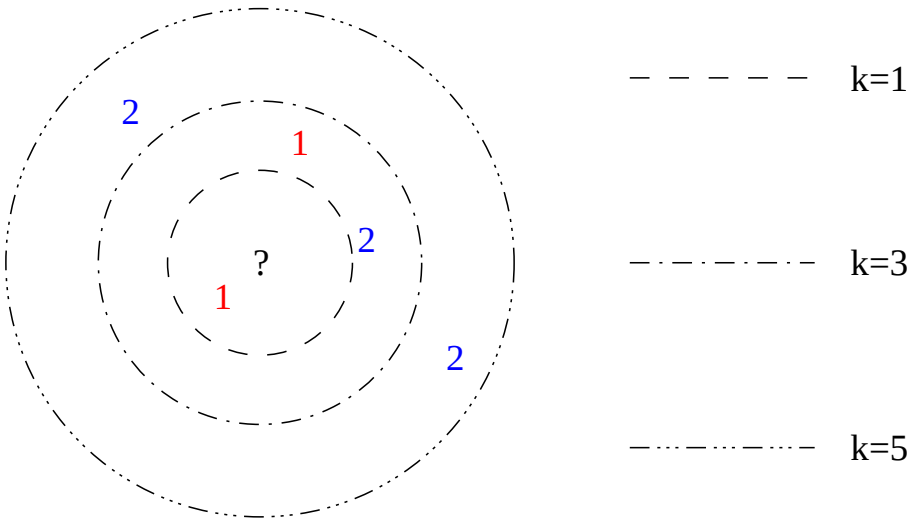
Nearest neighbour classification for problems with monotonicity constraints

Wouter Duivesteijn

Masters thesis

October 24, 2007

Thesis advisors:
Dr. K. Dajani & Dr. A.J. Feelders



Nearest neighbour classification for problems with monotonicity constraints

Wouter Duivesteijn

Masters thesis

October 24, 2007

Thesis advisors:
Dr. K. Dajani & Dr. A.J. Feelders

Contents

List of Algorithms

List of Tables

List of Figures

Chapter 1

Prefatory Matters

1.1 Introduction to the subject

Classification is the prediction of the class of an object on the basis of some of its attributes. One can think of prediction of good/bad credit for loan applicants using income, age, and so on, or spam/no spam for e-mail messages using percentage of words matching a given word, use of capital letters, etcetera. The basic idea is to build a classification model using a set of training examples.

For general classification problems, there are many techniques to do exactly what we just described. However in many applications of classification, we have a priori knowledge to the extent that, all else equal, an increase in an input variable should not lead to a decrease in class label. For example, if loan applicants A and B have the same attribute values, except that A has a higher income than B, then it would be surprising if B got the loan while A did not. Examples of other application domains in which we can have this type of knowledge are legal support systems, medicine (e.g. smoking increases the probability of vascular diseases), operations research and economics (e.g. house prices increase with the house area).

In this thesis, we want to develop a nonparametric classifier that satisfies such constraints, and has a good predictive performance. We choose for nonparametricity, since we want to assume as little as possible about our data; in particular we do not want to assume a certain set of parameters to summarize the data.

In the next section we give a general introduction to classification, for the benefit of those not familiar with the concept. In section 3 we describe some notation and definitions, used throughout the thesis. In section 4 we formulate the goal of this thesis, and describe the two-step approach to monotone classification problems we will use. Also in that section, we describe the organization of the rest of the thesis.

1.2 Introduction to classification

The previous section began with a brief statement what classification is about. In this section, we will give a more formal introduction on classification and a way to make a classifier.

Classification is a statistical procedure in which objects from an n-dimensional input space $\mathcal{X}$ are put into groups (*classes*) from a one-dimensional space $\mathcal{L}$, based on their attibutes and on a *training set* of previously labeled objects. The problem is: given a set of data $D = \{(x_1, \ell_1), \ldots, (x_n, \ell_n)\}$, produce a *classifier* $f : \mathcal{X} \to \mathcal{L}$, mapping an object $x \in \mathcal{X}$ to its class label $\ell \in \mathcal{L}$.

A classification problem is called *binary* if there are only two class labels.

1.2.1 Types of classifiers

There are many methods for classification. We distinguish two main types of methods: parametric and nonparametric classifiers.

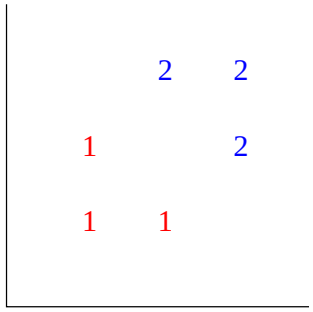
Parametric classifiers

A *parametric classifier* is a classifier which uses not the whole data set, but a set of parameters that are supposed to summarize the data. An example of such a method is a linear classifier, which classifies objects based on the value of a linear combination of the features: let $x \in \mathcal{X}$ and let w be a real vector of weights with the same dimension as x. Let w_0 be an extra weight. The class label ℓ of x according to the linear classifier f is $\ell = f(w_0 + w \cdot x)$.

Suppose for instance that we have a binary classification problem with a two-dimensional input space, and suppose that our data set is given in table 1.1. A plot is added for better insight. In the plot, the axes form the input space, and the data points are depicted by their class labels. The actual data point is at the lower left corner of the label.

Table 1.1: Example data set, with plot

x_1	x_2	ℓ
1	1	1
1	2	1
2	1	1
2	3	2
3	2	2
3	3	2



There are many ways to transform this data set into a classifier. One of them is based on the *naive Bayes* probabilistic model. One can view a classifier as a conditional probability model:

$$p(\mathcal{L} \mid \mathcal{X}_1, \mathcal{X}_2)$$

where we map $x \in \mathcal{X}$ to the $\ell \in \mathcal{L}$ for which this probability is the largest. To make these conditional probabilities more tractable, we will reformulate this model using Bayes' theorem:

$$p(\mathcal{L} \mid \mathcal{X}_1, \mathcal{X}_2) \;=\; \frac{p(\mathcal{L})\; p(\mathcal{X}_1, \mathcal{X}_2 \mid \mathcal{L})}{p(\mathcal{X}_1, \mathcal{X}_2)}$$

The denominator is independent from $\mathcal{L}$ and the $\mathcal{X}$-values are given, so we can ignore it. We can rewrite the numerator as a joint probability to obtain:

$$p(\mathcal{L}, \mathcal{X}_1, \mathcal{X}_2)$$

By applying the definition of conditional probability twice, we obtain:

$$\begin{aligned} p(\mathcal{L}, \mathcal{X}_1, \mathcal{X}_2) &= p(\mathcal{L})\; p(\mathcal{X}_1, \mathcal{X}_2 \mid \mathcal{L}) \\ &= p(\mathcal{L})\; p(\mathcal{X}_1 \mid \mathcal{L})\; p(\mathcal{X}_2 \mid \mathcal{L}, \mathcal{X}_1) \end{aligned}$$

So far this is all Bayes. The naive part is, that we assume conditional independence between the input variables given the class label. In other words, we assume:

$$\forall_{i \neq j} \quad p(\mathcal{X}_i \mid \mathcal{L}, \mathcal{X}_j) \;=\; p(\mathcal{X}_i \mid \mathcal{L})$$

So, we can express our joint model as:

$$p(\mathcal{L}, \mathcal{X}_1, \mathcal{X}_2) \;=\; p(\mathcal{L})\; p(\mathcal{X}_1 \mid \mathcal{L})\; p(\mathcal{X}_2 \mid \mathcal{L})$$

Naive Bayes' is usually applied on discrete data, and we assume to have data points for every point in our input space. We therefore can estimate values for $p(\mathcal{L})$ and $p(\mathcal{X}_i \mid \mathcal{L}), i = 1, 2$ from our training data, by calculating relative frequencies. If we do this for our example, we get:

$$\begin{array}{lll}
p(\mathcal{X}_1 = 1 \mid \mathcal{L} = 1) = \frac{2}{3}, & p(\mathcal{X}_1 = 2 \mid \mathcal{L} = 1) = \frac{1}{3}, & p(\mathcal{X}_1 = 3 \mid \mathcal{L} = 1) = 0 \\
p(\mathcal{X}_1 = 1 \mid \mathcal{L} = 2) = 0, & p(\mathcal{X}_1 = 2 \mid \mathcal{L} = 2) = \frac{1}{3}, & p(\mathcal{X}_1 = 3 \mid \mathcal{L} = 2) = \frac{2}{3} \\
p(\mathcal{X}_2 = 1 \mid \mathcal{L} = 1) = \frac{2}{3}, & p(\mathcal{X}_2 = 2 \mid \mathcal{L} = 1) = \frac{1}{3}, & p(\mathcal{X}_2 = 3 \mid \mathcal{L} = 1) = 0 \\
p(\mathcal{X}_2 = 1 \mid \mathcal{L} = 2) = 0, & p(\mathcal{X}_2 = 2 \mid \mathcal{L} = 2) = \frac{1}{3}, & p(\mathcal{X}_2 = 3 \mid \mathcal{L} = 2) = \frac{2}{3}
\end{array}$$

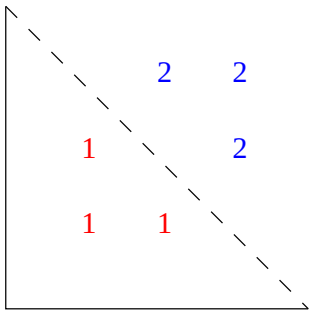
$$p(\mathcal{L} = 1) = \tfrac{1}{2},\; p(\mathcal{L} = 2) = \tfrac{1}{2}$$

Now suppose that we want to classify a new point $x = (2, 2)$. We calculate:

$$\begin{aligned}
p(\mathcal{L} = 1)\; p(\mathcal{X}_1 = 2 \mid \mathcal{L} = 1)\; p(\mathcal{X}_2 = 2 \mid \mathcal{L} = 1) &= \frac{1}{2}\frac{1}{3}\frac{1}{3} = \frac{1}{18} \\
p(\mathcal{L} = 2)\; p(\mathcal{X}_1 = 2 \mid \mathcal{L} = 2)\; p(\mathcal{X}_2 = 2 \mid \mathcal{L} = 2) &= \frac{1}{2}\frac{1}{3}\frac{1}{3} = \frac{1}{18}
\end{aligned}$$

Since we have equal probabilities for both class labels, we can pick both. By calculating all points for which this holds, we obtain figure 1.1.

