Stochastic Active Learning with Monotonicity Constraints

Tijmen Kolkman

February 24, 2015

Abstract

Active learning can in many cases speed up classification tasks by combining expert knowledge and knowledge about the structure of the data. When it is known that the class label increases or decreases with the attribute vectors we can exploit this feature to greatly decrease the number of labelled examples that is needed to construct a classifier. Here we study such algorithms both in general and in the case where data exhibits such special features. These monotone relations form the basis of the SMAL algorithm as described by Barile and Feelders in [1], which we will study in more detail. We describe a special case that can lead to unwanted behaviour in this algorithm and explore a number of possible alternative approaches that aim to prevent the occurrence of this special case. We propose a number of changes to the algorithm that aim to reduce the occurrence of this special case, possibly sacrificing some performance. Experimental results look promising as they show only a minor drop in performance across our toy datasets and even increased performance in some cases.

Chapter 1

Introduction

Active learning is a sub-field of machine learning which itself belongs to artificial intelligence. It differs from standard supervised learning in that the active learner is allowed to choose which data it wants to learn from. The hypothesis is that if the learner is allowed to decide for itself which attribute vectors' labels it wants to learn most, the algorithm will perform better with less training.

The goal in both active and supervised learning is to learn a so-called *classifier*, which assigns a class-label to each instance of the data-set. Here a class-label refers to an attribute that can be used to describe the individual data instances. For example, a mass spectrometer can analyse unknown chemical compounds and produce a histogram of relative mass-charge intensities. Since different ions have different mass-charge values, the shape of the histogram allows an expert in chemical science to identify which ions are present and label the chemical substance.

To learn the classifier learners need to be trained on a set of data for which the labels are known, this set is often called the *training set*. A key challenge of machine learning is to decide which part of the available data should be used for training and which for validation. The available data is generally a sample from a larger domain space and need not be entirely uniform, that is it might be skewed toward certain parts of the domain. It's therefore common practice in machine learning to set aside a part of the data for testing purposes to avoid over-tuning the classifier. If to much of the data is used for training the classifier might generalize poorly to new instances that weren't part of the original data set. If the training set is too small it might not represent the data well enough and lead to inaccurate classifiers. We note here a key difference between active-learning and supervised-learning; a supervised learner aims to build a classifier using all of

the provided training data, whereas an active-learner aims to do so while using only a subset of the training data.

Obtaining training data is usually a challenge. Since learners need some of the data for testing the classifiers that were learned, only a part can be used for the actual learning. In addition, labeled data is often scarce and expensive. In most cases each instance of a data-set needs to be labeled individually by a domain expert by hand, e.g. in chemical science, providing a bottle-neck for many learning applications. Active learners aim to overcome this problem by posing queries of unlabelled data to the domain experts. The queries are chosen in a clever manner so as to minimize the number of questions asked while obtaining a set of labels that represents the data well. In this way active learners attempt to achieve good performance while keeping the cost of obtaining labelled data to a minimum.

In most theoretical analysis, the domain experts are represented by *oracles*. An *oracle* is a black-box which allows a learner to observe the label of any instance of the data-set it wants. Sometimes there are variable costs associated with each instance, or not all labels might be available, as is the case in real life.

There are different sorts of labelling problems, but here we will only look at ordinal classification. That is we assume that labels can be compared and ranked in some meaningful way. For example, if a survey asks customers to rate their experience on a scale from 1 to 10, there exists a clear order amongst the replies. A customer who writes down 4 is clearly less satisfied about some issue than a customer who writes down 7. However, say we want to label the colors of different flowers based on their external properties. The labels *blue*, *green* and *yellow* do not supply any meaningful ordering information: We can not say that a blue flower is more "flower" than a yellow one, or rank different flowers based on their color.

In some cases, part of the domain knowledge is that there exists some relation between data and its labels. Specifically it is assumed that attributes have a positive (or negative) influence on the label. What this means is that if an instance from the data-set has attribute values that are at-least as high as the values of another instance, its label can't be lower. The main goal is to use this information in an active learning setting when formulating the queries that the learner will present to the oracle. The hypothesis is that the use of these so-called *monotonicity constraints* improves the quality of the queries and allows for learning better classifiers with fewer queries.

Specifically we will look at the *non-deterministic case*. We assume that the oracle is allowed to make mistakes when deciding the label of a query, leading to (possible) monotonicity violations in

the resulting labeling. This case has been briefly considered by Barile and Feelders in their work on monotonicity constraints [1]. They propose an approach that aims to recover the underlying monotone classifier when violations are encountered, based on a so-called *relabelling-algorithm*. This algorithm can be proven to always return a monotone labeling and on average finds a solution that yields better results than the original (possibly non-monotone) labelling. It can be shown however that in some cases this approach fails dramatically. We set out by considering a special case to illustrate this problem and show why the original approach fails. We design and test a number of alternative query strategies that aim to fix this problem and improve primarily the special case performance of the original algorithm,

Chapter 2

Preliminaries

We will now define the concepts discussed in the introduction more clearly. Our aim is to obtain, for a given data-set, a set of ordinal class-labels. That is, we want to learn the label $y \in \mathcal{Y}$ for each vector in the data-set, where $\mathcal{Y} = \{1, ..., K\}$ is an ordinal set of K class-labels. Our data-set X consists of attribute-vectors $\mathbf{x} = (x_1, ..., x_m) \in \mathcal{X}$, where $\mathcal{X}$ is commonly referred to as the *attribute space*. Without loss of generality we assume that the values of the attributes are taken from $\mathbb{R}$, so the *attribute space* is an m-dimensional space of real numbers $\mathcal{X} \subset \mathbb{R}^m$. We assume that the objects in our data-set, which we will call *vectors* are generated independently from some (unknown) probability distribution $P(\mathbf{x}, y)$. To clearly distinguish vectors and attribute values we will use the following notation; Bold-face letters will refer to vectors. Plain-face letters with subscripts will refer to attribute values. So for example $\mathbf{x}$ will be the vector consisting of attributes $x_1, ..., x_m$.

We will call the assignment of labels to vectors a function f. The function $f(\mathbf{x})$ assigns a label from $\mathcal{Y}$ to the vector $\mathbf{x}$, that is f is a function from X to $\mathcal{Y}$. Here we make the specific distinction between the data-set X and the attribute-space $\mathcal{X}$: The function assigns a label only to vectors that are present in the data-set, where sometimes the aim is to derive a general classifier for the entire attribute-space. Torvik [11], for example aims to derive the values of the entire monotone function over the attribute-space. The collection of all possible assignments will be called $\mathcal{F}$. We also define the function y that assigns to each vector its true label. The value $y(\mathbf{x})$ denotes the prior knowledge about our data and the correct label for vector $\mathbf{x}$. For brevity we use the subscript $y_{\mathbf{x}}$ when referring to the value of $y(\mathbf{x})$. Our goal is to build a *classifier* function h that predicts the labels of all vectors in the data-set. The classifier h is a function from $\mathcal{F}$, and assigns a value

from $\mathcal{Y}$ to each vector $\mathbf{x}$. The classifier should aim to predict $y_{\mathbf{x}}$ for as many labels as possible. To measure the accuracy of a classifier a loss-function $L(y, h(\mathbf{x}))$ is used, which reflects the cost of predicting $h(\mathbf{x})$ when the actual label is $y_{\mathbf{x}}$. An optimal classifier would be a function $h(\mathbf{x})$ that minimizes expected loss. A common choice as loss-function is *0/1-loss*, in which a wrongly classified vector incurs a fixed penalty of 1. For ordinal classification it makes sense to also take the order of the labels into account, that is if a label is more wrong it should incur a larger penalty. This feature is well captured by *absolute loss*, in which penalties are proportional to the distance from the true value. An alternative to using *abusolute loss* is *squared loss* (or *quadratic loss* in some cases), which has some advantages and some disadvantages over absolute loss. As the choice of loss-function is usually dictated by practical concerns we will not further elaborate here.

Our domain-knowledge is expressed by so called *monotonicity constraints* on our data. Intuitively this means that if a vector $\mathbf{x}$ has one or more attributes that have a greater value than the corresponding attributes of another vector $\mathbf{x}'$ that vector should get a greater label. That is, if we have some ordering amongst our vectors $\mathbf{x}$ and $\mathbf{x}'$, say $\mathbf{x} \geq \mathbf{x}'$, then the corresponding labels should exhibit a similar order relation, $y(\mathbf{x}) \geq y(\mathbf{x}')$. We define a *dominance relation* as a binary relation on X such that for any pair of vectors $\mathbf{x}, \mathbf{x}'$ we have that if $\mathbf{x}$ dominates $\mathbf{x}'$ every attribute of $\mathbf{x}$ should be greater than or equal to the corresponding attribute of $\mathbf{x}'$:

$$\mathbf{x}' \preceq \mathbf{x} \iff \forall_{j=1,\ldots,m} x'_j \leq x_j. \tag{2.1}$$

This dominance relation is a *partial order* (or *poset* for short) that is, it is both transitive and reflexive. We will use the notation $(Q, \preceq)$ to mean that set Q is a *poset* with dominance relation $\preceq$. We call two vectors $\mathbf{a}$ and $\mathbf{b}$ *comparable* if and only if we have that $\mathbf{a} \geq \mathbf{b}$ or $\mathbf{a} \leq \mathbf{b}$. If all vectors of a poset are *comparable* the order takes the form of a simple *chain*.

We call a function f *monotone* if for any two vectors $\mathbf{x}$ and $\mathbf{x}'$ when $\mathbf{x}$ dominates $\mathbf{x}'$ the value of $f(\mathbf{x})$ is also greater than the value of $f(\mathbf{x}')$:

$$\mathbf{x}' \preceq \mathbf{x} \implies f(\mathbf{x}') \leq f(\mathbf{x}). \tag{2.2}$$

Now if we say our data-set has *monotonicity constraints* we mean to say that for all *comparable* vectors the dominance relation equation (2.1) holds and that equation (2.2) holds for our true label function y. Since we want to build a classifier that closely resembles the function y it stands to reason that equation (2.1) should also hold for h.

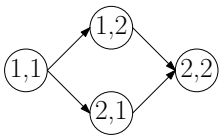

Figure 2.1: A partial order consisting of 4 vectors

To represent partial orders graphically we define an *order-graph* as follows: Let vertices represent vectors and directed edges represent order relations. If there is a directed edge from vertex x to vertex x' then vector $\mathbf{x}$ precedes vector $\mathbf{x}'$ in the order. To avoid clutter transitive edges are not shown. This means that if two vectors are *comparable* there should be a (directed) path from one to the other. To improve readability all order-graphs will be presented in such a way that successors are to the right of their predecessors. In other words all graphs can be read left-to-right in increasing order.

