\mathcal{D}}_{\min}$ and $\lambda^{\mathcal{D}}_{\max}$, as defined, are the minimal resp. maximal elements of $M(\mathcal{D})$.

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

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

(ii) $M(\mathcal{D}) = \{f : \lambda^{\mathcal{D}}_{\min} \leq f \leq \lambda^{\mathcal{D}}_{\max}$ *and* f *monotone*$\}$.

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 allocation rules we will call the *minimal rule* and the *maximal rule* respectively. In addition we have for every point $\mathbf{x}$ in the instance space bounds that any rule f must satisfy:

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

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

In Section 3 we will require the representation of our allocation rule to have a specific form, viz. the form of a classification tree or decision tree.

2.2 Quasi-monotone Allocation Rules

As can been seen in Makino *et al.* [10] for the two-class problem, it may be hard to find an exact solution to a monotone classification problem. Therefore, Makino *et al.* introduce the concept of quasi-monotonicity, which

[1]The proofs of all lemmas in this paper can be found in [12].

we generalize here to the k class problem. An allocation rule f will be called *quasi-monotone* for dataset $\mathcal{D} = (D, \lambda)$ if for all $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$

$$\mathbf{x} \leq \mathbf{x}' \text{ and } [\mathbf{x}, \mathbf{x}'] \cap D \neq \emptyset \Rightarrow f(\mathbf{x}) \leq f(\mathbf{x}'). \tag{2}$$

Recall that $[\mathbf{x}, \mathbf{x}']$ is the interval from $\mathbf{x}$ to $\mathbf{x}'$. So, for a quasi-monotone allocation rule (1) needs to hold only for pairs of instances that have at least one data-example in between them.

The set of quasi-monotone extensions of dataset $\mathcal{D}$ will be called $Q(\mathcal{D})$. It is clear that $M(\mathcal{D}) \subset Q(\mathcal{D})$, since monotonicity is stronger than quasi-monotonicity.

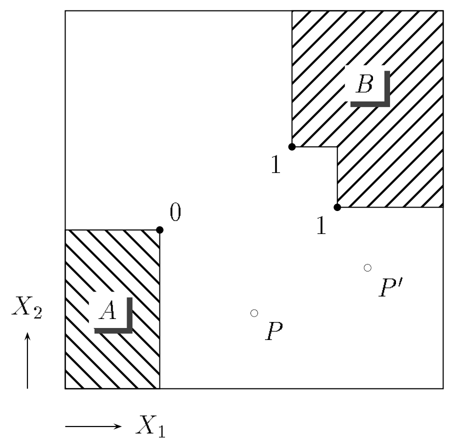

Figure 1: A quasi-monotone classification rule, that is not monotone

In Figure 1 we give an example of a quasi-monotone classification rule that is not monotone. In this example we have a dataset with two attributes X_1 and X_2 and two classes (0 and 1). Both attributes are numerical with values in some interval, say $[0, 1]$. The dataset contains three examples which have been marked in the figure with their classes, one example with class=0 and two with class=1. A quasi-monotone classification rule, that extends this dataset, is any rule that assigns class=0 to the points in the horizontally shaded area A, and class 1 to the points in vertically shaded area B. It does not matter what class is assigned to the points in the non-shaded area. So, if we assign class=1 to point P and class=0 to point P', then it follows from $P \leq P'$ and $1 = f(P) > f(P') = 0$ that a non-monotone classification rule results, which is quasi-monotone as long as it stays 0 at A and 1 at B.

Using the notation of Section 2.1 we can give a useful characterization of the concept of quasi-monotonicity and of the set $Q(\mathcal{D})$.

Lemma 2 *If $\mathcal{D} = (D, \lambda)$ is a monotone dataset, then*

$$Q(\mathcal{D}) = \{f : \lambda_{\min}^{\mathcal{D}} \leq f \leq \lambda_{\max}^{\mathcal{D}}, f \, \textit{quasi-monotone for } \mathcal{D}\}.$$

Thus, the minimal monotone allocation rule $\lambda_{\min}^{\mathcal{D}}$ for a dataset $\mathcal{D}$, is also the minimal quasi-monotone allocation rule.

If $f : \mathcal{X} \to \mathcal{C}$ is any allocation rule, we define the allocation rules $\hat{f}$ and $\check{f}$ as

$$\hat{f}(\mathbf{x}) = \max\{f(\mathbf{y}) : \mathbf{y} \leq \mathbf{x}\}$$

and

$$\check{f}(\mathbf{x}) = \min\{f(\mathbf{y}) : \mathbf{y} \geq \mathbf{x}\}$$

for $\mathbf{x} \in \mathcal{X}$. It is easy to see that, for all $\mathbf{x} \in \mathcal{X}$

$$\check{f}(\mathbf{x}) \leq f(\mathbf{x}) \leq \hat{f}(\mathbf{x})$$

and $\check{f}$ and $\hat{f}$ are monotone. In fact, it can be easily shown that $\hat{f}$ is the *least monotone major* of f, and $\check{f}$ is the *greatest monotone minor* of f. Using these functions $\check{f}$ and $\hat{f}$ we can give the following characterizations of monotonicity and quasi-monotonicity.

Lemma 3 *If $f : \mathcal{X} \to \mathcal{C}$ is an arbitrary allocation rule, then*

$$f \textit{ monotone} \quad \Leftrightarrow \quad \textit{for all } \mathbf{x} \in \mathcal{X} : \check{f}(\mathbf{x}) = \hat{f}(\mathbf{x})$$

Lemma 4 *If $\mathcal{D} = (D, \lambda)$ is a monotone dataset and $f : \mathcal{X} \to \mathcal{C}$ is an extension of λ, then*

$$f \textit{ quasi-monotone for } \mathcal{D} \quad \Leftrightarrow \quad \textit{for all } \mathbf{x} \in D : \check{f}(\mathbf{x}) = \hat{f}(\mathbf{x})$$

So, a monotone allocation rule coincides with its least monotone major and its least monotone minor on the whole instance space, while for a quasi-monotone rule this is only true for instances in the dataset.

In order to ensure the algorithms to work for both discrete and continuous instance spaces, we need one more concept that we will call $\mathbb{D}$-granularity. For a consistent dataset $\mathcal{D} = (D, \lambda)$ we define

$$\mathbb{D}_i = \{x_i | \mathbf{x} \in D\} \qquad \text{for } i = 1, \ldots, p$$

and

$$\mathbb{D} = \mathbb{D}_1 \times \mathbb{D}_2 \times \ldots \times \mathbb{D}_p.$$

Since D is finite, $\mathbb{D}_i$ and $\mathbb{D}$ are finite sets as well. In fact, $\mathbb{D}$ is a finite lattice with minimal element $d_{\min}$ and maximal element $d_{\max}$. Now, for each $\mathbf{x} \in \mathcal{X}$ with $\mathbf{x} \geq d_{\min}$ we define the $\mathbb{D}$-approximation $\tilde{\mathbf{x}}$ of $\mathbf{x}$ as follows:

$$\tilde{x}_i = \max\{d \in \mathbb{D}_i : d \leq x_i\} \text{ for } i = 1, \dots , p$$

and

$$\tilde{\mathbf{x}} = (\tilde{x}_1, \dots , \tilde{x}_p).$$

We will call an allocation rule $f : \mathcal{X} \to \mathcal{C}$ to be $\mathbb{D}$-*granular* for dataset $\mathcal{D}$, if for all $\mathbf{x} \in \mathcal{X}$ with $\mathbf{x} \geq d_{\min}$ we have $f(\mathbf{x}) = f(\tilde{\mathbf{x}})$. Thus, f is $\mathbb{D}$-granular if it is constant on all regions that have the same $\mathbb{D}$-approximation.

