Utrecht University

ICA-0367176

Exploiting Monotonicity Constraints in Active Learning

Author:
Pieter Soons

Supervisor:
Ad Feelders

April 27, 2013

Contents

Abstract

If one wants to build a classifier, but the available data is not classified, active learning can be a good solution. If the data in question has the monotonicity property, we can classify much more data using fewer queries. The data is said to be monotone if the class label is known to be increasing or decreasing in the attributes. The goal of active learning with monotonicity constraints is to use queried and monotonically inferred data to build a classifier. A central problem in active learning is finding the best query points. It has been suggested to use the concept of lower sets to search in the partially ordered sets (posets) that represent the dataset [3], [2], [8]. We show that this concept can also be used to select good query points in active learning with monotonicity constraints. Because counting lower sets becomes very costly quickly, it's hard to test this hypothesis on general posets. We provide a dynamic programming algorithm that counts all lower sets in linear time for the special case of a matrix order. We then show that this query strategy is at most a logarithmic factor worse than the optimal query strategy at classifying the whole dataset.

We contrast this strategy with a simple but good known greedy heuristic [1]. Testing shows that the lower-set-count strategy outperforms the greedy heuristic in classifying the whole dataset, but is worse when an a priori set maximum number of queries is made. Tests are performed on artificial and real datasets and the heuristics are judged based on both the quantity of inferred class labels and their quality, by means of building a classifier. These tests lead us to conclude that the greedy heuristic is very good and that the lower-set-count strategy is not a promising one for general posets. The latter conclusion is reached by reasoning that approximating lower set counts, which would be needed, will nullify the marginal edge over the greedy strategy it might have at best.

The most challenging part of creating artificial data is generating random monotone classification functions. We show how to do this without generating all of them using the Propp-Wilson algorithm. Furthermore, concerning datasets with more than two class labels, we show that when building a classifier after a restricted number of queries, the classifier can be improved substantially by using partially classified data entries in addition to the fully classified data.

Chapter 1

Introduction

Consider the staging of cancer. This is the process of assigning, for instance, a number from I to IV to a case to indicate the extent to which the cancer has spread, where stage IV is the most severe. It is used to estimate the expected survivability, but it is also used to indicate what treatment would be best to apply. This means incorrect staging can lead to incorrect treatment, which can be detrimental to the patient. What makes it difficult to stage correctly is that it is dependent on various different diagnostic tests that don't necessarily correlate much. Each of these tests in itself, though, has a well defined linear order from best case to worst case, for the most part. Examples of questions these tests answer are; What is the size of the tumor? How deeply has it penetrated? In how far has it invaded adjacent organs? How many lymph nodes has it metastasised to? Common sense suggests there is an increasing relationship between these measurements and the stage. That is, all else equal, if the size of the tumor increases, the stage does not decrease (become less severe).

It is not a novel idea to use machine learning to help with this process. An effective way of doing this, for example, would be to build a classifier that uses previous cases to assign a stage to a new case. The problem with this method, however, is that we don't always have enough of these cases, complete with a correct stage, readily available to build this classifier with. An alternative, is to take a collection of (generated) unstaged cases, and let domain experts (diagnosticians) stage them. Again, sample size is the problem. It is likely way too expensive to have enough of these cases labeled. In this thesis we will show how to combine the monotonicity of a problem like this with active learning, in order to strategically pick only a small number of cases to be labeled by experts, that nonetheless provide enough information to build an accurate classifier.

1.1 Active Learning and Monotonicity

The ultimate goal of any classification algorithm is to find a classification function $f(x)$, which takes an attribute vector $x = (x^1, x^2, \ldots, x^d)$ and produces the corresponding class label y. In reality it is of course not always clear what the correct class label is, but for the purpose of this thesis we will assume each attribute vector has a unique correct class label. In the staging example we used above, an attribute vector x is the collection of outcomes from the diagnostic tests of one patient and the class label y is the cancer stage (I - IV) of that patient.

In traditional classification we have available a training collection

$$T = \{(x_i, y_i)\}_{i=1}^{n}.$$

These examples can be used to estimate f, which in turn can be used to predict unlabeled vectors. A very simple example of such a function f would be one that assigns to unlabeled vector x the class y_i, when x_i is the vector in T that is most similar to x. This is called a nearest neighbour classifier.

In active learning it's typically assumed that we have no such T, but a set

$$X = \{(x_i)\}_{i=1}^{n}$$

of unlabeled vectors instead. There is, however, an oracle (e.g. a human annotator or domain expert, or an expensive algorithm) that will produce a class label for any vector in X when requested. The main idea is that it is expensive to make such a request and we are charged for each queried vector. Since we are not concerned with the probabilistic case of active learning, we can assume the provided label to be the correct one. In this thesis we will study a special case of active learning where some monotonic properties hold. We'll explain by continuing our cancer staging example.

We assume that for each diagnostic test we can find a linear order, such that, all other things equal, a higher (worse) outcome cannot result in a lower staging. To clarify a bit, if we have patients A and B with exactly the same test results, except for one test in which patient B got a worse result (e.g. the tumor is more more spread), then patient B must have a stage of cancer at least as high as patient A. It is of course quite common that on one test patient A scores better, but on another test B scores better. In that case the stage of patient A will tell us nothing about the stage of patient B and vice versa. If we order a set of attribute vectors by this principle, the result is a partial order on this set, a structure that can be thought of as a directed acyclic graph. This is useful in the following way: If we query the right vector, we might be able to infer a number of class labels from other vectors as well. We will explain what a partially ordered set (poset) is in more detail later.

Putting all this in mathematical terms, for each attribute let $\mathcal{X}^a, a = 1, 2, \ldots, d$ denote the set of possible values of x^a and likewise for $\mathcal{Y}$ and y. We assume that there is a linear order on $\mathcal{Y}$ and a partial order $\preceq$ on the input space $\mathcal{X}$, that can be derived from knowledge of the signs of the influences of the different $\mathcal{X}^a$ on $\mathcal{Y}$. This will typically result in a product order on $\mathcal{X}$, that is

$$x \preceq x' \Leftrightarrow \forall a : x^a \leq x'^a.$$

Together with the constraint

$$x \preceq x' \Rightarrow f(x) \leq f(x'),$$

this expresses that the class label assigned to the attribute vectors should be increasing in each attribute.

1.2 Research objective

The problem in active learning with monotonicity constraints is finding a good query strategy. A query strategy is an algorithm or heuristic that uses information about the structure of the partial order on the observed attribute vectors, and previous queries, to select the next vector to query. In [1], Barile and Feelders describe a greedy query strategy which maximizes the number of eliminated class labels in the worst case (for details on the 'GREEDY' heuristic see chapter 4). This heuristic performs well, but it doesn't use information about the structure of the whole poset to select a query. It has been suggested to use the concept of lower sets to search in posets [3], [2], [8]. In binary search, the fastest worst case search algorithm eliminates half of the search space with each query. The problem of finding the monotone classification function f is very similar. This leads to our main research objective:

Is maximizing the number of monotone classification functions that is eliminated in the worst case a good query strategy?

This strategy is optimized to find f as fast as possible. But, as we are charged significantly for each query, a good strategy needs to perform well after few queries, relative to the size of the poset. There is reason to believe this is the case, however. To perform well in the worst case, common sense suggests that a strategy will have to select a vector in the center of the poset. And the vector that eliminates close to half the number of possible monotone classification functions in the worst case is a good definition of the center of the poset. For general posets it is expensive to find this central vector exactly, which is why we pose the following sub-objectives:

Is there a special case poset for which there exists an algorithm that finds the central vector in a reasonable time?

Can we conclude anything about the number of queries this strategy needs to find the true classification function?

Is applying an approximation of this strategy to general posets a possibility that yields good results in a reasonable time?

We will see that monotone classification functions and so called lower sets are essentially the same thing for posets. Counting the number of lower sets is impractical for general posets, but approximating this number may be sufficient.

It is not trivial to define a measure of performance for the query strategies. To compare them we will look at both the quantity of the training set (number of labeled vectors) they yield after a certain number of queries, and their quality. To judge the quality, we will use a K-nearest neighbour classifier on the training set in conjunction with an unlabeled test set.

1.3 Structure of this thesis

The rest of this thesis is structured as follows. In the next chapter we will apply the ideas described so far to the simplest of posets, a chain. In chapter 3 a more realistic special case partial order is covered, namely matrix orderings. Most of the chapter deals with counting the number of lower sets. In chapter 4 we look at general posets and the GREEDY heuristic. To perform experiments, we need a random monotone classification function generator, which we will construct in chapter 5. Then, in chapter 6 we describe the way the experiments are set up, and show the result in chapter 7. Finally, in chapter 8 we look back at the research objectives and if and how they were fulfilled. In addition, we indicate possibilities for further research.

Chapter 2

Querying chains

In the previous chapter we started off with an example that had four class labels (cancer stages) as well as multiple attributes (diagnostic tests). The latter combined with the monotonic properties resulted in our data being partially ordered. Before we introduce our algorithms on posets in more detail, we will continue with the most basic poset; a chain. A chain is simply a linear order of n vectors and it is the order we get, for example, when there is only one attribute. To simplify things further, we will for now also restrict the problem to two class labels, i.e. binary version. In section 2.2 we discuss the more general case of multi-class chains.

2.1 Binary chains

The monotonicity property in this instance is easy to grasp. In the lower segment of the chain a certain number of vectors will have class label 0, and after the crossover point, each vector has class label 1. Because this crossover can be before the first vector or after the last, there are $n+1$ ways to label the chain. In other words, there are $n + 1$ monotone classification functions. We assume each of these is equally likely to occur. For the rest of this thesis, we will take *classification function* to mean *monotone classification function* (that is binary when applicable).

If we query a vector, one of two things will happen. Either the answer is class label 0, after which we can conclude that every vector lower on the chain also has class label 0. Or the answer is 1, after which we can infer that every vector higher on the chain also has class label 1. We mentioned before that it might be a good idea to query the central vector, which in this case is very clearly defined. To prove that the central vector is the best one to query let's look at what we are trying to accomplish. The GREEDY strategy we mentioned, is to query the vector that lets us infer the most other vectors in the worst case. On the other hand, we have the strategy that eliminates the largest number of possible classification functions.

But, in case of a binary chain, the number of possible classification functions is linear in the number of unlabeled vectors $(n+1)$, and therefore these strategies are the same.

We will now introduce some definitions that will be expanded upon in the next chapter. The downset $\downarrow(x)$ of a vector x is the set of vectors lower in the ordering $\preceq$ than x, and x itself. Similarly we can define the upset of x. The mathematically definitions are:

$$\downarrow(x_i) = \{x' \in X : x' \preceq x_i\}, \qquad \uparrow(x_i) = \{x' \in X : x_i \preceq x'\}$$

In case of a chain, this means:

$$|\downarrow(x_i)| = i \quad \text{and} \quad |\uparrow(x_i)| = n+1-i,$$
$$\text{assuming } x_i \leq x_j \Leftrightarrow i \leq j$$

The number of monotone classifications in which a point gets the label 0 is equal to the size of its upset: taking any point in its upset and assigning the label 1 to the downset of that point yields such a monotone classification. Likewise, the number of monotone classifications that assigns the label 0 to a point is equal to the size of its downset. This gives

$$P(y_i = 0) = \frac{|\uparrow(x_i)|}{|\downarrow(x_i)| + |\uparrow(x_i)|}, \qquad P(y_i = 1) = \frac{|\downarrow(x_i)|}{|\downarrow(x_i)| + |\uparrow(x_i)|}$$

Proposition 1. *Querying the central vector of a binary chain maximizes the number of class labels that is inferred in the worst case and the average case.*

Proof. The proof for the worst case is trivial, so we will only show the proof for the average case.

$$\begin{aligned}
\operatorname*{argmax}_i P(y_i = 0)|\downarrow(x_i)| + P(y_i = 1)|\uparrow(x_i)| &= \\
\operatorname*{argmax}_i \frac{|\uparrow(x_i)||\downarrow(x_i)|}{|\downarrow(x_i)| + |\uparrow(x_i)|} + \frac{|\downarrow(x_i)||\uparrow(x_i)|}{|\downarrow(x_i)| + |\uparrow(x_i)|} &= \\
\operatorname*{argmax}_i 2\frac{i(n+1-i)}{i+(n+1-i)} &= \\
\operatorname*{argmax}_i 2\frac{i(n+1)-i^2}{n+1} &= \\
\operatorname*{argmax}_i \frac{-2}{n+1}i^2 + 2i &
\end{aligned}$$

Take the derivative and equate to zero.

$$\begin{aligned}
\frac{d}{di} = \frac{-4}{n+1}i + 2 = 0 \\
\frac{4}{n+1}i = 2 \\
i = \frac{n+1}{2}
\end{aligned}$$

□

It is clear that just like a standard binary search, this process takes at most $O(\log n)$ queries to find the true classification function.

2.2 Non-binary chains

The problem gets a lot more complicated once we drop the binary restriction. Querying the central vector is very unlikely to yield the smallest or largest class label, so we can't expect to get any direct inferences from one query. What GREEDY does in this case is, instead of counting inferred labels, to count the number of labels that can be eliminated in the worst case as a result of querying a vector. For example, if we have a chain with $n = 5$ and $k = 3$, and we query x_3 and find $f(x_3) = 2$, then that's two eliminations for x_3. For x_1 and x_2 we can eliminate class label 3, and for x_4 and x_5 we can eliminate class label 1. In total this is six eliminated class labels. It is not hard to prove that by this performance measure, querying the central vector is still optimal.