Example 2.1. Let's assume for simplicity's sake that $\mathcal{X} = \{1, 2\}$ and $m = 2$. That is, we have binary attribute values and vectors of only two attributes. Our attribute-space $\mathcal{X}$ will then consist of the following four vectors $(1, 1)$, $(1, 2)$, $(2, 1)$ and $(2, 2)$ The partial order $(X, \preceq)$ can then be expressed by the order-graph seen in figure 2.1. From the figure we see that the vector $(1, 1)$ is a predecessor of all other vectors, and that the vector $(2, 2)$ succeeds all the others. We also see that while both vectors $(1, 2)$ and $(2, 1)$ are successors to $(1, 1)$, and both are predecessors to $(2, 2)$, neither can be compared to the other since there is no path from $(2, 1)$ to $(1, 2)$. In this example we listed the values of all attributes inside the vertex. For clarity we will usually only display the subscript of the vectors.

We define the following concepts which are related to partial orders: An *upset* $\uparrow(x_i)$ of x_i contains all elements of $\mathcal{X}$ that succeed x_i in the order: $\uparrow(x_i) = \{x_j \in X : x_i \preceq x_j\}$. Similarly, a *downset* $\downarrow(x_i)$ contains all elements that precede x_i: $\downarrow(x_i) = \{x_j \in X : x_j \preceq x_i\}$. To refer to the number of vectors in $\uparrow(x_i)$ we use $u(x_i)$ and, analogously, for the number of vectors in $\downarrow(x_i)$ we use $d(x_i)$. An *upper set* U of X is a subset of X that contains the upward closure of all of its elements: $x_i \in U, x_i \preceq x_j \implies x_j \in U$. That is, an *upper set* contains the *upsets* of all its elements. A *lower set* L of X is a subset of X that contains the downward closure of all its elements: $x_i \in L, x_j \preceq x_i \implies x_j \in L$. That is, a *lower set* contains the *downsets* of all its elements. Upper sets and lower sets are complementary, that is, if we have an upper set U then the complement $\bar{U} = X \setminus U$ is a lower set of X. We conclude with an example illustrating *upper sets* and *lower sets*, and the related concepts of *upsets* and *downsets*.

Example 2.2. In figure 2.1 if we want to check whether the set $U = \{(1,1),(1,2),(2,1),(2,2)\}$ is an upper set of X we need to make sure that for each of its elements the entire upset is included. The upset of the first element $(1,1)$ are all the elements that succeed it in the order, from figure 2.1 we see that these are the elements $(1,2),(2,1),(2,2)$, all of these are also included in U. We follow the same procedure for the other elements to find that U contains the upsets of all its elements and is therefore indeed an upper set of X. If we list all upper and lower sets from the simple order in figure 2.1 we get the following list of sets:

$uppersets$	$lowersets$
$\varnothing$	$\{(2,2),(2,1),(1,2),(1,1)\}$
$\{(2,2)\}$	$\{(1,2),(2,1),(1,1)\}$
$\{(1,2),(2,2)\}$	$\{(2,1),(1,1)\}$
$\{(2,1),(2,2)\}$	$\{(1,2),(1,1)\}$
$\{(1,2),(2,1),(2,2)\}$	$\{(1,1)\}$
$\{(1,1),(1,2),(2,1),(2,2)\}$	$\varnothing$

Each line also illustrates how the complement of an upper set is a lower set and visa versa. Also note that for each line we have that $U \cup L = X$, as it should considering U and L are complementary.

Chapter 3

Active Learning

3.1 Active Learning

In general, machine learning aims to build a classifier based on some sample from the data for which labels are already known, usually this sample is selected in one go, before the classifier is learned. Challenges are how to select the sample and how to use remaining data to test the classifier. For example if the chosen sample does not represent the data distribution well enough, the resulting classifier might not generalize well to other, similar datasets. On the other hand, if the sample is 'to general' the classifier might not learn enough distinction between different data vectors which leads to useless classifiers. Active learning is an approach that lets the learner decide for itself how to build the sample. What this means is that the learner has control over which vectors it wants to know the label of and use to learn its classifier. For example in a binary classification problem, it would make sense to only want to train on vectors that lie closer to the decision boundary, as they at more information than vectors that are farther removed. Where the goal in general machine learning is to build a classifier that give an accurate description of the data. Active learning aims to build an (accurate) classifier using as little of the data as possible. [2]

Settles [8] describes an *active learning cycle* as follows: First a sample is selected from the data based on some criteria, also known as a *informativeness measure*. The sample is presented to a domain expert, which in this setting is called an *oracle*, who will decide the label for each vector in the sample. The second step involves the learner updating its *model*, or current classifier, by adding the information it gained from learning the new labels. Next we again select a sample, this

time the informativeness of the individual vectors still remaining changes due to the changes in the model. Different types of (active) learners are distinguished by the methods they use to select the sample and the *informativeness measure* they use to decide what to query next. The three main selection scenarios identified by Settles [8] are: *membership query synthesis*, *stream-based selective sampling* and *pool-based sampling*. Each scenario describes a common way to sample the dataset and together they cover a large part of the learners seen in practice. The *informativeness measure* is the score function that is maximized to find the optimal vector to query next. In the literature the optimal query vector is denoted by $\mathbf{x}^*$ and the measure usually has the form:

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in X} S_\theta(\mathbf{x}), \tag{3.1}$$

where X contains all vectors in our sample and $S_\theta(\mathbf{x})$ is the score for vector $\mathbf{x}$ when using model θ. Each step of the learning process the active learner will select some sample from the dataset and for each of them calculate the score, based on its current model θ. The vector with the best score is selected and its label queried from the oracle. This new information is added to θ and the process repeats until the learner is satisfied with its classifier. This stopping criteria should theoretically be based on the learners confidence in its solution, however in practise it is more likely that learners stop well before to conserve resources.

3.1.1 Sampling settings

The sampling setting describes the method that is used to obtain the samples. There are three general settings, *synthesis*, *stream-based* and *pool-based* sampling. These three should capture all practical settings one might encounter while employing machine learning to learn classifiers. Stream- and pool-based approaches process actual instances of the data set, whereas the synthesis approach is allowed to form its own instances, granted that they fit somewhere into the feature space.

The most general setting is *membership query synthesis*, where the leaner is allowed to formulate and ask any query that fits its model. Typically the learner asks only queries that are sampled from some underlying distribution of the input space, but with *membership query synthesis* it is allowed to formulate a query de novo. This allows the learner to be very specific about the what it wants to learn resulting in both flexibility and accuracy. This approach is attractive because it can be shown that query synthesis is often both tractable and efficient for finite problem domains. The problem

is however that the domain should be well suited to handle a wide range of different queries. Take for example the case of handwriting-recognition. Based on its model the learner might decide to synthesize a hybrid character that would result in the most information gained. Such characters would however have no syntactical meaning and a domain expert would have difficulties interpreting and labelling them. Settles concludes that this approach might work best in fields where non-human experts are used that have a way of interpreting in-between queries. Think for example about a robot learning to judge its arm position based on the angles between joints. Every angle combination would result in a proper spatial location which could be measure by some outside observer, allowing the learner to synthesise whatever query it wants. The important aspect to this setting is that the learner is very specific about the vector it samples and queries. The value of a vector is already decided before it is either sampled or synthesised and subsequently queries.

Selective sampling is an alternative to the *membership query synthesis* approach. It is assumed that in this settings obtaining unlabelled vectors is inexpensive or free, so the learner can so to speak sample first and ask questions later. The decision on whether to query a sampled vector or not is usually based a query strategy which allows the learner to make biased random decision where more informative vectors are more likely to be queried. Another approach is to assign explicit regions op uncertainty, and only query the vectors that the learner is still uncertain about. The key point here is that the learner is allowed to sample many vectors and decides whether to query them on an vector-to-vector basis, therefore this approach is sometimes also called the *stream-based* or *sequential* approach.

The *pool-based sampling* setting assumes that there are two pools of data available, a small pool of labeled data and a larger pool of unlabeled data. The information in the labeled pool is used to create a model that decides which of the vectors from the unlabeled pool to query next. For many real-world applications it is possible to gather both these pools at once so no further sampling is necessary. Typically the selection of the next query point is done in a greedy manner by evaluating some informativeness criteria (based on the labeled data) until all of the unlabeled vectors have been processed.

3.1.2 Informativeness measure

To select the next vector to query the learner computes the result of equation 3.1 for each of the vectors in its sample. The process is also known as the *query strategy* as the decision which vector

to query is one of the critical aspects of the active learner. There are a large number of way to calculate the informativeness of the vectors in the sample, based on the (estimated) distribution of the data and the model that the learner is building. Settles [8] quite extensively describes a number of the most used approaches, discussing the pros and cons for each. The main challenge is to decide what constitutes 'most informative' query: We can query the vector that would make the current model change the most (that is the vector that maximizes the *Expected Model Change*) and hope that in this way we can converge to the best model in the quickest manner. We could also query the vector that would increase the learners confidence in its model the most (that is the vector that maximizes the *Expected Error Reduction*), aiming instead to quickly find a model the learner is most confident in. Both these approaches however can be computationally heavy since they require the model the be retrained each time we want to consider a new vector. Our main focus therefore will be instead on two other strategies, *Uncertainty sampling* and *Query-by-Committee*.

The most common and straightforward measure is *uncertainty sampling*. The key idea is that the leaner has available a model θ that will be used to predict the labels for each unlabeled vector and its confidence. This model θ can for example be obtained from some small set of prior available labeled data or by analysing data that has been labeled so far. The learner selects the vector about whose prediction it is least confident based on the value of its uncertainty measure. The form of this measure depends on the sort of classification (amount of labels in the problem) and the aim of the model (minimizing classification error, or log-loss etc. see [8]). In general, we want to find the instance that we are least certain about:

$$\mathbf{x}_H^* = \arg\max_{\mathbf{x}} \left[-\sum_i P_\theta(y_i|\mathbf{x}) \log P_\theta(y_i|\mathbf{x}) \right], \tag{3.2}$$

where $P_\theta(y_i|\mathbf{x})$ represents the probability of the label y_i given $\mathbf{x}$, under the current model θ. This measure is sometimes also revered to as the *entropy*, which is a measure of 'uncertainty' in the distribution. So, in fact, we are looking for the vector $\mathbf{x}$ that would remove the most 'uncertainty' from the solution if summed over the posterior probabilities of all possible labels $y_i \in Y$.

A different approach is *query-by-committee*. The learner trains a set number of models $\theta^{(i)}$ that are consistent with the labels it has learned so far. Together the models for a committee $\mathcal{C} = \{\theta^{(1)}, \ldots, \theta^{(C)}\}$, if a model no longer agrees with all labels that have been learned so far, as might happen after new information is added, the model is removed from $\mathcal{C}$. The committee votes on the labels of each the yet unknown query-candidates, allowing each member to predict the label of the yet unknown vectors. The candidate about which most disagreement exists after voting

is the next query, since learning its label would eliminated the greatest number of committees from $\mathcal{C}$. To implement a *query-by-committee* framework two things are necessary; The ability to construct a committee of rivalling models that each support different hypothesis, for example we can have a number of different decision boundaries that support some set of known vectors. Once the labels of additional vectors are learned, some boundary might no longer be valid and can be eliminated. The second condition is that the disagreement between the committee members must be quantifiable. For example with decision boundaries each remaining unlabeled vector will fall to either side and we can measure the disagreement by simply tallying votes. We can formalize the *query-by-committee* strategy as follows:

$$\mathbf{x}^*_{VE} = \arg\max_{\mathbf{x}} \left[-\sum_i \frac{V(y_i)}{C} \log \frac{V(y_i)}{C} \right], \tag{3.3}$$

where $V(y_i)$ is the number of 'votes' cast for y_i being the best label for $\mathbf{x}$ and C denotes the number of committees in $\mathcal{C}$. We are again looking at a measure of entropy, meaning we aim to query the vector that reduces the dissension the most.

