Quasi-monotone decision trees for ordinal classification

Rob Potharst, Jan C. Bioch, René van Dordregt

Department of Computer Science
Erasmus University Rotterdam
P.O. Box 1738, 3000 DR Rotterdam
The Netherlands
{potharst, bioch}@few.eur.nl
Technical Report EUR-FEW-CS-98-01

Abstract

In many classification problems the domains of the attributes and the classes are linearly orderded. Since the known decision tree methods generate non-monotone trees, these methods are not suitable for monotone classification problems. We already provided order-preserving tree-generation algorithms for multi-attribute classification problems with k linearly ordered classes in a previous paper. For real-world datasets it is important to consider approximate solutions to handle problems like speed, tree-size and noise. In this report we develop a new decision tree algorithm that generates quasi-monotone decision trees. This algorithm outperforms classical algorithms such as those of Quinlan with respect to prediction, and beats algorithms that generate strictly monotone decision trees with respect to speed. This report contains proofs of all presented results.

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 linearly ordered. For these class of problems classification rules often need to be order-preserving. In that case we have a monotone classification problem.

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.* [8] for the two-class problem with continuous attributes. In [7, 5] we already provided several algorithms for monotone classification problems with k classes and discrete or continuous domains. With one of these algorithms we can start from an already given non-monotone tree generated by any tree generating algorithm and end with a monotone decision tree. For practical reasons and real world datasets (speed, tree-size, noise) it appeared that it is also interesting to consider algorithms that generate decision trees which are not strictly monotone. In this report we present an algorithm that induces a socalled quasi-monotone decision tree. This concept is introduced by Makino *et al.* as an approximate solution to the two-class problem. Furthermore, they also generate monotone decision trees for the 2-class problem by repairing quasi-monotone decision trees. In this report we generalize the concept of quasi-monotonicity and we provide an algorithm that solves the k-class problem. This algorithm produces quasi-monotone decision trees that can be used as an approximate solution or as the input of one of our algorithms that generates a strictly monotone decision tree.

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 no 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.

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

2 Monotone and quasi-monotone classification rules

In this report we shall suppose our input space $X = X_1 \times X_2 \times \ldots \times X_n$ to be a product space of linearly ordered coordinate spaces $X_i, 1 \leq i \leq n$. Thus, X is a partially ordered space: for $x, y \in X$ we have $x \leq y$ iff $x_i \leq y_i$ for $1 \leq i \leq n$. If for some $x, y \in X$ we have neither $x \leq y$ nor $y \leq x$, we call y and x incomparable.

Furthermore, we have a finite ordered set $C = \{c_{\min}, \ldots, c_{\max}\}$ of classes. A function $f : X \to C$ is called a *classification rule*. A classification rule f is said to be *monotone* if

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

holds for all $x, y \in X$.

A monotone classification problem is the problem of finding a monotone classification rule that is only partially known, i.e. it is only known in a finite subset D of X.

Let D be a finite subset of X and let $\lambda : D \to C$ be the *labeling* function that assigns to each data point $x \in D$ its class $\lambda(x)$. Then the pair $\mathcal{D} = (D, \lambda)$ is called a *dataset*, and the pairs $(x, \lambda(x))$ with $x \in D$ are called the *data elements* of dataset $\mathcal{D}$. A dataset $\mathcal{D} = (D, \lambda)$ is *monotone* if

$$x \leq y \Rightarrow \lambda(x) \leq \lambda(y) \tag{2}$$

for all $x, y \in D$. If $\mathcal{D} = (D, \lambda)$ is a dataset, then $f : X \to C$ is an *extension* of $\mathcal{D}$, if $f(x) = \lambda(x)$ for all $x \in D$. For a monotone dataset $\mathcal{D}$, let $M(\mathcal{D})$ be the set of all monotone extensions of $\mathcal{D}$. Thus, if the monotone classification problem is the problem to find a monotone classification rule that is compatible with dataset $\mathcal{D}$, then $M(\mathcal{D})$ is the set of solutions to this problem.

As can been seen in Makino *et al.* [8] 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 we will generalize here to the k class problem. A classification rule f will be called *quasi-monotone* for dataset $\mathcal{D} = (D, \lambda)$ if for all $x, y \in X$

$$x \leq y \text{ and } [x, y] \cap D \neq \emptyset \Rightarrow f(x) \leq f(y). \tag{3}$$

It is clear from this definition that quasi-monotonicity for dataset $\mathcal{D} = (D, \lambda)$ implies quasi-monotonicity for dataset $\mathcal{D}' = (D', \lambda')$, if $D' = D$. 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. If we introduce the following notation, we can give useful characterizations of the concepts of quasi-monotonicity and monotonicity and of the sets $M(\mathcal{D})$ and $Q(\mathcal{D})$.