Proposition 2. *Querying the central vector of a chain maximizes the number of class labels that can be eliminated in the worst case.*

Proof. If we query vector x_i and $f(x_i) = j$ we can eliminate labels $j+1$ through k for vectors lower than x_i on the chain and labels 1 through $j-1$ for vectors higher than x_i. So we need to optimize the following expression:

$$\operatorname*{argmax}_i \min_j (i-1)(k-j) + (n-i)(j-1) \tag{2.2.1}$$

We minimize for j first:

$$\begin{aligned}
&\operatorname*{argmin}_j (i-1)(k-j) + (n-i)(j-1) = \\
&\operatorname*{argmin}_j ik - 2ij - k + j + nj - n + i = \\
&\operatorname*{argmin}_j (n-2i+1)j + ik + i - n - k = \\
&\operatorname*{argmin}_j (n-2i+1)j = \begin{cases} 1 & \text{if } i \le \frac{n+1}{2} \\ k & \text{if } i > \frac{n+1}{2} \end{cases}
\end{aligned}$$

Substituting 1 and k for j in Equation 2.2.1 yields

$$\min_j (i-1)(k-j) + (n-i)(j-1) = \begin{cases} (i-1)(k-1) & \text{if } i \le \frac{n+1}{2} \\ (n-i)(k-1) & \text{if } i > \frac{n+1}{2}, \end{cases}$$

which gives

$$\operatorname*{argmax}_i \min_j (i-1)(k-j) + (n-i)(j-1) = \frac{n+1}{2}.$$

□

Unlike in the binary case, GREEDY is not identical to maximizing the number of eliminated classification functions. This means we can now notice the difference in greediness between the two strategies. Maximizing the number of eliminated classification functions is less greedy in the sense that instead of trying for the most information that is practical now, it gathers as much information about the data distribution as possible that might lead to more inferred labels in a future query. It is, however, still a greedy strategy.

If we want to maximize the number of eliminated classification functions in the worst case, it turns out the central vector is not always optimal, although it seems to be always near it (for example if $n = 7$ and $k = 3$ the optimal query points are x_3 and x_5). If we go for the average case instead, we end up with a hard to solve equation, which we will demonstrate how to find.

First off, let's see how many monotone classification functions there actually are. Where we had one crossover point in the binary case, we now have $k-1$ of them. Each unique way of choosing $k-1$ crossover points out of all $n+1$ of them represents a unique classification function. Because it is possible that a certain class label doesn't occur in a classification function, the same crossover point can be chosen multiple times. The number of ways to choose $k-1$ out of $n+1$ where repetition is allowed and the order doesn't matter is equal to $\binom{n+k-1}{k-1}$.

Lemma 1. *The number of unique monotone classification functions on a chain with n vectors and k class labels is $\binom{n+k-1}{k-1}$.*

Although we can't always infer other vectors directly like in the binary problem, we can derive two smaller problems after each query. Say we query x_i and find $f(x_i) = j$ then all vectors lower than i can only have class labels smaller than or equal to j and vectors higher than i can only have class labels higher than or equal to j, which means these two segments of the chain are now completely independent with regards to monotonicity. Hence the number of possible classification functions that are left is:

$$\binom{i-1+j-1}{j-1}\binom{n-i+k-j}{k-j}$$

.

To find the query point that minimizes this number on average, we need to solve:

$$\operatorname*{argmin}_i \sum_{j=1}^{k} P(y_i = j)\binom{i-1+j-1}{j-1}\binom{n-i+k-j}{k-j}$$

The probability $P(y_i = j)$ is equal to the number of classification functions in which $f(x_i) = j$ divided by the total number of classification functions. The number of classification functions in which $f(x_i) = j$ is the same thing as the number of possible classification functions left after querying x_i and finding $f(x_i) = j$. Also, because we are minimizing, dividing by the constant $\binom{n+k-1}{k-1}$ is not going to matter. Hence, we get:

$$\operatorname*{argmin}_i \sum_{j=1}^{k} \binom{i-1+j-1}{j-1}^2\binom{n-i+k-j}{k-j}^2 \tag{2.2.2}$$

Solving this minimization problem is not trivial. Computation shows that this expression is equal to $\frac{n+1}{2}$ for the range of $n = [2, 100] \times k = [2, 30]$. We have not been able to prove, however, that $i = \frac{n+1}{2}$ is the general solution of this problem.

In the next chapter we will step up one dimension and apply the two strategies to a dataset with two attributes in stead of one, a much more realistic and useful scenario.

Chapter 3

Querying matrix orders

In this chapter we will consider another special case where $\mathcal{Y}$ is binary but the partial order $\preceq$ on X is like a matrix, or in fact, a product of two chains. This means that, say we have a matrix $A = [a_{i,j}]_{i=0,\ldots,N-1;j=0,\ldots,M-1}$, we can assign each attribute vector $x \in X$ to an entry of A such that $a_{i,j} \preceq a_{k,l} \Leftrightarrow i \leq k$ and $j \leq l$. Note that not every entry of A necessarily gets assigned a vector. Entries that don't will be called gaps.

In the previous chapter we introduced two query strategies that seem promising. In one of them we took a very greedy approach that maximized, with each query, the number of inferred class labels. In the other, we took a less greedy, but still greedy approach that maximized the number of eliminated monotone classification functions. While the GREEDY heuristic remains as easy to implement, the latter strategy, which we will call MATRIX, uses information about the whole matrix and is more complicated. The rest of this chapter will mostly focus on detailing the algorithm that performs this strategy.

3.1 Lower sets and upper sets

Before we continue, we will define some properties of posets. A partial order is a pair $P = (X, \preceq)$ where X is a set and $\preceq$ is a binary relation that is irreflexive, anti-symmetric and transitive. A lower set (also called ideal) of P is a subset $L \subseteq X$ such that if $x \in L$ and $x' \preceq x$ then $x' \in L$. Similarly an upper set is a subset $U \subseteq X$ such that if $x \in U$ and $x \preceq x'$ then $x' \in U$. If L is a lower set of P then $X \setminus L$ is an upper set of P. The downset $\downarrow(x)$ of an $x \in X$ is the minimal lower set that contains x, i.e. $\downarrow(x) = \{x' \in X : x' \preceq x\}$ and the upset $\uparrow(x)$ of $x \in X$ is defined as $\uparrow(x) = \{x' \in X : x \preceq x'\}$. For examples of lower sets see Figure 3.1 and 3.2.

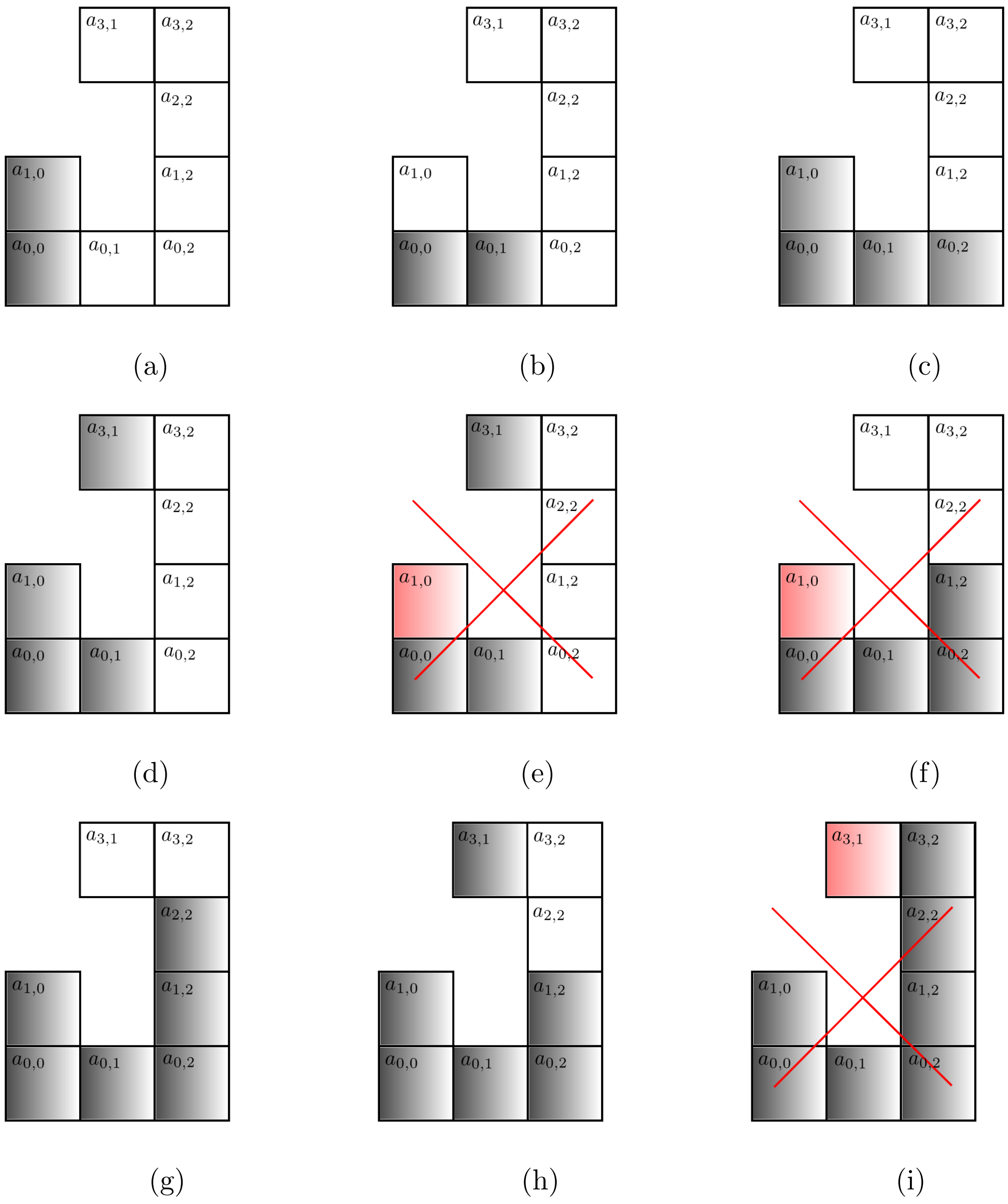


Figure 3.1: Examples of lower sets of a matrix with gaps, and subsets that aren't lower sets. A subset consists of the grey entries. The red entries would need to be included in the subset for it to become a lower set

3.2 Query strategies

Because we are working with binary classification functions, when we find out the class label of a certain vector x, we know immediately all the class labels of either

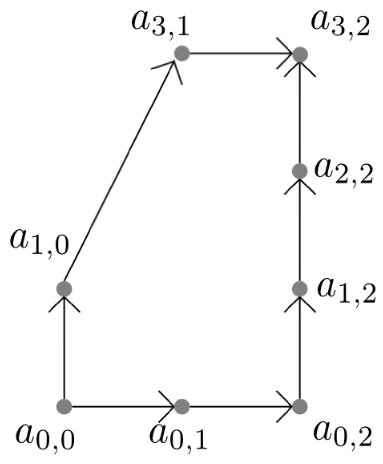


Figure 3.2: The graphical representation of order on the matrix in Figure 3.1. Arrows represent the $\preceq$ relation and all transitively implied arrows are omitted for clarity.

$\downarrow(x)$ or $\uparrow(x)$ depending on whether the observed class label is 0 or 1. This means that the way to maximize the worst case number of inferred class labels with one query is

$$x^* = \operatorname*{argmax}_{x \in X} \min(|\downarrow(x)|, |\uparrow(x)|)$$

.

This is the GREEDY heuristic for matrix posets. At the time of construction of the poset, $|\downarrow(x)|$ and $|\uparrow(x)|$ can be easily tracked and stored. This means running this heuristic can be done in linear time with respect to n, the size of the matrix poset.

To implement MATRIX, we can't just compare a local property for each vector, we need more information about the matrix poset as a whole. We need to know, for every vector x, in what proportion $p_x = P(f(x) = 0)$ of classification functions it has class 0 versus class 1. This depends on the structure of the whole poset. Assuming each classification function is equally probable, we can query the vector with p_x closest to 0.5, which maximizes the number of classification functions we can dismiss in the worst case. This strategy also maximizes the expected number of eliminated classification functions:

Proposition 3. *Vector $x^* = \operatorname{argmin}_x |p_x - 0.5|$ maximizes the expected number of eliminated classification functions.*

Proof. Let f' be the true classification function. If $f'(x) = 0$ we can eliminate all classification functions f with $f(x) = 1$, which is a number proportional to $1 - p_x$.

So to maximize the expected number of eliminated classification functions we need to maximize

$$\operatorname*{argmax}_x P(f'(x) = 0)(1 - p_x) + P(f'(x) = 1)p_x$$

.

Since $P(f'(x) = 0) = p_x$, this is equal to

$$\operatorname*{argmax}_x p_x(1 - p_x) + (1 - p_x)p_x$$

.

Taking the derivative of $2p_x(1 - p_x)$ and equating to zero gives:

$$\frac{d}{dp_x} = 2 - 4p_x = 0$$

$$p_x = \frac{1}{2}$$

The function $2p_x(1 - p_x)$ is mirrored in $p_x = \frac{1}{2}$ and monotone decreasing on both sides. Thus, the value x for which p_x is closest to $\frac{1}{2}$, x^*, maximizes it. □

3.2.1 Lower sets and classification functions

From now on we will not mention classification functions much. The reason for this is that in the binary case they are essentially the same thing as lower sets (or upper sets). We see this when we look at the constraint we put on the function f

$$x \preceq x' \Rightarrow f(x) \leq f(x')$$

Proposition 4. *Every binary monotone classification function on X corresponds to exactly one lower set of $(X, \preceq)$ and vice versa.*

Proof. Let's prove this for both directions. First we will prove: *If f is a binary classification function for X, then the set $L = \{x \in X : f(x) = 0\}$ is a lower set of $(X, \preceq)$*
Take an $x \in X$ and $x' \in L$ with $x \preceq x'$. This means $f(x') = 0$ and because f is monotone we know $f(x) = 0$ as well. Then also $x \in L$, concluding if $x' \in L$ and $x \preceq x'$ then $x \in L$, which is the definition of a lower set. Leaving to prove:

If L is a lower set of X, then the function

$$f(x) = \begin{cases} 0 & \textit{if } x \in L, \\ 1 & \textit{if } x \notin L \end{cases}$$

is a binary monotone classification function for X.

Take an $x, x' \in X$ with $f(x') = 0$ and $x \preceq x'$. This means $x' \in L$ and because of the definition of a lower set $x \preceq x'$ implies $x \in L$ as well. Then also $f(x) = 0$, and thus f must be a binary monotone classification function for X. □

3.2.2 Counting lower sets

For a general poset, counting lower sets is a known #P-complete problem [9]. However, if the poset is a matrix with gaps, we can find a dynamic programming (DP) algorithm that counts the number of lower sets in $O(NM)$ time (N and M are the height and width of the matrix).