3.2 Noisy Oracles

The basic assumption of most active learners is that their oracles will always return the true label for the queried vectors. An interesting practical consideration is how the learner should handle oracles that do not have this guarantee. In practice, the assumption that oracles don't make mistakes might be too strict. Human domain experts become tired or distracted and it is not unthinkable that they make a mistake now and then. The aim of active learning is to build a representative classifier by asking as few queries as possible. This makes the information gained by each query more valuable, as each query should constitute as much information as possible. However this poses a potential problem if this information can no longer be relied upon, in the sense that the oracle might return the incorrect label. Some potential approaches that deal with this problem have been investigated. A learner needs to be able to judge the quality of its labeled instances, and decide how it handles potential noisy answers form the oracle. Important considerations are whether to keep re-query the same vector if it is suspected to be noisy or to simply abandon it to query something else. Sheng et al. [9] use an approach where multiple oracles are used to improve the reliability of queried labels by selectively re-querying.

Another possible solution here is to use a *query-by-committee* approach that lets competing

models vote on the currently labeled vectors. If there is a lot of disagreement on a particular label, the learner might decide to discard its label and re-query. There are however numerous problems with this approach. Firstly, since we cannot simply select models that are consistent with the current set of labeled instances as with the *query-by-committee* approach, how do we choose which models form the committee? Secondly, should each model in the committee get the same vote, or can some models be more right than others? Furthermore, can we simply choose to discard labels we suspect to be incorrect? There is always a possibility that the suspected label was, in fact, the correct label for that vector, which would lead us to discard useful information. Some of these questions are considered by Barile and Feelders in the discussion of their active learning algorithms for stochastic oracles in [1].

Chapter 4

Active Learning with Monotonicity Constraints

4.1 Active Learning with monotone functions

The goal in (standard) active learning is to find those vectors that represent the dataset well enough that a classifier can be learned while only part of the labels are uncovered. In active learning with monotone functions the learning of the classifier is done implicitly while we are learning. Our goal is to find query-candidates that are representative of the dataset. However we aim to learn all we can directly each time we query for a label, instead of building a sample. What this means is, that each time a label is observed we use the knowledge that we are working with monotone functions to infer more information about our classifier directly. A monotone function is a function whose values only increase (or decrease) as the function progresses. If we have a function $f(x)$ that is monotonically increasing we know that for a pair of values x, x' we have that if $x \leq x'$ it follows that $f(x) \leq f(x')$, figure 4.1 shows a function that is monotonically increasing. If we know the value y of some point on a monotonically increasing function we can conclude that any points above will have values that are *at least* y, while points below will have values that are *at most* y. This observation is what we will use to learn additional labels from the labels we observe from the oracle which should greatly decrease the number of queries it takes to learn the entire classifier. While both monotonically increasing and decreasing functions can be used in active learning, most literature focusses solely on increasing functions. This should however not be viewed as a restriction, as the conversion from increasing to decreasing functions is trivial and

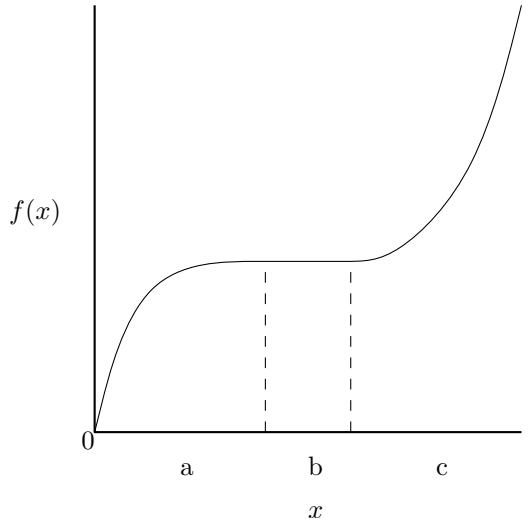

Figure 4.1: The function $f(x)$ is monotonically increasing. Intervals a and c are **strictly** monotonic, meaning $x < x' \implies f(x) < f(x')$.

all arguments made also hold in the decreasing case.

The key assumption in active learning with monotonicity constraints is that attribute values have a strictly positive or negative influence on the class-labels. As discussed previously, if our dataset has positive monotonicity constraints, a higher value on an attribute should not result in a lower classification. In the case of negative constraints this relation would be reversed, a higher attribute should not lead to a higher class-label. This assumption of monotonicity is rather strong, but also provides a powerful tool when used in combination with active learning.

In practice there are many examples where we can find monotonicity constraints, for example consider a bank which receives two loan applications a and b. If applicant a has a better credit rating than applicant b, it would be strange that if b's loan is approved, a's would be be denied. Intuitively one could say that once someone has been approved for a loan, every person with a better credit rating should automatically also be approved. Another example comes from the realty business; a housing agent judges property values by comparing it to properties that have similar attributes. This process however is mostly done by looking at said property in person, which is both costly and time consuming. Important factors in the pricing of a house are things like lot size, floor area and the number of bathrooms, where larger quantities of any result in a higher price. Clearly these attributes have a positive influence on the final price, making this field a good candidate for learning with monotonicity constraints. By having the learner choose which houses are best to get appraised, or queried, the workload of a housing agent can be greatly reduced.

Many applications have such intuitive relations between attribute values and class-labels and active learners with monotonicity constraints aim to exploit these relations to select better query-candidates. In this section we will review some of the work done in this area that forms the basis for our query strategies. We will start with the general case where we assume an unfailing oracle that returns the true label for each query. Later we will discus the case of a noisy oracle which has a non-zero probability of returning a label other than the true label. Different authors use different nomenclature; what we call a labeling, others might refer to as classifier, model or simply function. A labeling will be assumed to be monotone, that is equation 2.2 holds, if not otherwise specified. For a dataset A we will use the notation $(A, \preceq)$ to indicate that it is a poset with an associated ordering.

4.1.1 Deterministic case

We start with deterministic oracles, that is to say oracles that always return the true label, and review the work of Barile and Feelders [1]. They derive an expression for computing the optimal query-point by analysing the intuitive case of a dataset ordered as a simple chain. Unfortunately solving the expressions turns out to be a hard problem. Therefore they develop a worst-case heuristic which they implement in the MAL algorithm. In terms of the concepts of chapter 3.1 we have a sample consisting of the entire dataset and for each vector we want to evaluate some expression that measures the value of learning its label. The model θ will simply consist of the partial classifier we have built so far and each cycle it's extended by adding the new information.

In this strategy all vectors are considered to be potential query-candidates. The key assumption is that the best candidate is the one which allows us to infer the most labels. Or to put it differently, we look for the candidate that, once its label is known, eliminates the most potential labels for all the other vectors. For each candidate we want to estimate the number of labels that can be inferred once its labels is known. That is we want to calculate:

$$\mathbb{E}[N(\mathbf{x}_i)] = \sum_{l=1}^{l=k} \hat{P}(y_i = l) I(y_i = l), \tag{4.1}$$

where $\hat{P}(y_i = l)$ is the estimated probability that the true label of $\mathbf{x}_i$ is l and $I(y_i = l)$ is the number of labels that can be inferred.

Since we do not know the distribution of the oracle the best approach is to estimate the probabilities by using the monotonicity constraints. Without any further domain knowledge all

monotone classifications are equally likely and we can say that the set of all monotone functions is uniformly distributed. In some fields one might expect that extremes, like cases where all vectors have the same true label, are not as likely as the cases were the labels are spread more uniformly. However, this is domain specific and it makes sense to ignore this aspect in favour of a more general approach. For each query-candidate we can calculate the probability of its label by counting how often each label occurs. The probability of vector $\mathbf{x}_i$ having the label l is then given by:

$$P(y_i = l) = \frac{C(y_i = l)}{C}, \tag{4.2}$$

where C is the total number of monotone functions, and $C(y_i = l)$ is the number of monotone functions where $\mathbf{x}_i$ gets assigned the label l. To find the best query-candidate we now only need to find the maximum of the expression for the expected number of inferred labels. The only problem that remains is finding the counts for the number of inferred labels and the number of monotone functions. We will illustrate how these counts can be related to the concepts of up and downsets.

Suppose we have dataset and ordering $(X, \preceq)$ of n vectors which takes the form of a chain, that is $x_1 \preceq x_2 \preceq ... \preceq x_n$. We are looking for labels that respect the monotonicity constraint, given by: $x_i \preceq x_j \implies y_i \leq y_j$. For simplicity we will take all functions to be binary for now, that is $Y = \{1, 2\}$. To get a monotone classification we can assign the label 1 to any initial segment of the chain and the label 2 to the remainder. We also include the empty segment, which would correspond to the case where all vectors get assigned the label 2, giving us a total of $n + 1$ monotone classifications. To count the number of classifications where a vector gets the label 1 (or the label 2) consider the following: Every time a vector $\mathbf{x}$ is assigned the label 1 all its predecessors also need to get assigned the label 1, otherwise we would violate monotonicity. We know that only vectors that succeed it in the order can potentially have the label 2, meaning that in every monotone classification where $\mathbf{x}$ gets the label 1 at least some of its successors will have the label 2. We also know that if a vector gets label 2, all its successors must also get the label 2. We can now conclude that the number of classifications where $\mathbf{x}$ gets the label 1 is equal to the number of successors it has. Note that all the successors of a vector $\mathbf{x}$ belong to its upset $\uparrow(\mathbf{x})$. The number of items in $\uparrow(\mathbf{x})$, and therefore the number of classifications where $\mathbf{x}$ gets assigned the label 1, is given by: $u(\mathbf{x})$. The reverse holds true for the number of times a vector is assigned the label 2. Instead of items in the *upset* we count items in the *downset* of $\mathbf{x}$ and we find that the number of times a vector $\mathbf{x}$ is assigned the label 2 is equal to $d(\mathbf{x})$. The probabilities can now be computed by plugging these results back into equation 4.2. Note that for a chain, we have that

for an arbitrarily chosen vector $\mathbf{x}$ all other vectors are either in $\uparrow(\mathbf{x})$ or in $\downarrow(\mathbf{x})$. We find that the number of monotone functions is equal to $u(\mathbf{x}) + d(\mathbf{x})$, which is exactly $n + 1$.

$$P(y_i = 1) = \frac{u(\mathbf{x}_i)}{u(\mathbf{x}_i) + d(\mathbf{x}_i)} \qquad P(y_i = 2) = \frac{d(\mathbf{x}_i)}{u(\mathbf{x}_i) + d(\mathbf{x}_i)}.$$