Definition 1 If $D \subset X$ and $\lambda : D \to C$ is a partially defined function on X, we define the functions $\hat{\lambda} : X \to C$ and $\check{\lambda} : X \to C$ as follows[1]: for any $x \in X$

$$\hat{\lambda}(x) = \begin{cases} \max\{\lambda(y) : y \leq x, y \in D\} & \text{if } x \geq y \text{ for some } y \in D \\ c_{\min} & \text{otherwise,} \end{cases}$$

and

$$\check{\lambda}(x) = \begin{cases} \min\{\lambda(y) : y \geq x, y \in D\} & \text{if } x \leq y \text{ for some } y \in D \\ c_{\max} & \text{otherwise.} \end{cases}$$

Applying this definition to the labeling λ of a monotone dataset $\mathcal{D} = (D, \lambda)$, gives us the following characterization of $M(\mathcal{D})$ and $Q(\mathcal{D})$:

Lemma 1 *If $\mathcal{D} = (D, \lambda)$ is a monotone dataset, and $\mathcal{F}$ is the set of classification rules $f : X \to C$, then*

a) for all $x \in X$: $\hat{\lambda}(x) \leq \check{\lambda}(x)$,

b) $\hat{\lambda}, \check{\lambda} \in M(\mathcal{D})$

c) $M(\mathcal{D}) = \{f \in \mathcal{F} : \hat{\lambda} \leq f \leq \check{\lambda}, f \, monotone\}$

d) $Q(\mathcal{D}) = \{f \in \mathcal{F} : \hat{\lambda} \leq f \leq \check{\lambda}, f \, quasi\text{-}monotone \ for \ \mathcal{D}\}$.

Proof: The a), b) and c) part of this lemma have already been proved in our technical report [7]. So here, we only prove d).

We first prove the $\subset$ part. For this we need to prove that for every quasi-monotone f we have $\hat{\lambda} \leq f \leq \check{\lambda}$. we will first show that for all $x \in X$, we have $f(x) \geq \hat{\lambda}(x)$. If $\hat{\lambda}(x) = c_{\min}$, the inequality is obvious. So we concentrate on the case that for some $y \in D$ we have $y \leq x$. In that case

$$\lambda(y) = f(y) \leq f(x) \tag{4}$$

[1] Since C is a finite set, we take here and in the remainder of this report a maximum and minimum rather than a supremum and infimum.

where the equality stems from the fact that f is an extension of λ, and the inequality follows from the quasi-monotonicity of f with respect to $\mathcal{D}$. Note, that since $y \in D$, we must have $[y, x] \cap D \neq \emptyset$. Since (1) holds for all $y \in D$ with $y \leq x$, it must also hold for the y which the left hand side is maximal. So, by the definition of $\hat{\lambda}$ we have $\hat{\lambda}(x) \leq f(x)$. A similar reasoning shows that $f(x) \leq \check{\lambda}(x)$. This proves the $\subset$ part of the lemma.

To prove the $\supset$ part of the lemma, we need to show that a quasi-monotone f that satisfies $\hat{\lambda} \leq f \leq \check{\lambda}$ must be an extension of λ. Now, it is easy to see from the above Definition 1 that for $x \in D$:

$$\hat{\lambda}(x) = \lambda(x) = \check{\lambda}(x).$$

Thus, if $\hat{\lambda} \leq f \leq \check{\lambda}$, then $f(x) = \lambda(x)$ for $x \in D$, which means that f is an extension of λ. □

Thus, the minimal (maximal) monotone classification rule $\hat{\lambda}(\check{\lambda})$ for a dataset $\mathcal{D}$, is also the minimal (maximal) quasi-monotone classification rule. The following lemma will be used in the proof of the correctness of our algorithms.