First we consider the simple case where A has no gaps. Interestingly, the number of lower sets then is $\binom{N+M}{N}$, which looks a lot like the number we got for chains with general k. But of importance is the way we calculate that number if we are to incorporate gaps in the matrix into our algorithm.

Dynamic programming [4] is a method for solving complex problems by breaking them down into simpler sub-problems with essentially the same structure as the main problem. Repeating this process eventually leads to a potentially exponential number of trivial problems, that, combined, form the solution. If applied naïvely, this might not help much at all, but if the problem has the right structure (known as *optimal substructure*) many of these sub-problems will be the same. So if we make sure each sub-problem is calculated only once, this method can result in very elegant and fast solutions.

If $\mathcal{L}(A)$ is the set of lower sets of A we can define $D(y, x) = |\mathcal{L}(B_{y,x})|$ where $B_{y,x} \subseteq A$ with elements $\{a_{i,j} | i < y, j > x\}$, and so $D(N, -1) = |\mathcal{L}(A)|$. Here $D(N, -1)$ is the main problem and the goal is to find a way to write it as one or more similar but smaller problems. The indexing on D is somewhat awkward; this is because of the way we need to construct it, which will hopefully become clear soon. Basically, $D(y, x)$ represents the number of lower sets in A if we would remove all entries on rows y and higher, and columns x and lower. Partition $\mathcal{L}(B_{y,x})$ in sets $\mathcal{L}_k \subseteq \mathcal{L}(B_{y,x})$ ($k = 0 \ldots y$) such that for each lower set L,

$$\text{if } b_{i,x+1} \in L \Leftrightarrow i < k \ (i = 0 \ldots y - 1) \text{ then } L \in \mathcal{L}_k$$

In other words, fix all (and only) entries below row k in the lowest column of $B_{y,x}$, namely $x + 1$, to be in the lower set, group all lower sets meeting this constraint and do this for all k.

Because each lower set in $\mathcal{L}_k$ has the same configuration of column $x+1$ and by definition no lower set $\mathcal{L}_{k'}$, ($k' \neq k$) can have the same configuration, it holds that $\mathcal{L}_k \cap \mathcal{L}_{k'} = \emptyset$ for all $k \neq k'$. Furthermore, together, the sets $\mathcal{L}_k$ cover all possible configurations of column $x+1$ and by definition each $\mathcal{L}_k$ consists of all possible lower sets in $\mathcal{L}(B_{y,x})$ with that configuration. Therefore, $\bigcup_{k=0}^{y} \mathcal{L}_k = \mathcal{L}(B_{y,x})$.

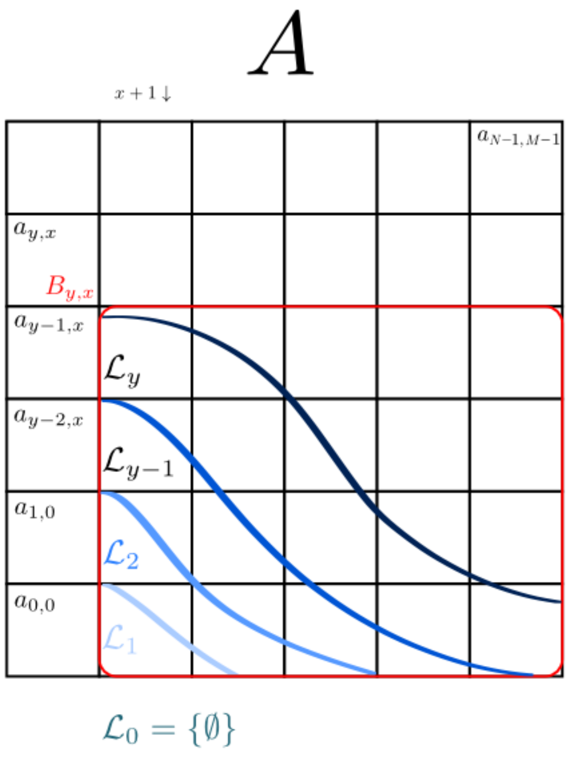


Figure 3.3: An example of how submatrix $B_{y,x}$ is split into sets of lower sets $\mathcal{L}_k$. Each $\mathcal{L}_k$ contains all lower sets that include $a_{k-1,x+1}$. With a little imagination, the curved lines depict the contour of some lower set in the respective $\mathcal{L}_k$.

Lemma 2. $|\mathcal{L}_k| = D(k, x+1) \quad (x < M-1, y > 0)$

Proof. If we remove column $x+1$ from every lower set in $\mathcal{L}_k$ it will not cause any duplicates because we fixed that column, so this will not affect the size of the set. Also, we know there are no entries from rows k or higher in any lower set from $\mathcal{L}_k$, because none contain $b_{k,x+1}$, a lower element. So, lower sets in $\mathcal{L}_k$ minus column $x+1$ only contain entries from rows lower than k and columns higher than $x+1$. This means every lower set in $\mathcal{L}_k$ minus column $x+1$ is also a lower set in $\mathcal{L}(B_{k,x+1})$. The other way around, the only entries that could be in a lower set from $\mathcal{L}_k$ that cannot be in a lower set from $\mathcal{L}(B_{k,x+1})$ are in column $x+1$ below k. But we know that those are in every set from $\mathcal{L}_k$, and because this is the lowest column in B, they put no further requirements on which higher entries must or can be in the lower set. That means every lower set in $\mathcal{L}(B_{k,x+1})$ is also a lower set in $\mathcal{L}_k$ minus column $x+1$. This concludes $|\mathcal{L}_k| = |\mathcal{L}(B_{k,x+1})| = D(k, x+1)$. $\square$

Lemma 3. $D(y,x) = \sum_{k=0}^{y} D(k, x+1) \quad (x < M-1)$

Proof. Adding up all $|\mathcal{L}_k|$ gives

$$D(y,x) = |\mathcal{L}(B_{y,x})| = \left|\bigcup_{k=0}^{y} \mathcal{L}_k\right| = \sum_{k=0}^{y} |\mathcal{L}_k| = \sum_{k=0}^{y} D(k, x+1)$$

. □

Theorem 1. $D(y,x) = D(y, x+1) + D(y-1, x) \quad (x < M-1, y > 0)$

Proof. From Lemma 3 we have $D(y,x) = \sum_{k=0}^{y} D(k, x+1)$. Hence, we can write

$$D(y,x) = D(y,x+1) + \sum_{k=0}^{y-1} D(k,x+1) = D(y,x+1) + D(y-1,x)$$

. □

Then all we need is start values for our recursion, which are $D(0,x)$ and $D(y, M-1)$. Both of these represent the number of lower sets of an empty matrix, which is $|\{\emptyset\}| = 1$. This seems to be an elaborate way to describe Pascal's rule, which it is, but next we will build upon this recursion to account for gaps.

$$A$$

$$\begin{array}{c|c|c|c|c|}
70 & 35 & 15 & 5 & \mathbf{1} \\
\hline
35 & 20 & 10 & 4 & \mathbf{1} \\
\hline
15 & 10 & 6 & 3 & \mathbf{1} \\
\hline
5 & 4 & 3 & 2 & \mathbf{1} \\
\hline
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\
\hline
\end{array}$$

Figure 3.4: All values $D(y,x)$ for a four by four matrix. There are 70 lower sets.

3.2.3 Gaps in the matrix

Say we have a gap at $a_{y-1,x+1}$. Then the first time it will influence a value of D when we build it up from our start values is when it is first included in B, which is at $B_{y,x}$ and thus $D(y,x)$. We will assume all other values of D up to this point are correct. If we take every lower set we would count if the gap wasn't there and remove $a_{y-1,x+1}$ from them, then we would clearly include every lower set we need to count in the updated $D(y,x)$. The issue is the duplicates this would produce, which we can count as follows.

If an entry on row $y-1$ was in a lower set, removing $a_{y-1,x+1}$ from it is not going to result in a duplicate because $a_{y-1,x+1}$ could not have been absent before. Also, if removing $a_{y-1,x+1}$ results in a duplicate, then $a_{y-1,x+1}$ was in that lower set, and consequently every entry below it as well. So we need a count for the number of lower sets that has every entry below $a_{y-1,x+1}$ and no entries in rows $y-1$ or higher, which conveniently is $D(y-1,x+1)$.

This means $D(y,x) = D(y,x+1) + D(y-1,x) - D(y-1,x+1)$ is the correct value for $D(y,x)$ if there is a gap at $a_{y-1,x+1}$ and our assumption that all lower values of D are correct is true. By induction this is true because the start values are still correct, since empty matrices can't have gaps. The final recursion is as follows:

$$D(y,x) = \begin{cases} 1 \qquad \text{if } y = 0 \text{ or } x = M-1, \\ D(y,x+1) + D(y-1,x) - A(y-1,x+1) \cdot D(y-1,x+1) \text{ otherwise} \end{cases}$$

where $A(y-1,x+1)$ equals 1 when $a_{y-1,x+1}$ is empty (a gap) and 0 otherwise.

If we make sure to only call $D(y,x)$ once for each x and y, then it will be called at most NM times. The function itself consists of four function calls that take constant time since we already accounted for calls to D. Thus, computing all values of D can be done in $O(NM)$ time.

3.2.4 Class label probabilities

The final problem is finding the values p_x for each vector x, or in other words, the proportion of lower sets that include x. Define $\mathcal{L}(y,x)$ as the set of lower sets containing $a_{y,x}$. If $a_{y,x}$ is empty, then we don't need this number for the algorithm, but we do need it for recursive purposes. In this case $\mathcal{L}(y,x)$ will be all lower sets that would be lower sets if we were to add $a_{y,x}$ to them. So, to be clear, lower sets in $\mathcal{L}(y,x)$ will then not include $a_{y,x}$, but must include all entries smaller than $a_{y,x}$ and may include larger entries with respect to $\preceq$.

Unfortunately, the numbers in D are not directly related to the counts we need. That said, there is one trivial case where we *can* use them directly, namely,

$$A$$

$$\begin{array}{c|c|c|c|c|}
58 & 29 & 15 & 5 & 1 \\ \hline
29 & 14 & 10 & 4 & 1 \\ \hline
15 & 10 & 6 & 3 & 1 \\ \hline
5 & 4 & 3 & 2 & 1 \\ \hline
1 & 1 & 1 & 1 & 1 \\ \hline
\end{array}$$

Figure 3.5: All values $D(y,x)$ for a four by four matrix with one gap (in pink). There are 58 lower sets, 12 less than without the gap. The numbers in red are the ones affected by the inclusion of the gap.

$|\mathcal{L}(N-1,x)| = D(N,x)$. The difference between $|\mathcal{L}(y-1,x)|$ and $D(y,x)$ is that, while both include $a_{y,x}$ and all smaller entries in all the lower sets they count, $|\mathcal{L}(y,x)|$ also needs to take into account entries higher on the y-axis, whereas $D(y,x)$ excludes them explicitly. That means, however, that on the top row, these values are equal. Later we will see how we can use these as a starting point to calculate the remaining values, but we will prove this first.

Lemma 4. $|\mathcal{L}(N-1,x)| = D(N,x)$

Proof. $D(N,x)$ is equal to $|\mathcal{L}(B_{N,x})|$, where $B_{N,x} \subseteq A$ with elements $\{a_{i,j} | i < N, j > x\}$. Take C to be the set of entries in A but not in $B_{N,x}$. We can add all entries in C to all lower sets in $B_{N,x}$ and they will remain lower sets. Say this updated $B_{N,x}$ is B'. $a_{N-1,x}$ is a maximal entry in C (i.e. there is no entry in C strictly larger with respect to $\preceq$). This gives $a_{N-1,x} \in L \Leftrightarrow L \in \mathcal{L}(B')$. And thus, $|\mathcal{L}(N-1,x)| = |\mathcal{L}(B')| = |\mathcal{L}(B_{N,x})| = D(N,x)$. $\square$

The next step is finding a recursion for $|\mathcal{L}(y,x)|$. In this case finding a way to write $|\mathcal{L}(y,x)|$ as a function of $|\mathcal{L}(y+1,x)|$ would be convenient. It's obvious that $\mathcal{L}(y+1,x) \subseteq \mathcal{L}(y,x)$, so if we know how many lower sets contain $a_{y,x}$ but not $a_{y+1,x}$ we are done. Imagine A consisting of four quadrants. All lower sets in $\mathcal{L}(y,x) \setminus \mathcal{L}(y+1,x)$ contain all entries in the quadrant $i \le y, j \le x$ (the downset of $a_{y,x}$) and none of them contain any entries in the quadrant $i > y, j > x$.

Furthermore, a lower set in $\mathcal{L}(y,x) \setminus \mathcal{L}(y+1,x)$ can have any combination of entries from the quadrant $i \le y, j > x$ (quadrant D) as long as that combination would be a lower set in that quadrant if it were isolated. The same holds true for the remaining quadrant, $i > y, j \le x$, quadrant E; well almost, but we will get to the details shortly. From here on, when we say a lower set in a quadrant, we will mean a lower set in that quadrant as if it were isolated.

So every lower set in $\mathcal{L}(y,x) \setminus \mathcal{L}(y+1,x)$ is a combination of a lower set in quadrant D and quadrant E (plus of course all entries in the first quadrant). In reverse, it's also the case that every combination of a lower set in quadrant D and quadrant E is a lower set in $\mathcal{L}(y,x) \setminus \mathcal{L}(y+1,x)$, since no entry in quadrant D is comparable to any entry in quadrant E with respect to $\preceq$. This means that if we know the count for lower sets in these two quadrants, multiplying them will yield $|\mathcal{L}(y,x) \setminus \mathcal{L}(y+1,x)|$. We already have a way to count the lower sets in quadrant D, namely $D(y+1,x)$, and we can do something similar for quadrant E.