To calculate to expected number of inferred labels we also need to know the number of inferred labels when $\mathbf{x}_i$ gets assigned label l, $I(y_i = l)$. Imaging we presented vector $\mathbf{x}_i$ to the oracle and learn that its true label is 1. Due to the monotonicity constraint in equation 2.2 we know that all vectors that are a predecessor of $\mathbf{x}_i$ must also get the label 1. The predecessors of $\mathbf{x}_i$ are those vectors that belong to $\downarrow(\mathbf{x}_i)$ and therefore the number of inferred labels can be expressed simply as $I(y_i = 1) = u(\mathbf{x}_i)$. The case where $\mathbf{x}_i$ true label is 2 is similar but reversed; if $\mathbf{x}_i$ gets the label 2 all its successors must also get the label 2 and we find: $I(y_i = 1) = d(\mathbf{x}_i)$. By plugging all this back into equation 4.1 we find the following expression for the number of inferred labels $N(\mathbf{x}_i)$ when querying $\mathbf{x}_i$:

$$\mathbb{E}[N(\mathbf{x}_i)] = P(y_i = 1)d(\mathbf{x}_i) + P(y_i = 2)u(\mathbf{x}_i) = \frac{u(\mathbf{x}_i)d(\mathbf{x}_i)}{u(\mathbf{x}_i) + d(\mathbf{x}_i)} + \frac{d(\mathbf{x}_i)u(\mathbf{x}_i)}{u(\mathbf{x}_i) + d(\mathbf{x}_i)}. \tag{4.3}$$

The best query-candidate is of course the vector that infers the most labels and hence the vectors that maximizes this expression, which we can find analytically: Let c_a be the total number of possible assignments, that is $u(\mathbf{x}_i) + d(\mathbf{x}_i) = c_a$, then:

$$\max(\mathbb{E}[N(\mathbf{x}_i)]) = \max(2\frac{u(\mathbf{x}_i)d(\mathbf{x}_i)}{c_a}). \tag{4.4}$$

Note that $d(\mathbf{x}_i) = c_a - u(\mathbf{x}_i)$ we can simplify this expression by substituting either $u(\mathbf{x}_i)$ or $d(\mathbf{x}_i)$;

$$\max(\mathbb{E}[N(\mathbf{x}_i)]) = \max(u(\mathbf{x}_i)n - u(\mathbf{x}_i)^2). \tag{4.5}$$

The right-hand-side of this expression is a quadratic equation and we can find the the maximum by setting the first-order derivative to 0 and solving for $u(\mathbf{x}_i)$ (mind that the $d(\mathbf{x}_i)$ in the following expression is the first derivative with respect to $\mathbf{x}_i$ and not the number of items in this downset):

$$\frac{d}{d(u(\mathbf{x}_i))}[u(\mathbf{x}_i)n - u(\mathbf{x}_i)^2] = 0.$$

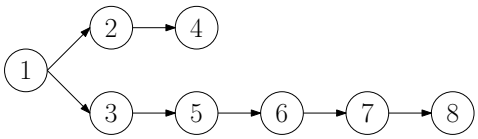

Figure 4.2: A poset consisting of 8 vectors.

Solving this equation for $d(\mathbf{x}_i)$ and $u(\mathbf{x}_i)$ we find: $u(x_i) = \frac{n}{2}$, and by definition of n: $d(\mathbf{x}_i) = \frac{n}{2}$. The maximum of the expected number of inferred labels occurs when we choose the vector $\mathbf{x}_i$ for which the *upset* and the *downset* are of equal size, that is $d(x_i) = u(x_i)$. To put it in other words, the best vector to query is the vector which lies most *in the middle* of the chain, which intuitively shouldn't come as a surprise. In the worst-case we would still be able to infer half of labels by asking only one query. This result is very similar to the binary search strategy used for searching for an element in a sorted list; we can infer all labels by querying only $log(n) + 1$ times, if we iteratively select the candidate that is in the middle and remove candidates that already have had their labels inferred.

Extending this strategy to general posets is a little more involved as we can't simply count the member of the up and down sets. We will first illustrate the problem by applying the strategy described above to the poset in figure 4.2. Observe that this time not all vectors are comparable and we have an ordering that consists of two chains joined at the beginning. Intuitively the best query-candidate would be one of the vectors from the lower chain, say vector $\mathbf{x}_5$ or $\mathbf{x}_6$, since they would infer the most labels in the worst-case. If we would apply the binary search strategy however we find that $u(\mathbf{x}_5) = 3$ and $d(\mathbf{x}_5) = 2$, and for the other $u(\mathbf{x}_6) = 2$ and $d(\mathbf{x}_6) = 2$. Both not equal and in fact also not "most equal". If we look at $\mathbf{x}_2$ we find $u(\mathbf{x}_2) = 1$ and $d(\mathbf{x}_2) = 1$. Clearly vector $\mathbf{x}_2$ is "more equal" than vectors $\mathbf{x}_5$ or $\mathbf{x}_6$ and hence the strategy would decide vector $\mathbf{x}_2$ is the optimal query-candidate. The problem here is that the strategy no longer considers all monotone classifications for each case. If we look at $\mathbf{x}_2$ for example, we see that $\uparrow(\mathbf{x}_2)$ only has 1 member, namely $\mathbf{x}?4$. Assigning the labels 2 and 1 (for the empty set) to $\mathbf{x}_4$ gives two different classifications, but we could also assign the label 2 to any of the segments of the lower branch (vectors $\mathbf{x}_3$, $\mathbf{x}_5$, $\mathbf{x}_6$,$\mathbf{x}_7$ and $\mathbf{x}_8$) and get a monotone classification. Clearly we can no longer just count members of the up and downsets to find all monotone classifications. We need a measure that allows us to also consider all the vectors that are in all the other branches of the order. The answer is to count the number of upper and lower sets. Look again at the order in figure 4.2, if we assign the label 2 to any final segment of any or both of the two branches and the label 1 to the rest we find a monotone classification. Assigning the label 2 to vectors $\mathbf{x}_4$, $\mathbf{x}_7$ and $\mathbf{x}_8$ gives a

monotone classification. Or we could assign the label 2 only to the lower branch $\mathbf{x}_3$, $\mathbf{x}_5$, $\mathbf{x}_6$,$\mathbf{x}_7$ and $\mathbf{x}_8$, which is also a monotone classification. It turns out that these final segments are all upper sets of the order; assigning the label 2 to all members of an upper set and the label 1 to all members of its complement lower set yields a monotone classification. If we want to know how many times a vector gets assigned the label 2 we only need to count the number of upper sets in which it's included. Conversely, if we want to know how many times a vector gets assigned the label 1 we count the number of lower sets in which it's included. Hence we can rewrite the probabilities in terms of upper and lower sets: a vector $\mathbf{x}_i$ gets assigned the label 1 (respectively label 2) exactly as many times as it is included in a lower set (respectively upper set). Clearly there is a one-to-one correspondence between lower sets and monotone binary classifications. Plugging the new counts into equation 4.2 yields:

$$P(y_i = 1) = \frac{L(\mathbf{x}_i)}{L(\mathbf{x}_i) + U(\mathbf{x}_i)} \qquad\qquad P(y_i = 2) = \frac{U(\mathbf{x}_i)}{L(\mathbf{x}_i) + U(\mathbf{x}_i)},$$

where $L(\mathbf{x}_i)$ denotes the number of lower sets that include $\mathbf{x}_i$, and $U(\mathbf{x}_i)$ the number of upper sets that include $\mathbf{x}_i$. Using the same approach as before we can determine the best vector to query by maximizing the expected number of inferred labels. Unsurprisingly we find a similar result; we should query the vector for which the number of times it appears in an upper set is closets to the number of times it appears in a lower set. If we turn back to our example, the number of times $\mathbf{x}_2$ appears in a lower set is 12, while the number of times it appears in an upper set is 7. If we calculate the values for vectors $\mathbf{x}_5$ and $\mathbf{x}_6$ we find: $L(\mathbf{x}_5) = 12$ and $U(\mathbf{x}_5) = 7$, and $L(\mathbf{x}_6) = 9$ and $U(\mathbf{x}_6) = 10$. Clearly we should ask the oracle for the label of $\mathbf{x}_6$ as our intuition predicted at the start. Extending this concept to general classifications requires a nested sequence of lower sets of $(Q, \preceq)$, say $L_1 \subseteq L_2 \subseteq \cdots \subseteq L_{k-1}$. We can now build a monotone function by assigning to each of the vectors in the lower sets its associated label, that is:

$$f(x) = \begin{cases} j & \text{if } x \in L_j \setminus \bigcup_{i<j} L_i, j = 1, \ldots . k-1 \\ k & \text{otherwise} \end{cases} \tag{4.6}$$

Unfortunately counting the number of lower sets of a partial order is a hard problem and therefore Barile and Feelders [1] propose a worst-case heuristic to solve this problem. The main result of the strategy discussed still holds, although we cannot directly calculate the optimal point. We still know that our best bet is to find a query-candidate that is somewhere "in the middle". This

is the basis for the heuristic. We want a strategy that still maximizes the number of labels that can be inferred in the worst-case, where the worst-case is the potential label that infers the fewest labels. For each vector $\mathbf{x}_i$ we determine the size of its upset and its downset, giving us the amount of labels inferred if we assign label 1 and label 2 respectively. The worst case then is if the oracle answers the label belonging to the smallest set. The heuristic selects the query-candidate for which the worst-case yields the best result, that is:

$$x^* = \arg\max_{\mathbf{x} \in X} \min\{d(\mathbf{x}), u(\mathbf{x})\}.$$

Note how this heuristic still prefers vectors that are near the middle of the order: Vectors near the edges of the order will have an up- or downset that is much larger than the other and get a low value for the inner expression. Whereas this expression will be greatest for vectors that have up- and downset of near equal size, which are the vectors near the middle. To illustrate, for the order in figure 4.2, the vectors for which the smaller of its up- and downset has the greatest value are $\mathbf{x}_6$ and $\mathbf{x}_5$, which is in line with both the results of the counting strategy as well as our intuition. In the non-binary case, we can no longer assume we can always infer the true label from an oracle answer. We therefore look at the number of labels that can be excluded from the interval $[l_i, h_i]$ for each x_i, where each vector has an initial interval of $[1, k]$. To count the number of labels that are eliminated when x_i gets the label $y_i = y$ we use:

$$N(\mathbf{x}_i, y) = \sum_{\mathbf{x}_j \in \downarrow(\mathbf{x}_i)} (h_j - y)_+ + \sum_{\mathbf{x}_j \in \uparrow(\mathbf{x}_i)} (y - l_j)_+, \tag{4.7}$$

where $z_+ = max(0, z)$. The vector we want to query is the vector that eliminates the most labels in the worst-case:

$$x^* = \arg\max_{\mathbf{x}_i \in X} \min_{y \in [l_i, h_i]} \{N(\mathbf{x}_i, y)\}. \tag{4.8}$$

4.1.2 Monotone Active Learner

The MAL algorithm keeps two lists: Q contains vectors whose labels are known and their labels, while U contains vectors whose label is not yet determined. For each label MAL keeps an upper and a lower bound of labels that are still valid. Each iteration of the main loop of the MAL algorithm will calculate the score (equation 4.7) for all $\mathbf{x}_i \in U$. The vector with the best score is selected and queried, added to Q and removed from U. Once the label is known, the information