Lemma 2 *Let $\mathcal{D} = (D, \lambda)$ be a monotone dataset and let $x \in X$ be an inputvector with $x \notin D$. If we define $\lambda^+(x) \in C$ such that $\hat{\lambda}(x) \leq \lambda^+(x) \leq \check{\lambda}(x)$ and $\lambda^+(y) = \lambda(y)$ for $y \in D$, then for the dataset $\mathcal{D}^+ = (D^+, \lambda^+)$ with $D^+ = D \cup \{x\}$ the following statements hold:*

a) $\mathcal{D}^+$ is a monotone dataset,

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

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

Proof:

a) If $y \in D$ with $y \leq x$, then, according to the definition of $\hat{\lambda}$, we have $\lambda^+(y) = \lambda(y) \leq \hat{\lambda}(x) \leq \lambda^+(x)$. In the same vein, if $y \in D$ with $y \geq x$, then by the definition of $\check{\lambda}$ we have $\lambda^+(y) = \lambda(y) \geq \check{\lambda}(x) \geq \lambda^+(x)$. Both cases satisfy (2), so $\mathcal{D}^+$ is equally monotone as $\mathcal{D}$.

b) If $f \in M(\mathcal{D}^+)$, then f is an extension of λ^+, and, by implication, it is an extension of λ. Thus, $f \in M(\mathcal{D})$.

c) Let $f \in Q(\mathcal{D}^+)$, then f is an extension of both λ^+ and λ. Furthermore, f is quasi-monotone for $\mathcal{D}^+$, so for $\mathcal{D}$ as well, since $D \subset D^+$. Thus, $f \in Q(\mathcal{D})$. □

If $f : X \to C$ is any classification rule, the above definition for the special case $D = X$ boils down to

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

and

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

for $x \in X$. In that case, we can easily see that, for all $x \in X$

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

and $\check{f}$ and $\hat{f}$ are monotone functions. 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 useful characterizations of monotonicity and quasi-monotonicity.

Lemma 3 *If $f : X \to C$ is an arbitrary classification rule, then*

$$\begin{aligned} f \textit{ monotone} \quad &\Leftrightarrow \textit{ for all } x \in X : \check{f}(x) = \hat{f}(x) \\ &\Leftrightarrow \textit{ for all } x \in X : \check{f}(x) = f(x) \\ &\Leftrightarrow \textit{ for all } x \in X : \hat{f}(x) = f(x) \end{aligned}$$

Proof: The first equivalence is an easy consequence of the other two equivalences, together with (5). We will now prove the second equivalence. The third can be treated analogously.

We start with the $\Rightarrow$ part. If $x \leq y$ then, since f is monotone, we also have $f(x) \leq f(y)$. So according to the definition of $\check{f}$, $f(x) \leq \check{f}(x)$. Together with (5), this implies the right side of the equivalence. In order to prove the converse, we just note that if $f(x) \leq \check{f}(x)$, then for each $y \geq x$, we must have $f(y) \geq f(x)$, which is the definition of monotonicity. $\square$

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

$$\begin{aligned} f \textit{ quasi-monotone for } \mathcal{D} \quad &\Leftrightarrow \textit{ for all } x \in D : \check{f}(x) = \hat{f}(x) \\ &\Leftrightarrow \textit{ for all } x \in D : \hat{f}(x) \leq \lambda(x) \leq \check{f}(x). \end{aligned}$$

Proof: The equivalence of both right members follows easily from (5) and the fact that for $x \in D$ we have $f(x) = \lambda(x)$. So we are ready if we prove the equivalence of the definition of quasi-monotonicity with the second of these criteria. Let f be a quasi-monotone extension of λ. If $y \leq x$ with $x \in D$, then $[y, x] \cap D \neq \emptyset$, so we must have $f(y) \leq f(x) = \lambda(x)$. Since this inequality holds for every $y \leq x$, we also have $\hat{f}(x) \leq \lambda(x)$. Analogously, we must have $\lambda(x) \leq \check{f}(x)$. This proves one part.

Conversely, let $x \leq y$ be such that $[x, y] \cap D$ is non-empty. Then we must have $x \leq z \leq y$, with $z \in D$. By hypothesis we then have $\hat{f}(z) \leq \lambda(z) \leq \check{f}(z)$. Thus, using the definition of $\hat{f}$ and $\check{f}$, we have $f(x) \leq f(z) \leq f(y)$. It follows, that f is quasi-monotone. $\square$

In the rest of this report we will only look at a certain type of classification rules, namely binary decision trees. In [7] we developed algorithms for constructing monotone decision trees. In this report, we will show how to construct quasi-monotone decision trees.