3 Methods for monotone data

Classification or decision trees have long been used for classification problems. Well-known introductions to this field can be found in [7] and [14]. 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 feature or attribute only. Within this class of univariate decision trees, we will only consider so-called *binary* trees. For such trees, at each node a split is made using a test of the form

$$X_i \leq c \ (\text{or} X_i < c)$$

for some $c \in \mathcal{X}_i, 1 \leq i \leq n$. Thus, for a binary tree, in each node[2] the associated set $T \subset \mathcal{X}$ is split into the two subsets $T_\ell = \{\mathbf{x} \in T : x_i \leq c\}$ and $T_r = \{\mathbf{x} \in T : x_i > c\}$. An example of a univariate binary decision tree is the following:

[2]By slight abuse of language in the sequel we will make no distinction between a node or leaf and its associated subset.

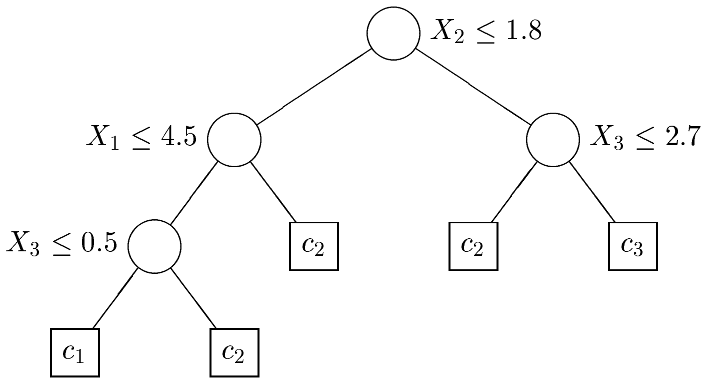

Figure 2: Univariate Binary Decision Tree: Example

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

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

the first and the last of which are classified as c_1 and c_3 respectively, and the remaining regions as c_2. The allocation rule that is induced by a decision tree $\mathcal{T}$ will be denoted by $f_{\mathcal{T}}$.

Lemma 5 *If $\mathcal{X}$ is an instance space with continuous features 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 $\mathcal{T}$,*

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

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

Here we use the expression $\overline{\mathcal{X}}$ instead of $\mathcal{X}$, because in some cases $\mathcal{X}$ would have to be extended with infinity-elements in order to have a representation of form (3) for each node or leaf.

If $\mathcal{X}$ is an instance space with discrete features, then any subset T associated with a univariate binary decision tree $\mathcal{T}$ on $\mathcal{X}$ will satisfy

$$T = \{\mathbf{x} \in \mathcal{X} : a \leq \mathbf{x} \leq b\} = [a, b] \tag{4}$$

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. Below we will call $\min(T) = a$ the *minimal element*[3] and $\max(T) = b$ the *maximal element* of T. Together, we call these the *corner* elements of the node T.

3.1 Testing the Monotonicity of a Decision Tree

In this subsection we describe an efficient algorithm for testing whether a given decision tree $\mathcal{T}$ is monotone or not. A naive way to test the monotonicity of a decision tree $\mathcal{T}$ would be to check all pairs of instances $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$, determine $f_{\mathcal{T}}(\mathbf{x})$ and $f_{\mathcal{T}}(\mathbf{x}')$ by throwing them through the tree and check whether we find a non-monotonicity like $\mathbf{x} \leq \mathbf{x}'$ and at the same time $f_{\mathcal{T}}(\mathbf{x}) > f_{\mathcal{T}}(\mathbf{x}')$. Of course, this method would be very time consuming and, in the continuous case, even sheer impossible. Fortunately, there is a straightforward manner to test the monotonicity using the maximal and minimal elements of the leaves of the decision tree:

```
for all pairs of leaves $T, T'$:
    if $\Big(f_{\mathcal{T}}(T) > f_{\mathcal{T}}(T')$ and $\min(T) < \max(T')\Big)$ or
        $\Big(f_{\mathcal{T}}(T) < f_{\mathcal{T}}(T')$ and $\max(T) > \min(T')\Big)$
    then stop: $\mathcal{T}$ not monotone
```

It is easy to check that a decision tree is passed through the above algorithm without stopping, if and only if the tree is monotone.

3.2 The Direct Method

In this subsection we will describe the algorithm proposed in [12] for the induction of a monotone binary decision tree from a monotone dataset. The algorithm has been tested extensively on artificial and real world data, see [13] for an application to a bankruptcy problem. We will first describe the algorithm for the case of a discrete feature space. At the end of the section we will indicate what changes are needed to run this algorithm in the continuous case.

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,

[3] In the continuous case this definition implies $\min(T) \notin T$, but that does not lead to any complications.

- 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.

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 3.