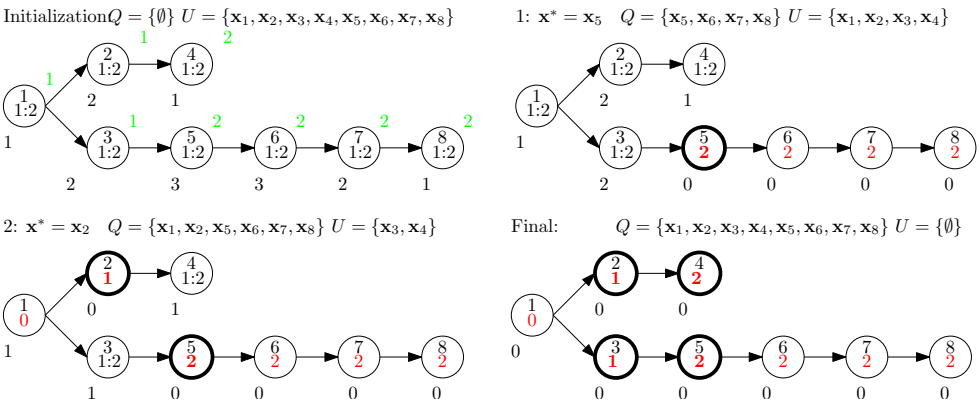

Figure 4.3: Initialization and two steps of the MAL algorithm. Indicated below each vector index are the bounds, the first pane shows the true labels in the upper-right corner in green. The the lower-left of each vector the score for that vector is show (resulting from equation 4.7). The bold circles indicate vectors that are selected for querying, single (red) number indicate labels that are predicted by the classifier and bold number are queried whereas other are inferred.

is used to infer tighter bounds on the remaining unlabelled points. If the interval of a vector vanishes, that vector is also added to Q together with the inferred label and removed from U. This process is repeated until all vectors are labelled or until a maximum of iterations has been reached. See appendix A for a listing and a more detailed description. Example 4.1 illustrates a the working of MAL.

Example 4.1. In figure 4.1.2 we show the working of the MAL algorithm on an example order. At initialization the list Q will be empty and the list U of query-candidates will hold all vectors. Each vector has associated an upper and a lower bound $[l_i, h_i]$, of labels that are still valid for the vectors $\mathbf{x}_i$, we start out with the bounds $[1, 2]$ for all vectors. If we apply equation 4.7 to each vector we find that both vectors $\mathbf{x}_5$ and $\mathbf{x}_6$ infer 3 labels in the worst-case. We arbitrarily pick $\mathbf{x}^* = \mathbf{x}_5$ and observe its label to be 2, adding $(\mathbf{x}_5, 2)$ to Q and $\mathbf{x}_5$ is removed from U. At this point the algorithm infers the labels of all vectors in $\uparrow(\mathbf{x}_5) = \{\mathbf{x}_6, \mathbf{x}_7, \mathbf{x}_8\}$ to be 2 as well meaning we can also add those to Q and remove them from U. The situation is shown in the second pane. Scores of vectors whose labels are known are set to 0 and we find that we again have two query-candidates with equal scores, vectors $\mathbf{x}_2$ and $\mathbf{x}_3$. This time we pick $\mathbf{x}^* = \mathbf{x}_2$ and observe the label 1. We add $\{\mathbf{x}_2, 1\}$ to Q and since its label was 1 we infer all labels from $\downarrow(\mathbf{x}_2) = \{\mathbf{x}_1\}$. This situation is shown in the third pane. We are now left with two single vectors that only infer their own labels so the algorithm finishes by querying for both. The final situation is shown in the fourth pane. The vectors in the bold circles had their labels determined by the oracle. The rest have had their labels inferred. Observe that the vectors that the algorithm chooses to query

are all towards the centre of the order.

4.1.3 Related work

Dasgupta [3] gives an analysis of this class of strategies which he classifies as *generalized binary search* or GBS for short. Let H denote a hypothesis class and let $\hat{H}$ denote the effective hypothesis class for a dataset X that is yet unlabelled. π will denote the probability distribution over our class of effective hypothesis $\hat{H}$. The goal is to find the hypothesis $h \in \hat{H}$ that is consistent with the true (or *hidden*) labels of the vectors in X, in as few queries as possible. For a binary set of labels $\{1, 2\}$ assume that we have already queried for the labels of a number of vectors up until the ith vector $\mathbf{x}_i$. The set S is a subset of the hypotheses from $\hat{H}$ that are still consistent with the labels of the vectors that have been queried so far: $S \subset \hat{H}$. Let S_i^+ be the extension of S where $\mathbf{x}_i$ gets assigned the label 2 and lets S_i^- be the extension of S where $\mathbf{x}_i$ gets assigned the label 1. The greedy strategy that Dasgupta [3] proposes is to select the $\mathbf{x}_i$ for which the probability for the sets S_i^+ and S_i^- are most equal, that is $\pi(S_i^+) = 0.5$. Dasgupta [3] continues by proving that the upper bound for the number of queries needed by a GBS strategy is equal to: $4Q^* \ln \frac{1}{\min_h[\pi(h)]}$, where Q^* is the number of queries needed by the optimal strategy. In particular, if the distribution is uniform, the upper bound reduces to $4 \ln |\hat{H}|$. The class of strategies described by Dasgupta [3] has clear similarities to the strategy proposed by Barile and Feelders [1]. Limit the effective hypothesis class $\hat{H}$ to monotone functions that are valid on the data $(Q, \preceq)$ and note again the correspondence between lower sets and monotone classifications. The sets S_i^- and S_i^+ are respectively the number of lower sets that include $\mathbf{x}_i$ and the number of upper sets that include $\mathbf{x}_i$. Our goal is now to find the vector $\mathbf{x}_i$ for which the probabilities of these sets are most equal. In other words we look to find the vector which appears in the most equal number of upper and lower sets.

4.1.4 Stochastic case

Up until now we have made the assumption that oracles behave in a deterministic manner and always return the same true label. In this section we will review the case where an oracle can be incorrect and might not return identical labels when presented repeatedly with the same query. The former point especially is important when considering the strategy employed so far, as it may present problems with the monotonicity of the classifications. Take for instance a chain with some hidden monotone classification. We present a vector $\mathbf{x}_i$ to the oracle as before and receive a label y_i. Based on this label we infer the labels of points in the up- and downsets of $\mathbf{x}_i$ and advance to

the next iteration. Next we query $\mathbf{x}_j$ and receive the label y_j. If everything goes as before (the deterministic case) we should end up with labels such that either $y_i \leq y_j$ if $\mathbf{x}_i \preceq \mathbf{x}_j$ or $y_j \leq y_i$ if $\mathbf{x}_j \preceq \mathbf{x}_i$, and the monotonicity constraint is satisfied. However, since the oracle might return labels that are incorrect we might also end up with either $y_i > y_j$ if $\mathbf{x}_i \preceq \mathbf{x}_j$ or $y_j > y_i$ if $\mathbf{x}_j \preceq \mathbf{x}_i$, which both do not satisfy the constraint in equation 2.2. Furthermore, if we infer labels from an incorrect answer all of the inferred labels might also be incorrect. The main problem we have to consider is how to deal with *monotonicity violations* in our (partial) classifier as the monotonicity assumption was crucial for the inference done by the learner. We do note that we assume that our oracle always does contribute some knowledge, meaning that the expert is expected to perform better than random assignment of class labels to vectors. While this assumption is consistent with intuition (and may seem trivial), Torvik [11] shows that in the binary case taking an error-rate that is larger than 50% radically transforms the problem. He shows that if the error-rate q is below 0.5 our objective is equivalent to minimizing the number errors in the classifier. If however q is greater than 0.5 it becomes equivalent to *maximizing* the number of errors. It is therefore important to make the restriction that our oracles have at least some knowledge of their domain.

Barile and Feelders [1] propose to solve the problem of monotonicity violation by repairing the (partial) classifier that the algorithm is learning. They use a stochastic framework by Kotlowski [5] that captures the non-deterministic behaviour of the new oracle and allows for the underlying monotone function to be recovered. Finally they employ an *empirical risk minimization approach* to find the best estimate of the *Bayes classifier* (the monotone function that best describes the underlying distribution).

In the stochastic framework the original monotonicity constraint (equation 2.2) no longer holds due to the fact that the underlying function is no longer monotone (in this case, the oracle), but we do assume that the underlying distribution still exhibits so-called *stochastic dominance*. This new constraint is less strict in the sense that we can no longer claim that the value of the classifier is monotone, but instead we assume that the probability distributions $P(\mathbf{x}, y)$ for different values of $\mathbf{x}$ can *dominate* each other. This concept is illustrated in figure 4.4, the (cumulative) distribution to the right is dominated by the distribution on the left for each value of y. In our setting this could be interpreted as follows: Take two vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. If we have that $\mathbf{x}_i \preceq \mathbf{x}_j$ then according to *stochastic dominance* it should be more probable that $\mathbf{x}_j$ gets a higher label than $\mathbf{x}_i$. We define

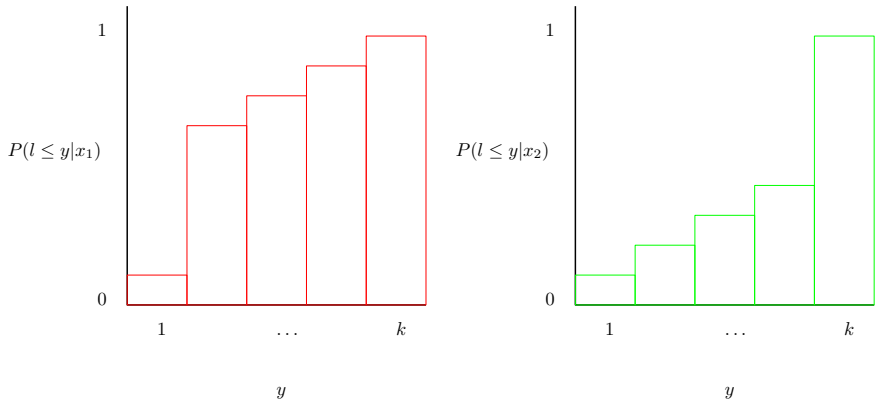

Figure 4.4: Two graphs showing the cumulative probability distributions for two vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ of some label l being smaller than the value of y. The green distribution is clearly dominated by the red distribution.

this relation as follows:

$$\mathbf{x}_i \preceq \mathbf{x}_j \implies \forall j : P(y \leq j|\mathbf{x}_i) \geq P(y \leq j|\mathbf{x}_j), \tag{4.9}$$

where $j = 1, ..., k$. That is to say, if $\mathbf{x}_i$ precedes $\mathbf{x}_j$ in the order, the cumulative probability distribution over all labels for $\mathbf{x}_i$ should be *above* the distribution for $\mathbf{x}_j$.