Figure 1.1: The resulting linear classifier



All data points below the dashed line (that is called the *decision boundary*) are assigned class label 1, and all data points above it are assigned class

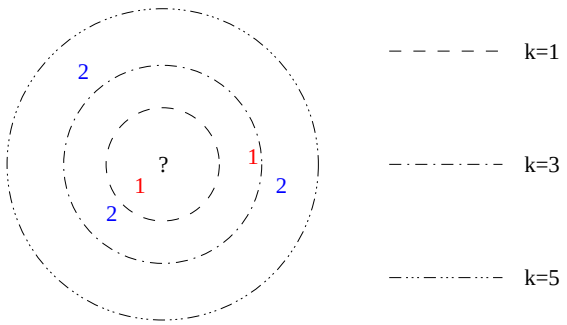
label 2. The decision boundary is a straight line, which corresponds to a linear classifier as described on page 11. In this case, $w_0 = 0$, the vector $w = (1, 1)$, and $f(\cdot) = 1 + \mathbf{1}\{\cdot \leq 4\}$. Under the naive Bayes assumption, the decision boundary is always a straight line, so the resulting classifier is linear.

Non-parametric classifiers

A *nonparametric classifier* is a classifier which memorizes all data, and uses a distance metric d to decide how similar new data points are to existing examples. An example of such a method is the k-nearest neighbour procedure (kNN). Let k be a positive integer, typically small, and let $x \in \mathcal{X}$ be the data point to be classified. We find the k objects of which we know the label, that are nearest to x (according to the metric d). The data point x is assigned the class most common among these objects. For a binary classification problem, we usually choose k to be odd, to avoid ties. If there is a tie for the most common class, we apply one of many possible tie-breaking rules. We could for instance randomly choose one of the most common classes, or instead look at the $k + i$ nearest neighbours, where we let i be the smallest positive integer for which there is no tie for the most common class.

For a typical example of a k-nearest neighbour classifier, suppose we have a two-dimensional input space, let d be Euclidian distance, and suppose we have a data set from which a small piece is shown in figure 1.2.

Figure 1.2: Part of an example data set



We want to assign a new data point, depicted by a question mark, to a class. The five nearest data points are shown. Now if $k = 1$, we assign the point

to class 1, since that is the label of the nearest data point. If $k = 3$, we look at the three data points in the second inner circle, and assign the new point to class 1 since this is the most common class among the three points. If however $k = 5$, we look at the five data points in the outer circle, and assign the new point to class 2 since this is the most common class among the five points.

Less parametric classifiers

With the above definitions, we have made a rigid distinction between parametric and non-parametric classifiers. However, some parametric classifiers are more parametric than others, in the sense that they use a smaller set of parameters to summarize the data. In our linear classifier example, all information about a data point is brought back to one single linear combination of its components, with the vector of weights w and weight w_0 as the only parameters. To illustrate a 'less parametric' classifier, we consider a certain type of *neural network*. In these classifiers, we calculate M linear combinations of the input variables $x_1, \ldots, x_D$:

$$h_i(x) = \sum_{j=1}^{D} w_{ij}^a x_j \qquad \text{for } i = 1, \ldots, M$$

We then transform each linear combination with a certain function g, usually non-linear:

$$g_i(x) = g(h_i(x)) \qquad \text{for } i = 1, \ldots, M$$

Finally we compute a transformation f of a linear combination of the resulting values:

$$f(x) = s\left(\sum_{i=1}^{M} w_i^b g_i(x)\right),$$

where e.g. if we only have class labels 0 and 1, s usually is the logistic function rounded to the nearest integer.

So, the class label ℓ of x is the value of $f(x)$ according to the neural network classifier with the function g, the number M of linear transformations in the first step, and both vectors of weights w^a and w^b. By our definition, this classifier is clearly parametric, but it is also clear to see that it is far more complex than the linear classifier.

1.2.2 Classifier performance

We can only measure the performance of a classification procedure on a certain data set (since clearly, if there is no data available, we cannot tell whether the classifier is right or wrong). If we want to do so, then we partition our data set into two sets. The first set is called the *training data*. We use this data, including the class labels, to build a classifier according to the classification procedure we are testing. The second set is called the *test data*. For each object x in the test data, we compare $f(x)$ to the real label ℓ. The partitioning is necessary, since if we would build the classifier on the whole data set, it would be prone to overfit, and thus the test would give inaccurate results.

The performance of a classifier is expressed as a total cost, which is to be minimized. Every time the classifier misclassifies a data point, we add a certain amount to the total cost. There are again many ways to do this. One could for instance simply count the number of misclassifications.

As an example of how we can use this train-and-test procedure to measure the performance of a classifier in a quite sophisticated way, we consider K-*fold cross-validation*. In this technique, we partition the data into K subsets. One of these subsets is used as the test data, and the rest of the subsets become the training data, calculating the total cost as we just described. We repeat this procedure K times, in such a way that each subset is used as the test data exactly once. Finally, we combine the K results into one prediction of the classifier performance. The special case where K is equal to the number of data points is called *leave-one-out cross-validation*, for obvious reasons.

1.3 Definitions and notation

Suppose we have an observed data set, consisting of n points in a k-dimensional input space $\mathcal{X} = \times\mathcal{X}_i$, and their corresponding class labels taking values in a one-dimensional space $\mathcal{L}$. We denote the i^{th} data point in our data set by x_i, and its m^{th} attribute is denoted by $x_{i,m}$. We denote the class label of the data point x_i by $\ell(x_i)$. Finally, we denote the observed data points by $D = \{(x_i, \ell(x_i))\}_{i=1}^{n}$.

We will assume a total order on $\mathcal{L}$. On $\mathcal{X}$, we will assume a partial order unless stated otherwise. We will not make any assumptions on this partial order, however in practical examples it is often a product order. This corresponds to the situation where single attributes have a positive or negative influence on the class label.

A pair of points $(x_i, \ell(x_i))$ and $(x_j, \ell(x_j))$ from D is called non-monotone if $x_i \leq x_j$ and $\ell(x_i) > \ell(x_j)$, or $x_j \leq x_i$ and $\ell(x_j) > \ell(x_i)$.

We define the *monotonicity violation graph* (MVG) to be the directed graph
$G = (V, E)$ with:

- $V = \{1, 2, \ldots, n\}$, and:
- $(i, j) \in E$ if the points $(x_i, \ell(x_i))$ and $(x_j, \ell(x_j))$ are non-monotone and $\ell(x_i) > \ell(x_j)$[1].

The monotonicity violation graph is the graph of a strict partial order, since it is both

- anti-symmetric: $(i, j) \in E \Rightarrow (j, i) \notin E$, and
- transitive: $(i, j) \in E$ and $(j, k) \in E \Rightarrow (i, k) \in E$.

These properties follow immediately from the total ordering on the class labels.

In an MVG, two points x and y are said to be *neighbours* if either $(x, y) \in E$ or $(y, x) \in E$.

[1] If we would omit this last condition, we would obtain two edges between every two non-monotone points

In an MVG, a point y is said to be *minimal* if there is no $x < y$ such that x is a neighbour of y in the MVG. Conversely, a point y in an MVG is said to be *maximal* if there is no $z > y$ such that z is a neighbour of y in the MVG.

Given an order on the input space and an order on the class labels, a *monotone classifier* is a classifier f such that for every two objects $x_1, x_2 \in \mathcal{X}$ we have:

$$x_1 \leq x_2 \quad \Rightarrow \quad f(x_1) \leq f(x_2)$$

Whenever we consider a relabeling of the data point x_i, we denote its label *before* relabeling by $\ell(x_i)$, and its label *after* relabeling by $\ell'_j(x_i)$. The subscripted character j is used if we want to discuss more than one relabeling of a data set; if this is not the case we will omit it.

For a relabeling, we also sometimes use as shorthand notation a comma-separated list with elements of the form $x_i \to \ell$. This means that x_i has label ℓ after relabeling, and all unmentioned data points keep their original labels.

1.4 Thesis objective

As stated in section 1.1, we want to develop a nonparametric classifier that satisfies monotonicity constraints. We tackle this problem using a two-step approach.

1.4.1 Two-step approach

In this thesis, we will focus on the following two-step approach to build a nonparametric monotone classifier:

Step 1 Relabel the data in order to make it monotone. Minimize the total cost. This produces the monotone classifier with the lowest total cost on the training data. This classifier is however only defined on the observed data points.

Step 2 Extend the classifier to the whole input space, not just the observed x-values.

In step one, the resulting classifier will depend on the *cost matrix* C, where $C(\ell_i, \ell_j)$ is the "cost" of allocating a point with label ℓ_i to label ℓ_j. When talking about specific labels i and j, we also will use the shorthand notation C_{ij} for the cost of relabeling a data point with label i to label j.

If we take the 0/1 cost matrix (in other words: we simply count the number of label changes):

$$C(\ell_i, \ell_j) = \mathbf{1}\{\ell_i \neq \ell_j\},$$

then we are in the case discussed in [5]. In this paper, two algorithms for monotonizing data sets with as few label changes as possible are discussed; one greedy heuristic and one exact algorithm. We will describe the latter in section 2.2.

1.4.2 Thesis goal

In this thesis, we will develop a method for nonparametric monotone classification. We use the two-step approach described in the previous subsection, for which:

- in step one, we will work with both 0/1 and general cost matrices;
- in step two, we will use variants of the kNN algorithm that maintain monotonicity.

1.4.3 Organization of the rest of this thesis

The rest of this thesis is organized as follows. First, in the following section, we will discuss some work related to the topic of this thesis.

Then, in the next chapter, we describe the exact relabeling algorithm from [5], and extend it to an algorithm that performs our first step. We will obtain different results for binary classification and classification with a general number of classes. We discuss the implementation of these algorithms, and show some special theoretical results for the case where we have a total order on the input space.

In chapter 3, we propose some variants of an algorithm that performs our second step, and algorithms that combine these two steps. Also in that chapter, we discuss the implementation of our algorithms, and test them on several real data sets.

Finally, in chapter 4 we draw conclusions from our experiments, and describe some topics that deserve further research.

1.5 Related work

The monotonicity constraint is rather common in practice. Hence, several data mining and machine learning techniques have been modified to handle such constraints. If we restrict our attention to work that deals specifically with the nonparametric monotone classification problem, two publications are relevant.