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

split($T$, var $\mathcal{D}$):
   $\mathcal{D} := \text{update}(\mathcal{D}, T)$;
   if $\mathcal{H}(T, \mathcal{D})$ then
      assign class label $\mathcal{L}(T, \mathcal{D})$ to leaf $T$
   else
      begin
           $(T_\ell, T_r) := \mathcal{S}(T, \mathcal{D})$;
           split $(T_\ell, \mathcal{D})$;
           split $(T_r, \mathcal{D})$
      end
```

Figure 3: Monotone Tree Induction Algorithm

In this algorithm outline there is one aspect that we have not mentioned yet: the *update rule.* In the algorithm 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.

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

In addition to the update rule, we need to specify a splitting rule, a stopping rule and a labeling rule. Together these are then plugged into the algorithm of Figure 3 to give a complete description of the algorithm under consideration.

We start with describing the update rule. When this rule fires, the dataset $\mathcal{D} = (D, \lambda)$ will be updated: at most two elements will be added to the dataset, each time the update rule fires. As soon as a node T is accessed, either the minimal element of T or the maximal element, or both will be added to $\mathcal{D}$, provided with a well-chosen class labeling. If both these corner elements of T already belong to D, nothing changes. Here is the complete update rule:

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

Figure 4: The Standard Update Rule

So, when a minimal element of node T is added to the dataset, it gets the highest possible class label. In contrast, a maximal element that is added to the dataset will receive the lowest possible class label. The reason for this choice has to do with the desire to produce a small tree. It speeds up the course towards homogeneous leaves.

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

$$\mathcal{S}(T, \mathcal{D}) = (T_\ell, T_r) \quad \begin{array}{l} \text{with } T_\ell = \{\mathbf{x} \in T : x_i \leq c\} \\ \text{and } T_r = \{\mathbf{x} \in T : x_i > c\} \end{array} \tag{5}$$

for some $i \in \{1, \ldots, p\}$, and some $c \in \mathcal{X}_i$. Furthermore, the splitting rule must satisfy the following requirement: i and c must be chosen such that

$$\exists \mathbf{x}, \mathbf{x}' \in D \cap T \text{ with } \lambda(\mathbf{x}) \neq \lambda(\mathbf{x}'), \mathbf{x} \in T_\ell \text{ and } \mathbf{x}' \in T_r. \tag{6}$$

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 $\min(T)$ and the maximal element $\max(T)$ of T belong to D. Now, as a stopping rule we will use:

$$\mathcal{H}(T, \mathcal{D}) = \begin{cases} \textbf{true} & \text{if } \lambda(\min(T)) = \lambda(\max(T)), \\ \textbf{false} & \text{otherwise.} \end{cases} \tag{7}$$

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

$$\mathcal{L}(T, \mathcal{D}) = \lambda(\min(T)) = \lambda(\max(T)). \tag{8}$$

For the proof that this algorithm works we will need two lemmas. The first of these lemmas tells us that if we add an instance to a dataset while giving it a class label that is in between the lower and upper bounds that are given by the dataset as it is now, the dataset remains monotone. The second lemma tells us that if the minimal and maximal element of a node both have the same class label, then we can make this node into a leaf with that class label.

Lemma 6 *Let $\mathcal{D} = (D, \lambda)$ be a monotone dataset with $D \subset \mathcal{X}$ and $\lambda : D \to \mathcal{C}$. Let $\mathbf{x}^+$ be an arbitrary instance vector with $\mathbf{x}^+ \notin D$, and let $c \in \mathcal{C}$ be such that*

$$\lambda_{\min}^{\mathcal{D}}(\mathbf{x}^+) \leq c \leq \lambda_{\max}^{\mathcal{D}}(\mathbf{x}^+).$$

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

$$\begin{cases} D^+ = D \cup \{\mathbf{x}^+\} \\ \lambda^+(\mathbf{x}) = \begin{cases} \lambda(\mathbf{x}) & \textit{for } \mathbf{x} \in D \\ c & \textit{for } \mathbf{x} = \mathbf{x}^+ \end{cases} \end{cases}$$

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) $M(\mathcal{D}^+) \subset M(\mathcal{D})$.

(iv) $Q(\mathcal{D}^+) \subset Q(\mathcal{D})$.

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 monotone allocation rules $f \in M(\mathcal{D})$ we have for all $\mathbf{x} \in T = \{\mathbf{x} \in \mathcal{X} : a \leq \mathbf{x} \leq b\}$*

$$f(\mathbf{x}) = c.$$

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

Theorem 1 *Let $\mathcal{X}$ be a finite instance space with discrete features and let $\mathcal{D} = (D, \lambda)$ be a monotone dataset on $\mathcal{X}$. If the functions $\mathcal{S}, \mathcal{H}, \mathcal{L}$ satisfy the requirements* (5),(6),(7) *and* (8)*, then the algorithm of Figure 3 together with the update rule of Figure 4 will generate a monotone decision tree $\mathcal{T}$ with $f_{\mathcal{T}} \in M(\mathcal{D})$.*

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 6, 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

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

The update rule guarantees, that the minimal and maximal element of each node, where the stopping rule fires, are members of the dataset. For such a node, Lemma 7 asserts there is only one labeling possible. 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 $M(\mathcal{D}_k)$ will consist of just one member: $f_{\mathcal{T}}$. The process must be finite since we have a finite instance space $\mathcal{X}$, and each D_i must be a subset of $\mathcal{X}$. $\square$

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 [7].

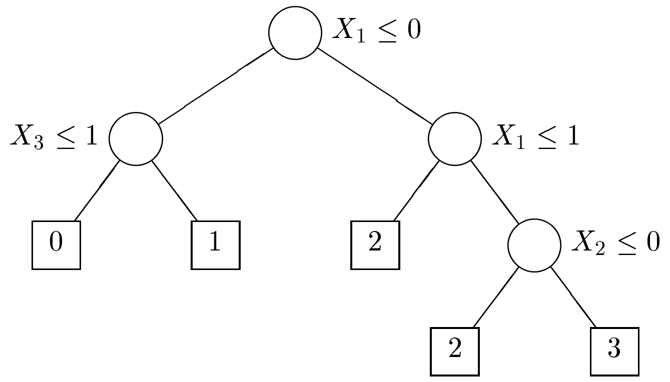

Figure 5: Monotone Decision Tree for the Bank Loan Dataset

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 criterion we will use entropy, see [14]. 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 $T_\ell = [000, 022]$ 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 $1.92 - 0.97 = 0.95$. 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 $T_\ell = [000, 021]$ 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 5 which is easily checked to be monotone.

A useful variation of the above algorithm is the following. We change the update rule to

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

Figure 6: Update Rule: a variation

thus, only adding the minimal and maximal elements of a node T to the dataset if the node is *homogeneous*, i.e. if

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

The splitting rule, stopping rule and labeling rule remain the same. 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 algorithm is equal to Quinlan's C4.5-algorithm, until one hits upon a homogeneous node; from then on our algorithm starts adding the corner elements $\min(T)$ and $\max(T)$ to the dataset, enlarging the tree somewhat, but making it monotone. We call this process *cornering.* Thus, the algorithm of Figure 6 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:

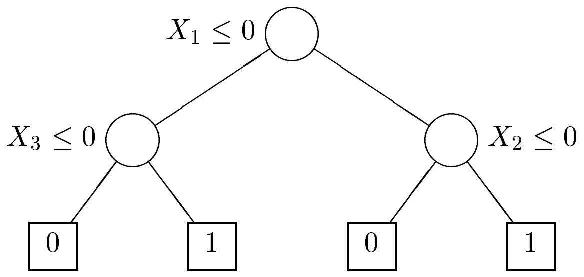

Figure 7: Non-monotone Decision Tree

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 Figure 6 to it. We add the maximal element of the third leaf 101 to the dataset with the value $\lambda^{\mathcal{D}}_{\min}(101) = 1$. The leaf is subsequently split and the resulting tree is easily found to be monotone:

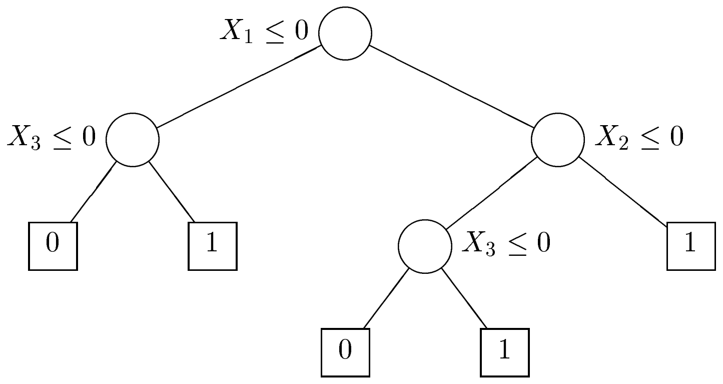

Figure 8: The above tree, but repaired

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:

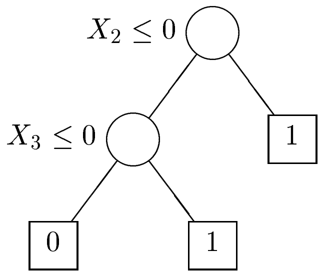

Figure 9: Monotone Tree produced by the Standard Algorithm

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.

The main algorithm of this section further suggests a new impurity measure to be used as an attribute selection criterion. First note, that for each $T = \{\mathbf{x} \in \mathcal{X} : a \leq \mathbf{x} \leq b\}$ with $T \cap D \neq \emptyset$ we have

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

This can be seen as follows: let $\mathbf{x}_0$ be an element of $T \cap D$, then

$$\lambda^{\mathcal{D}}_{\max}(a) \leq \lambda(\mathbf{x}_0) \leq \lambda^{\mathcal{D}}_{\min}(b).$$

We now define the *variation* of the dataset on T as

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

the number of different class labels that are possible within node T minus one. 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 criterion. However, 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.

3.2.1 Changes Needed for Continuous Attributes

Here we will sum up the changes that need to be made to the described algorithms in case one or more of the attributes is continuous. For simplicity of notation we will assume that all attributes $X_i, 1 \leq i \leq p$, are continuous on a finite or infinite subinterval $\mathcal{X}_i$ of $\mathbb{R}$. If in practice, some of the attributes are discrete while others are continuous, the reader can easily adapt the described procedures to that situation.

Thus, we assume that we have an infinite instance space $\mathcal{X} = \mathcal{X}_1 \times \ldots \times \mathcal{X}_p$, with $\mathcal{X}_i$ a subinterval of $\mathbb{R}$, the set of real numbers. However, the dataset $\mathcal{D} = (D, \lambda)$ will always be finite. In particular, let us assume that attribute X_i has values

$$x_i^{(1)} < x_i^{(2)} < \ldots < x_i^{(k_i)}$$

in the dataset D, where k_i is the number of different values that attribute X_i has in the dataset D. Of course, $k_i \leq |\mathcal{D}|$. In fact, with probability one we have $k_i = |\mathcal{D}|$, but, because of rounding off, in practice $k_i < |\mathcal{D}|$ will often occur. Now, we define

$$\mathcal{X}_i^{\mathcal{D}} = \{x_i^{(1)}, \ldots, x_i^{(k_i)}\}$$

and

$$\mathcal{X}^{\mathcal{D}} = \mathcal{X}_1^{\mathcal{D}} \times \mathcal{X}_2^{\mathcal{D}} \times \ldots \times \mathcal{X}_p^{\mathcal{D}}.$$

Thus, $\mathcal{X}^{\mathcal{D}}$ is a finite space which includes all instances in D, and which is discrete. So we have mapped the classification problem with infinite instance space $\mathcal{X}$ onto a classification problem with finite space $\mathcal{X}^{\mathcal{D}}$. Using the methods of this section we can generate a decision tree for the classification problem on $\mathcal{X}^{\mathcal{D}}$. The final step then will be to translate this decision tree on $\mathcal{X}^{\mathcal{D}}$ to a decision tree on $\mathcal{X}$.

Let $\mathcal{T}$ be a binary monotone decision tree on $\mathcal{X}^{\mathcal{D}}$, generated by one of the methods of this section using dataset $\mathcal{D}$. Each test of this tree will have either the form

$$X_i \leq x_i^{(j)} \tag{9}$$

for some j with $1 < j \leq k_i$, for some $i \in \{1, \dots, p\}$. With a test of the form (9) $j = k_i$ is impossible since in that case one of the splitted sets would be empty.

Now, we replace each test of the form (9) by

$$X_i \leq \frac{x_i^{(j)} + x_i^{(j+1)}}{2}.$$

These changes will give us a binary decision tree on $\mathcal{X}$ that classifies the dataset $\mathcal{D}$ correctly.

As an example, let us assume we have a dataset with one continuous attribute X_1, while all other attributes are discrete. Let us further assume that X_1 has values

$$0.51 \quad 0.98 \quad 1.43 \quad 2.87 \quad 3.11$$

in the dataset. With these values, seen as discrete values, a decision tree is built which happens to have two nodes in which X_1 plays a role: in one node we have a test $X_1 \leq 0.98$ and in the other node we have $X_1 \leq 2.87$. Both tests are subsequently replaced by $X_1 \leq (0.98 + 1.43)/2$ or $X_1 \leq 1.205$ and $X_1 \leq (2.87 + 3.11)/2$ or $X_1 \leq 2.99$ respectively. This is similar to applying a continuity correction when approximating a discrete distribution by a continuous distribution in statistics.

As a final remark, note that in practice it is usually advisable to discretize continuous attributes, since working with too many values per attribute leads to prohibitive computing times.

3.3 An Indirect Method

In this subsection we present an alternative to the method of Section 3.2 using the concept of quasi-monotonicity. According to this method, we first build a quasi-monotone tree using an algorithm that appears to be somewhat faster than the direct algorithm. Subsequently, this quasi-monotone tree is tested for monotonicity. If it is monotone already, we are done. If not, we can use the repairing algorithm from Section 3.1 to fix it. As shown in Section 2.2 such a quasi-monotone decision tree can only have minor local non-monotonicities that are relatively easy to fix by splitting up a few more leaves. The main advantage of this method is that it is slightly faster than the direct method on most datasets. Another advantage is that it works for continuous attributes as well as for discrete attributes: we do not have to make special arrangements like those in Section 3.2.1. Just like the direct algorithm of Section 3.2, this method also needs a completely monotone dataset. The algorithm presented here for building quasi-monotone decision trees was proposed by Makino [10] for two class problems and was generalized by Potharst[12] to k-class problems. It was tested on artificial and real world data by these authors.

In this section our decision trees will have splits of the form $x_i < c$ for some $c \in X_i, 1 \leq i \leq p$. Thus, in each node the associated set $T \subset \mathcal{X}$ is split into the two subsets $T_\ell = \{\mathbf{x} \in T : x_i < c\}$ and $T_r = \{\mathbf{x} \in T : x_i \geq c\}$.

We shall now show how we can generate $\mathcal{D}$ a quasi-monotone binary decision tree $\mathcal{T}$ from a monotone dataset. As noted above, for such an algorithm we need a splitting rule $\mathcal{S}$, a stopping rule $\mathcal{H}$ and a labeling rule $\mathcal{L}$. 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 10.

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

split($T, \mathcal{D}$):
    if $\mathcal{H}(T, \mathcal{D})$ then
        assign class label $\mathcal{L}(T, \mathcal{D})$ to leaf $T$
    else
        begin
            $(T_\ell, T_r) := \mathcal{S}(T, \mathcal{D})$;
            $\mathcal{D}_\ell :=$ update($\mathcal{D}, \ell$);
            $\mathcal{D}_r :=$ update($\mathcal{D}, r$);
            split ($T_\ell, \mathcal{D}_\ell$);
            split ($T_r, \mathcal{D}_r$)
        end
```