With this in mind we can begin our search for the underlying monotone function. It should be noted that we assume that the underlying function that the oracle uses to generate its responses is still monotone and that the oracle simply makes mistakes somewhere along the line. This also means that we assume that there is some "true" monotone function that we are trying to recover by probing the oracle in clever spots. We will call this function t. In a sense we now have two parallel goals. On the one hand there is the active learner who aims to piece together a classifier by repeatedly querying for new labels. On the other hand we have our stochastic framework whose aim it is to make sure this *partial classifier* we are learning stays monotone. It is important to keep this distinction in mind, as this will be the basis for the final algorithm (SMAL). We will start by working with the stochastic framework part to find a way to as it were "re-monotonize" the *partial classifier*.

Kotlowski [5] shows that the *Bayes classifier* can be found by finding the function that minimizes the *Bayes risk*, that is the function that expresses the expected loss for each potential function. We call the collection of all monotone functions on our data $\mathcal{F}$ and we want to find the

function $f^* \in \mathcal{F}$ that best matches the true function t. We can formulate this as follows:

$$f^*(\mathbf{x}) = \arg\min_{j \in Y} \sum_{y \in Y} L(y, j) P(y|\mathbf{x}).$$

Where $L(x, j)$ represents the loss incurred if our function predicts y and the true label is j and $P(y|\mathbf{x})$ represents the probability distribution of observing the label y if querying vector $\mathbf{x}$. It can be show that if we use a convex loss-function and if the distribution satisfies the stochastic order constraints, the function f^* is monotone [5]. That is, given these conditions, we have the following for any two vectors $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$\mathbf{x}_i \preceq \mathbf{x}_j \implies f^*(\mathbf{x}_i) \leq f^*(\mathbf{x}_j).$$

In other words, if we can estimate f^* we are guaranteed to end up with a classification that satisfies the monotonicity constraint in equation 2.2, which was our initial goal.

4.1.5 Stochastic Monotone Active Learner

We return to the work of Barile and Feelders [1], who show a possible approach to estimating f^* using a *risk minimization approach*. They continue by implementing this "relabeling" step into the MAL algorithm to solve the problem of *monotonicity violations*. The result is a two-step algorithm they dub SMAL (for *Stochastic Monotone Active Learner*), which shows encouraging results in both artificial- and real-data experiments. The first step is the same as in the MAL algorithm, we look for the best query-candidate and request its label. In the second step the (partial) classification we have so far is passed to a *relabeling-algorithm* which returns the best estimate for the true classification.

The *Bayes classifier* can be estimated by minimizing the loss:

$$\hat{f}(x) = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{n} L(y_i, f(\mathbf{x}_i)),$$

where n is the total number of vectors in the dataset. $L(y_i, f(\mathrm{x}_i))$ is the loss incurred by function f when predicting the label for the ith vector. Here y_i indicates the label of the function we want to "monotonize". That is, the (partial) classification the learner has learned up until now. The *relabeling-algorithm* aims to find the function $\hat{f}$ that minimizes absolute loss and returns for each labelled vector an interval $[l_i, h_i]$ of optimal labels (For a more thorough description see chapter

8).

The SMAL algorithm keeps a list Q of vectors that are labelled so far and a list U of unlabelled vectors. Each iteration of the main-loop starts with a call to the *relabeling-algorithm*, which assigns to each vector in Q a minimum and a maximum label (l, u). For both the lower and the upper classifications we calculate the score and average to find the optimal query-point, the new label is added to Q and removed from U. No inference is done while the main-loop runs, since we can't guarantee that the values in (l, u) are the correct labels. Instead the loop continues until either U is empty or a maximum number of iterations has been reached. If there are any vectors $\mathbf{x}_i$ in Q for which $l_i = h_i$ at termination of the main-loop, we assign them their label and infer what labels we can. The algorithm returns the upper and lower bound for all vectors in Q. The key point is that once we find a conflict in Q the relabelling-algorithm will solve this conflict and return the best estimate for the "true" monotone function. Results indicate that this estimate is closer to the "true" monotone function after relabelling than it was before relabelling (see appendix A). Conflicts arise naturally while progressing through U and as Q grows it becomes more likely that mistakes made by the oracle introduce monotonicity violations which the relabelling repairs.

Chapter 5

Analysis and extension of SMAL

5.1 Problem Description

There is a potential problem with the SMAL algorithm which lies in the way the monotonicity violations are handled. Each iteration of the main-loop will pass the partial classifier to the relabelling-algorithm to "re-monotonize" any violations.

As mentioned, the algorithm does not perform any inference during the loop. This is done because the relabelling-algorithm might change any label if the change will result in a classifier with smaller empirical loss. Therefore it can not be guaranteed that any inferred information stays valid across multiple steps of the iteration. If for example we find a vector which gets assigned the smallest label, we might infer that all vectors in its downset necessarily also need to be assigned the lowest label. If the next iteration changes the label of this vector to the largest label, we can no longer make any inference about the vectors in its downset. Hence the choice was made to do no inference until the main loop terminates, after which it is certain the (partial) classifier will no longer change. However, we still compute scores for each query-candidate, and

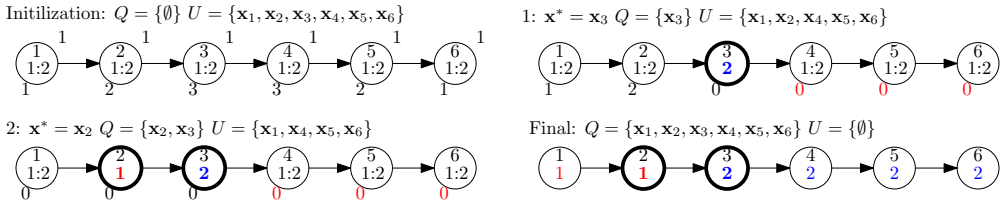

Figure 5.1: A chain of vectors $\mathbf{x}_1 \preceq \mathbf{x}_2 \preceq \cdots \preceq \mathbf{x}_6$. To the upper-right corner the first pane shows the true label. Below the vector index for each vector the bounds can be found. To the lower-left corner of the first 3 panes the score is displayed.

during this computation any label that can be eliminated before calculation will be removed. This is in fact desirable, for if we didn't, the scores would not properly reflect the "information gained" for each query-candidate. It is this implicit inference step that can potentially lead to very poor performance, as we will show in this section. We will start out with a simple example to illustrate the problem and continue to derive a more general description. In the second part we will propose a number of alternative query strategies that remedy the problem and aim to avoid poor performance in worst-case scenarios.

The following example will show a simple case that will provide some insight into why this problem occurs.

Example 5.1. Consider the chain depicted in figure 5.1. Like before, we can skip the first relabelling step and compute the score with equation 4.7. Vectors $\mathbf{x}_3$ and $\mathbf{x}_4$ are tied with 3 eliminated labels, we make the arbitrary choice for $\mathbf{x}^* = \mathbf{x}_3$ and request its label. This time the oracle makes a mistake and instead of the correct label we observe the label 2 for $\mathbf{x}_3$ (indicated in bold and blue). As we are not finished there is no inference step yet and the algorithm computes the scores for the next set of query-candidates. However, the scores for the successors of $\mathbf{x}_3$ are calculated based on the information that their predecessor's label is 2 and all default to zero. This is desired behaviour, since had the label for $\mathbf{x}_3$ been correct, little would have been gained in requesting the label of any of its successors, and we expect the relabelling step to repair this error in some future iteration. If we turn to the working of the relabelling-algorithm, we see that, since its objective is to minimize absolute-loss, it can loosely be said that it aims to change as few labels as possible. Since $\mathbf{x}_3$ received the highest label, the relabelling-algorithm would only consider changing its label if any of its successors had a label that violates monotonicity. The error made by the oracle can thus only be corrected if SMAL chooses any of the vectors in $\uparrow(\mathbf{x}_3)$ as query. Unfortunate they all get a score of zero, because since $\mathbf{x}_3$ has the label 2, the algorithm concludes that no information is gained by querying anything that succeeds it in the order. This means that at termination of the main-loop the algorithm will conclude that $\mathbf{x}_3$ was labelled correctly and it can be safely used to infer labels. The situation at termination can be seen in the final panel: The algorithm returns a classifier which predicts the labels shown in the lower-right corner, only predicting the correct answer one out of three times.

From example 5.1 we can conclude that, clearly, we cannot simply rely on the relabelling-algorithm to catch all errors. If we get unlucky and the error happens to infer the labels of all vectors above or below the query-point the error will never be discovered, in the binary case the

SMAL algorithm effectively reduces to MAL. We continue by investigating ways to remedy this unwanted behaviour. If we can somehow increase the number of monotone functions that are generated by the relabelling-algorithm the chances of finding dissension amongst them increases. We start by looking into the cause of the malfunction in SMAL and look for ways to counter this. We also look to investigate the effect of considering more than just the minimal and maximal monotone functions returned by the relabelling-algorithm.

The effect of this is most clearly seen in the binary case. Assume we have a chain where the true labels of all vectors are 1. The SMAL algorithm will select the vector that is most in the middle for querying and observe its label. If the algorithm observes the label 2 it will infer that the label of all vectors in the upper part of the chain are also 2, effectively mis-classifying over 50% of the vectors after just one query. More importantly, the next query will always be taken from the lower part of the chain, as the vectors in the upper part already all classified. Whatever label we observe next has no influence on the labels in the upper part: If we observe the label 1, we only infer something about the lower part of the chain. If we observe the label 2, all vectors succeeding our query-point are inferred to be 2 but that doesn't change the labels of the previously mis-classified vectors. Effectively by mis-classifying the first vectors we have excluded half of the chain for consideration and inferred the incorrect label for all of them.

At this point we can establish an upper bound on the number of mis-classifications the algorithm makes in the worst-case scenario. For the worst-case we assume that all the labels that we observe from the oracle are the incorrect labels. For simplicities sake we take our order to be a chain of n vectors and let our set of labels Y be $\{1, 2\}$. Each vector either belongs to the *majority class*, if its label is the most common (most occurring) on the chain, and to the *minority class* otherwise. Now let the set $M = \{m_1, m_2, \dots\}$ be the index of the vectors the algorithm chooses to query, that is $\mathbf{x}_{m_1}$ is the first vector the algorithm queries. Since we have a chain we can assume that the algorithm will always query in the middle of the set of unlabeled vectors and due to the problem described above we know that after each query, half of the chain will be dismissed by the algorithm. By definition we know that the labels on the chain are monotone (increasing) and we therefore know that a vector in the middle of the chain should always belong to the *majority class*. We will now show that in the worst-case the number of mis-classified vectors is equal to the number of vectors that belong to the *majority class*. First we know that in the initial step the algorithm will select the vector $\mathbf{x}_{m_1}$ whose true label belongs to the **majority class**, but since we are assuming the worst-case we will observe it as belonging to the **minority class**. The algorithm

will not technically infer any labels until termination of the main loop, however since the information of certain vectors will no longer change we will for simplicity assume that inference is done right after observation. There are now two possible cases: The *minority class* has the lowest label. Or the *minority class* has the highest label. It is easy to see that the only difference between the two is that in the case where the *minority class* has the label 1 the algorithm infers the labels of the vectors in its downset. And visa versa if the label is 2 it infers the vectors in its upset. Going with the first base where the *minority class* has label 1, we thus infer that all vectors in the downset of $\mathbf{x}_{m_1}$ belong to the **minority class**, that is $\forall \mathbf{x}_i \in \downarrow (\mathbf{x}_{m_1}) : y_i = y_{m_1}$. The next step is to show that for a given sub-chain $\{m_i + 1, \ldots, n\}$, that is the chain resulting from removing all vectors in the downset of $\mathbf{m}_i$, we again predict the *minority class* for all vectors in the remainder of $\downarrow (\mathbf{x}_{m_i+1})$. This proof is trivial, as we know that by definition all vectors belong to the *majority class* and that we will therefore observe the *minority class* for any vector in the remaining chain when we query. We find that a given query-point $\mathbf{x}_{m_i+1}$ will result in observing the *minority class* and the inference of the *minority class* for all the vectors in its downset. At termination we either have a chain consisting of one or two vectors. If there is one vector left the algorithm will always observe the *minority class*. If there are two vectors left the algorithm either picks the lowest or the highest vector in the order. In the latter case the algorithm observes the *minority class* and can infer the remainder. In the former case the algorithm requires an additional step as no inference will be done. The above shows that in the worst-case scenario where the algorithm always observes the incorrect label it will predict the *minority class* for every vector. Therefore in the worst-case the algorithm mis-classifies all vectors that actually belong to the *majority class* and we can express an upper bound on the fraction of errors in the solutions as $\frac{|majority|}{n}$ where $|majority|$ is the number of vectors belonging to the *majority class*.