1.5.1 Ordinal Learning Model by Ben-David

The earliest work in this area known to me is the Ordinal Learning Model (OLM) of Ben-David [2]. He constructs a *rule base* R from a training data set E. We have $R \subseteq E$. The set R is composed of *consistent* and *redundant* data points.

Consitency refers to the monotonicity constraint. The algorithm sequentially adds points from the training set to R. If an example violates monotonicity with a point in R, then it is discarded. Examples may also be redundant with respect to the current R, because of the labeling strategy: for a new point x_0, OLM finds all $x_i \in R$ such that $x_i \leq x_0$, and allocates x_0 to the largest class among these points. Now if $x \in R$, and x' is a data point such that $x \leq x'$ and $\ell(x) = \ell(x')$, then x' does not affect the labeling of new instances. So, x' is redundant with respect to x. If there is no $x_i \in R$ such that $x_i \leq x_0$, then x_0 is allocated to the class of its nearest neighbour in R.

1.5.2 Algorithms minimizing L_1 and L_2 cost by Dykstra et al.

Dykstra, Hewett, and Robertson [4] propose a procedure that minimizes L_1 cost: $C_{ij} = |j - i|$, subject to $x_i \leq x_j \Rightarrow f(x_i) \leq f(x_j)$. The algorithm requires the computation of $k - 1$ isotonic regressions to find the (non-unique) optimal solutions, where k is the number of classes.

They also provide an algorithm that minimizes L_2 cost: $C_{ij} = |j - i|^2$. This algorithm requires the performance of a single isotonic regression. Notice that for binary classification, these cost matrices are equal to the matrix for 0/1 cost.

Chapter 2

Step One: making a monotone data set

2.1 Overview of step one

This chapter will deal with the first step of our two-step approach: relabel the data to make it monotone, minimizing the total cost corresponding to a certain cost matrix.

First, in the next section, we look at the problem in the 0/1 cost matrix case, i.e. the case where we simply want to minimize the number of relabeled data points. We discuss a known algorithm that solves the problem.

Then, in section three, we look at the problem in the general cost matrix case. Here, we distinguish between the binary classes case and the general case, and for the former we extend the algorithm from the previous section to solve this problem.

In section four, we implement the algorithms discussed in sections two and three.

Finally, in section five, we discuss some interesting properties that hold if we assume a total order on the input space instead of a partial order.

2.2 The 0/1 cost matrix case

This section describes an algorithm that was already described in another paper. Therefore this section will have a large intersection with [5]. In this section we take the 0/1 cost matrix, so

$$C(\ell_i, \ell_j) = \mathbf{1}\{\ell_i \neq \ell_j\}.$$

As Rademaker, De Baets, and De Meyer [11] observe, a maximum weight independent set in the monotonicity violation graph corresponds to a maximum size monotone subset of the data. Relabeling the complement of the maximum weight independent set results in a monotone data set with the lowest total cost possible. It is always possible to find a consistent relabeling of this complement.

The monotonicity violation graph is defined to contain a node for all distinct points in D. However, it is obvious that points that do not violate the monotonicity constraint with any other point will always be part of the maximum weight independent set, and can from now on be ignored.

The monotonicity violation graph is the graph of a partial order. For such a graph, a maximum weight independent set corresponds to a maximum antichain in the corresponding partial order, that can be computed in $\mathcal{O}(n^3)$ time by solving a minimum flow problem on a transportation network that is easily constructed from the comparability graph (see [10]). In the next subsection, we describe the transformation of the monotonicity violation graph $G = (V, E)$ to the corresponding transportation network $G' = (V', E')$. Thereafter, we will show how we find the maximum weight independent set and how this relates to the minimum number of points that have to be relabeled.

2.2.1 Transforming the MVG

The standard network flow algorithms assume that weights are associated with the edges, but the monotonicity violation graph has weights associated with the vertices. Hence we transform vertices to edges by vertex splitting:

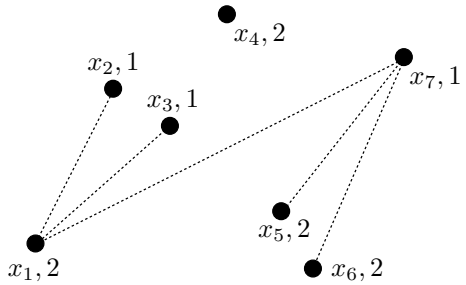
$$V' = \bigcup_{i \in V} \{i_a, i_b\} \cup \{s, t\}$$

where s is the source, and t is the sink of the transportation network.

The edge set E' contains edges $i_a i_b$ for all $i \in V$, and edges $i_b j_a$ for all $ij \in E$. Added to that, E' contains edges $s i_a$ for all minimal points x_i, and edges $j_b t$ for all maximal points x_j. All edges have upper capacity ∞. The edges of the form $i_a i_b \in E'$ have lower capacity one, and all other edges of E' are assigned lower capacities of zero.

To illustrate this construction, we consider the data set given in figure 2.1.

Figure 2.1: Data set with MVG



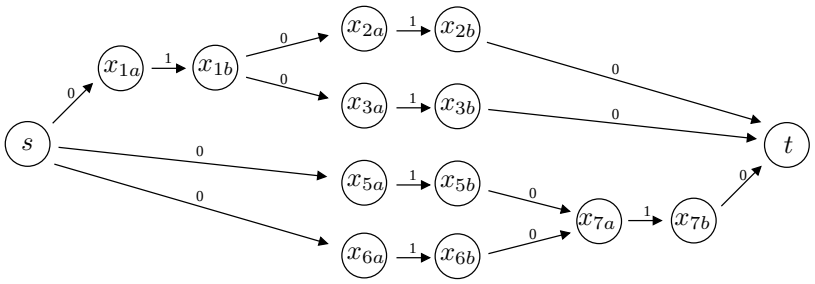
Assume we have a product ordering: for any two points $a = (a_1, a_2)$ and $b = (b_1, b_2)$, we have

$$a \leq b \qquad \Leftrightarrow \qquad a_1 \leq b_1 \text{ and } a_2 \leq b_2.$$

The set of dotted lines connecting the points indicate the monotonicity violations. The corresponding monotonicity violation graph is $G = (V, E)$ with $V = \{1, 2, 3, 5, 6, 7\}$ and $E = \{(1,2), (1,3), (1,7), (5,7), (6,7)\}$. Notice that $4 \notin V$ since x_4 does not violate monotonicity with any other point. Figure 2.2 depicts the transportation network associated with G.

Each of the data points is represented by an edge with lower capacity of one and an upper capacity of ∞. The connections to the source and the sink, as well as the edges representing the monotonicity violations, are assigned lower capacities of zero and upper capacities of ∞. Since all upper capacities are ∞, only lower capacities are shown in the figure.

Figure 2.2: Transportation network corresponding with the data set in figure 2.1



2.2.2 Finding the maximum weight independent set

The problem of finding the maximum weight independent set in G can now be solved by finding the minimum flow value f_{val}^{min} in G'. **(author?)** show that this can be done by finding the maximum capacity of an s,t-cut in G', that is,

$$f_{val}^{min} = \max_{S,T} \left(\sum_{\substack{ab \in E' \\ a \in S, b \in T}} lc(ab) - \sum_{\substack{ab \in E' \\ a \in T, b \in S}} uc(ab) \right),$$

where S,T is an s,t-cut of $G' = (V', E')$, meaning that $\{S,T\}$ forms a partition of V' while $s \in S$ and $t \in T$, and where $lc(ab)$ and $uc(ab)$ denote the lower and upper capacity of edge $ab \in E'$. For a proof, see [10].

Finally, the set complement[1] to the maximum weight independent set is the set of points that need to be relabeled to get monotone data. For the network flow on the previous page, we find $f_{val}^{min} = 4$, so the weight of the maximum independent set M is 4. This set is obtained by the S,T-cut, where $S = \{s, x_{1a}, x_{1b}, x_{2a}, x_{3a}, x_{5a}, x_{6a}\}$, $T = V' \backslash S$, and $M = \{2,3,5,6\}$. Hence, we find that the set of points we need to relabel to make D monotone is $V \backslash M = \{1,7\}$.

[1] with respect to V, so disregarding points that did not violate the monotonicity constraint with any other point

2.3 The general cost case

In the previous section we introduced an algorithm for relabeling non-monotone data, minimizing the number of label changes. In this section, we want to extend this algorithm to minimize the cost according to a general cost matrix. The only assumptions we want to make on this matrix are:

$$\begin{array}{lll} C(\ell_i, \ell_j) = 0 & \text{if} & \ell_i = \ell_j \\ C(\ell_i, \ell_j) > 0 & \text{if} & \ell_i \neq \ell_j \end{array} \tag{1}$$

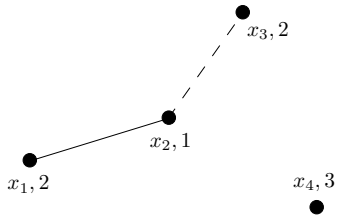
In the previous section, we did not pay much attention to the following claim:

Claim 1. *Although the monotonicity violation graph is defined to contain a node for all distinct points [...], it is obvious that points that do not violate the monotonicity constraint with any other point [...] can be ignored in the sequel.*

For the 0/1 cost matrix case, this indeed is obvious. Unfortunately, for the general cost matrix case, the claim does not hold. We will now show a counterexample.

Assume again a product ordering (see page 26). Now suppose we have only three distinct class labels, and a cost matrix such that $C_{13}, C_{23} \ll C_{12}, C_{21}$. Consider the data set shown in figure 2.3.

Figure 2.3: Data set with MVG



The solid line is the monotonicity violation graph (MVG) of this data set. Now x_3 and x_4 do not violate the monotonicity constraint with any other point. However, if we ignore them, we will find $x_2 \to 3$ as optimal relabeling (i.e. a relabeling with the lowest cost), thus creating a new monotonicity violation between x_2 and x_3 (the dashed line).

In reality, the optimal relabeling for this data set is $x_2 \to 3, x_3 \to 3$. We conclude that we cannot ignore the points with degree zero in the MVG, in the general cost matrix case with more than two distinct class labels. This of course does not mean that the problem in this case cannot be solved; we merely conclude that our approach is not applicable, and thus this case is beyond the scope of this thesis.

2.3.1 The binary classes case

The data set in the counterexample on the previous page had three distinct class labels. In this section, we will show that claim 1 holds in the general cost matrix case with exactly two distinct class labels.