Figure 10: Quasi-monotone Tree Induction Algorithm

In this algorithm outline again an *update rule* is mentioned. Like in

```
update $(\mathcal{D}, \text{side})$ :
    **if** side $= \ell$ **then**
        **begin**
            $\mathcal{D}_\ell := (D_\ell, \lambda_\ell)$;
            **return** $\mathcal{D}_\ell$
        **end**;
    **if** side $= r$ **then**
        **begin**
            $\mathcal{D}_r := (D_r, \lambda_r)$;
            **return** $\mathcal{D}_r$
        **end**
```

Figure 11: The update rule

the algorithms of Section 3.2, 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 quasi-monotonicity of the resulting tree. As opposed to the algorithm of Section 3.1 where we worked with only one *global* dataset, in this algorithm we work with *local* datasets in the following sense: each time we make a split the dataset is also split into two parts: a left dataset and a right dataset. To each of these datasets vital information from the other dataset is added by projecting points from the other side to this side. How this projection is executed will be described below.

Each time the splitting rule $\mathcal{S}$ splits a node T into a left node T_ℓ and a right node T_r, the dataset $\mathcal{D} = (D, \lambda)$ must accordingly be split into a dataset $\mathcal{D}_\ell = (D_\ell, \lambda_\ell)$ and a dataset $\mathcal{D}_r = (D_r, \lambda_r)$. This is done by the update rule, which is described in Figure 11.

Here D_ℓ and D_r are defined as

$$\begin{aligned} D_\ell &= (D \cap T_\ell) \cup \pi_\ell((D \cap T_r) \setminus D_{\max}), \\ D_r &= (D \cap T_r) \cup \pi_r((D \cap T_\ell) \setminus D_{\min}). \end{aligned}$$

In these formulae the projections π_ℓ and π_r are defined as follows. Suppose $S_{i,c}$ splits T into T_ℓ and T_r. Thus, $T_\ell = \{\mathbf{x} \in T : x_i < c\}$ and $T_r = \{\mathbf{x} \in T : x_i \geq c\}$. Then, for $\mathbf{x} \in T_r$ we define $\pi_\ell(\mathbf{x}) = \mathbf{x}' \in T_\ell$ as

$$x'_j = \begin{cases} x_j & \text{for } j \neq i \\ \max\{d \in \mathbb{D}_i : d < c\} & \text{for } j = i. \end{cases} \tag{10}$$

On the other hand, for $\mathbf{x} \in T_\ell$ we define $\pi_r(\mathbf{x}) = \mathbf{x}' \in T_r$ as

$$x'_j = \begin{cases} x_j & \text{for } j \neq i \\ c & \text{for } j = i. \end{cases} \tag{11}$$

Furthermore, for $A \subset X$ we define $\pi_\ell(A) = \bigcup_{a \in A} \pi_\ell(a)$ and $\pi_r(A) = \bigcup_{a \in A} \pi_r(a)$.

The sets $D_{\min}$ and $D_{\max}$ are defined as $D_{\min} = \{\mathbf{x} \in D : \lambda(\mathbf{x}) = c_{\min}\}$ and $D_{\max} = \{\mathbf{x} \in D : \lambda(\mathbf{x}) = c_{\max}\}$. Finally, the labelings λ_ℓ and λ_r are defined as follows:

$$\lambda_\ell(\mathbf{x}) = \begin{cases} \lambda(\mathbf{x}) & \text{for } \mathbf{x} \in D \cap T_\ell \\ \lambda^{\mathcal{D}}_{\min}(\mathbf{x}) & \text{for } \mathbf{x} \notin D \cap T_\ell, \end{cases} \tag{12}$$

and

$$\lambda_r(\mathbf{x}) = \begin{cases} \lambda(\mathbf{x}) & \text{for } \mathbf{x} \in D \cap T_r \\ \lambda^{\mathcal{D}}_{\max}(\mathbf{x}) & \text{for } \mathbf{x} \notin D \cap T_r. \end{cases} \tag{13}$$

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 with $T_\ell = \{\mathbf{x} \in T : x_i < c\}$ and $T_r = \{\mathbf{x} \in T : x_i \geq c\}$ for some $i \in \{1, \ldots, p\}$, and some $c \in \mathcal{X}_i$, while

$$T_\ell \text{ and } T_r \text{ are non-empty.} \tag{14}$$

Furthermore, the splitting rule must satisfy the following requirement: i and c must be chosen such that

$$\exists \mathbf{x}, \mathbf{x}' \in D \cap T \text{ with } \lambda(\mathbf{x}) \neq \lambda(\mathbf{x}'), \mathbf{x} \in T_\ell \text{ and } \mathbf{x}' \in T_r. \tag{15}$$

The stopping rule $\mathcal{H}(T, \mathcal{D})$ will return **true** only if the node T is homogeneous, i.e. if for all $\mathbf{x}, \mathbf{x}' \in D$ we have $\lambda(\mathbf{x}) = \lambda(\mathbf{x}')$. In that case node T is made into a leaf. Finally, the labeling rule $\mathcal{L}(T, \mathcal{D})$ will assign this uniform class to a new leaf.
Now we can formulate the main result of this subsection.

Theorem 2 *If $\mathcal{D} = (D, \lambda)$ is a monotone dataset on instance space $\mathcal{X}$ and if the functions $\mathcal{S}, \mathcal{H}, \mathcal{L}$ satisfy* (12), (13), (14) and (15)*, then the algorithm specified in Figure 10 and Figure 11 will generate a quasi-monotone decision tree $\mathcal{T}$ with $f_{\mathcal{T}} \in Q(\mathcal{D})$.*

Again, 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 $t = \{x_i < c\}$. As noted above, obvious candidates for attribute-selection and cut-off point calculation are the well-known impurity measures like entropy, Gini or the twoing rule, see [7]. Below, we will give an example that makes use of the entropy measure.

Before we prove the above theorem we will present the following lemma.

Lemma 8 *Let $T \subset X$ be a subset of X, and let $\mathcal{D} = (D, \lambda)$ be a monotone dataset with $D \subset T$. Furthermore, let $S_{i,c}$ be a split of T into T_ℓ and T_r, and let $\mathcal{D}_\ell$ and $\mathcal{D}_r$ be defined by* (12) *and* (13)*. Then we have*

a) $\mathcal{D}_\ell$ and $\mathcal{D}_r$ are monotone datasets on T_ℓ and T_r respectively.

Furthermore, let $f : T \to C$ be a $\mathbb{D}$-granular function on T, let $f_\ell = f|T_\ell$ (resp. $f_r = f|T_r$) be the restriction of f to T_ℓ (resp. T_r). Then we have

b) if f_ℓ is quasi-monotone with respect to $\mathcal{D}_\ell$ and f_r is quasi-monotone with respect to $\mathcal{D}_r$, then f is quasi-monotone with respect to $\mathcal{D}$.