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

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

and

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

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 $x \in X$ with $x \geq d_{\min}$ we define the $\mathbb{D}$-approximation $\tilde{x}$ of x as follows:

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

and

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

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

3 Quasi-monotone decision tree induction

In this section we will show how quasi-monotone decision trees can be induced from a monotone dataset. In this report 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. Furthermore, we will only consider *binary* trees. For binary trees, at each node a split is made using a test of the form $x_i < c$ for some $c \in X_i, 1 \leq i \leq n$. Thus, for a binary tree, in each node the associated set $T \subset X$ is split into the two subsets $T_\ell = \{x \in T : x_i < c\}$ and $T_r = \{x \in T : x_i \geq c\}$. An example of a univariate binary decision tree is the following (each $X_i = [0, 3)$):

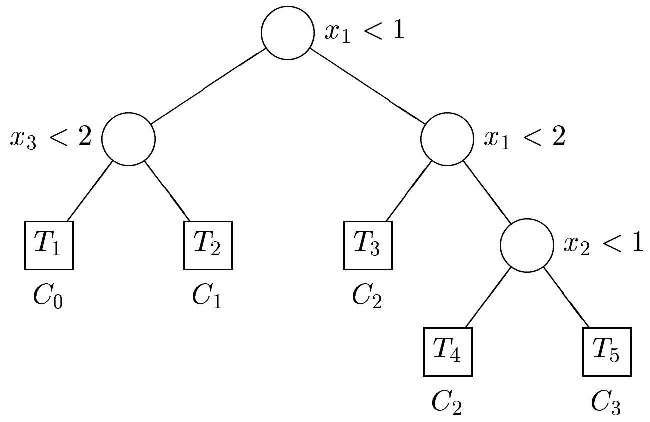

Figure 1: Univariate Binary Decision Tree: Example

This tree splits the input space $X = [0, 3)^3$ into the five regions

$$\begin{aligned}
T_1 &= \{x \in X : x_1 < 1, x_3 < 2\} \\
T_2 &= \{x \in X : x_1 < 1, x_3 \geq 2\} \\
T_3 &= \{x \in X : 1 \leq x_1 < 2\} \\
T_4 &= \{x \in X : 1 \leq x_1 < 2, x_2 < 1\} \\
T_5 &= \{x \in X : 1 \leq x_1 < 2, x_2 \geq 1\}
\end{aligned}$$

while region T_1 is classified as C_0, region T_1 as C_1, etc.

We shall now show how we can generate from a monotone data set $\mathcal{D}$ a quasi-monotone binary decision tree $\mathcal{T}$. This process is also called *inducing* a binary decision tree $\mathcal{T}$ from a dataset $\mathcal{D}$. According to Breiman *et al.* [6] 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.

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(X, D_0):
    split(X, D_0)