We shall need the following concept, as some sort of 'proving device'. Let a data set and its MVG be given. A relabeling is called a *contagious relabeling* if for all points x the following holds: either $\ell'(x) = \ell(x)$, or there is a point y such that $\ell'(x) = \ell(y)$ and y and x are neighbours in the MVG.

Now the following lemma will lead us to the proof of claim 1:

Lemma 1. *We can ignore points with degree zero in the MVG, if we allow only contagious relabelings.*

It suffices to prove that we cannot create a new monotonicity violation between any point with degree zero in the MVG and any point that is relabeled.

Proof. Let a data set and its MVG be given, and a contagious relabeling. Let z be a point with degree zero in the MVG, let x be a point that is being relabeled, and let y be a point such that $\ell'(x) = \ell(y)$ and y and x are neighbours in the MVG.

Suppose x and z are incomparable. We now cannot create a new monotonicity violation between them.

Suppose x and z are comparable. By symmetry, we can restrict ourselves to the case where $x < z$. We now have two different cases:

- Suppose $y \leq x$. By transitivity of a partial order, $y \leq z$. Since z has degree zero in the MVG, $\ell(y) \leq \ell(z)$. So $\ell'(x) = \ell(y) \leq \ell(z)$.
- Suppose $y > x$. Since x and y are neighbours in the MVG, $\ell(x) > \ell(y)$. Since z has degree zero in the MVG, x and z are not neighbours, so $\ell(x) \leq \ell(z)$. So $\ell'(x) = \ell(y) < \ell(x) \leq \ell(z)$.

In both cases we arrive at the conclusion that $\ell'(x) \leq \ell(z)$, and since $x < z$ we do not create a new monotonicity violation between x and z. Since y and x are neighbours in the MVG, they must be comparable. Thus we have proven that in any case, a contagious relabeling will not create new monotonicity violations with points with degree zero in the MVG. Therefore we can ignore these points. □

Notice that we can relax our limitations on z: the whole reasoning also holds if there is no path in the MVG between x and z. It follows that we can apply a contagious relabeling on a connected component of the MVG, ignoring all points not in that component. This observation comes in handy when proving the last lemma we need:

Lemma 2. *If we have a cost matrix satisfying equation* 1 *and a data set with only two distinct class labels, every optimal relabeling is contagious.*

Proof. Suppose that we have a cost matrix satisfying equation 1 and a data set with only two distinct class labels, and suppose that there is an optimal relabeling that is not contagious. Since there are only two class labels, we know that every relabeling automatically is contagious on each connected component of the MVG. So there must be a data point x with degree zero in the MVG, such that $\ell(x) = i \neq j = \ell'(x)$. However, from our observation directly after lemma 1, we see that the relabeling of x is not induced by any contagious relabeling. So, we can make another relabeling which is equal to the optimal relabeling, except that it leaves the labels of all points with degree zero in the MVG the way they were. Since our cost matrix satisfies equation 1, we have that $C_{ij} > 0$. Hence the total cost of the new relabeling is lower than the total cost of the optimal relabeling, which is a contradiction. □

If we have a general cost matrix and only two distinct class labels, then by lemma 2 every optimal relabeling is contagious. By lemma 1, we can hence ignore the points with degree zero in the MVG, which proves claim 1 in this case.

2.3.2 A relabeling algorithm for the binary classes case

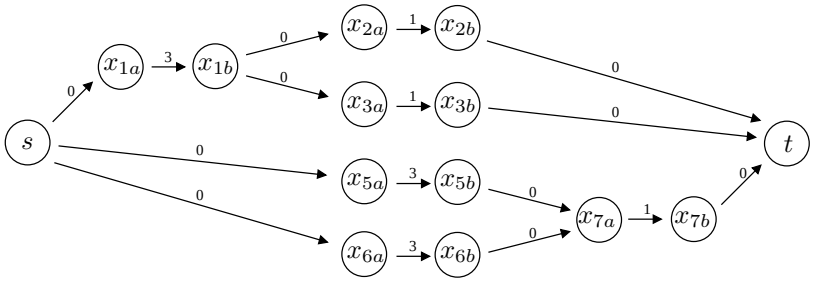
In the previous section, we proved that we can ignore points with degree zero in the MVG, if we have a general cost matrix and only two distinct class labels. This means that we can easily adjust the algorithm described in the previous section for this case. Since we are in the binary classes case, assume we have the following cost matrix:

$$C = \begin{pmatrix} 0 & p \\ q & 0 \end{pmatrix} \quad \text{with } p, q > 0. \tag{2}$$

Just like in the 0/1 cost matrix case, we build a transportation network. The capacities of the edges are as described on page 26, except that the edges of the form $i_a i_b \in E'$ have lower capacity pw_i if $\ell(x_i) = 1$, and lower capacity qw_i if $\ell(x_i) = 2$. The rest of the algorithm is the same as in the 0/1 cost matrix case.

Consider again the data set given on page 26, and suppose we have the cost matrix from equation 2 with $p = 1, q = 3$. We obtain the transportation network shown in figure 2.4. For the flow in this network, we find $f^{\min}_{\text{val}} = 9$, corresponding to the S,T-cut with $S = \{s, x_{1a}, x_{5a}, x_{6a}\}$, $T = V'\backslash S$, and $M = \{1, 5, 6\}$. We find that the set of points we need to relabel to make D monotone is $V\backslash M = \{2, 3, 7\}$, with total cost 3.

Figure 2.4: The transportation network under a different cost matrix



2.4 Implementation

Our algorithms are implemented in MATLAB, and our pseudocode uses many of MATLAB's built-in functions. See the MATLAB website ([8], [9]) for help.

We first implemented the algorithm 'toBeRelabeled', whose pseudocode is given in algorithm 1. The algorithm takes a data set, and returns the labels of a set with minimum weight of data points that need to be relabeled to make the data monotone.

In line 4, the auxiliary function 'normalize' is called on the input space. This function applies on every column c of the input space the following transformation:

$$c'_i = \frac{c_i - \text{mean}(c)}{\text{std}(c)}, \quad i = 1, \ldots, n$$

where 'mean' and 'std' are standard MATLAB functions that compute the mean and standard deviation of a vector. By normalizing the values of each attribute, we make sure that the selection of nearest neighbours by a certain metric does not depend on the unit in which the attribute is described. If we do not normalize, the concept 'nearest neighbours' is meaningless.

Lines $6, \ldots, 10$ build the adjacency matrix M for the MVG: the matrix entry M_{ij} is equal to 1 if $x_i \leq x_j$ and $\ell(x_i) > \ell(x_j)$, and 0 otherwise.

Lines $11, \ldots, 23$ find all the points that do not violate monotonicity with any other point, and discard them.

Lines $24, \ldots, 28$ build the edge set of the MVG from the adjacency matrix.

Line 29 uses the algorithm 'grMaxStabSet' from the grTheory toolbox for MATLAB, to be found at [6]. This algorithm solves the maximal independent set problem we described in section 2.2.2. It implicitly works with the 0/1 cost matrix, but it also accepts a vector of weights as optional argument. For a binary classification problem, we can easily construct such a vector given a general cost matrix, so in this case the algorithm can also work with general cost matrices.

Algorithm 1 toBeRelabeled(D)

```
    dimensions ← size(D)
    n ← dimensions(1)
    k ← dimensions(2)-1
    x ← normalize(D(:, 1 : k))
 5: y ← D(:, k + 1)
    mvgAdjMatrix ← zeros(n)
    for i = 1 to n do
      for j = 1 to n do
        if all(x(i, :) ≤ x(j, :)) & y(i) > y(j) then
10:       mvgAdjMatrix(i, j) ← 1
    nonViolatingPoints ← []
    for i = 1 to n do
      if sum(mvgAdjMatrix(i, :))+sum(mvgAdjMatrix(:, i))==0 then
        nonViolatingPoints ← [nonViolatingPoints; i]
15: if size(nonViolatingPoints)==n then
      return []
    else
      tBR ← 1 : n;   tBR(nonViolatingPoints)←[]
      mvgAdjMatrix(nonViolatingPoints,:)←[]
20:   mvgAdjMatrix(:,nonViolatingPoints)←[]
      n ← n− size(nonViolatingPoints)
      x(nonViolatingPoints,:)←[]
      y(nonViolatingPoints)←[]
      mvgEdges ← []
25:   for i = 1 to n do
        for j = 1 to n do
          if mvgAdjMatrix(i, j)==1 then
            mvgEdges ← [mvgEdges; i j]
      maxIndependentSet ← grMaxStabSet(mvgEdges)
30:   tBR(:,maxIndependentSet)←[]
      return tBR
```

2.4.1 Implementation for binary classification

The algorithm we just described finds out which points need to be relabeled, but we do not know yet which new label these points should receive. If we have a binary classification problem however, the problem is easy. For every data point, there is only one label that the point does not have, so we have no choice in relabeling.

The pseudocode for the algorithm 'monotonify' is given in algorithm 2.

Algorithm 2 monotonify(D)

```
    tbr ← toBeRelabeled(D)
    dimensions ← size(D)
    k ← dimensions(2) −1
    x ← D(:, 1 : k)
5:  y ← D(:, k + 1)
    y(tbr) ← 3 − y(tbr)
    return [x y]
```

Line 6 performs the actual relabeling, where we assume 1 and 2 to be the only labels.

2.4.2 Classification with an arbitrary number of classes

In the general case, there may be many choices for a new labeling. This section is dedicated to one of them, and thus gives an implementation of the relabeling algorithm for the general classes case. Notice that this implementation only works with the 0/1 cost matrix, since in section 2.3 we showed that our two-step approach does not work with a general cost matrix in the general classes case.

Throughout this section, we assume to have an MVG for a data set D with vertex set V and edge set E, and a set r of data points that need to be relabeled in order to make the data set monotone.

Lemma 3. *If* E *is not empty, then there must be a point in* r *which is either a minimal or a maximal point in the MVG.*

Proof. We shall prove the lemma by contradiction. Suppose E is not empty, and r contains only points that are neither minimal nor maximal points in the MVG.

The set r cannot be empty, since this would imply that the data set is monotone, and E would be empty.

Let $y \in r$. Our assumption on r implies that y is neither minimal nor maximal. Now there must be an $x \in V$ such that $x < y$, x is a neighbour of y in the MVG, and x is a minimal point in the MVG. Also there must be a $z \in V$ such that $z > y$, z is a neighbour of y in the MVG, and z is a maximal point in the MVG. Because of our assumption on r, $\{x, z\} \cap r = \emptyset$.