Using this lemma, we easily prove the above theorem.

Proof of the theorem : Lemma 8a guarantees that with each split of a node T into T_ℓ and T_r we get two new datasets $\mathcal{D}_\ell$ and $\mathcal{D}_r$ that are both monotone. This guarantees the existence of a quasi-monotone f on T_ℓ and T_r. Since $\mathbb{D}$ is finite, the number of possible splits is finite, and the tree must necessarily be finite. Now, in each leaf T of the finished tree, we have: $\mathcal{D}_T$ is homogeneous. So $f_{\mathcal{T}}(\mathbf{x}) = k$, for all $\mathbf{x} \in T$. This state of affairs trivially satisfies the definition of quasi-monotonicity: $f_{\mathcal{T}}$ is quasi-monotone for $\mathcal{D}_T$ on leaf T. Since this is the case for each leaf, from Lemma 8b we infer that $f_{\mathcal{T}}$ must be quasi-monotone on X. $\square$.

We will now use the presented algorithm to generate a quasi-monotone decision tree for the dataset of Table 1. As an impurity measure we will use entropy. Starting in the root of the tree we have $T = \mathcal{X} = [000, 333)$. Since $\mathbb{D}_1 = \{0, 1, 2\}$, $\mathbb{D}_2 = \{0, 1\}$, $\mathbb{D}_3 = \{1, 2\}$ we have $3 \times 2 \times 2 = 12$ possible splits. Of these twelve only four satisfy criteria (14) and (15), namely $x_1 < 1$, $x_1 < 2$, $x_2 < 1$ and $x_3 < 2$. First, consider the split generated by the test $x_1 < 1$. Now, $\mathcal{D}_\ell = \{$011:0, 002:1, 012:1$\}$. The last element of this dataset stems from the projection of the element 112 : 2 of the original dataset $\mathcal{D}$, using the fact that $\lambda^{\mathcal{D}}_{\min}(012) = 1$. Next, $\mathcal{D}_r = \{$102:2, 112:2, 202:2, 212:3$\}$ where the first element stems from the projection of 002:1 and the fact that $\lambda^{\mathcal{D}}_{\max}(102) = 2$. Note, that the elements 001 and 212 of $\mathcal{D}$ are not projected since they belong to $D_{\min}$ and $D_{\max}$ respectively. The entropy of this split can be calculated as $\frac{3}{7} \times 0.9183 + \frac{4}{7} \times 0.8113 = 0.8571$, so the decrease in entropy of the other three splits $x_1 < 2$, $x_2 < 1$ and $x_3 < 2$ can be calculated as 0.5886, 0.6712 and 1.0647 respectively. Since the first and the last split give the highest decrease in entropy, we pick just one of these, e.g. $x_1 < 1$, as first split of the decision tree.

Proceeding with the left node $T = [000, 133)$ with dataset $\{001{:}0,\ 002{:}1,\ 012{:}1\}$, we first note that only two possible splits satisfy criteria (14) and (15), namely $x_2 < 1$ and $x_3 < 2$. The second of these gives the greatest decrease in entropy and leads to a homogeneous $\mathcal{D}_\ell$ and $\mathcal{D}_r$, namely $\mathcal{D}_\ell = \{011{:}0,\ 011{:}0\}$ and $\mathcal{D}_r = \{002{:}1,\ 012{:}1\}$. Thus, node $T = [000, 133)$ is split into the leaves $[000, 132)$ with class 0 and $[002, 133)$ with class 1.

Proceeding in this manner we end up with the decision tree in Figure 12. This decision tree is in fact not only quasi-monotone but even monotone.

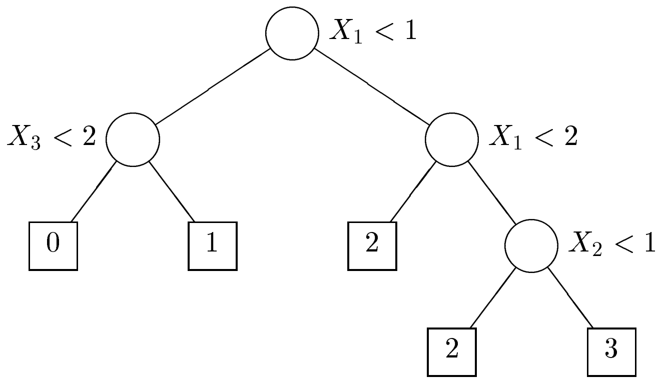

Figure 12: (Quasi-)monotone Decision Tree for the Bank Loan dataset

Furthermore, it represents the same allocation rule as the decision tree of Figure 5.

4 Methods for non-monotone data

The algorithms discussed so far work for monotone datasets. Even if the true underlying relation is monotone, the observed data may, as a consequence of noise, not be. Furthermore, sometimes we simply require that the allocation rule be monotone, even if we believe that the underlying relation is not. In that case the task is to find a monotone model with good predictive performance.

In this section we look at two approaches that can handle non-monotone and inconsistent datasets.

4.1 The Weighted Sum Method

Ben-David [2], proposes a tree induction algorithm that is similar to well-known algorithms such as C4.5 and CART. The important difference with those algorithms is that the splitting rule includes a measure of the degree of monotonicity of the tree in addition to the usual impurity measure.

To this end a $k \times k$ symmetric non-monotonicity matrix M is defined, where k equals the number of leaves of the tree constructed so far. The m_{ij} element of M equals 1 if leaf T_i is non-monotonic with respect to leaf T_j and 0 otherwise. Clearly, the diagonal elements of M are 0. A non-monotonicity index I is defined as follows

$$I = \frac{W}{k^2 - k},$$

where W denotes the sum of M's entries, and $k^2 - k$ is the maximum possible value of W for any tree with k leaves [2]. Note however that this maximum can only be achieved if there are at least k distinct classes.

Based on this non-monotonicity index the order-ambiguity-score of a decision tree is defined as follows

$$A = \begin{cases} 0 & \text{if } I = 0 \\ -(\log_2 I)^{-1} & \text{otherwise} \end{cases}$$

Finally the splitting rule is redefined to include the order- ambiguity-score

$$S = E + \rho A,$$

where S denotes the total-ambiguity-score to be minimized, E is the well-known entropy measure, and ρ is a parameter that expresses the importance of monotonicity relative to inductive accuracy. The quality of each split is determined by computing its total-ambiguity-score, where A is the order-ambiguity-score of the tree that results from the split.