split(T, var D):
    if H(T, D) then
        assign class label L(T, D) to leaf T
    else
        begin
            (T_ℓ, T_r) := S(T, D);
            D_ℓ := update(D, ℓ);
            D_r := update(D, r);
            split (T_ℓ, D_ℓ);
            split (T_r, D_r)
        end
```

Figure 2: Quasi-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 quasi-monotonicity of the resulting tree.

As noted in the beginning of this section, 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 2 to give a complete description of the algorithm under consideration.

Each time the splitting rule 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 the following figure:

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 3: The update rule

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, $T_\ell = \{x \in T : x_i < c\}$ and $T_r = \{x \in T : x_i \geq c\}$. Then, for $x \in T_r$ we define $\pi_\ell(x) = x' \in T_\ell$ as

$$x'_j = \begin{cases} x_j & \text{for } j \neq i \\ \max\{d \in \mathbb{D}_i : d < c\} & \text{for } j = i. \end{cases} \tag{5}$$

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

$$x'_j = \begin{cases} x_j & \text{for } j \neq i \\ c & \text{for } j = i. \end{cases} \tag{6}$$

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

$$\lambda_\ell(x) = \begin{cases} \lambda(x) & \text{for } x \in D \cap T_\ell \\ \hat{\lambda}(x) & \text{for } x \notin D \cap T_\ell, \end{cases} \tag{7}$$

and

$$\lambda_r(x) = \begin{cases} \lambda(x) & \text{for } x \in D \cap T_r \\ \check{\lambda}(x) & \text{for } x \notin D \cap T_r. \end{cases} \tag{8}$$

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

$$T_\ell \text{ and } T_r \text{ are non-empty.} \tag{9}$$

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

$$\exists x, y \in D \cap T \text{ with } \lambda(x) \neq \lambda(y), x \in T_\ell \text{ and } y \in T_r. \tag{10}$$

The stopping rule $\mathcal{H}(T, \mathcal{D})$ will return **true** only if the node T is homogeneous, i.e. if for all $x, y \in D$ we have $\lambda(x) = \lambda(y)$. 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 theorem of this report.

Theorem *If $\mathcal{D} = (D, \lambda)$ is a monotone dataset on input space $\mathcal{X}$ and if the functions $\mathcal{S}, \mathcal{H}, \mathcal{L}$ satisfy* (3), (4), (5) and (6), *then the algorithm specified in Figure 2 and 3 will generate a quasi-monotone decision tree $\mathcal{T}$ with $f_{\mathcal{T}} \in Q(\mathcal{D})$.*

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 $t = \{x_i < c\}$. Obvious candidates for attribute-selection and cut-off point calculation are the well-known impurity measures like entropy, Gini or the twoing rule, see [6].

Before we prove the above theorem we will present the following lemma, on which the proof rests.

Lemma 5 *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* (7) *and* (8). *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 $\mathcal{D}$.

Proof: The a) part is a direct consequence of Lemma 2 and the choices that we made for the labeling of newly added data points in (7) and (8). Obviously, in every case a newly added data point gets a class-label that satisfies the assumption of Lemma 2. Thus, all newly formed datasets stay monotone, and remain compatible with the original dataset.

Next, we prove the b) part. Thus, from

$$\text{for all } x \in D_\ell : \hat{f}_\ell(x) \leq \lambda_\ell(x) \leq \check{f}_\ell(x) \tag{11}$$

and

$$\text{for all } x \in D_r : \hat{f}_r(x) \leq \lambda_r(x) \leq \check{f}_r(x) \tag{12}$$

it must be deduced, that

$$\text{for all } x \in D : \hat{f}(x) \leq \lambda(x) \leq \check{f}(x). \tag{13}$$

Note, that

$$\hat{f}_\ell(x) = \max\{f_\ell(y) : y \leq x\} = \max\{f(y) : y \in T_\ell, y \leq x\}$$

while

$$\hat{f}(x) = \max\{f(y) : y \in T, y \leq x\}.$$

Because of the way T is split into T_ℓ and T_r we have: if $x \in T_\ell$ and $y \leq x$, then also $y \in T_\ell$. Thus, we have

$$\begin{cases} \hat{f}_\ell(x) = \hat{f}(x) & \text{for } x \in T_\ell \\ \hat{f}_\ell(x) \leq \hat{f}(x) & \text{for } x \in T_r \end{cases} \tag{14}$$

and analogously,

$$\begin{cases} \check{f}_\ell(x) = \check{f}(x) & \text{for } x \in T_\ell \\ \check{f}_\ell(x) \geq \check{f}(x) & \text{for } x \in T_r. \end{cases} \tag{15}$$

By interchanging ℓ and r, we get analogous (in)equalities for $\hat{f}_r$ and $\check{f}_r$.

By the construction of $\mathcal{D}_\ell$ and $\mathcal{D}_r$ from $\mathcal{D}$, we also have

$$x \in D \cap T_\ell \Rightarrow x \in D_\ell, \lambda(x) = \lambda_\ell(x) \tag{16}$$

and

$$x \in D \cap T_r \Rightarrow x \in D_r, \lambda(x) = \lambda_r(x). \tag{17}$$