Define $E(y,x) = |\mathcal{L}(B_{y,x})|$ where $B_{y,x} \subseteq A$ with elements $\{a_{i,j} | i > y, j < x\}$. Then $E(-1,M) = D(N,-1) = |\mathcal{L}(A)|$. And, again, mirroring the recursion for $D(y,x)$ we get

$$E(y,x) = \begin{cases} 1 & \text{if } y = N-1 \text{ or } x = 0, \\ E(y,x-1) + E(y+1,x) - A(y+1,x-1) \cdot E(y+1,x-1) & \text{otherwise} \end{cases}$$

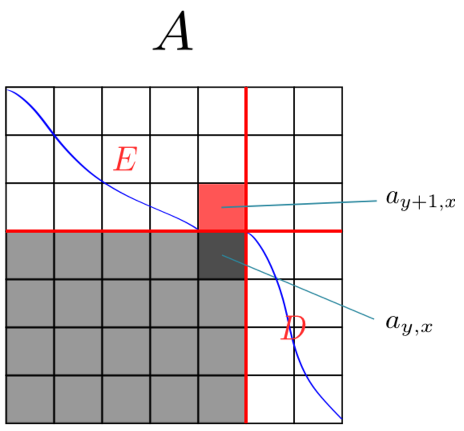


Figure 3.6: All lower sets that contain $a_{y,x}$, but not $a_{y+1,x}$ are a combination of a lower set in quadrant D and a lower set in quadrant E.

Theorem 2. $|\mathcal{L}(y,x)| = |\mathcal{L}(y+1,x)| + D(y+1,x) \cdot \big(E(y,x+1) - E(y+1,x+1)\big) \quad (y < N-1)$

Proof. We already mentioned $D(y+1,x)$ equals the number of lower sets in quadrant D and how each of those is part of some lower set in $\mathcal{L}(y,x) \setminus \mathcal{L}(y+1,x)$, but that it's a little different for quadrant E. This is because $a_{y+1,x}$ is in quadrant E and consequently if we use $E(y,x+1)$, the mirror of $D(y+1,x)$, to count the lower sets in quadrant E, we could count some that are a part of $\mathcal{L}(y+1,x)$. The lower sets we don't want to count are the ones that contain all entries $a_{y+1,j}$ for $j < x+1$, which is represented by the number $E(y+1,x+1)$. So the number of lower sets in quadrant E that we're interested in is equal to $E(y,x+1) - E(y+1,x+1)$. As discussed, $\mathcal{L}(y,x) \setminus \mathcal{L}(y+1,x)$ consists of all lower sets that are a combination of the downset of $a_{y,x}$, a lower set in quadrant D, and a lower set in quadrant E. Therefore, $|\mathcal{L}(y,x)| = |\mathcal{L}(y+1,x)| + D(y+1,x) \cdot \big(E(y,x+1) - E(y+1,x+1)\big)$ □

This gives us the recursion $C(y,x) = |\mathcal{L}(y,x)|$, which divided by the total number of lower sets yields us the percentages we need:

$$C(y,x) = \begin{cases} D(N,x) & \text{if } y = N-1, \\ C(y+1,x) + D(y+1,x) \cdot \big(E(y,x+1) - E(y+1,x+1)\big) & \text{otherwise} \end{cases}$$

3.2.5 Performance Analysis

We will show that this heuristic can indeed be performed in linear time, and we will draw a conclusion about the lower bound expected number of queries we need to find the true classification function as well. If we calculate $C(0,x)$, this recursion will get called $O(N)$ times. Doing this for each x results in $O(NM)$ calls. We know that storing all values of D and thus also E can be done in $O(NM)$ time. Since we accounted for each call to C and each call to D and E, each call to C takes no more than constant time otherwise. This means that we can calculate the lower set count for each vector in $O(NM)$ time, which is equal to $O(cn)$, where c is a constant.

It is clear that our heuristic will not always be the fastest route to the true classification function (hypothesis). This is because it doesn't take in account the way a query will modify the set of possible classification functions. We can conclude something, however, on how fast GREEDY finds the true hypothesis. Dasgupta [5] analyzes a greedy active learning strategy that can be summarized as follows. Let H denote a hypothesis class, and let $\widehat{H}$ denote the effective hypothesis class for a given sample of unlabeled points $x_1, \ldots, x_n$ (for example, the class of all monotone binary-valued functions on $x_1, \ldots, x_n$). Let π denote a probability distribution over $\widehat{H}$. The objective is to determine the unique $h \in \widehat{H}$ that is consistent with all the hidden labels, by querying just a few of them. Let $S \subseteq \widehat{H}$ be the set of hypotheses that is consistent with the labels queried so far. For each unlabeled x_i, let S_i^+ be the hypotheses which label x_i positive and S_i^- the ones which label it negative. The proposed greedy strategy is to pick the x_i for which these sets are most nearly equal in probability, that is, the x_i for which $\min\{\pi(S_i^+), \pi(S_i^-)\}$ is largest. Dasgupta [5] shows that if the optimal query strategy (that is, the one that requires the fewest queries in expectation to determine h) requires Q^* queries in expectation, then the expected number of queries needed by the greedy strategy is at most $4Q^* \ln 1/(\min_h \pi(h))$. In particular, if π is the uniform distribution, it requires at most $4 \ln |\widehat{H}| Q^*$ queries in expectation. The selection procedure of GREEDY is identical to the one Dasgupta describes and since we use a uniform distribution we can conclude that GREEDY find the correct hypotheses in at most $4 \ln |\widehat{H}| Q^*$ queries.

This DP algorithm has no direct translation to matrices (posets) of higher dimensions. The DP is based on summing the values $\mathcal{L}_k$ (sets of lower sets), where k is linear in n. In three dimensions this k grows combinatoric with n, which is problematic. In the next chapter we will drop all restrictions and look into arbitrarily structured posets.

Chapter 4

Querying general posets

In this chapter we will look at the heuristics from the previous chapter in the light of general posets. To use the MATRIX heuristic the dataset had the constraint of having only two attributes as well as two class labels. We will now drop both constraints and extend the heuristics as needed. We would still like to test the heuristic that uses the lower set count. We will call this general version of MATRIX, COUNT-LS. But as we know, counting lower sets is #P-complete, so when we do, we are restricted to small posets.

To count the number of lower sets in a poset we used an algorithm by Steiner [11] that does this in $O(nl)$ time, where n is the size of the poset and l is the number of lower sets. Aside from this, we also need to know, for each vector x, how many lower sets contain that vector. This is accomplished by counting the lower sets in $X \setminus \downarrow (x)$. Note that this heuristic doesn't make as much sense for cases with more than two class labels as with just two. The idea was to be able to eliminate close to half of the number of classification functions with every query, in order to eliminate all of them in logarithmic time. With more than two class labels, however, lower sets like we defined them have no one to one correspondence with classification functions. So while picking a vector that is in about half the lower sets might still be a good way to select a central vector, it has become somewhat arbitrary.

Luckily, GREEDY, the best performing heuristic tested on matrix posets (see chapter 7) does translate well to general posets. This heuristic uses the size of the up- and down set for each vector to select the best one. Just looking at this number only makes sense for two class labels, so we need to extend this to higher k's. What GREEDY uses as a qualifier is the information gained by querying a certain vector, or analogous, the number of class labels we can eliminate. In case of more than two class labels this elimination number and the size of the up/downset are not necessarily the same. It could be that the first query yields a middle class label, in which case no class labels in either the up- or the downset can be inferred.

Fortunately, it's not hard at all to just count the elimination number instead. The only thing we need to do is, for each vector in the poset, keep track of their possible class labels. Let $[l_i, h_i]$ be the interval of possible class labels for x_i, then if we query x_i and observe label y_i we update the intervals as follows:

1. $\forall x_j \in \;\uparrow(x_i) : l_j \leftarrow \max(l_j, y_i)$
2. $\forall x_j \in \;\downarrow(x_i) : h_j \leftarrow \min(h_j, y_i)$

Now let $N(x_i, y)$ denote the elimination number when $y_i = y$, then

$$N(x_i, y) = \sum_{x_j \in \uparrow(x_i)} (y - l_j)_+ + \sum_{x_j \in \downarrow(x_i)} (h_j - y)_+$$

where $z_+ = \max(0, z)$.

Then the vector that eliminates the maximum class labels in the worst case is:

$$x^* = \operatorname*{argmax}_{x_i \in X} \min_{y \in [l_i, h_i]} N(x_i, y).$$

We will conclude this chapter with an example of this heuristic, using the graph we introduced in the last chapter, shown with their respective identifiers i in Figure 4.1. Let's say $k = 3$. Then if we query x_2, there are three possibilities: If y_2 turns out to be 0, we can eliminate class labels 1 and 2 from vectors in the downset of x_2, which amounts to 6 eliminations. If $y_2 = 1$, we can eliminate 3 labels this way, but also class label 0 for the upset of x_2, which is 4 more eliminations, totalling 7. Finally, if $y_2 = 2$, we can eliminate 8 class labels from the upset. All elimination numbers $N(x_i, y)$ are shown in Table 4.1. GREEDY picks the vector that has the largest of the minimum elimination numbers (the bold numbers), which in this graph is either x_2 or x_3. After querying, finally, we update the class label intervals. In the next chapters we will show how the experiments will be set up, starting with an algorithm to generate random monotone classification functions.

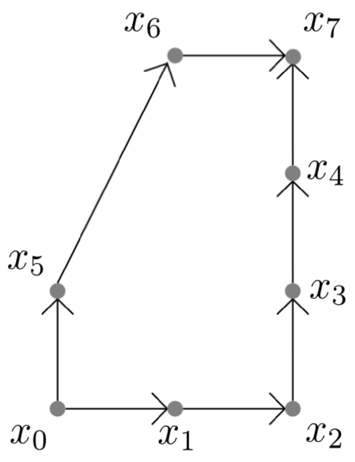


Figure 4.1: poset P

Table 4.1: Elimination numbers $N(x_i, y_i)$ for the poset P in Figure 4.1 when $k = 3$. The the minima are in bold and the GREEDY heuristic will select x_2 or x_3.

i	$y_i = 0$	$y_i = 1$	$y_i = 2$
0	**2**	9	16
1	**4**	7	10
2	**6**	7	8
3	8	7	**6**
4	10	7	**4**
5	6	5	**4**
6	**4**	5	6
7	16	9	**2**

Chapter 5

Random monotone classification functions

In the next chapter we will discuss how we are going to set up our experiments. These will consist of randomly generated datasets in addition to real-life datasets. For the generated datasets, we do not only need to generate the structure of the data (i.e. the poset), but also draw a random monotone classification function from the uniform distribution. In this chapter we will discuss an interesting way to do this, using a type of algorithm that can be used to solve all kinds of probabilistic problems. As we already have a way to generate all lower sets of a given poset, generating a random classification function uniformly for $k = 2$ should be trivial. We would just pick a lower set at random and assign label 0 to items in that lower set or 1 otherwise. If it is to be of practical value though, we would like to have a faster sampling algorithm than that, since this requires enumerating all lower sets and their number tends to become very large. Besides, it is not immediately clear how we can use this method to draw random classification functions for values of k larger than 2. In [10], Stegeman and Feelders suggest using the Propp-Wilson algorithm to generate random lower sets. We will now look at this method and later show how we can extend it to also be useful for $k > 2$. In [7], Häggström explains the Propp-Wilson algorithm more generally.

5.1 The Propp-Wilson algorithm

The main idea of the Propp-Wilson algorithm is to set up a Markov chain in which each state corresponds to one element of the distribution to sample and vice versa. In our case this means a state for every possible classification function. Then by performing a random walk in this state space for a certain amount of time it will output a random element.

One of the requirements for this method to be useful of course, is that we don't need to generate the state space for the Markov chain in advance. What we do need, is a state to start in, a way to move from one state to the next (i.e. an update function), and a stop condition that guarantees the state we end up in is a uniformly random one.

5.1.1 Update function

Let's start with the update function. For now, we will keep things simple and look at $k = 2$. This means the state space consists of all lower sets of a given poset $(X, \preceq)$. To determine the next state at time t we will take an $x_t \in X$ at random and flip a coin. If the coin comes up heads, we add x_t to the current lower set if that results in a lower set, or else the current state will not change. If the coin comes up tails, we remove x_t from the current lower set if that results in a lower set, or else the current state will not change.

More exactly, if $L \in \mathcal{L}$ is a lower set and ϵ is a uniformly distributed random variable in the interval $[0, 1)$, the update function ϕ_t at time t is:

$$\phi_t(L, \epsilon) = \begin{cases} L \setminus x_t & \text{if } L \setminus x_t \in \mathcal{L} \text{ and } \epsilon < \frac{1}{2}, \\ L \cup x_t & \text{if } L \cup x_t \in \mathcal{L} \text{ and } \epsilon \geq \frac{1}{2}, \\ L & \text{otherwise} \end{cases}$$

where $\mathcal{L}$ is the collection of all lower sets of X, i.e. $\mathcal{L}$ is the state space.

5.1.2 Choosing a start state

One way of making sure the pick of start state doesn't influence the randomness of the algorithm is to run a chain from every possible start state. This works because if we use the same random variables ϵ for all chains, the state of all chains will start to become more similar, until eventually they are all in the same state (they converge). And once they are, they will be locked together from then on. To see why this is true, take an x that is in the current lower set of some chains but not in others. There is an $x_t = x$ at some point, and after that, x will be in all current lower sets of all chains or in none.

Of course, running every chain is nonsensical. Fortunately though, if we choose our Markov chain and update function correctly, we won't have to do this. Suppose our states form a partially ordered set $(\mathcal{L}, \preceq_L)$ and the update function is monotone, that is, $L_1 \preceq_L L_2 \to \phi_t(L_1, \epsilon) \preceq_L \phi_t(L_2, \epsilon)$. Then, if we have three start states L_1, L_2, L_3 with $L_1 \preceq_L L_2 \preceq_L L_3$, as soon as L_1 and L_3 are identical, we know for certain L_2 must be as well. Now, if our partial order has a smallest and greatest element, by running only chains from those two states as start states, chains from all other start states get "sandwiched" in between like this. If we order our lower sets by set inclusion, we get a partially ordered set with smallest element $\emptyset$ and greatest element X. We will prove in the non-binary case that the update function is monotone for this poset.

5.1.3 Convergence

One might think we would be done now. We would run our two chains until they converge to a certain state and output it. While every state has a chance to get output in this way, it would not, however, result in a uniformly random chosen state. To achieve true randomness, we need to run the chains a random amount of time as follows. We pick a time T, run the two chains from time $-T$ till time 0 and if by then they have converged we output the current state. If they have not, we pick a larger T (e.g. $2T$) and try again until they do converge at time 0. If one were to implement this algorithm, and this is also the reason why we start our run at times smaller than 0, they must make sure to use the same random variables in each run as well. So, if ϵ_t is the random variable used at time t in the first run, ϵ_t must be used at time t in each consecutive run.