We know that $x < y$ and $y < z$, so $x < z$. Since x and y are neighbours in the MVG and $x < y$, we know that $\ell(x) > \ell(y)$. Since y and z are neighbours in the MVG and $y < z$, we know that $\ell(y) > \ell(z)$. Thus, we know that $\ell(x) > \ell(z)$. But this means that x and z are neighbours in the MVG, and so either x or z needs to be relabeled to make the data set monotone. This contradicts the fact that $\{x, z\} \cap r = \emptyset$. □

The proposed relabeling algorithm works in the following way: while r is not empty, let:

$$M^- = \{x \in r \mid x \text{ is a minimal point in the MVG}\}$$

Now for each $x \in M^-$, make $\ell(x)$ equal to the minimum of the labels of the neighbours of x in the MVG. This resolves all monotonicity violations for x, so we can remove all edges with x as endpoint from E, and remove x from r. Then, let:

$$M^+ = \{x \in r \mid x \text{ is a maximal point in the MVG}\}$$

Now for each $x \in M^+$, make $\ell(x)$ equal to the maximum of the labels of the neighbours of x in the MVG. This resolves all monotonicity violations for x, so we can remove all edges with x as endpoint from E, and remove x from r. End while.

For every run of this while-loop, we remove at least one element from r, because of lemma 3. Since we continue until r is empty, all monotonicity violations are resolved if the algorithm terminates, and since the size of r is decreased in every step, we are sure that the algorithm will terminate.

In every step of this while-loop, we perform two relabelings. For each of these relabelings, each data point is either not relabeled or relabeled to the label of a neighbour of the point in the MVG. So, all these relabelings are contagious (see section 2.3.1). For contagious relabelings, we have lemma 1, which says that these relabelings cannot induce new monotonicity violations.

2.4.3 Implementation of the general case

The pseudocode of the algorithm from the previous subsection is given in algorithm 3. The pseudocode almost literally implements the steps of the algorithm as described on the previous page, with some added if-then-else loops to prevent the algorithm from crashing.

Algorithm 3 generalMonotonify(D)

```
    ⟨lines 1,...,5 from algorithm 1⟩
    r ← toBeRelabeled(D)
    ⟨lines 6,...,10 from algorithm 1⟩
    while all(size(r))≥[1 1] do
 5:   mMin ← []
      for i = 1 to length(r) do
        p ← r(i)
        if sum(mvgAdjMatrix(:,p))==0 then
          mMin ← [mMin p]
10:   for i = 1 to length(mMin) do
        p ← mMin(i)
        ℓ ← y(p)
        for j = 1 to n do
          if mvgAdjMatrix(p,j)==1 then
15:         if y(j) < ℓ then
              ℓ ← y(j)
            mvgAdjMatrix(p,j) ← 0
        y(p) ← ℓ
      r ← r\ mMin
20:   if all(size(r))≥[1 1] then
        mMax ← []
        for i = 1 to length(r) do
          p ← r(i)
          if sum(mvgAdjMatrix(p,:))==0 then
25:         mMax ← [mMax p]
        for i = 1 to length(mMax) do
          p ← mMax(i)
          ℓ ← y(p)
          for j = 1 to n do
30:         if mvgAdjMatrix(j,p)==1 then
              if y(j) > ℓ then
                ℓ ← y(j)
              mvgAdjMatrix(j,p) ← 0
          y(p) ← ℓ
35:     r ← r\ mMax
    return [x y]
```

2.5 Total ordering

In the previous section, we concluded that we cannot use our two step approach with a general cost matrix and more than two classes. In this section, we will show some results in the case where we have a total ordering on our data set. This case occurs for example when there is a single input attribute, or when we assume a lexicographical order.

Throughout this section, we will assume that our data points are sorted such that $x_1 \leq x_2 \leq \ldots \leq x_n$. Therefore the statements $i < j$ and $x_i < x_j$ are equivalent.

2.5.1 Property of the connected components

Now that we have a total ordering, it turns out that every connected component of the MVG will consist only of consecutive x_i's. Formally:

Lemma 4. *If x_i is in the same connected component of the MVG as x_j for a certain $j \geq i+2$, then $\forall_{k \in \{i,\ldots,j\}} x_k$ is also in that connected component.*

Proof. We shall prove the lemma by contradiction. Suppose that x_i is in the same connected component as x_j but x_k is not, with $j \geq i+2$ and $i < k < j$. Since x_i and x_j are in the same connected component, we must have one of four different situations:

1. x_i and x_j are neighbours in the MVG;
2. $\exists_{h<i}$ s.t. x_h and x_i, and x_h and x_j are neighbours in the MVG;
3. $\exists_{m>j}$ s.t. x_m and x_j, and x_m and x_i are neighbours in the MVG;
4. $\exists_{h<i,m>j}$ s.t. x_h and x_i, x_h and x_m, and x_m and x_j are neighbours in the MVG;

By symmetry, case 3 is the same as case 2, so we do not have to consider it.

In case 1, since x_i and x_j are neighbours in the MVG and $x_i < x_j$, $\ell(x_i) > \ell(x_j)$. Since x_k is not in the same connected component as x_i and x_j, and since $x_i < x_k < x_j$, we know that $\ell(x_k) > \ell(x_i)$ and $\ell(x_k) < \ell(x_j)$.

Combining these inequalities gives us $\ell(x_k) > \ell(x_i) > \ell(x_j) > \ell(x_k)$, which is a contradiction.

In case 2, since x_h and x_j are neighbours in the MVG and $x_h < x_j$, $\ell(x_h) > \ell(x_j)$. Since x_k is not in the same connected component as x_i and x_j and consequently as x_h, and since $x_h < x_i < x_k < x_j$, we know that $\ell(x_k) > \ell(x_h)$ and $\ell(x_k) < \ell(x_j)$. Combining these inequalities gives us $\ell(x_k) > \ell(x_h) > \ell(x_j) > \ell(x_k)$, which is a contradiction.

In case 4, since x_h and x_m are neighbours in the MVG and $x_h < x_m$, $\ell(x_h) > \ell(x_m)$. Since x_k is not in the same connected component as x_i and x_j and consequently as x_h and x_m, and since $x_h < x_i < x_k < x_j < x_m$, we know that $\ell(x_k) > \ell(x_h)$ and $\ell(x_k) < \ell(x_m)$. Combining these inequalities gives us $\ell(x_k) > \ell(x_h) > \ell(x_m) > \ell(x_k)$, which is a contradiction. □

So, in the case of a total ordering, any connected component of the MVG must consist of one or more consecutive x_i's.

2.5.2 Upper and lower records

Using the lemma from the previous subsection, we can rather easily determine the connected components of the MVG. Before we describe how this is done, we need some definitions. For $i = \{1, \ldots, n\}$, define:

- $a_i = \max_{j=\{1,\ldots,i\}} \ell(x_j)$, i.e. the highest label in $\{x_1, \ldots, x_i\}$
- $b_i = \min_{j=\{i,\ldots,n\}} \ell(x_j)$, i.e. the lowest label in $\{x_i, \ldots, x_n\}$

We also call the a-values the *upper records* starting at x_1, and the b-values the *lower records* starting at x_n. One can compute these records in $\mathcal{O}(n)$ time by simply walking over the labels from x_1 to x_n for the a_i and from x_n to x_1 for the b_i.

We can deduce some properties of the upper and lower records directly from their definitions:

1. $\forall_{i=1}^{n}\ a_i \geq \ell(x_i)$ and $b_i \leq \ell(x_i)$, so we must have $a_i \geq b_i$;
2. $\forall_{i=1}^{n-1}\ a_i \leq a_{i+1}$;
3. $\forall_{i=1}^{n-1}\ b_i \leq b_{i+1}$.

2.5.3 Calculating the connected components

We will now show how we can compute the connected components of the MVG. This is done in $\mathcal{O}(n)$ time.

The connected components of the MVG can be computed by traversing over our data from x_1 to x_n and applying the following properties:

1. if $a_i = b_i$, then $\{x_i\}$ is a connected component of the MVG, i.e. x_i is not connected to any other data point;
2. if $a_i > b_i$, then x_i and x_{i+1} are in the same connected component of the MVG if and only if $b_{i+1} < a_i$.

Because of lemma 4, this description gives all possible information about the connected components of the MVG. Of course, we need to show that these properties actually hold.

Lemma 5. *The above mentioned properties hold.*

Proof.

1. Suppose that $a_i = b_i$. Property 1 of the records show us that $\ell(x_i) \leq a_i$ and $b_i \leq \ell(x_i)$, so $\ell(x_i) \leq a_i = b_i \leq \ell(x_i)$. Thus, $a_i = \ell(x_i) = b_i$, and we have two properties:
   - $a_i = \max_{j=\{1,\ldots,i\}} \ell(x_j) = \ell(x_i)$, so $\forall_{j \leq i}\, \ell(x_j) \leq \ell(x_i)$;
   - $b_i = \min_{j=\{i,\ldots,n\}} \ell(x_j) = \ell(x_i)$, so $\forall_{j \geq i}\, \ell(x_j) \geq \ell(x_i)$.

   Hence, x_i does not violate monotonicity with any other data point, and forms a connected component of the MVG by itself.
2. Suppose that $a_i > b_i$.
   - If $b_{i+1} < a_i$, then there are $h \leq i < i+1 \leq k$ such that $\ell(x_h) = a_i$ and $\ell(x_k) = b_{i+1}$. So we have $x_h < x_k$ and $\ell(x_h) = a_i > b_{i+1} = \ell(x_k)$, so x_h and x_k are neighbours in the MVG. Hence, by lemma 4, x_i and x_{i+1} are in the same connected component of the MVG.

- If $b_{i+1} \geq a_i$, then by definition of a_i and b_i,

$$\max_{j=\{1,\ldots,i\}} \ell(x_j) \quad \leq \quad \min_{k=\{i+1,\ldots,n\}} \ell(x_k)$$

  Therefore, no data point from $\{x_{i+1}, \ldots, x_n\}$ can violate monotonicity with any other data point from $\{x_1, \ldots, x_i\}$, so x_i and x_{i+1} are not in the same connected component of the MVG.

Hence, x_i and x_{i+1} are in the same connected component of the MVG if and only if $b_{i+1} < a_i$.