Now, let $x \in D$ be an arbitrary element of D. We will treat the cases $x \in D \cap T_\ell$ and $x \in D \cap T_r$ separately.

(i) $x \in D \cap T_\ell$: The first inequality of (13) follows from (14), (16) and (11). To prove the second inequality of (13) we must show that for all $y \in T_r$ with $y \geq x$ we have $f(y) \geq \lambda(x)$. So, let y be any element of T_r with $y \geq x$. If $x \in D_{\min}$, then $\lambda_\ell(x) = \lambda(x) = c_{\min}$; so, in that case we trivially have $f(y) \geq \lambda(x)$. Next, we will suppose, that $x \notin D_{\min}$. In that case, let $x' = \pi_r(x)$. Then, $x' \in D_r$ and $y \geq x'$. So, according to (12), we have $\lambda_r(x') \leq f_r(y)$. Together, we have $\lambda(x) \leq \lambda_r(x') \leq f_r(y) = f(y)$, where the first inequality follows from $x' \geq x$ and the fact that λ_r is a monotone labeling. The equality follows from $y \in T_r$. This proves the first case.

(ii) $x \in D \cap T_r$: In this case, the second inequality of (13) follows easily from (12), (17) and the analogon of (15) for $\check{f}_r$. The first inequality can be proven as follows: Because of (12), we only need to show that for all $y \in T_\ell$ with $y \leq x$ we have $f(y) \leq \lambda(x)$. So, let y be an element of T_ℓ with $y \leq x$. If $x \in D_{\max}$, then $\lambda_r(x) = \lambda(x) = c_{\max}$ so automatically, $f(y) \leq \lambda(x)$. If $x \notin D_{\max}$, let $x' = \pi_\ell(x)$ so $x' \in D_\ell$. We now have two possibilities: either $y_i \leq x'_i$ or $y_i > x'_i$. In the first case, we have $y \leq x'$, so $f(y) = f_\ell(y) \leq \lambda_\ell(x') \leq \lambda(x)$, so we are ready. In the second case, we still have $y \in T_\ell$, so we must have $x'_i < y_i < c$. Now, we define y' as follows:

$$y'_j = \begin{cases} y_j & \text{if } j \neq i \\ x'_i & \text{if } j = i. \end{cases}$$

We now have $y' \leq x'$, and $f(y) = f(y')$, due to the $\mathbb{D}$-granularity of f. Thus, in this case we have $f(y) = f(y') \leq \lambda_\ell(x') \leq \lambda(x)$, where the first inequality follows from (11) and $y' \leq x'$. The second inequality follows from the monotonicity of λ_ℓ. Thus, in both cases the first inequality of (13) is proved. □

Using the lemma, we easily prove the above theorem.

Proof of the theorem : Lemma 4a 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 existance 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 $f_\mathcal{T}(x) = k$, for all $x \in T$. 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 4b we infer that $f_\mathcal{T}$ must be quasi-monotone on X. $\Box$.

4 Example

In this section we will 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, see [Quinlan]. Starting in the root of the tree we have $T = X = [000, 333)$. 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 criterium (9), 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 $\hat{\lambda}(012) = 1$. 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 $\check{\lambda}(102) = 2$. 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 criterium (8), 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 of for the dataset given in Table 1, see Figure 4. This decision tree is in fact not only quasi-monotone but even monotone.

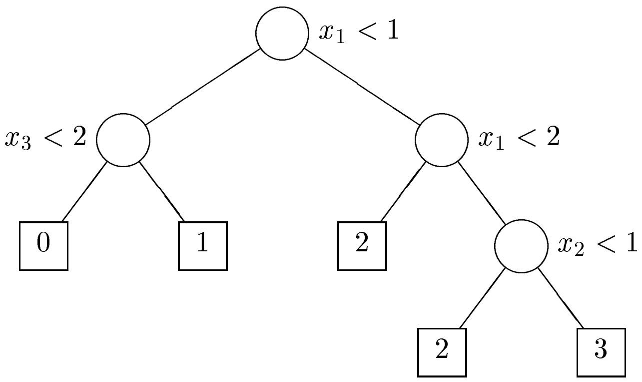

Figure 4: (Quasi-)monotone Decision Tree: Example

5 Experiments

We did some experiments to get an idea of the usefulnes of the algorithm for quasi-monotone trees in the k classes case (we call this algorithm QMT for short), and to compare its performance with a standard decision tree algorithm such as ID3 on the one hand, and our standard algorithm for monotone decision trees [7, 5] on the other hand (MT for short). As a first experiment, we wanted to investigate the size and the degree of non-monotonicity of the trees generated by QMT in comparison with the other methods, and the speed of QMT compared to the other methods. We generated random monotone datasets with 10, 20, 30, etc. examples and built trees with each of those datasets, using three different methods: ID3, MT, and QMT. For each number of examples we generated five different datasets, each from a universe with five attributes, each having three possible values, while all data elements were evenly devided over four classes. The results of this experiment are shown in Table 2 and 3. The figures shown are averages over the five datasets.

examples	ID3	MT	QMT
10	0.6	0.8	1.0
20	1.0	1.8	2.0
30	1.4	3.4	3.2
40	2.2	4.6	4.4
50	2.4	7.4	6.0
60	2.8	7.0	6.0
70	3.6	10.2	6.8
80	4.4	14.2	8.6
90	4.4	15.0	9.0
100	5.6	19.0	10.6
150	8.0	36.6	14.4

Table 2: Speed comparisons of considered methods

	# non-monotone		perc. non-monotone	