If we look closer at the way the relabelling-algorithm works, we can see that, in fact, some vectors will never get assigned a different label in any relabelling. Errors are repaired by finding the monotone function f that has the smallest loss when compared to the original non-monotone input. For example, in figure 5.1 if all vectors have a true label of $y = 1$ and lets say we observe the correct labels for all vectors but y_3. The solution with the smallest loss would be to simply change the label of the vector $\mathbf{x}_3$ to be $y_3 = 1$. Now consider the case where the labels for all vectors following $\mathbf{x}_3$ are also incorrect that is, we have the labels $(1, 1, 2, 2, 2, 2)$. There are 4 incorrect labels in this labeling, namely the labels for vectors $\mathbf{x}_3$, $\mathbf{x}_4$, $\mathbf{x}_5$ and $\mathbf{x}_6$. However this labeling is a proper monotone function and the relabeling-algorithm terminates, having found a solution with

0 loss. Clearly we need a violation of *monotonicity* to have any chance of correcting mis-labelings.

As the number of *comparable* vectors decreases and we consider problems with more labels the effect of this problem diminishes. Fewer comparable vectors means fewer labels inferred for each observed label which in turn means fewer mis-classifications. More potential labels means that there is a smaller chance that parts of the order are considered "classified", since this only happens when the interval of potential labels for a vector is reduced to only one value. So while this problem is less pervasive in general partial orders with more than two labels, the SMAL algorithm can potentially mis-classify parts of the data if it gets "unlucky". The root of the problem lies in the fact that the relabeling-algorithm will not always help resolving problems under circumstances where the incorrect labeling doesn't contain any monotonicity violations. Therefore our approach will be to add an additional step to the algorithm that will attempt to "repair" potential problems by maximizing the probability of introducing a violation to the current solution.

Firstly consider our chain example in figure 5.1 where all vectors have that $y_{\mathbf{x}} = 1$. We again observe the label 2 for $\mathbf{x}_3$. Relabeling won't have any effect until we have a conflict, and the only way to have a conflict is to either observe the label 2 for a vector from the downset of $\mathbf{x}_3$ or the label 1 for a vector from its upset. Having either would result in a classification that violates monotonicity and will cause the relabeling-algorithm to change the labels of some vectors. In this case, if we would query again and observe the label of $\mathbf{x}_4$ to be 1, the relabeling algorithm finds two optimal relabelings: $y_{\mathbf{x}_3} = 1$ and $y_{\mathbf{x}_4} = 1$. Or $y_{\mathbf{x}_3} = 2$ and $y_{\mathbf{x}_4} = 2$. SMAL would then proceed to use both to calculate the next query-point, picking either $\mathbf{x}_5$ or $\mathbf{x}_2$ as the next query. Our aim is thus to query vectors that will have the best probability of yielding more than 1 relabelling. We will now look further into identifying such vectors.

Assume $Y \in \{1, \ldots, k\}$ and let Q again denote the set of points for which a label has been observed and let $(Q, \preceq)$ be the *poset* made up of these points. We are looking for minimal and maximal elements from $(Q, \preceq)$ for which we have observed the labels 1 and k respectively. Each of these points have the potential of excluding parts of the solution space from consideration by the learner and therefore should be considered for "reparation". To put it differently, we need to query a point such that, if added to Q, is likely to cause a violation of monotonicity. Assume we have some suspect vector $\mathbf{x}_u$ that is a maximal element of $(Q, \preceq)$ and we observe its label to be k. To violate monotonicity we need to observe a label the is a successor to $\mathbf{x}_u$ and has a label smaller than k. This clearly means we need to query a vector that is a member of $\uparrow(\mathbf{x}_u)$. Similarly, let $\mathbf{x}_l$ be a minimal element of $(Q, \preceq)$, we want to observe a vector from $\downarrow(\mathbf{x}_l)$ with a label larger than

1. We define B^+ to be the set of vectors in the upsets of any maximal element of $(Q, \preceq)$, that is:

$$B^+ = \bigcup_{\mathbf{x}_u \in X} [\uparrow (\mathbf{x}_u) \setminus \mathbf{x}_u] \text{ s.t. } \mathbf{x}_u \text{ is a maximal element of } (Q, \preceq). \tag{5.1}$$

The set B^+ contains all elements that succeed any element that lies on the end of a branch of our current ordered set of labelled vectors (Q), see figure 5.2. Note that the maximal element itself is excluded from B^+ as we do not want the learner to re-query this item. Similarly we define B^- to be the set of vectors in the downsets of any minimal element of $(Q, \preceq)$, that is:

$$B^- = \bigcup_{\mathbf{x}_l \in X} [\downarrow (\mathbf{x}_l) \setminus \mathbf{x}_l] \text{ s.t. } \mathbf{x}_l \text{ is a minimal element of } (Q, \preceq). \tag{5.2}$$

In addition we define B to be the union of B^+ and B^-:

$$B = B^+ \cup B^-. \tag{5.3}$$

We can use the *stochastic dominance* assumption in equation 4.9 to find the vector that has the largest probability of causing a conflict. Recall that we have $\mathbf{x}_i \preceq \mathbf{x}_j \implies \forall j : P(y \leq j|\mathbf{x}_i) \geq P(y \leq j|\mathbf{x}_j)$, for two vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. Let $\mathbf{x}_u$ be a maximal element from $(Q, \preceq)$, we can find the vector $\mathbf{x}_c^u$ with the greatest probability of causing a conflict with $\mathbf{x}_u$ by maximizing the cumulative probability of observing a smaller label:

$$\mathbf{x}_c^u = \arg\max_{\mathbf{x}_c \in \uparrow(\mathbf{x}_u)} P(y_c < y_u|\mathbf{x}_c). \tag{5.4}$$

Similarly we can find the vector $\mathbf{x}_c^l$ that maximizes the probability of causing a conflict with a minimal element $\mathbf{x}_l$ in $(Q, \preceq)$:

$$\mathbf{x}_c^l = \arg\max_{\mathbf{x}_c \in \downarrow(\mathbf{x}_l)} P(y_l < y_c|\mathbf{x}_c). \tag{5.5}$$

To find the optimal conflict vector $\mathbf{x}_c^*$ the learner needs to compute which of the vectors in B has the greatest probability of causing a conflict with any of the minimal and maximal elements from $(Q, \preceq)$. Let MU be the set of all maximal elements in $(Q, \preceq)$ and let ML be the set of all minimal elements in $(Q, \preceq)$. Then $\mathbf{x}_c^*$ is an element from B that maximizes either 5.4 or 5.5 over

all elements of MU and ML:

$$\mathbf{x}_c^* = \arg\max_{\mathbf{x}_c \in B} \left[\max_{\mathbf{x}_u \in MU} P(y_c < y_u | \mathbf{x}_c), \max_{\mathbf{x}_l \in ML} P(y_l < y_c | \mathbf{x}_c) \right]. \tag{5.6}$$

Unfortunately there is no way to explicitly compute the probabilities in this expression. However we can theorize that as the cumulative distribution of a given vector is always dominated by its successor (by definition of stochastic dominance), the vector for which we have the best chance to observe a label that is smaller than that of its predecessor should be its direct successor. That is, if we have a vector $\mathbf{x}_i$ and its direct successor $\mathbf{x}_j$ than querying the label of $\mathbf{x}_j$ gives us the best chance of observing a label that is smaller than that of $\mathbf{x}_i$. If our order $(X, \preceq)$ is a chain, we can use the stochastic order constraints to find a $\mathbf{x}_u^*$ and a $\mathbf{x}_l^*$, the optimal conflict vectors for the maximal and the minimal element respectively: Assume we have the maximal element of $(Q, \preceq)$ $\mathbf{x}_u$ with observed label $y_u = k$. B^+ will contain the upset of our maximal element, that is $\uparrow(\mathbf{x}_u) \subset B^+$, and we know our query-point must come from there. For each pair of vectors $\mathbf{x}_i, \mathbf{x}_j \in \uparrow(\mathbf{x}_u)$ we know from equation 4.9 that if $\mathbf{x}_i < \mathbf{x}_j$ then $P(y_i \leq l|\mathbf{x}_i) \leq P(y_j \leq l|\mathbf{x}_j) \forall l \in \{1, k\}$ and therefore $P(y_i \leq y_u|\mathbf{x}_i) \leq P(y_j \leq y_u|\mathbf{x}_j)$. Clearly this means that if we want to maximize the probability of generating a conflict we should query the vector that is a direct successor of $\mathbf{x}_u$, since stochastic dominance tells us that the probability of finding a smaller label can only decrease (or at best stay the same) if we move toward more distant successors of $\mathbf{x}_u$. The element of X that is a direct successor of $\mathbf{x}_u$ is the smallest element from its upset, excluding $\mathbf{x}_u$ itself, that is: if we want to observe a label $y_c < y_u$ we should pick the vector $\mathbf{x}_c$ that is the smallest element of $\uparrow(\mathbf{x}_u) \setminus \mathbf{x}_u$. Note that $\mathbf{x}_c$ is a member of the set B^+. A similar argument can be made for the minimal element of $(Q, \preceq)$ $\mathbf{x}_l$ with observed label $y_l = 1$, where we want to query a vector that is direct predecessor of $\mathbf{x}_l$.