5.2 Nested lower sets

5.2.1 State definition and update function

Now that we looked at how the Propp-Wilson algorithm works, let us generalize this algorithm to any value of k. Classification functions don't correspond directly with lower sets any more. Now a classification function (state S) is represented by a number of nested lower sets $S = L_0^S, L_1^S, ..., L_{k-2}^S$ with the constraint

$$\forall i, j : i < j \to L_i^S \subseteq L_j^S \tag{5.2.1}$$

This gives the classification function:

$$f(x_i) = \begin{cases} \text{argmin}_j (x_i \in L_j^S) & \text{if } x_i \in L_{k-2}^S, \\ k-1 & \text{otherwise.} \end{cases}$$

The states form a partial order $(\mathcal{S}, \preceq_S)$ with

$$S_1 \preceq_S S_2 \leftrightarrow \forall i : L_i^{S_1} \subseteq L_i^{S_2}$$

Start states are $\forall i : L_i = \emptyset$ and $\forall i : L_i = X$ which are respectively a smallest and greatest element of this partially ordered set. The update function is:

$$\phi_t(S,\epsilon) = \begin{cases} L_0^S, L_{rem}^S(x_t) \setminus x_t, ..., L_{k-2}^S & \text{if } L_{rem}^S(x_t) \setminus x_t \in \mathcal{L} \text{ and } \epsilon < \frac{1}{2}, \\ L_0^S, L_{add}^S(x_t) \cup x_t, ..., L_{k-2}^S & \text{if } L_{add}^S(x_t) \cup x_t \in \mathcal{L} \text{ and } \epsilon \geq \frac{1}{2}, \\ S \qquad \text{otherwise,} \end{cases}$$

where $L_{rem}^S(x_t) = \min_j(L_j^S | x_t \in L_j^S)$ and $L_{add}^S(x_t) = \max_j(L_j^S | x_t \notin L_j^S)$.

Proposition 5. *If state S satisfies the constraint in Equation 5.2.1, then $\phi_t(S,\epsilon)$ satisfies Equation 5.2.1.*

Proof. If by applying ϕ, x_t is added to L_i^S, then because $L_i^S = \max_j(L_j^S | x_t \notin L_j^S)$, x_t is in all L_j^S for $j > i$ and thus Equation 5.2.1 holds. If by applying ϕ, x_t is removed from L_j^S, then because $L_j^S = \min_i(L_i^S | x_t \in L_i^S)$, x_t is not in any L_i^S for $i < j$ and thus Equation 5.2.1 holds. $\square$

5.2.2 Monotonicity of the update function

This time we will also actually prove that the update function is monotone, that is:

Proposition 6.

$$S_1 \preceq_S S_2 \rightarrow \phi_t(S_1,\epsilon) \preceq_S \phi_t(S_2,\epsilon) \tag{5.2.2}$$

Proof. Let's start with the cases were we remove an element. We will take r_1 to be the index of $L_{rem}^{S_1}(x_t)$ and r_2 for the index of $L_{rem}^{S_2}(x_t)$. Note that $L_{r_1}^{S_1}$ and $L_{r_2}^{S_2}$ are the only lower sets that can change and thus violate monotonicity.

In case none of the states change, i.e., removing x_t from $L_{r_1}^{S_1}$ or $L_{r_2}^{S_2}$ would not result in a lower set, Equation 5.2.2 obviously holds.

A change in S_1 would give $L_{r_1}^{S_1} \setminus x_t \subset L_{r_1}^{S_1} \subseteq L_{r_1}^{S_2}$ and so can not falsify Equation 5.2.2 either.

This leaves a change in S_2. We need to make sure that when $L_{r_2}^{S_2}$ gets smaller, $L_{r_2}^{S_1}$ will still be a subset of it. First, suppose that $r_1 = r_2$. Then $L_{r_2}^{S_1} \subseteq L_{r_2}^{S_2}$, so $L_{r_2}^{S_2} \setminus x_t \in \mathcal{L}$ implies $L_{r_2}^{S_1} \setminus x_t \in \mathcal{L}$, which means $L_{r_2}^{S_1}$ will be reduced accordingly. So now that we can assume $r_1 \neq r_2$, it has to be that $r_1 > r_2$. This is true because r_2 is the lowest ranked L^{S_2} that contains x_t, and so x_t can definitely not be in a lower ranked L^{S_1}. From $r_1 > r_2$ and $L_{r_1}^{S_1} = \min_j(L_j^S | x_i \in L_j)$ it follows that $x_t \notin L_{r_2}^{S_1}$ and thus $L_{r_2}^{S_1} \subseteq L_{r_2}^{S_2} \setminus x_t$.

The proof for the adding part of the equation is analogous, so we will just briefly show the most interesting part where S_1 becomes larger. Let a_1 be the index of

$L^{S_1}_{add}(x_t)$ and a_2 the index of $L^{S_2}_{add}(x_t)$. a_1 is the highest ranked L^{S_1} that does not contain x_t, and so x_t must be in every higher ranked L^{S_2}, implying $a_1 \geq a_2$. Similar as before, if $a_1 = a_2$ then $L^{S_1}_{a_1} \subseteq L^{S_2}_{a_1}$ and thus $L^{S_1}_{a_1} \cup x_t \in \mathcal{L}$ implies $L^{S_2}_{a_1} \cup x_t \in \mathcal{L}$. Leaving $a_1 > a_2 \rightarrow x_t \in L^{S_2}_{r_1} \rightarrow L^{S_1}_{a_1} \cup x_t \subseteq L^{S_2}_{a_1}$. □

5.2.3 Irreducibility and aperiodicity

Finally, there are two conditions on the Markov chain, that have to be met in order to guarantee the Propp-Wilson algorithm produces random draws. we need to show that our Markov chain is irreducible and aperiodic. A Markov chain is irreducible if each state is reachable from each other state in a finite number of steps. To see this, we will verify that from each state S we can get to the empty state $\forall i : L^S_i = \emptyset$ and in turn from the empty state to each state S in a finite number of steps.

Proposition 7. *The Markov chain formed by the states S is irreducible.*

Proof. Take the size of a state S to be the total number of elements in all L^S_i. Then each S that is not the empty state can be made smaller if the update function takes the lowest ranked L^S_i with $L^S_i \neq \emptyset$ and removes a maximal element from it. This means from each state, by applying a finite number of updates, we can reach a state of size zero, which has to be the empty state. Looking at the update function, it is easy to see that it is reversible, meaning that for every update on every state, there is an update that reverses its effect. So we can conclude that if from every state the empty state is reachable, then from the empty state we can reach every state and thus the chain is irreducible. □

A state in a Markov chain is said to have a period of k if a return to a state always has to occur in an integer multiple of k steps. A state with period $k = 1$ is aperiodic, and if each state in a Markov chain is aperiodic then the chain is aperiodic.

Proposition 8. *The Markov chain formed by the states S is aperiodic.*

Proof. Each state can be updated such that it stays in that state. This means each state can be returned to in an arbitrary number of steps, and therefore, our Markov chain is aperiodic. □

With the help of this algorithm it is easy to generate datasets on which to experiment. Although this method is much faster than naïvely drawing a random classification function it can still take quite a while for the convergence to happen. The posets we generated for this thesis should give you an indication of which sizes are still fast to compute, but this algorithm was one of the bottlenecks for testing on much bigger posets. In the next chapter we will show the complete setup of our experiments.

Chapter 6

Setup of Experiments

To see how the greedy active learning heuristic that uses monotonicity performs we will run a number of tests on both artificially generated and real datasets and compare its performance to other heuristics. In this chapter we will show how we set up the datasets as matrix and general posets. We will also explain how we perform these tests and measure the performance of the heuristics.

6.1 Dataset preparation

There are two ways we use to generate posets. For matrix posets we first decide on its size and then fill it with random integer vectors. For general posets we first generate the data vectors, which are random floating point numbers between zero and one and let the structure of the poset follow from that. As we will explain, this gives us control over the comparability of the poset. We will also test our algorithm on real datasets. To do this, we'll use some commonly known datasets that are relabeled in such a way that a minimal total label adjustments results in a monotone dataset. For the exact relabeling procedure, see Feelders [6].

6.1.1 Matrix posets

There are several parameters we use that impact the structure of the data. There is of course the size of the matrix, for which we have parameters N and M. We want to test on fairly large matrices if possible and in that case it shouldn't matter too much if we stick to square matrices, so we will use m to denote the size of the two dimensions. The parameter for the maximal number of queries will be q.

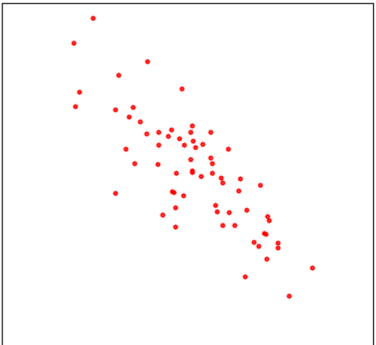
As we discussed, entries in the matrix represent possible attribute vectors, but not every vector is necessarily available as a possible query. To capture this we will specify a density parameter $d \in [0, 1]$. If we then generate $m^2 d$ random data vectors with attribute values $[0, m)$ rounded down to the nearest integer, we also allow for recurring vectors. Once we know the structure of the poset we can use the random classification function generator from chapter 5 to complete each dataset.

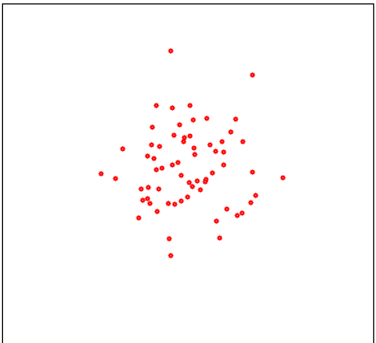
6.1.2 General posets

For tests performed on randomly generated general posets we used the following parameters. The size of the poset n, the number of class labels k, and c, which stands for comparability. We define the comparability of a poset as the percentage of comparable pairs of vectors, expressed as a number in the range $[0, 1]$.

We see no immediate disadvantage in limiting our testing of artificial data to two-dimensional data, implicating two attributes. Having more attributes generally leads to differently structured posets, for example with a lower comparability. But we will see how to use random numbers generated from bivariate normal distributions with different correlations, to create the different structures. Also, having two attributes has the nice property of making the data vectors easy to plot, which will be useful later on.

Let us look into bivariate normal distributions a bit. The key difference between two one-dimensional normal distributions and one two-dimensional one is that the latter comes with a covariance matrix. To illustrate what we can do with this, picture an XY plot of a reasonably large sample from two normalized one-dimensional normal distributions. This will look like a disc around the origin. The covariance matrix allows us to generate random data that looks like an ellipse with axis rotated 45 degrees around the origin. For examples, see Figure 6.1. Different ellipses will have different average comparabilities. In fact, every value of the covariance, ranging from -1 to 1, corresponds to a unique average comparability. It might be possible to find this exact relation, but for our purpose an approximation will suffice. We fit a large sample of experimental covariance-comparability pairs to a trigonometric function that predicts all other pairs very well. This allowed us to approximate for every desired comparability value, the covariance we would need to set with good accuracy.


(a) correlation = −0.82, comparability = 0.2


(b) correlation = 0, comparability = 0.5

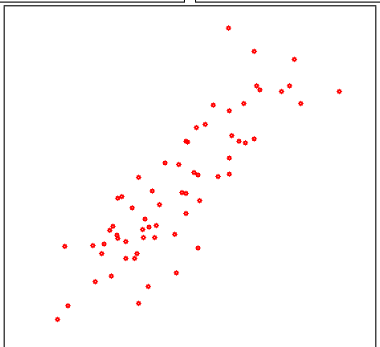

(c) correlation = 0.82, comparability = 0.8

Figure 6.1: Samples from bivariate normal distributions with different correlations

6.2 Nearest Neighbour as performance measure

The output of our query algorithms is a number of vectors for which we have the correct label, courtesy of the oracle in combination with monotone inference. One way to measure the quality of this output is to simply count the number of labeled attribute vectors we found, but for all we know, a large number of these vectors is very similar while not representative of the whole dataset. So we would like a better way to judge if a certain set of vectors tells us more about the structure of the dataset than some other set of vectors. This problem is related to classical classification. The classification problem is, given a set of vectors with known class labels, to predict the class labels of future vectors as best as possible. Whereas what we want to know, is which set predicts future vectors best, given a classification algorithm. To briefly lay out how we do this; we first of all separate the dataset in two parts. We will then only allow vectors from one part to be queried (the trainable set) and use the other part (the test set) to estimate how well the classification algorithm can classify future data vectors from the same distribution. To illustrate this, the way we construct our training vectors is as follows:

1. Randomly separate the dataset into two parts: A trainable set T_U and a test set D.
2. Associate with every vector $x \in T_U$ a set of possible class labels C_x.
3. Use a heuristic to pick a vector $x \in T_U$ to query.
4. Update $C_{x'}$ for each $x' \in T_U$ as implied by the observed class label of x.
5. Put x and all other vectors x' with $|C_{x'}| = 1$ in training set T_K and remove them from T_U. Repeat 3-5 for the desired number of queries.
6. Use T_K to build a classifier and test it on D.

Note that T_K can be relatively small compared to T, in particular when querying is very costly and as such the number of queries is small. We will of course not use the vectors in T_U as extra test data, in order to avoid biased results.

As we are mostly interested in difference of performance and not so much the absolute performance of the different heuristics, we don't need to use the best classifier possible. For our purposes the simple, yet effective, K-nearest neighbour (K-NN) algorithm will suffice.

For those who don't know exactly what K-NN does, we will summarise. To classify a given vector $x \in D$, K-NN looks at the K vectors in the training set T_K that are closest to x, i.e. its K nearest neighbours. Then the class labels of those vectors are used to predict the class label for x. The key property of an NN algorithm is its distance function, which defines how close two vectors are.

As evident, there are some choices to be made in implementing K-NN. However, they are not especially important to us for the reason mentioned above. Therefore, and also to make sure we are not fitting our choices to the used datasets, we will not test many different options but simply take a conventional approach and stick with it. For instance, we will choose to split the dataset 2:1, where the training set takes the larger share.