□

So, we have an efficient way to calculate the connected components of the MVG. This result is potentially useful, since efficient computation of the MVG may be serviceable in the next step. If we restrict ourselves to contagious relabelings the efficient calculation of the connected components is particularly useful; on page 30 we showed that in this case we can relabel each connected component of the MVG, without taking the other components into account.

Chapter 3

Step Two: making a monotone classifier

3.1 Overview of step two

This chapter deals with the second step of our two-step approach: extend the classifier we made in the previous chapter to the whole input space. While the previous chapter was highly theoretical, this chapter is much more focused on experiments.

First, in the next section, we discuss the algorithms we designed to solve the problem. These are variants of the k-nearest neighbour algorithm, modified so that the newly labeled points will comply with the monotonicity constraint. We describe the variants, and give an example that shows that the variants behave differently.

In section three, we discuss the implementation of the algorithms from section two. We meticulously describe what is implemented, and discuss the corresponding pseudocode.

The last two sections are dedicated to experiments. In section four, we experiment with the algorithms from section three on binary classification problems. We describe the experiments and the data sets the experiments are run on, and discuss the results. In section five, we repeat the experiments from the section before on classification problems with four classes.

3.2 Monotone kNN variants

We want our new classifier to be nonparametric, and the canonical example of such a classifier uses the kNN algorithm. Now, if we want to classify a new data point given a certain data set, standard kNN does not take into account the restraints imposed on the label of the new point by monotonicity (see the description of the standard kNN algorithm in section 1.2.1). We propose two variants of kNN that do comply with such restraints.

3.2.1 The algorithms

In both the algorithms, let $D = \{(x_i, \ell_i)\}, \quad i = 1, \ldots, n$ be the data set, monotonified by the algorithms from section 2.4. Also, let x_0 be the data point we want to classify, and let k be a positive integer.

We first need an auxiliary algorithm that calculates the labels that the new data point is allowed to have according to the data set. The implementation of this algorithm is given as algorithm 4.

Algorithm 4 allowedLabels(D, x_0)

```
t ← max{ℓ_i | (x_i, ℓ_i) ∈ D}
b ← min{ℓ_i | (x_i, ℓ_i) ∈ D}
for i = 1 to n do
  if x_i ≤ x_0 and ℓ_i > b then
5:   b ← ℓ_i
  if x_i ≥ x_0 and ℓ_i < t then
    t ← ℓ_i
return {b, b + 1, ..., t}
```

Now we can give our kNN variants. The implementation of monotone kNN variant 1 is given as algorithm 5. This variant simply takes the k nearest neighbours, discards those whose labels are not allowed, and does majority voting on the remaining labels. Thus, this variant actually uses at most k neighbours. In the case where there are no remaining labels, we invoke a tie breaking rule. These are described in general in subsection 3.2.3.

The implementation of monotone kNN variant 2 is given as algorithm 6. This variant always uses exactly k neighbours. It takes the k nearest neigh-

Algorithm 5 Monotone kNN variant 1 (D, x_0, k)

$\ell^- \leftarrow \min(\text{allowedLabels}(D, x_0))$
$\ell^+ \leftarrow \max(\text{allowedLabels}(D, x_0))$
sort D according to ascending euclidean distance of x_i to x_0
labels $\leftarrow \{\ell_i \mid (x_i, \ell_i) \in$ the first k elements of $D\}$
allowed $\leftarrow \{\ell_i \in \text{labels} \mid \ell^- \leq \ell_i \leq \ell^+\}$
return label, selected from allowed by majority voting

bours, discards the n_1 neighbours whose labels are not allowed, adds the $k+1^{st}, \ldots, k+{n_1}^{th}$ nearest neighbours (so that we have k candidates again), discards the n_2 neighbours whose labels are not allowed, and so on until we keep k labels. The algorithm then does majority voting on the labels.

This monotone kNN variant with data set D and point x_0 is equal to the standard kNN procedure with data set D' and point x_0, where D' is D minus all data points with labels that are not allowed for x_0.

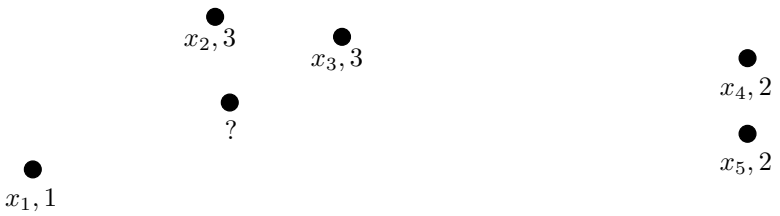
Algorithm 6 Monotone kNN variant 2 (D, x_0, k)

$\ell^- \leftarrow \min(\text{allowedLabels}(D, x_0))$
$\ell^+ \leftarrow \max(\text{allowedLabels}(D, x_0))$
sort D according to ascending euclidean distance of x_i to x_0
votes $\leftarrow \emptyset$
repeat
  $r \leftarrow k-\text{length}(\text{votes})$
  labels $\leftarrow \{\ell_i \mid (x_i, \ell_i) \in$ the first r elements of $D\}$
  $D \leftarrow D \backslash \{$the first r elements of $D\}$
  newvotes $\leftarrow \{\ell_i \in \text{labels} \mid \ell^- \leq \ell_i \leq \ell^+\}$
  votes $\leftarrow$ votes $\cup$ newvotes
until length(votes)$= k$
return label, selected from votes by majority voting

3.2.2 Example: the difference between the variants

Depending on the data, the two variants we just defined may behave quite differently. Suppose for instance that we have the data set from figure 3.1.

Figure 3.1: Data set



The allowed labels for our new data point are restricted by data points x_1 and x_4, so we can only choose from $\{1, 2\}$. Let $k = 3$.

Monotone kNN variant 1 will find the three nearest neighbours to be x_2, x_3, and x_1, so we get the labels $3, 3$, and 1. The threes are not allowed, so we discard them. We then do majority voting on the set $\{1\}$, and assign our new data point to class 1.

Monotone kNN variant 2 will follow the same reasoning up to discarding the threes. But then, since we just discarded two labels, we will add the next two nearest neighbours to our pool, i.e. x_4 and x_5. We get the labels 2 and 2 next to the 1 we already had, and all of these are allowed. We then do majority voting on the set $\{1, 2, 2\}$, and assign our new data point to class 2.

3.2.3 Tie breaking rules

Both monotone kNN variants and standard kNN finally assign the data point to a label by majority voting on a set of labels. However, in this voting process, we may get a tie. Such a tie needs to be broken, and for this we propose two tie breaking rules. Tie breaking rule 1 considers the set of all labels that are tied for the first place, and picks one of these at random. Tie breaking rule 2 adds extra points to the pool, one at a time, until the tie for the first place is broken.

When using monotone KNN variant 1, we may encounter the pathological case where we try to do majority voting on an empty set. Tie breaking rule 2 still solves this case. For tie breaking rule 1 we consider the set of all labels the data point is allowed to have given the data set, and picks one of these at random.

We will revisit the tie breaking rules in section 3.4.

3.3 Implementation for binary classification

In chapter 2, we described an algorithm that indicates which points need to be relabeled to make a given data set monotone. In the previous section, we described two algorithms that extend a monotone classifier which is defined only on the data set to the whole input space. All is set to combine these algorithms into one large algorithm that performs monotone kNN given a data point and a (not necessarily monotone) data set.

3.3.1 Standard kNN

The implementation of standard kNN is given as algorithm 7.

Lines 4 and 5 use the MATLAB code for manipulating matrices; keep in mind that the colon denotes a range. So, line 4 selects all rows and columns $1, \ldots, k$, and line 5 selects all rows and column $k+1$.

Line 6 uses the nearestneighbour function, to be found at [3]. This function gives the indices of the k points in x with the smallest euclidean distance to x_0. The rest of the code in this line takes the labels of these points and sorts them.

Line 10 uses the auxiliary algorithm 'voteCount', whose pseudocode can be found in subsection 3.3.4. This algorithm takes the sorted labels found by the nearestneighbour algorithm (a $1 \times k$-matrix), and returns a $2 \times k$-matrix whose first row contains the distinct labels, and the second row contains the number of times this label appears in the input argument. The columns of the matrix are sorted such that the second row is non-increasing.

In line 12, we simply test whether there is more than one distinct label. If not, the line after would crash our algorithm.

If the condition in line 13 is met, we have a tie and need to apply a tie breaking rule. Which rule is applied depends on the switch 'TBvar' that is an input argument to the algorithm, and needs to have either value 1 or 2.

Line 21 simply says: pick one of the best labels at random.

Algorithm 7 standardKNN(D, x_0, k, TBvar)

```
    dimensions ← size(D)
    n ← dimensions(1)
    k ← dimensions(2)-1
    x ← D(:, 1 : k)
 5: y ← D(:, k + 1)
    ℓ ← sort(y(nearestneighbour(x_0, x, k)))
    if size(ℓ)==1 then
      return ℓ(1)
    else
10:   votes ← voteCount(ℓ)
      maxVotes ← votes(2, 1)
      if all([1 1]<size(votes)) then
        if votes(2, 2)==maxVotes then
          switch TBvar
15:         case 1
              i ← 1
              while all([1 i]<size(votes)) & votes(2, i + 1)==maxVotes
                i ← i + 1
              end while
20:           bestLabels ← votes(1, 1 : i)
              l ← bestLabels(1+fix(length(bestLabels)·rand(1)))
            case 2
              while votes(2, 2)==maxVotes
                k ← k + 1
25:             ℓ ← sort(y(nearestneighbour(x_0, x, k)))
                votes ← voteCount(ℓ)
                maxVotes ← votes(2, 1)
              end while
              l ← votes(1, 1)
30:       end
          return l
        else
          return votes(1, 1)
      else
35:     return votes(1, 1)
```

The code in the case starting at line 22 actually acts in the following way: while we still have a tie, try again with $k = k + 1$.

3.3.2 Monotone kNN

The implementation of monotone kNN is given as algorithm 8. It has a large intersection with the implementation of standard kNN.