Note that W is a rather crude measure of the degree of non-monotonicity of a tree, since each non-monotonic leaf pair has equal weight. A possible improvement would be to weight the different leaves according to their probability of occurrence. The matrix M' could now be defined as follows. The

m_{ij} element of M' equals $p(T_i) \times p(T_j)$ if leaf T_i is non-monotonic with respect to leaf T_j and 0 otherwise, where $p(T_i)$ denotes the proportion of cases in leaf T_i. The non-monotonicity index becomes

$$I' = \frac{W'}{(k^2 - k)/k^2} = \frac{W'}{1 - 1/k},$$

where W' is again the sum of the entries of M', and the maximum is attained when all possible leaves are non-monotonic with respect to each other and occur with equal probability $1/k$. W' is an estimate of the probability that if we draw two points at random from the feature space, these points turn out to lie in two leaves that are non-monotonic with respect to each other. Note that $p(T_i) \times p(T_j)$ is an upperbound for the degree of non-monotonicity between node T_i and T_j because not all elements of T_i and T_j have to be non-monotonic with respect to each other.

The most straightforward way to measure the degree of non-monotonicity of a tree would be to use it to label all data, and simply count the number of non-monotonic pairs created by the labeling. This is however computationally rather demanding since this should be performed for the collection of trees that results from applying each possible split.

4.2 A Generate-and-Test Approach

The use of a measure of monotonicity in determining the best split, as discussed in the previous section, has certain drawbacks. Monotonicity is a *global* property, i.e. it involves a relation between different leaf nodes of a tree. If the degree of monotonicity is measured for each possible split during tree construction, the *order* in which nodes are expanded becomes important. For example, a depth-first search strategy will generally lead to a different tree then a breadth-first search. Also, and perhaps more importantly, a non-monotonic tree may become monotone after additional splits.

In view of these drawbacks, we consider an alternative approach in this section. Rather than *enforcing* monotonicity during tree construction, we generate many different trees and *check* if they are monotonic. The collection of trees may be obtained by drawing bootstrap samples from the training data, or making different random partitions of the data in a training and test set. This approach allows the use of a standard tree algorithm except that the minimum and maximum elements of the nodes have to be recorded during tree construction, in order to be able to check whether the final tree is monotone. This approach has the additional advantage that one can estimate to what extent the assumption of monotonicity is correct, by looking at the proportion of monotone trees versus non-monotone trees obtained.

The tree algorithm used is in many respects similar to the CART program as described in [7]. The program makes binary splits and uses the gini-index as splitting criterion. Furthermore it uses cost-complexity pruning [7] to generate a nested sequence of trees from which the best one is selected on the basis of test set performance. During tree construction, the algorithm records the minimum and maximum element for each node. These are used to check the whether a tree is monotone. In Figure 13 we give pseudo-code for the tree construction algorithm with recording of the corner elements of each node.

```
growtree($\mathcal{X}$,$\mathcal{D}$ : training sample):
    $T_0 := \mathcal{X}$
    **for** $i \in \{1, \ldots, p\}$
        $\min_i(T_0) := -\infty$
        $\max_i(T_0) := +\infty$
    split($T_0$,min($T_0$), max($T_0$), $\mathcal{D}$)

split($T$, min($T$), max($T$), $\mathcal{D}$):
    **if** leaf($T, \mathcal{D}$) **then**
      assign class label $\mathcal{L}(T, \mathcal{D})$ to $T$
    **else**
      $\mathcal{S} :=$ allsplits($T, \mathcal{D}$)
      $(j^*, c^*) := \arg\max_{(j,c)\in\mathcal{S}}$ quality($j, c, T, \mathcal{D}$)
      $T_\ell := \{\mathbf{x} \in T : x_{j^*} < c^*\}$
      $T_r := \{\mathbf{x} \in T : x_{j^*} \geq c^*\}$
      **for** $i \in \{1, \ldots, p\}$
          $\min_i(T_\ell) := \min_i(T)$
          $\max_i(T_r) := \max_i(T)$
          **if** $i = j^*$
              $\max_i(T_\ell) := c^*$
              $\min_i(T_r) := c^*$
          **else**
              $\max_i(T_\ell) := \max_i(T)$
              $\min_i(T_r) := \min_i(T)$
      split ($T_\ell$, min($T_\ell$), max($T_\ell$), $\mathcal{D}$)
      split ($T_r$, min($T_r$), max($T_r$), $\mathcal{D}$)
```

Figure 13: Tree Induction Algorithm with recording of node corners

The function *leaf* determines whether a node should be turned into a leaf. This is the case when the node is homogeneous, all examples in the node have identical attribute values, or the node contains too few examples to be split any further. A class label is assigned to the leaf, by default based on the majority rule.

The minimum and maximum element of root node T_0 are set to $-\infty$ and ∞ respectively. The updating of corner elements proceeds as follows. The minimum of T_ℓ is identical to that of T, and the same goes for the maximum of T_r. For the maximum of T_ℓ and the minimum of T_r, x_{j^*} (the split attribute) is set to c^* (the split value) and for all other attributes they are the same as the maximum and minimum of T respectively.

Determining the non-monotonic pairs of leaf nodes is straightforward: take any pair (T, T') with $f_{\mathcal{T}}(T) > f_{\mathcal{T}}(T')$ and check if $\min(T) < \max(T')$. If this is the case, then add (T, T') to the list of non-monotonic leaf-pairs.

In the next section we illustrate this algorithm by applying it to an economic dataset concerning house prices.

Symbol	Definition
DISTR	type of district, four categories ranked from bad to good
SURF	total area including garden
RM	number of bedrooms
TYPE	1. apartment
	2. row house
	3. corner house
	4. semidetached house
	5. detached house
	6. villa
VOL	volume of the house
STOR	number of storeys
GARD	type of garden, four categories ranked from bad to good
GARG	1. no garage
	2. normal garage
	3. large garage

Table 2: Definition of attributes for house pricing example

4.3 Application to House Pricing

In this section we illustrate the resampling approach described in the previous section. We discuss its application to the prediction of the price-category of a house in the city of Den Bosch (a medium sized Dutch city with approximately 120,000 inhabitants).

The attributes $x_1, x_2, \ldots, x_p$ are characteristics of the house. They have been selected on the basis of interviews with experts of local house brokers, and advertisements offering real estate in local magazines. The monotonicity constraint makes sense for this application, since the better the characteristics of a house, the higher the asking price. The most important attributes are listed in Table 2.

It is a relatively small data set with only 119 observations used for illustrative purposes only. Of all 7021 distinct pairs of observations, 2217 are comparable, and 78 are non-monotonic. For the purpose of this study we have discretized the dependent variable (asking price) into the classes *below median* (euro 157,955) and *above median*. After this discretization of the dependent variable 9 pairs of observations are non-monotonic.

In order to determine the effect of application of the monotonicity constraint we repeated the following experiment 100 times. The dataset was randomly partitioned (within classes) into a training set (60 observations) and test set (59 observations). The training set was used to construct a sequence of trees using cost-complexity pruning. From this sequence the best tree was selected on the basis of error rate on the test set (in case of a tie, the smallest tree was chosen). Finally, it was checked whether the tree was monotone and if not, the upperbound W' for the degree of monotonicity (as defined in Section 4.1) was computed.

Out of the 100 trees thus constructed, 57 turned out to be monotone and 43 not. The average misclassification rate of the monotonic trees was 14.93% against 14.94% for the non-monotonic trees. Thus, the predicitive accuracy was virtually identical.