As mentioned, we need to decide on a distance function and how exactly a class is derived from a set of neighbours. We will define the distance between two vectors in units of standard deviation. To be more precise, we first find, for each attribute individually, its standard deviation on the whole training set. This way we can derive the distance between two vectors in terms of the separate attributes.

Then we combine these by taking the square root of the sum of their squares, which is of course the euclidean distance.

This method might lead to a number of nearest neighbour candidates that is larger than K in case the furthest ones are equally close. We could decide on the last slots randomly, but in order to keep the algorithm deterministic, we will include all of them in that case. Finally, to decide on a class for x, we take the weighted average of the nearest neighbours' classes, where each vector's weight is equal to one over its distance to x. The class is then rounded to the nearest integer.

6.3 Partial information

For matrix posets and binary class labels the K-NN algorithm as described works well. For general posets, however, it leaves something to be desired. We are maximizing the number of class labels we can eliminate, but our K-NN process is only concerned with unambiguously labeled vectors. In other words, the two processes are out of tune. This is not an issue when we work with two class labels, but if we want to experiment with larger values of k, we want to either maximize the number of completed labels during the active learning process, or make better use of all the information acquired otherwise. Our intuition why this might not just be a small issue is as follows. If, for example, the heuristic GREEDY does what it was intended to do, it will tend to start by taking vectors in the center of the poset. These vectors have a very slim chance of returning a class label that is either the highest or the lowest when the range of class labels gets wider. The consequence is that, although we should gain a lot of information about the class labels of related vectors, none of these vectors will have their label uniquely determined. Of course we query more than one vector, so later queries should provide the needed additional information. However, this might take a while, because now these vectors are not so attractive any more for future iterations of the greedy heuristic, so instead of seeking to complete what we need to know in order to include more vectors into the training set, it is plausible that it rather tends to avoid doing this. Early experiments confirmed that this is indeed a problem for GREEDY, as soon as k got larger than three, whereas a completely random query strategy was not affected by it as much.

We tried heuristics that optimize the number of vectors to be included into the training set. Just straight up maximizing the number performs abysmally, even worse than random. Most of the time there would be only one label gained, and by making sure in that case the pick would be random, it started performing a little bit better than random, but far from significantly. Even adding a touch of this optimizing strategy to GREEDY did not do well. It seems clear that instead of looking for more complete information we should explore the possibility of using the partial information we get in a more efficient way.

Therefore, we propose to add vectors to the training set even though there is uncertainty about their class label. Of course it wouldn't be wise to just add them as soon as we know a little bit, so we have to strike a balance. We will introduce a new parameter a which is a number between 0 and 1 that stands for ambiguity. Then if, for a vector x, we have $|C_x| \leq \max(a \cdot k, 1)$, we will add x to T_K, but keep it in T_U as long as $|C_x| > 1$. So, for example, if $a = 0.5$ and $k = 7$ we will add vectors for which the possible number of class labels is at most three.

Now we still have to reflect the uncertainty of these vectors in our K-NN algorithm. We chose to do this as follows. If a partial vector x is chosen as a nearest neighbour, we add this vector once for each class label in C_x, but multiply its weight by $1/|C_x|$. This will only count as a total of one vector towards the K in K-NN, of course. See Algorithm 1 for how classification is now done exactly.

Algorithm 1 The function Classify that predicts a class label for vector x using (partial) label information from the training set.

```
function CLASSIFY(x)
    $y \leftarrow 0$
    $weight \leftarrow 0$
    $NearestNeighbours \leftarrow$ GETNEARESTNEIGHBOURS(x)

    for all $nn$ in $NearestNeighbours$ do
        for all $j$ in $C_{nn}$ do
            $y \leftarrow y + j * 1/(|C_{nn}| * \text{DISTANCE}(x, nn))$
            $weight \leftarrow weight + 1/(|C_{nn}| * \text{DISTANCE}(x, nn))$
        end for
    end for

return ROUND($y/weight$)
end function
```

6.4 Heuristics

To conclude this chapter, we will list the heuristics we will apply to determine the query point. For the matrix posets we will of course test M, where we maximize the number of classification functions that can be eliminated in the worst case. We will also test the more greedy and elegant heuristic GREEDY that classifies the maximum number of vectors in the worst case. For general posets MATRIX

Table 6.1: Names and description of the query heuristics

Heuristic	Description
GREEDY	Maximizes the number of eliminated possible class labels in the worst case
MATRIX **COUNT-LS**	Maximizes the number of eliminated possible classification functions in the worst case
R(ANDOM)	Queries a random vector and applies monotone inference like GREEDY and MATRIX
R-NOINF	Queries a random vector and does not apply monotone inference
ALL	Queries all vectors in the training set immediately

is replaced by COUNT-LS, which does the same thing but way slower, and hence will only be tested on small sets.

Aside from these heuristics that use monotonicity to select a query vector we will also test some trivial methods to compare them to. An obvious one to use as such, is selecting a random vector (R), which is a naïve machine learning algorithm that still uses inference on monotonicity. And to establish a lower bound on possible heuristics, we will also include the naïve algorithm R-NOINF that does not even use inference. This means just the queried items will end up in the training set. Finally, as an upper bound, we will query all items immediately, i.e. $T_K = T$ (called ALL). In the next chapter the results of the performed experiments are shown.

Chapter 7

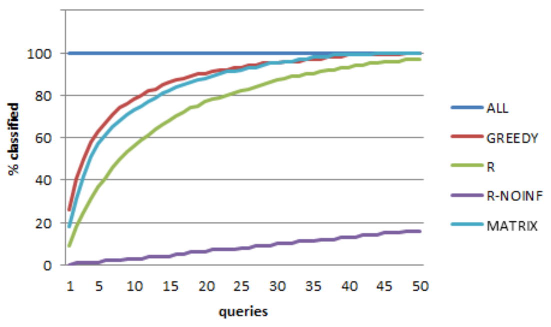
Experimental Results

In this chapter we show a number of illustrative test cases. The depicted graphs are all average results after running the tests described in the previous chapter a hundred times. For an overview table of these and more results see Appendix A.

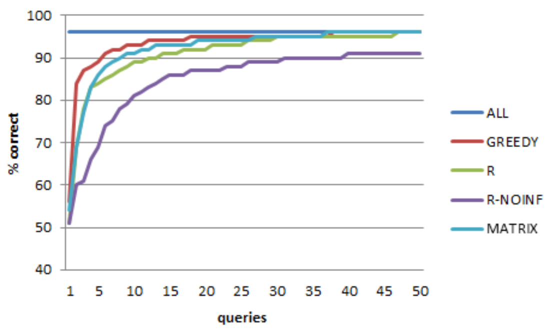
7.1 Matrix Posets

We start with the tested matrix posets. In Figures 7.1 and 7.2 we see the results on two generated datasets. The results are shown in pairs of graphs. The first one (a) shows the percentage of known class labels after a certain number of queries and the second one (b) shows how useful these class labels are, which is measured by how well the nearest neighbour algorithm described previously, predicts the class labels of the test set. The percentage on the vertical axis here, represents the number of test vectors that had their class label predicted correctly. The horizontal line at the top of the graph (ALL) indicates how well it does when all class labels in the training set are known. In Figure 7.1a we can see that after 40 queries (almost) all 500 class labels in the training set are known and in Figure 7.1b we can see that after only 15 queries most of the class labels in the test set are predicted correctly. We have one real world dataset that conformed to the restrictions of two class labels and two attributes (after relabelling that is). Results of the tests on the "Ohsumed" set, the dataset in question, are shown in Figure 7.3

The pattern in these graphs is very consistent for all tests we performed on the matrix posets and the different algorithms perform mostly as expected. If we compare random querying with and without monotone inference, it's apparent that the monotone inference is helping a lot, not only in gaining more information but also information that is evidently useful for the k-NN classification. Also, as they were designed to do, the two more selective algorithms we tested infer class labels at a much faster rate than the random ones. That said, the performance of

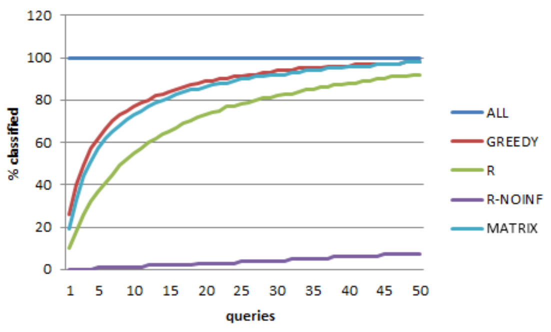


(a) % labeled vectors

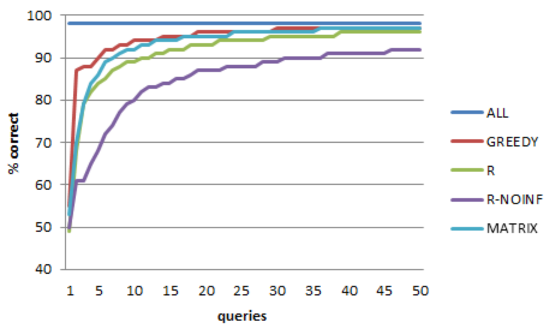


(b) accuracy on test set

Figure 7.1: The average result of tests on 100 randomly generated artificial datasets $(k = 2, m = 50, d = 0.2, [n = 500])$

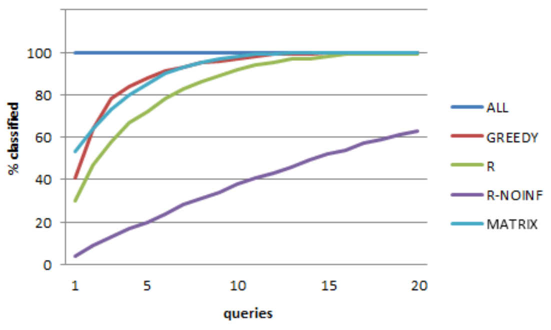


(a) % labeled vectors

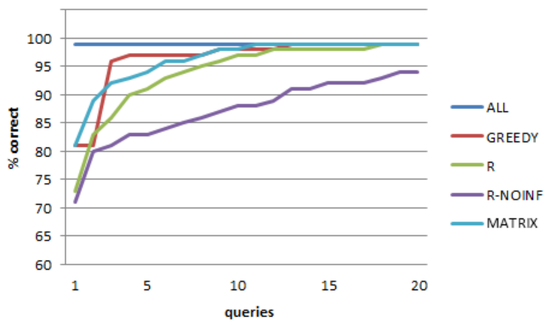


(b) accuracy on test set

Figure 7.2: The average result of tests on 100 randomly generated artificial datasets $(k = 2, m = 50, d = 0.5, [n = 1250])$



(a) % labeled vectors



(b) accuracy on test set

Figure 7.3: The average result of 100 tests on dataset Ohsumed $(k = 2, n = 156, c = 0.66)$

R, if we look at the K-NN graphs, is surprisingly good. MATRIX and GREEDY still clearly deliver better results however. This leads us to the comparison of MATRIX and GREEDY. In almost all the results we got, GREEDY shows that being greedy pays off until most of the labels are classified, but as consistently, MATRIX starts outperforming GREEDY in the end. This is not so strange, as the only thing MATRIX is geared to do is finding all class labels as soon as possible. That said, GREEDY does not lag far behind in that category. In table 7.1 we show this phenomenon with exact figures.

Table 7.1: average # queries needed to classify stated % of vectors over 100 samples

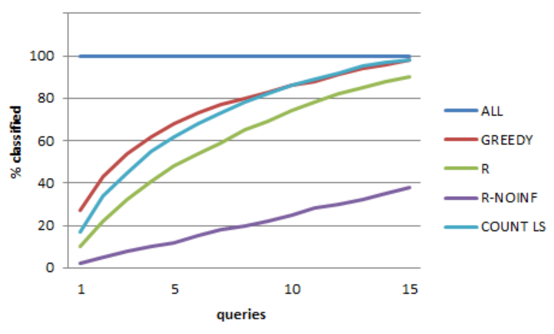
	$m = 50, d = 0.2$				$m = 50, d = 0.5$			
% labeled	20%	50%	80%	100%	20%	50%	80%	100%
GREEDY	1	3	11	45	1	3.1	12	67
MATRIX	2.1	3.8	13.4	42	2.1	3.8	14	56
R(ANDOM)	1.1	8	23	56	1.1	8.3	27	95

	Ohsumed			
% labeled	20%	50%	80%	100%
GREEDY	1	1.2	3.2	15
MATRIX	1	1	4	13
R(ANDOM)	1	2.2	6.3	24

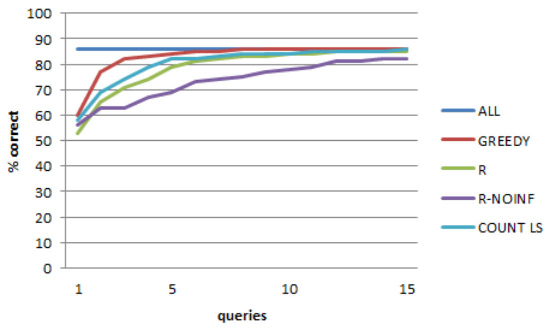
7.2 General Posets

To start off with general posets we will test a hundred randomly generated datasets of small size, where we can count the number of lower sets and as such relate to our previous testing (see Figure 7.4). Unsurprisingly, the result mimics the matrix results. This makes us inclined to predict that for larger sets, as long as we stick to two class labels, it will remain true that GREEDY does best after a relatively small number of queries.

Next we will show the results for some larger sets, where we can experiment with more class labels and thus our partial information proposal. In Figures 7.5, 7.6 and 7.7 you will find the graphs for 3, 5 and 7 class labels respectively. For biasing reasons mentioned before we chose to keep the ambiguity value fixed at 0.5, which seems to be the most natural value. Using partial information (ambiguity) indeed seems to make a substantial difference. It gives GREEDY a solid edge over R instead of being flat out worse than the random strategy in case of 7 class labels. To compare the strategies more accurately, see Table 7.2. We couldn't use the same benchmarks as before for obvious reasons, so this time we looked at the percentage of correctly labeled test cases after a certain number of queries.