Algorithm 8 monotoneKNN(D, x_0, k, kNNvar, TBvar)

```
    a ← allowedLabels(D, x_0)
    b ← min a
    t ← max a
    if b==t then
 5:   return b
    else
      ⟨lines 1, ..., 5 from algorithm 7⟩
      ℓ ← nearestLabels(x, y, x_0, k, b, t)
      if kNNvar==2 then
10:     q ← k
        while size(ℓ)< k do
          q ← q + k−size(ℓ)
          ℓ ← nearestLabels(x, y, x_0, q, b, t)
        k = q
15:   ℓ ← sort(ℓ)
      ⟨lines 7, ..., 24 from algorithm 7⟩
              ℓ ← sort(nearestLabels(x, y, x_0, q, b, t))
      ⟨lines 26, ..., 38 from algorithm 7⟩
```

The case in line 5 is the case in which we have no choice for the label, so that makes it easy.

Line 8 uses the auxiliary algorithm 'nearestLabels', whose pseudocode can be found in subsection 3.3.4. This algorithm does the same as the nearest-neighbour procedure we described in the previous subsection, except that it discards labels that are not allowed; the lowest and highest allowed label (b and t) are input arguments for the algorithm.

The statement in line 14 is necessary if we end up in tie breaking rule 2. In this rule k is increased, so we need to work with the updated k.

3.3.3 Cross-validation

We implemented one variant of cross-validation: 10-fold cross-validation. Its pseudocode is given als algorithm 9.

Notice that 10-fold cross-validation is not well-defined for monotone kNN. After we built a monotone classifier from our training data, we start to classify our test data. However, if we classify the m^{th} point from our test data, we may want to take into account the $1^{st}, \ldots, m-1^{th}$ points from our test data. Since we already classified these points, their label may affect the allowed labels for the points we still are to classify. But then, the order in which we classify the test data points becomes important.

In order to avoid these problems, our implementation of 10-fold cross-validation for monotone kNN simply ignores the points from the test data we already classified. This may lead to monotonicity violations in our predictions, and it will probably lead to a reduced effect of our taking monotonicity into account, whether that effect would be positive or negative.

Line 6 uses the standard MATLAB function 'randperm', that takes a positive integer n and returns a random permutation of $1, \ldots, n$.

10-fold cross-validation partitions the data in ten subsets of equal size, but the total number of data points need not be a multiple of 10. The code in line 10 ensures that the extra points are also in one of the subsets. In our experiments we do not stratify our samples with respect to the classes.

3.3.4 Auxiliary algorithms

This subsection contains the pseudocode for all auxiliary algorithms. What these algorithms do is not explained in this subsection, since that would be a rerun of subsections 3.3.1 and 3.3.2.

The pseudocode for the algorithm 'voteCount' is given in algorithm 10.

Algorithm 9 tenFoldCrossvalidate(D, k, kNNvar, TBvar)

```
    ⟨lines 1,...,5 from algorithm 1⟩
    stdKNNerr ← 0
    monKNNerr ← 0
    subsetsize ← ⌊n/10⌋
 5: extrapoints ←  mod (n, 10)
    permutation ← randperm(n)
    for i = 1 to 10 do
      interval ←  1 + (i − 1)·subsetsize : i·subsetsize
      if i ≤ extrapoints then
10:     interval ← [interval n − i + 1]
      currentX ← x(interval,:)
      realLabels ← y(interval)
      restX ← x;  restX(interval,:)←[]
      restY ← y;  restY(interval)←[]
15:   D′ ← [restX restY]
      D″ ← monotonify([restX restY])
      for j = 1 to size(interval) do
        stdLabel ← standardKNN(D′, currentX(j,:), k, TBvar)
        monLabel ← monotoneKNN(D″, currentX(j,:), k, kNNvar, TBvar)
20:     if realLabels(j) ≠ stdLabel then
          stdKNNerr ← stdKNNerr +1
        if realLabels(j) ≠ monLabel then
          monKNNerr ← monKNNerr +1
    return stdKNNerr, monKNNerr
```

Algorithm 10 voteCount(labels)

```
    count ← [labels(1);1]
    cnt ← 1
    for i = 2 to size(labels) do
      if labels(i)==count(1,cnt) then
5:      count(2,cnt) ← count(2,cnt)+1
      else
        count ← [count [labels(i);1]]
        cnt ← cnt +1
    count ← count^T
10: count ← [count(:, 2) count(:, 1)]
    count ← sortrows(count)
    count ← [count(:, 2) count(:, 1)]
    count ← count^T
    return fliplr(count)
```

Keep in mind that the labels are supposed to be sorted. The variable 'count' builds the matrix that the algorithm is supposed to return, while the variable 'cnt' indicates which column of 'count' we are filling right now.

The right-hand side of line 1 basically is MATLAB's notation for a 2×1-matrix.

At line 8 we have completed counting the labels. The six lines hereafter only perform the sorting of the matrix the algorithm requires, using standard functions of MATLAB.

The pseudocode for the algorithm 'nearestLabels' is given in algorithm 11.

Algorithm 11 nearestLabels(x, y, x_0, k, b, t)

```
    labels ← y(nearestneighbour(x_0, x, k))
    toBeIgnored ← []
    for i = 1 to size(labels) do
      if labels(i)< b | t <labels(i) then
5:      toBeIgnored ← [toBeIgnored i]
    labels(toBeIgnored) ← []
    return labels
```

3.4 Experiments for binary classification

Now that we have implemented monotone kNN algorithms, it is time to assess their performance on some data sets, which we assume to represent monotone classification problems.

3.4.1 The data sets

We obtained seven data sets that represent monotone problems. Six of these come from the UCI data repository to be found at [1], and the seventh can be found in [7].

Some characteristics of the data sets are given in table 3.1. The mentioned number of attributes is after modification of the data sets, as described in the following paragraphs.

Table 3.1: Data sets with some characteristics

Data set	# points	# attributes	Comparability
Australian credit approval	690	4	0.7162
AutoMpg	392	7	0.4009
Boston housing	506	12	0.1910
Haberman's survival	306	3	0.3123
Machine (cpu performance)	209	6	0.4950
Pima indians diabetes	768	8	0.0732
Windsor housing	546	11	0.2737

Some of the mentioned data sets did represent monotone regression problems, and to transform them into classification problems we discretized the dependent variables into two classes. We assigned a data point to class 1 if its value for the dependent variable is lower than the mean, and to class 2 otherwise.

The last column of the table displays the comparability of the data sets; this is the number of pairs that are comparable, divided by the total number of pairs.

For some data sets the number of incomparable pairs was very high, so in order to obtain more interesting results we dropped some columns. The

dropped attributes are:

- Australian: all but columns $7, 8, 9, 10$, and 15[1], where the latter represents the class label;
- Boston: Charles River dummy variable;
- Machine: vendor name, model name, estimated relative performance.

Some attributes have a negative influence with respect to the class variable. On each attribute x with a negative correlation coefficient, we perform the transformation $x_i = 1/x_i$ for $i = 1, \ldots, n$. This transformation has been applied to the following attributes:

- AutoMpg: Cylinders, Displacement, Horsepower, and Weight;
- Boston: Crime rate, Non-retail business acres, Nitric oxides concentration, Units built prior to 1940, Accessibility to highways, Property-tax rate, Pupil-teacher ratio, and Lower status;
- Haberman: Year of operation;
- Machine: Cycle time.

3.4.2 Testing for monotonicity

We assume the problems the data sets are derived from to be monotone. It could be useful for interpretation of the results of our experiments, to quantify the 'monotonicity' of the data sets.

Our test for monotonicity centers around the number of monotonicity violations. We first count the number of non-monotone pairs in our data set. Then we generate a thousand data sets, by copying the attributes and taking a random permutation of the class labels. We then count the number of non-monotone pairs in these data sets. This way, we obtain some sort of probability distribution of the number of non-monotone pairs. The null hypothesis is that the process is not monotone, and we reject this hypothesis if the number of non-monotone pairs in our original data set lies in the left tail of this distribution.

[1] this data set concerns credit card applications. To protect confidentiality, all attribute names have been changed to meaningless symbols.

For each data set we computed the number of non-monotone pairs, the mean number of non-monotone pairs in the thousand generated data sets, the corresponding standard deviation, and the score. This score is the number of generated data sets that the original data set beats[2], so it must be an integer between 0 and 1000 and of course, we would like it to be as close to 1000 as possible. The results can be found in table 3.2.

Table 3.2: Monotonicity test results: two classes

Data set	# pairs	Mean	Std.dev.	Score
Australian	8087	42649.0	2258.1	1000
AutoMpg	21	7716.5	733.3	1000
Boston	309	6113.0	612.5	1000
Haberman	1784	2848.2	379.4	999
Machine	86	2714.0	311.4	1000
Pima	482	4923.9	569.2	1000
Windsor	429	10187.0	875.1	1000

The good news is: all data sets are quite monotone. The bad news is that these results do not help us to interpret experimental results, since we cannot distinguish between the monotonicity of the data sets (since every data set is far in the tail of these probability distributions).

3.4.3 Experiments

On every data set, we tested our monotone kNN against standard kNN, for $k = 1, 3, 5$. We first calculated the results of leave-one-out cross-validation on the AutoMpg, Haberman, and Pima data sets. It turned out that the results for all monotone kNN algorithms[3] were equal. Therefore, we decided to test only monotone kNN variant 1 with tie breaking rule 1.

In these tests, we measure the performance of the classifiers using the error rate: the number of misclassified data points divided by the total number of data points. This corresponds to the 0/1 cost matrix case. We applied 10-fold cross-validation.

[2] meaning that the number of non-monotone pairs is smaller

[3] Two variants times two tie breaking rules yields four algorithms.

The results of the experiments can be found in table 3.3. For every data set, the best result for every algorithm is colored blue.

Table 3.3: Error rate $\pm$ standard deviation of the folds

Australian	$k = 1$	$k = 3$	$k = 5$
std kNN	0.233 ± 0.040	0.212 ± 0.038	0.183 ± 0.036
mon kNN	0.164 ± 0.053	0.164 ± 0.053	0.164 ± 0.053

AutoMpg	$k = 1$	$k = 3$	$k = 5$
std kNN	0.087 ± 0.034	0.092 ± 0.050	0.089 ± 0.062
mon kNN	0.079 ± 0.035	0.084 ± 0.042	0.084 ± 0.053

Boston	$k = 1$	$k = 3$	$k = 5$
std kNN	0.205 ± 0.096	0.231 ± 0.077	0.233 ± 0.099
mon kNN	0.188 ± 0.079	0.204 ± 0.088	0.188 ± 0.093

Haberman	$k = 1$	$k = 3$	$k = 5$
std kNN	0.330 ± 0.090	0.265 ± 0.090	0.268 ± 0.074
mon kNN	0.284 ± 0.095	0.285 ± 0.090	0.295 ± 0.084

Machine	$k = 1$	$k = 3$	$k = 5$
std kNN	0.153 ± 0.102	0.153 ± 0.092	0.167 ± 0.084
mon kNN	0.148 ± 0.111	0.152 ± 0.116	0.157 ± 0.119

Pima	$k = 1$	$k = 3$	$k = 5$
std kNN	0.293 ± 0.055	0.257 ± 0.067	0.259 ± 0.075
mon kNN	0.262 ± 0.055	0.259 ± 0.078	0.258 ± 0.065

Windsor	$k = 1$	$k = 3$	$k = 5$
std kNN	0.276 ± 0.089	0.276 ± 0.068	0.264 ± 0.052
mon kNN	0.218 ± 0.057	0.211 ± 0.061	0.209 ± 0.062

Monotone kNN performs better than standard kNN on the Australian, AutoMpg, Boston, Machine and Windsor data sets. Standard kNN performs better than monotone kNN on the Haberman and Pima data sets.