examples	ID3	QMT	ID3	QMT
10	2.6	0.0	10.7	0.00
20	8.0	2.4	11.8	0.78
30	11.8	1.6	8.1	0.33
40	12.4	2.4	5.7	0.23
50	15.0	2.2	3.5	0.14
60	15.8	1.8	3.4	0.10
70	18.4	0.4	2.6	0.02
80	19.6	1.0	2.0	0.03
90	18.2	1.0	1.6	0.03
100	19.6	0.4	1.2	0.01
150	12.6	0.0	0.4	0.00

Table 3: The degree of non-monotonicity of the generated trees

In Table 2 the speed is in seconds running time on our computer. From this table we may conclude that, although the size of quasi-monotone trees is usually somewhat larger than those generated with our best monotone algorithm, they are produced much faster. Moreover, as we can see in Table 3, the degree of non-monotonicity of the quasi-monotone trees produced is very low. This degree is measured by the number of non-monotone pairs of leaves, and by its percentage with respect to the total number of those pairs.

In comparison, the degree of non-monotonicity ID3 trees produced with monotone datasets is much higher, although it also decreases as the datasets become larger. This effect can be explained by the fact that quasi-monotonicity becomes closer to monotonicity when the dataset becomes larger, as can be seen from the definition.

We ran a second experiment to test the accuracy of QMT-trees in comparison with those generated by the other methods. Again, we generated random monotone datasets, in this case with 30, 60 and 90 examples each. We used the same universe as in the first experiment. This time, on each dataset we ran a 3-fold cross-validation experiment with all three methods: we split the dataset in three equal parts, built ID3, MT and QMT-trees on two thirds and tested the grown trees on the remaining one third of the dataset. Again, for each number of examples we used five different datasets, so the results are averaged over these five datasets. Table 4 summerizes the result.

	perc. correctly classified		
examples	ID3	MT	QMT
30	35.0	46.7	46.7
60	15.8	57.0	56.0
90	18.2	61.8	62.7

Table 4: Accuracy of tree growing methods on monotone datasets

From Table 4 it can be seen that in our experiments the quasi-monotone trees match the predictive power of the monotone trees, while both by far outperform ID3.

6 Complexity of generating (quasi)-monotone trees

Note. This section is still in preparation (19/2/98).

7 Discussion

We have provided a decision-tree algorithm that generates quasi-monotone decision-trees for monotone datasets with discrete or continuous attributes and k classes. This generalizes an algorithm of Makino *et al.* [8]. It appears that our algorithm outperforms classical decision tree algorithms with respect to prediction and is faster than algorithms that generate strictly monotone trees while the sizes of trees are comparable. In another Technical Report [7, 5] we provide an algorithm MT that can be used to repair non-monotone decision trees that have been generated by other methods. So our algorithm for inducing quasi-monotone decision trees can be used as a bootstrap for the algorithm MT or as an approximate solution for the problem of generating order-preserving decision trees. The complexity of generating (quasi-)monotone decision trees is not known. However, in Section 6 of this report we have proved that the complexity of the problem of generating monotone decision trees for two-valued attributes and 2 classes is at least of the same order as that of the dualisition of a positive Boolean function. This is a well-known open problem discussed in [4]. It is known that this problem can be solved in quasi-polynomial time.

Currently, we are investigating the problem of improving tree-size and speed by mixing existing methods that generate non-monotone trees with those that generate (quasi)-monotone decision trees. Furthermore, we study the effect of the effect of noise in the dataset.

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] Bioch J.C., Ibaraki T., (1995). Complexity of identification and dualisation of positive Boolean functions. In Information and Computation, vol. 123(1) 50-63.

[5] Bioch, J.C., Potharst, R., (1997). Decision trees for monotone classification problems. In K. van Marcke and W. Daelemans eds., Proceedings of the 9th Dutch Conference on Artificial Intelligence (NAIC97), 361-370, University of Antwerpen.

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

[7] Potharst R., Bioch J.C., Petter T., (1997). Monotone decision trees. Tech. Rep. eur-few-cs-97-07, (37pp). Dept.of Computer Science, Erasmus University Rotterdam. Can be downloaded from website `http://www.cs.few.eur.nl/few/people/potharst`.

[8] 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.

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