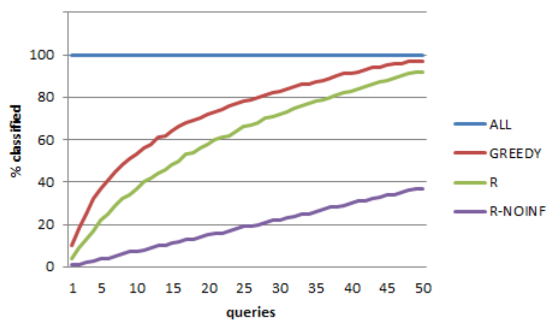


(a) % labeled vectors

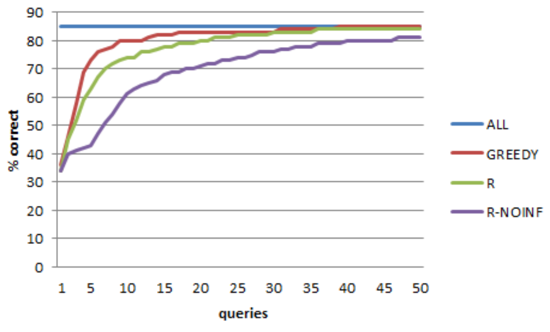


(b) accuracy on test set

Figure 7.4: The average result of tests on 100 randomly generated artificial datasets ($k = 2, n = 60, c = 0.5$)

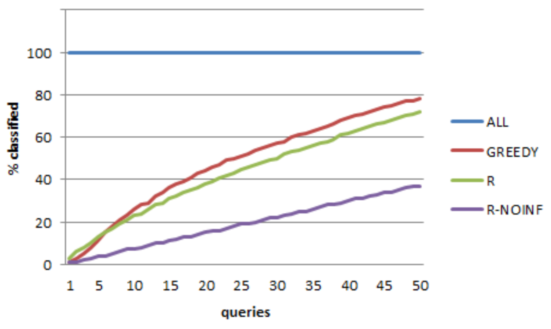


(a) % labeled vectors

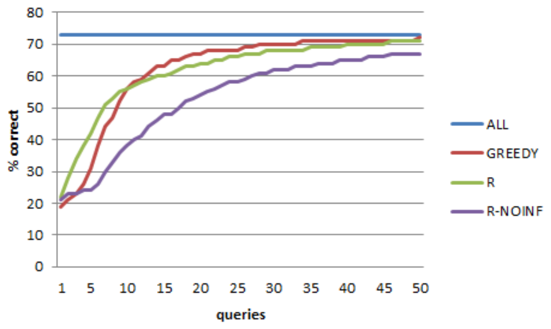


(b) accuracy on test set

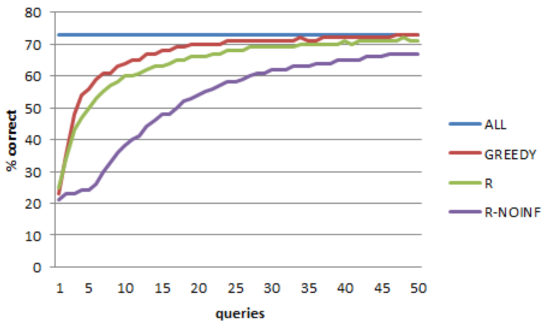
Figure 7.5: The average result of tests on 100 randomly generated artificial datasets ($k = 3, n = 200, c = 0.5$)



(a) % labeled vectors

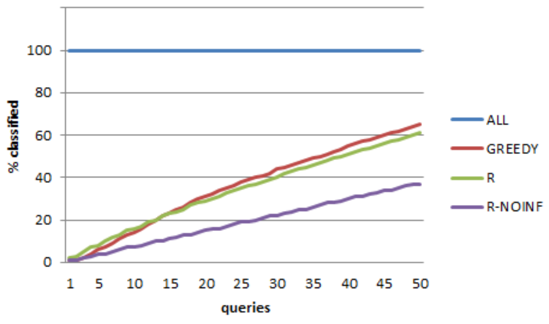


(b) accuracy on test set

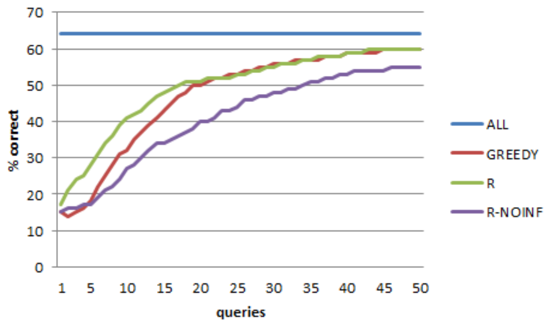


(c) accuracy on test set with ambiguity

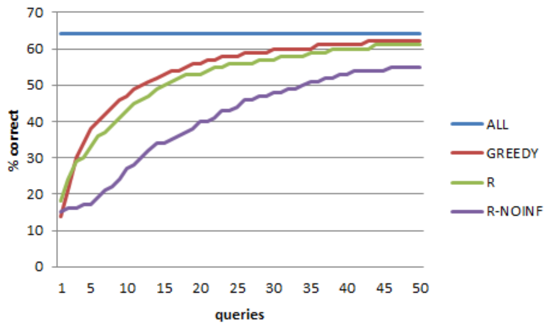
Figure 7.6: The average result of tests on 100 randomly generated artificial datasets ($k = 5, n = 200, c = 0.5$)



(a) % labeled vectors



(b) accuracy on test set



(c) accuracy on test set with ambiguity

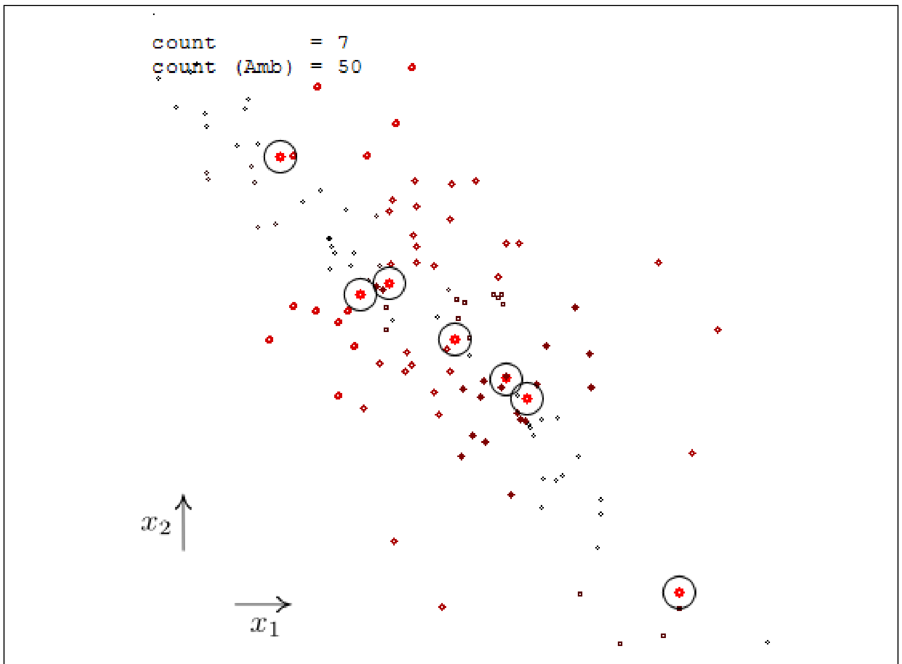
Figure 7.7: The average result of tests on 100 randomly generated artificial datasets ($k = 7, n = 200, c = 0.5$)

Table 7.2: % correctly labeled test vectors after stated number of queries over 100 samples. The max means using the maximum size training set ($n = 200, c = 0.5$)

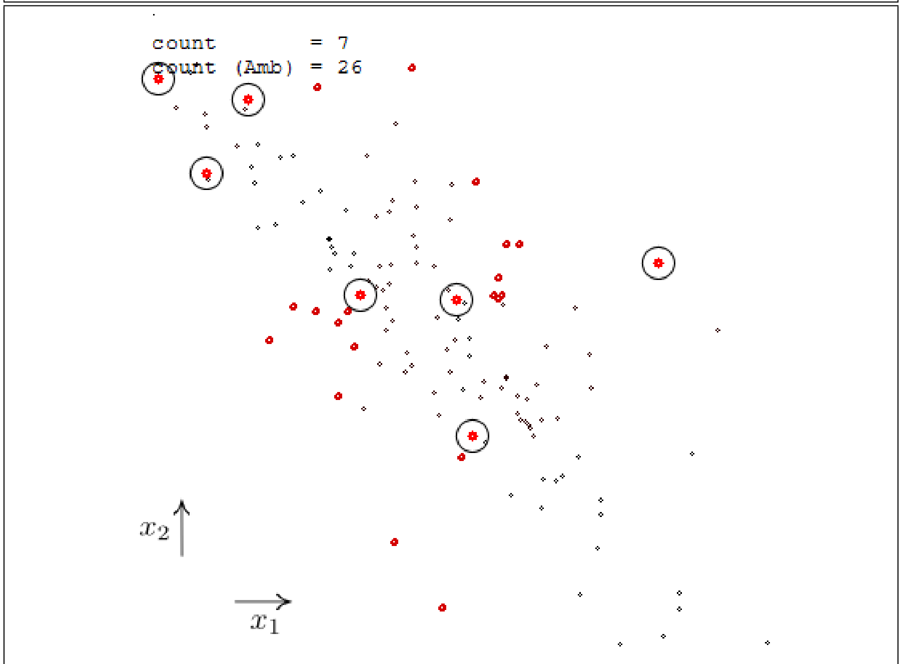
	$k = 5$ (max = 73)				$k = 7$ (max = 64)			
q	5	10	20	50	5	10	20	50
GREEDY	31	56	67	72	18	32	50	60
R(andom)	42	56	64	71	28	41	51	60
GREEDY (Amb)	56	64	70	73	38	47	56	62
R(andom) (Amb)	50	60	66	71	33	43	53	61

One might wonder why using partial information helps GREEDY so much more than it helps R. In Figure 7.8 you will find a plot that attempts to visualize why this is the case. The red and black dots in the plot each represent a vector of a randomly generated dataset after 7 queries. The larger and more red the vectors are, the more of the 7 possible class labels of that vector have been eliminated.

The $\preceq$ relation is constructed as follows. If a vector x_1 is both lower and to the left of x_2, then $x_1 \preceq x_2$. In the top image we see the results of the GREEDY heuristic and in the bottom the results of R. In both cases only 7 class labels are fully determined, but for GREEDY there are a lot more with partial information, 50 for GREEDY and only 26 for R. As you can see, the GREEDY heuristic does a very good job at selecting central vectors and they are spaced apart from each other pretty well also. This explains the gain of partial information and in turn the increased value of using that information. R however, tends to select the border vectors of which there are equally many, or more. The result is that not much at all is learned about the central area. This plot is not an anomaly either. Almost all plots made with parameters like these show the same phenomena.



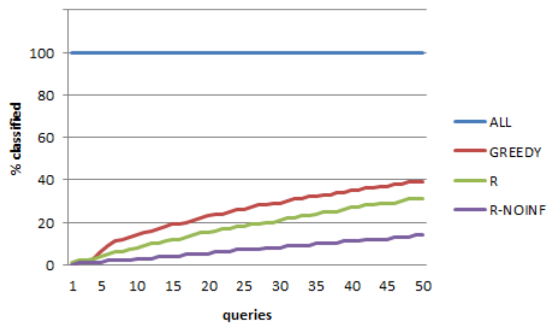
(a) GREEDY



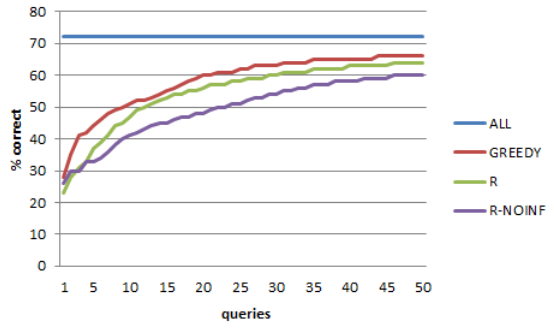
(b) R

Figure 7.8: Vector plot after 7 queries ($n = 100, k = 7, c = 0.2, q = 7$). Circled points have been queried. The bigger and more red a point is, the more certain we are of its class label

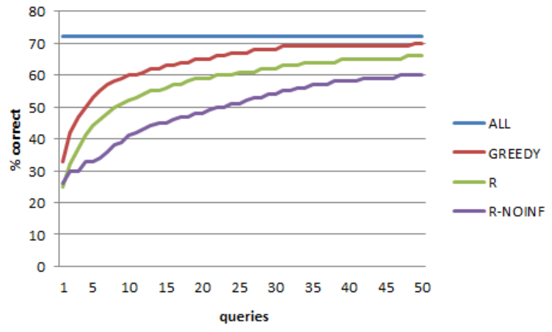
Finally, we tested on a collection of real datasets. In Figures 7.9 and 7.10 we show one with a small number of class labels and one with a larger number. For results on more datasets, see Appendix A.