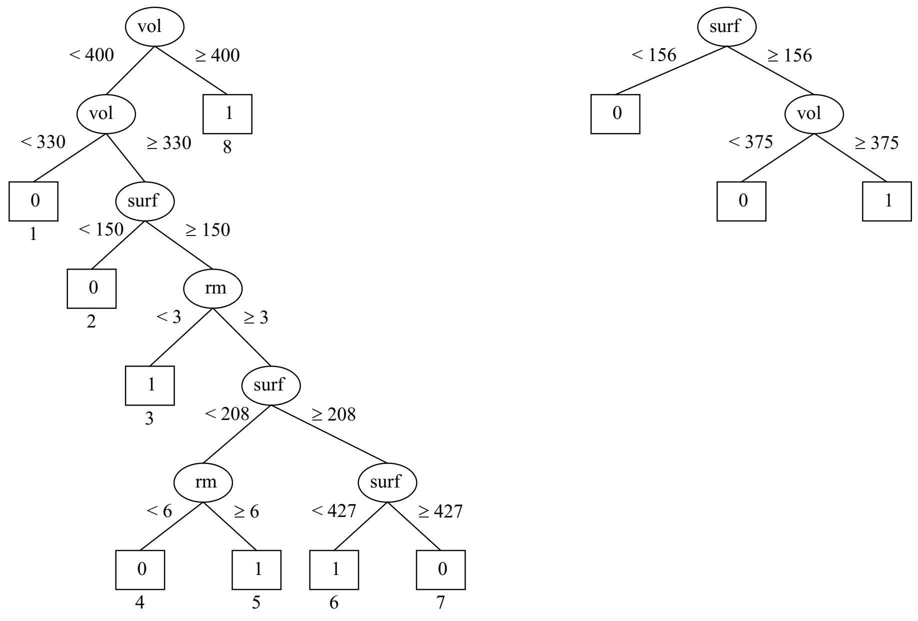

Figure 14: Example of non-monotonic (left) and monotonic tree

Figure 14 depicts one of the 43 non-montonic trees (left part of the figure) and one of the 57 monotonic trees (right part of the figure) obtained in the experiment. Class label 0 corresponds to prices below the median and label 1 to prices above median. The number of a leaf is given directly below it. It is easily verified that the leaf-pairs (3,4), (3,7), (5,7) and (6,7) of the left tree are non-monotone. The degree of non-monotonicity W' (see Section 4.1) of this tree is only about 1%. The right tree is monotone and has only 3 leaf nodes. The estimated error of the non-monotonic tree shown is 15.3%, and the estimated error of the monotonic tree 13.6%.

The average degree of non-monotonicity W' of the non-monotonic trees was about 1.6%, which is quite low, the more if we take into consideration that this is an upper bound. Another interesting comparison is between the average sizes of the trees. On average, the monotonic trees had about 3.19 leaf nodes, against 6.95 for the non-monotonic trees. Thus, the monotonic trees are considerably smaller and therefore easier to understand. The variability around the mean number of leaf nodes can be used as a measure of the stability of the trees generated. For the monotone trees, the variance of the number of leaf nodes was 0.91 against 5.05 for the non-monotone trees. Clearly then the monotone trees are more stable upon repeated sampling than there non-monotone counterparts.

5 Discussion

Monotonicity is a common type of constraint on models in data mining. Furthermore, monotonicity may be an important requirement for explaining and justifying model outcomes. We have investigated the use of monotonicity constraints in classification tree algorithms.

We have presented algorithms that work on monotone data only, as well as algorithms that work on both monotone and non-monotone data. The former could be made more widely applicable by developing sensible methods to make a non-monotone data set monotone by making as few adjustments to the data as possible.

For non-monotone data we have presented two algorithms, the weighted-sum method and a generate-and-test algorithm. In preliminary experiments with the generate-and-test algorithm on house pricing data, we have found that the predictive performance of monotone trees was comparable to the performance of the non-monotone trees. However, the monotone trees were much simpler and therefore more insightful and easier to explain. Furthermore, the monotone trees proved to be more stable upon repeated sampling.

This provides interesting prospects for applications where monotonicity is an absolute requirement, such as in many selection decision models.

An interesting, as yet unexplored, approach for non-monotone data would be to use a pruning method that prunes towards monotone subtrees of the initially grown tree. One could create a nested sequence of monotone subtrees of the initial tree, and select from this sequence the tree with the best predictive accuracy on a test set. Another interesting extension of the work surveyed in this paper is to consider multivariate classification trees, where each split may be based on more than one attribute.

References

[1] A. Ben-David. Automatic generation of symbolic multi-attribute ordinal knowledge-based dsss: methodology and applications. *Decision Sciences*, 23:1357–1372, 1992.

[2] A. Ben-David. Monotonicity maintenance in information-theoretic machine learning algorithms. *Machine Learning*, 19:29–43, 1995.

[3] A. Ben-David, L. Sterling, and Y. Pao. Learning and classification of monotonic ordinal concepts. *Computational Intelligence*, 5:45–49, 1989.

[4] J. Bioch. Dualization, decision lists and identification of monotone discrete functions. *Annals of Mathematics and Artificial Intelligence*, 24:69–91, 1998.

[5] J. Bioch and T. Ibaraki. Complexity of identification and dualization of positive boolean functions. *Information and Computation*, 123:50–63, 1995.

[6] J. Bioch and V. Popova. Rough sets and ordinal classification. In A. S. H. Arimura, S. Jain, editor, *Algorithmic Learning Theory*, Lecture Notes in Artificial Intelligence 1968, pages 291–305. Springer, 2000.

[7] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees (CART).* Wadsworth, 1984.

[8] H. Daniels and B. Kamp. Application of MLP networks to bond rating and house pricing. *Neural Computation and Applications*, 8:226–234, 1999.

[9] O. Harrison and D. Rubinfeld. Hedonic prices and the demand for clean air. *Journal of Environmental Economics and Management*, 53:81–102, 1978.

[10] K. Makino, T. Susa, K. Yano, and T. Ibaraki. Data analysis by positive decision trees. In Y. Kambayashi et al., editors, *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications (CODAS)*, pages 257–264, Kyoto, Japan, December 1996. World Scientific.

[11] H. Mukarjee and S. Stern. Feasible nonparametric esimation of multiargument monotone functions. *Journal of the American Statistical Association*, 89(425):77–80, 1994.

[12] R. Potharst. *Classification using Decision Trees and Neural Nets.* PhD thesis, Erasmus University Rotterdam, 1999.

[13] R. Potharst and J. Bioch. Decision trees for ordinal classification. *Intelligent Data Analysis*, 4(2):97–112, 2000.

[14] J. Quinlan. *C4.5 Programs for Machine Learning.* Morgan Kaufmann, 1993.

[15] T. Robertson, F. Wright, and R. Dykstra. *Order Restricted Statistical Inference.* Wiley, 1988.

[16] S. Wang. A neural network method of density estimation for univariate unimodal data. *Neural Computation & Applications*, 2:160–167, 1994.