Notice that taking monotonicity into account is the most profitable for lower k, in particular for $k = 1$.

3.5 Experiments for the general case

Of the data sets we used in section 4, Australian, Haberman, and Pima already had two class labels. The other four data sets represented monotone regression problems, so we can transform them into classification problems with any number of classes we want. For our experiments we used four classes, and divided the data points evenly over the classes.

3.5.1 Testing for monotonicity

Since many points have received another label than in section 3.4, we redid the test for monotonicity from section 3.4.2. The results can be found in table 3.4.

Table 3.4: Monotonicity test results: four classes

Data set	# pairs	Mean	Std.dev.	Score
AutoMpg	74	11557.0	816.4	1000
Boston	691	9175.8	739.4	1000
Machine	167	4070.2	359.2	1000
Windsor	1328	15302.0	970.2	1000

3.5.2 Experiments

We again measure the performance of the classifiers using the error rate. Again, we tested with 10-fold cross-validation. We tested monotone kNN variant 2 against standard kNN, for $k = 1, 3, 5$, with tie breaking rule 1. The results from our experiments can be found in table 3.5.

We also tested the two monotone kNN variants against each other. On all these test instances, the difference in error rate was smaller than 0.013, in both directions. In other words: the difference is negligible.

With four classes instead of two, monotone kNN performs better than standard kNN on all data sets.

Table 3.5: Error rate $\pm$ standard deviation of the folds

AutoMpg	k = 1	k = 3	k = 5
std kNN	0.222 ± 0.063	0.248 ± 0.068	0.240 ± 0.064
mon kNN	0.214 ± 0.038	0.230 ± 0.053	0.225 ± 0.051

Boston	k = 1	k = 3	k = 5
std kNN	0.500 ± 0.101	0.496 ± 0.125	0.501 ± 0.143
mon kNN	0.439 ± 0.121	0.439 ± 0.133	0.427 ± 0.122

Machine	k = 1	k = 3	k = 5
std kNN	0.397 ± 0.100	0.397 ± 0.141	0.415 ± 0.146
mon kNN	0.339 ± 0.145	0.334 ± 0.184	0.344 ± 0.154

Windsor	k = 1	k = 3	k = 5
std kNN	0.523 ± 0.107	0.536 ± 0.091	0.524 ± 0.085
mon kNN	0.478 ± 0.081	0.467 ± 0.085	0.463 ± 0.076

Notice that monotone kNN also performed better than standard kNN on all these data sets when we had two classes. However, the difference between the two algorithms is larger, especially on the Boston and Machine data sets.

Chapter 4

Conclusions

4.1 Summary and conclusions

We have discussed algorithms for building a nonparametric monotone classifier. This classifier is built in two steps: first we make the data set monotone, and then we make a classifier out of that monotone data set.

In the case of binary classification, the data can be made monotone under both 0/1 and general cost matrices. In the case where we have an arbitrary number of class labels, the monotone data set can be made under the 0/1 cost matrix. We showed that there are theoretical objections against using our approach (involving the use of the MVG) for this case under general cost matrices.

The classifiers we made out of the monotone data set are modifications of the standard kNN classifier. We made multiple variants of such classifiers with different behaviour in theory, but they turned out to behave more or less the same on real data sets.

We tested our monotone kNN classifier against standard kNN on seven data sets that we assumed to represent monotone binary classification problems. Monotone kNN performed better on five of these sets, and standard kNN performed better on the other two. We then modified four of the data sets, such that the classification problems is no longer binary, but has four classes. Monotone kNN performed better than standard kNN on all of these sets.

Taking monotonicity into account is more profitable for lower k, especially for $k = 1$. This may be because standard kNN tends to overfit if $k = 1$, while the data points that cause this overfitting probably have labels that are not allowed if we take monotonicity into account.

4.2 Related work revisited

This section compares our work with the related work we have described in section 1.5.

4.2.1 Ben-David revisited

The OLM has two important shortcomings from which our work does not suffer:

- the composition of the final rule base depends on the order in which the examples are processed. In particular, it is not guaranteed that the final rule-base contains the largest monotone subset of the training set;
- due to the nearest neighbour rule invoked when there are no $x_i \leq x_0$, non-monotone prediction results are possible.

4.2.2 Dykstra et al. revisited

The algorithms by Dykstra et al. minimizes absolute and squared error cost. These cost matrices both assume more than just an order on the classes. In our work, these classes usually are labeled $1, \ldots, c$ (where c is the number of classes) for convenience, but that does not imply that performing numerical operations on them is meaningful. If we do not have more information on the nature of the order on the classes, then minimization of misclassification cost is more appropriate.

The algorithms by Dykstra et al. make extensive use of isotone regression. If we have a total order on the input space, there is an $\mathcal{O}(n)$ algorithm for isotone regression. Our work is not that computationally efficient. In section 2.5, we do give some results that may lead to a computational efficient algorithm for this case.

4.3 Further research

Not only did this thesis answer some questions, it also raised some. Here are some topics that need further research.

- Section 2.4.3 gives a relabeling algorithm for the case where we have an arbitrary number of classes. In this case, there may (and probably will) be many optimal relabelings when minimizing 0/1 cost. It may be interesting to develop other relabeling algorithms for this case, and compare their behaviour. For instance, how do the results change if we treat the maximal points first and then the minimal points, instead of the other way around?
- Our implementation of the general classes case works only in the 0/1 cost matrix case. The general classes case with a general cost matrix remains an open problem. It will be interesting to develop a relabeling algorithm for the general cost case that minimizes a general cost matrix.
- Section 2.5 gives an algorithm with which we can quickly determine the connected components of the MVG in the case where we have a total order on our input space rather than a partial order. The properties we proved in that section may be useful for solving the open problem of the general classes case with a general cost matrix, or for improving on the algorithms we described in this thesis, in terms of computational complexity.

4.4 Bibliography

[1] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007. http://www.ics.uci.edu/~mlearn/MLRepository.html

[2] Arie Ben-David. Automatic Generation of Symbolic Multiattribute Ordinal Knowledge-Based DSSs: Methodology and Applications. *Decision Sciences*, 23 (6): 1357–1372, 1992.

[3] R.G. Brown. nearestneighbour.m - kNN implementation for MATLAB. http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12574&objectType=FILE

[4] R. Dykstra, J. Hewett, and T. Robertson. Nonparametric, Isotonic Discriminant Procedures. *Biometrika*, 86 (2): 429–438, 1999.

[5] Ad Feelders, Marina Velikova, and Hennie Daniels. Two polynomial algorithms for relabeling non-monotone data. Technical Report UU-CS-2006-046, Institute of Information and Computing Sciences, Utrecht University, 2006. http://www.cs.uu.nl/research/techreps/repo/CS-2006/2006-046.pdf

[6] S.P. Iglin. grTheory - graph theory toolbox for MATLAB. http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=4266&objectType=FILE

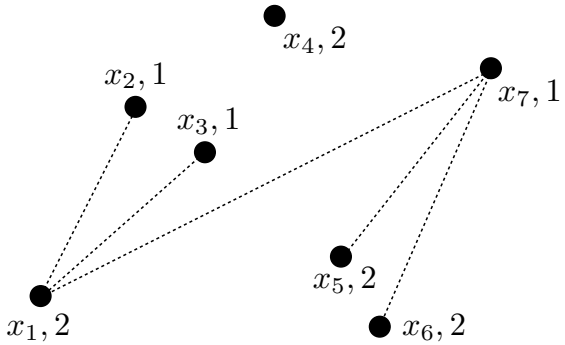
[7] Gary Koop. *Analysis of Economic Data*. Wiley, 2nd edition, 2004.

[8] MATLAB Central. http://www.mathworks.com/matlabcentral/

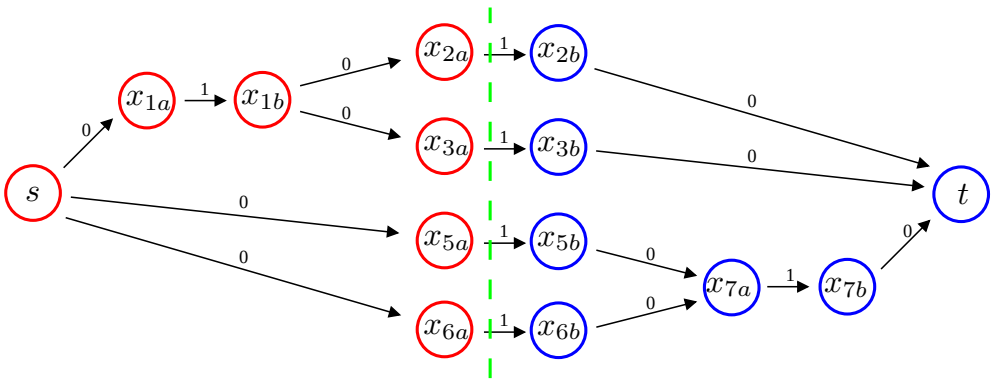
[9] MATLAB Software. http://www.mathworks.com/products/matlab/

[10] R.H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. *Graphs and Order*, pages 41–101, 1985.

[11] M. Rademaker, B. De Baets, and H. De Meyer. Data sets for supervised ranking: to clean or not to clean. In *Proceedings of the fifteenth Annual Machine Learning Conference of Belgium and The Netherlands: BENELEARN 2006*, pages 139–146, Ghent, Belgium, 2006.



$$C = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}:$$



$$C = \begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix}:$$