(a) % labeled vectors



(b) accuracy on test set



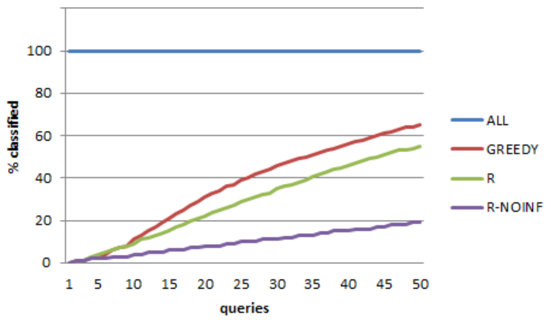
(c) accuracy on test set with ambiguity

Figure 7.9: The average result of 100 tests on dataset HPrice ($k = 4, n = 364, c = 0.26$)

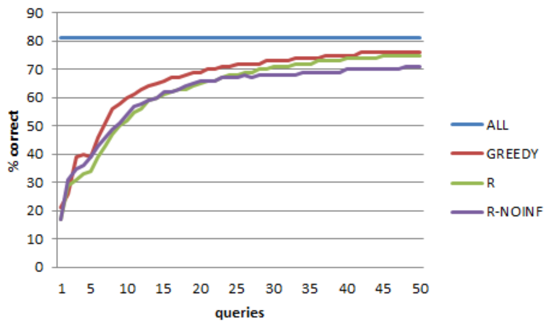
Table 7.3: average # queries needed to classify stated % of vectors over 100 samples

	Hprice				AutoMPG			
% labeled	20%	50%	80%	100%	20%	50%	80%	100%
GREEDY	17	72	160	234	3.4	34	74	129
R(andom)	18	93	176	254	6.2	44	85	137

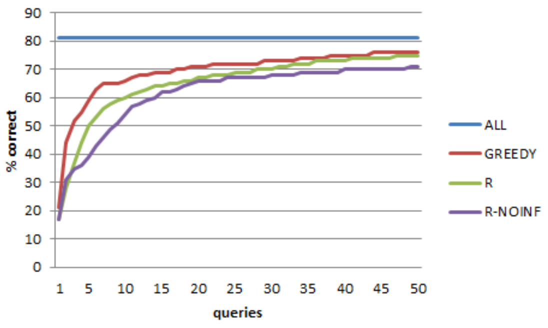
The difference between allowing ambiguity is less dramatic than in the artificial datasets. In stead of a necessity for GREEDY to perform better than random it is merely an improvement on it. This mostly seems to imply that GREEDY au naturel performs better on real datasets than generated data when the number of class labels is not small. In fact, on all ten datasets tested, GREEDY never



(a) % labeled vectors



(b) accuracy on test set



(c) accuracy on test set with ambiguity

Figure 7.10: The average result of 100 tests on dataset AutoMPG ($k = 7, n = 261, c = 0.81$)

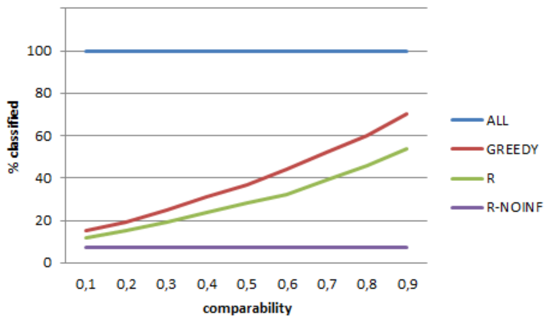
Table 7.4: % correctly labeled test vectors after stated number of queries over 100 samples

	Hprice (max = 72)				AutoMPG (max = 81)			
queries	5	10	20	50	5	10	20	50
GREEDY	44	51	60	66	39	60	69	76
R(andom)	37	47	56	64	34	52	65	75
GREEDY (Amb)	53	60	65	70	59	66	71	76
R(andom) (Amb)	44	52	59	66	50	60	67	75

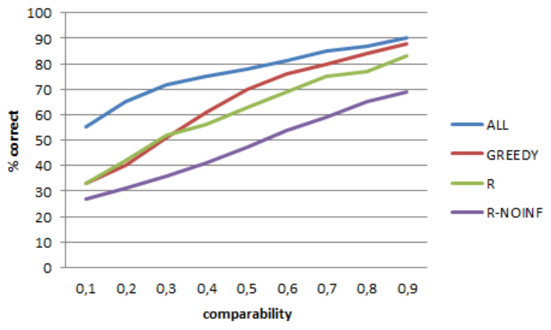
performed worse than random, either with partial information or without. Most of the sets however had only four class labels in which case the drawback of not using partial information is not that large.

We have not changed the comparability parameter so far and these results suggest partial information could be more useful in less comparable sets. To get a better sense of this parameter, take a look at Figure 7.11 in which instead of the number of rounds we see the comparability on the x-axis. The number of queries has been fixed at seven in this case.

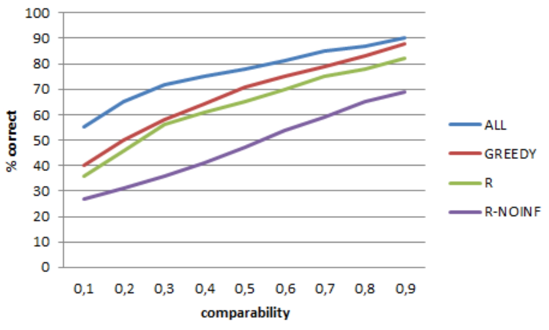
Although it seems to be the case indeed, that partial information is more useful in less comparable sets, it is somewhat of an unfair comparison. It's rather obvious that ambiguity helps more on sets where seven queries yield a lot less information. After all, when almost all vectors are labeled completely, there is not much room left for partial information. After further testing where we reduce the number of queries as the comparability increases, the difference was still there but much less apparent. The evidence is not sufficient to really distinguish the effects the different parameters have on the results. Altogether we can however conclude that using the partial information will not hurt performance most of the time and improves it on average.



(a) % labeled vectors



(b) accuracy on test set



(c) accuracy on test set with ambiguity

Figure 7.11: For each value of c (comparability), the average result of tests on 100 artificial datasets $(k = 4, n = 200, q = 7)$

Chapter 8

Conclusion and discussion

8.1 Conclusion

In the introduction, we formulated a number of objectives centered around active learning query strategies on partially ordered sets. We will now review these objectives and see if and how they were achieved, in a slightly different order.

Is there a special case poset for which there exists an algorithm that finds the central vector in a reasonable time?

We found that we had to apply some pretty harsh constraints to the dataset (poset) in order to solve this problem exactly in a way that is practical. Not only were we limited to binary classifications, but also the number of attributes of the dataset was limited to two. Given these restrictions however, we found the linear algorithm MATRIX that derives the "central" vector as defined. This gave us a framework in which we could compare this query strategy to GREEDY.

Can we conclude anything about the number of queries this strategy needs to find the true classification function?

We have shown that MATRIX is at most a logarithmic factor worse than the optimal query strategy at finding the true classification function on average. More precisely, if the average depth of the optimal query tree is Q^* and H is the set of classification functions then MATRIX needs $4Q^* \ln |H|$ queries on average to find the correct classification function.

Is maximizing the number of eliminated possible monotone classification functions in the worst case a good query strategy?

The short answer is: yes, but not as good as GREEDY. Tests showed a definitive pattern when we compared the two strategies. While they both do a good job at gaining useful information about the classification function at a fast rate, they do so notably different. Where GREEDY performs better in earlier queries, MATRIX catches up in the very end to find the true classification function before

GREEDY does in general. If it is feasible to use this strategy, because the goal is to classify the whole dataset and the right restrictions apply or the dataset is small enough, then MATRIX is a good option. But, of course, most of the time the restrictions are too strong, in which case an approximation of MATRIX would be required.

Is applying an approximation of MATRIX to general posets a possibility that yields good results in a reasonable time?

Because the focus of the problem lies on doing few queries, which GREEDY is so clearly better at, we feel the answer to this question is a clear 'no'. Even if the goal is to find the true classification function, the approximation penalty will likely at least cancel out the edge MATRIX has. This is mainly the case, though, because GREEDY does so well, not because MATRIX is a bad heuristic.

We will conclude with our findings about the GREEDY heuristic and our view on why it performs well. We showed that GREEDY not only delivers a sizeable labeled training set, but also provides labels that carry useful information. We saw this by applying a very basic classification algorithm. GREEDY is not only way ahead of R in the size of the training set, but also in the accuracy on the test set. We touched briefly on the fact that we didn't discuss many alternative query heuristics. This is because all the ones we tried were either almost indistinguishable from GREEDY or just performed worse. It seems very hard to find a better performing heuristic without additional information about the dataset it's used on. Why is this heuristic so effective? Because it balances the quantity of the labels in its upset and downset, it is good at finding labels in the central area of the poset. This is a good feature since no matter what label is found, useful information can be inferred, whereas for border vectors, in the worst case, which is also the most likely case, a much less useful border label is found. Furthermore, because it looks for as much information as possible with every query, the queries quickly spread out across the center of the poset (we see this in Figure 7.8). This may of course be detrimental for later queries, but the greediness of this approach is certainly a positive feature, since the goal is making few queries. The fact that it performs well even after a lot of queries is more surprising.

Testing showed that with an increasing number of class labels, in order to keep up the performance of GREEDY, we needed to use all the gathered information instead of just the usual fully labeled vectors. We called this classification using partial information and we defined the ambiguity parameter that dictated the minimum amount of information that could be used in this way. To use the partial information, we took the fairly intuitive approach of creating a dummy item for each remaining possible class label with adjusted weights in the training set.

8.2 Further research

In each of the subjects we discussed there is room to explore. We touched on chains briefly mainly for the purpose of setting up the problem in an easy to understand way, but there are definitely still questions to be answered. In particular in the relatively simple area of chains more exact results could be pursued. We already raised the question if a proof can be found for the seemingly valid proposition about querying the central vector of a non-binary chain in the average case (Equation 2.2.2). Also, we know the classification of a monotone binary chain is tightly related to binary search in a list. Can we draw a similar comparison when the chain is non-binary? And what can we conclude about the worst case or expected number of queries needed to find the true classification function for such a chain?

In the chapter on matrix posets, we've gone into more detail and got nice results where binary classification function are concerned. But this, of course, leaves the question if something similar could be done for non-binary ones. We have found an upper bound on the number of queries needed to find the true classification function, but it could very well be improved, since it's not very tight. Also, it would be interesting to explore if datasets with more than two dimensions can be made to adhere to the matrix restrictions. We think that datasets with more than two dimensions already fit into a matrix as long as their poset graph is planar.

For general posets a lot of follow up questions come to mind. We did mention it might be hard to find a better heuristic than GREEDY, but it remains a fairly simple one and a more complex one that outperforms it probably exists. Furthermore, we left a number of variables and other settings fixed that don't have to be, in order to focus on the important variables. We assumed the distribution of classification functions to be uniform for example. Also the classifier we used is very basic and different classifiers could produce different results. We introduced the ambiguity parameter and purposely didn't fiddle around with its value, but it might be interesting to do so. We strongly feel the value of 0.5 that we used is a good default choice, but the only motivation for this is that low values like < 0.2 and high values like > 0.8 make little sense for obvious reasons.

A big next step is of course to consider the probabilistic version of the problem. There are several steps where we assumed things weren't probabilistic, but they could well be. For example, the oracle can be faulty at times. Especially when human experts are used as an oracle it's more realistic to assume the result will sometimes violate monotonicity. But also the real world data doesn't have to be perfectly monotone. Take the house pricing problem, for example, where we have to estimate the worth of houses. A bigger house in a better neighbourhood should be worth more, but it doesn't have to be. This is of course why we relabeled the real-life datasets to be monotone, but it would be nice if we could do without this.

Appendix A

Additional test results

Table A.1: Basic properties of tested real datasets, #att stands for number of attributes and c stands for comparability

Name	n	k	#att	c
AutoMPG	261	7	4	0.81
CPU	139	4	6	0.48
Haberman	204	2	3	0.33
HPrice	364	4	11	0.26
Ohsumed	157	2	11	0.66
Pima	512	2	8	0.07

Dataset origins:

The Pima Indians Diabetes (Pima), Haberman's Survival (Haberman), Computer Hardware (CPU), and Auto MPG data sets have been taken from the UCI machine learning repository. The Ohsumed dataset is available from the LETOR website (http://research.microsoft.com/en-us/um/beijing/projects/letor/). Finally, the House Pricing dataset (HPrice) originates from the free pdf database (http://freepdfdb.com/doc/windsor-canada).

Table A.2: Test results for tested real datasets. All results are averages over 100 repetitions. The first three columns of numbers are the number of queries it takes to classify 20%, 50% and 80% of the training set. The last three columns of numbers are the percentage of correctly labeled test vectors after 5, 10 and 20 queries.

Name	Heuristic	20%	50%	80%	5	10	20	max
AutoMPG	GREEDY	3.4	34	74	39%	60%	69%	91
	R(andom)	6.2	44	85	34%	52%	65%	
	GREEDY (Amb)				59%	66%	71%	
	R(andom) (Amb)				50%	60%	67%	
CPU	GREEDY	7.3	31	65	45%	67%	69%	74
	R(andom)	12	39	79	40%	52%	65%	
	GREEDY (Amb)				59%	68%	70%	
	R(andom) (Amb)				49%	61%	66%	
Haberman	GREEDY	1.1	5.3	20	85%	89%	90%	91
	R(andom)	2	8.5	25.3	84%	86%	89%	
	GREEDY (Amb)							
	R(andom) (Amb)							
HPrice	GREEDY	17	72	160	44%	51%	60%	72
	R(andom)	18	93	176	37%	47%	56%	
	GREEDY (Amb)				53%	60%	65%	
	R(andom) (Amb)				44%	52%	59%	
Ohsumed	GREEDY	1	1.2	3.2	97%	98%	99%	99
	R(andom)	1	2.2	6.3	91%	97%	99%	
	GREEDY (Amb)							
	R(andom) (Amb)							
Pima	GREEDY	12	116	256	74%	77%	78%	79
	R(andom)	24	102	222	72%	75%	77%	
	GREEDY (Amb)							
	R(andom) (Amb)							

Bibliography

[1] Nicola Barile and Ad Feelders. Active learning with monotonicity constraints. In *Proceedings of SDM*, pages 756–767. SIAM / Omnipress, 2012.

[2] R. Carmo, J. Donadelli, Y. Kohayakawa, and E. Laber. Searching in random partially ordered sets. *Theor. Comput. Sci.*, 321(1):41–57, June 2004.

[3] Yan Chen. An efficient search algorithm for partially ordered sets. In *Proceedings of the 2nd IASTED international conference on Advances in computer science and technology*, ACST'06, pages 91–94, 2006.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (2nd ed.).*

[5] Sanjoy Dasgupta. Analysis of a greedy active learning strategy. In *NIPS*, 2004.

[6] Ad Feelders. Monotone relabeling in ordinal classification. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 803–808. IEEE Computer Society, 2010.

[7] Olle Häggström. *Finite Markov Chains and Algorithmic Applications.*

[8] Nathan Linial and Saks. Searching ordered structures. *J. Algorithms*, 6(1):86–103, 1985.

[9] J. Scott Provan and Michael O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.*, 12(4):777–788, 1983.

[10] Luite Stegeman and Ad Feelders. On generating all optimal monotone classifications. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, ICDM '11, pages 685–694. IEEE Computer Society, 2011.

[11] G. Steiner. An algorithm to generate the ideals of a partial order. *Operations research letters*, 5(6), 1986.