In summary, if we want to maximize the probability of generating a conflict in our partial classifier we should pick either a predecessor of a minimal element of $(Q, \preceq)$, or a successor of any of its maximal elements. The element with the highest probability of causing a conflict will be either the **direct** predecessor of the minimal element $(Q, \preceq)$, or the **direct** successor of the maximal element of $(Q, \preceq)$. There is currently no theoretical basis on how to proceed, since we cannot explicitly compute the cumulative distributions for the observed labels. In addition, both $\mathbf{x}_u^*$ and $\mathbf{x}_l^*$ need not be unique so there may be multiple optimal solutions. We consider a number of query-strategies that use different approaches for estimating the best vectors in B to query. We aim to devise strategies that maximize the probability of discovering any mis-classified vectors

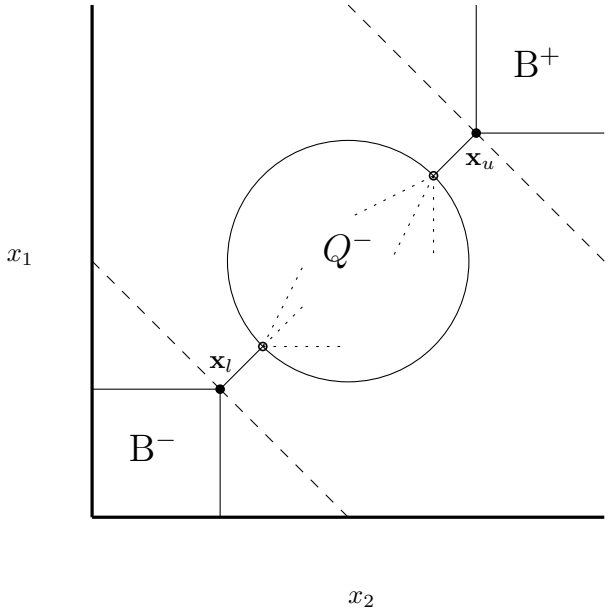

Figure 5.2: Illustration of the boundary zone. The axis represent the two attributes x_1 and x_2. The maximal and the minimal vectors all lie on the dashed diagonal line, the dots represent two such vectors $\mathbf{x}_l$ and $\mathbf{x}_u$. The circle in the centre marked Q^- is the set Q without its minimal and maximal elements.

while respecting the active learning tenet of minimizing the number of queries.

Our new algorithm SMAL+ will introduce an additional intermediate step that will apply the alternative query-strategies if some "repair condition" is met. Additionally we propose a number of different approaches for combining the number of optimal monotone functions that are generated by the relabelling-algorithm.

SMAL+ is supplied a list of unlabelled vectors U, a list of vectors that have been observed so far Q and a maximum number of iterations. The algorithm listed in 1 has a similar structure to the SMAL algorithm, but there are two major differences: In lines 5-12 denoted by R are all optimal monotone functions. The score H is calculated over all monotone functions. On line 3-4 we apply the new query-strategies if the repair condition is met. This condition will depend on the set Q of labels observed so far. If the condition isn't met, the original strategy is used in line 12. The algorithm returns an interval of possible labels (l, u) for each vector. Example 5.2 illustrates the new algorithm.

Example 5.2. In figure 5.3 we have a partial ordering on 8 points. Assume binary labels $\{1, 2\}$ and $Q = \{(\mathbf{x}_3, 1)\}$ after one iteration. For this example we set all true labels to be 1. Say SMAL+ chooses $x^* = x_5$ in its second iteration and observes the (incorrect) label $y_5 = 2$. We now have $Q = \{(\mathbf{x}_3, 1), (\mathbf{x}_5, 2)\}$. Lets say that in the third iteration the "repair condition" is met and the

Algorithm 1 SMAL+(Q,U,max)

while $max > 0 \wedge U \neq \emptyset$ **do**
  $R \leftarrow Relabel(Q)$
  if Repair condition **then**
    $x^* \leftarrow QueryStrategy(Q, U)$
  else
    for all $x \in \mathcal{U}$ **do**
      for all $r \in R$ **do**
        $H_r(x) \leftarrow$ query value of x using r
      end for
    end for
    $H(x) \leftarrow \sum_{r \in R} (H_r(x))/|R|$
    $x^* \leftarrow \arg\max_{x \in \mathcal{U}}(H(x))$
  end if
  $y^* \leftarrow \mathcal{O}(x^*)$
end while
$(l, u) \leftarrow Relabel(Q)$
return (l, u)

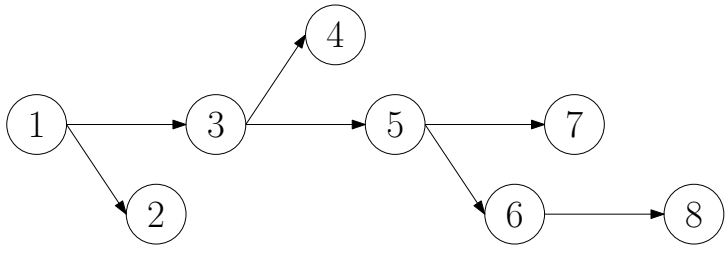

Figure 5.3: Partial order example

algorithm takes the alternative route trying to introduce a monotonicity violation to $(Q, \preceq)$. Our boundary set would consist of the union of the sets containing the vectors below $\mathbf{x}_2$ and above $\mathbf{x}_5$, that is $B = \{\mathbf{x}_1, \mathbf{x}_6, \mathbf{x}_7, \mathbf{x}_8\}$. We know from *stochastic dominance* that vectors near the edge of a *poset* are most likely to be observed having the minimum or maximum labels, so it makes sense to pick either a successor of $\mathbf{x}_5$ or a predecessor of $\mathbf{x}_2$ as our best bet to introduce violations. However, as we do not know the distribution used by the oracle we can only guess at which holds the optimal point. We can however make some observations that may prove helpful in designing the query heuristics; One could argue that while we do not know the distribution for the observed labels we can say something about their relative structure. Since the distributions of vectors near the edges tend to "collapse" more toward the minimum or maximum label, we would want to query a point that is nearest to the centre as its distribution would be more spread out as it were. In this case, querying vector $\mathbf{x}_1$ would likely result observing the label $y_1 = 1$ and not introduce a violations. On the other hand the observed label for $\mathbf{x}_5$ is more likely to be more equally distributed across the spectrum. In conclusion querying farther from the edges has more potential of creating the violations we are looking for. Another point to consider is that, since the main damage done by mislabelled vectors is done when we use their information to infer more labels. It would therefore be more detrimental if we observe the incorrect label for a vector that infers the labels of many vectors, as opposed to observing the label of a vector that infers but few. From this we can conclude that we should look more closely at the labels of vectors that can potentially do a lot of damage to our classifier. Finally we note that the objective of active learning is to learn a classifier faster than a regular learning algorithm. It should therefore also be an objective of the heuristics not to waste too many queries on reducing the risk of errors. While querying repeatedly near the same location in our order might improve our confidence in that region, it also costs precious queries, which may cause the learner to neglect other parts.

5.2 Query strategies

The observations made during example 5.2 seem to indicate that we need to focus our querying on the minimal and maximal vectors that have the largest up- and downsets. We propose a number of query-strategies that use this observation and we also define a random-pick strategy to serve as a base-line for testing.

5.2.1 Random pick

To establish a baseline for comparison the learner selects an arbitrary vector every time the repair condition is met. The learner picks from the set $X \setminus Q$, that is the vectors that are not yet labelled, and observes its label. Note that this strategy uses no additional information and has an equal probability of selecting any unlabelled vector as x^*. Our query-strategies should at least perform better than random pick.

$$\mathbf{x}_r^* = \text{sample}(X \setminus Q), \tag{5.7}$$

where $sample(x)$ is a function that randomly selects a vector from set x according to a uniform distribution.

5.2.2 Conservative approach

As observed in example 5.2, our best chance of finding a conflict is by querying the direct neighbour of the minimal and maximal elements of Q. For this approach we weigh the severity of an incorrectly observed vector by the number of vectors whose label it infers. As such a vector that has a large upset (downset) posses a greater risk, as it might potentially predict an incorrect label for all of its successors (predecessor). Our conservative (optimal) query point $\mathbf{x}_{con}^*$ is given by the vector in B that has the largest up- or downset:

$$\mathbf{x}_{con}^* = \arg\max \left[\max_{\mathbf{x}^+ \in B^+} u(\mathbf{x}^+), \max_{\mathbf{x}^- \in B^-} d(\mathbf{x}^-) \right]. \tag{5.8}$$

Recall that $u(\mathbf{x})$ indicates the size of the upset for vector $\mathbf{x}$ and similarly $d(\mathbf{x})$ the size of the downset. This vector should have a decent chance of causing a conflict and with that a monotonicity violation and also, in case it turns out to repair an error, provide the biggest pay-out of recovered labels. This strategy aims to maximize the number of conflicts and hence the number of monotone functions generated by the relabelling-algorithm at the cost of expending queries. The expected results are that the algorithm learns a classifier that predicts a smaller number of labels than SMAL, but also makes fewer errors.

5.2.3 Midway approach

Since queries are expensive, we do not want to spend too much time recovering errors that might not even be there. Querying the direct neighbour of the minimal or maximal element adds no information to the solution: In the case we find a label that causes no conflicts it will be either lower or higher, which in the best case only tightens the bounds of some points. In the case that we do find a conflict, we actually throw away some information (granted that this might have been incorrect information). It stands to reason that we should pick a point from a branch in our order that, in worst case, would add the most information to the solution. As this is exactly the vector that the SMAL algorithm would have chosen if $X = B$ we have:

$$\mathbf{x}^*_{mid} = \max \left[\underset{\mathbf{x}_i^+ \in B^+}{\arg\max} \min\{d(\mathbf{x}_i^+), u(\mathbf{x}_i^+)\}, \underset{\mathbf{x}_i^- \in B^-}{\arg\max} \min\{d(\mathbf{x}_i^-), u(\mathbf{x}_i^-)\} \right]. \tag{5.9}$$

That is, we select the vector that would be best to query if we only consider the vectors in B. If the vector we select does create a conflict we have repaired a potential error in the solution. If we do not find a conflict the learner will always add some information to the solution by selecting an optimal query point for the set B. This strategy should yield intermediate results: Intuitively selecting a query-point that is further from the minimal and maximal element should reduce the probability of getting a conflict. However, the selected query-point should perform better if the minimal and maximal elements had their correct labels observed.

5.2.4 Random boundary pick

In order to investigate whether the additional calculations done by the heuristics in equations 5.8 and 5.9 are justifiable, *random boundary pick* picks a random vector from the boundary-set. The strategy has a similar structure to the heuristic in equation 5.7, but instead only samples B:

$$\mathbf{x}^*_{rb} = \text{sample}(B). \tag{5.10}$$

By picking from the boundary-set the strategy guarantees that the algorithm has a chance of visiting regions that the original SMAL algorithm does not. The resulting classifier should exhibit less extreme cases than the SMAL-algorithm, where a large number of mis-classifications are made. Performance is however expected to be poorer than the heuristics in equations 5.8 and 5.9 as he randomly selected vectors can be both bad for efficiency (as was approached in the midway

heuristic) and error prevention (which the conservative heuristic optimizes).

5.3 Score Functions

Additionally, since we aim to find conflicts in the (optimal) monotone functions returned by the relabelling-algorithm we investigate the effect of considering more than just the minimal and maximal relabellings. We aim to see whether it's cost-effective to include multiple optimal monotone functions in the calculation of vectors score with the heuristic in equation 4.7.

5.3.1 Min/Max optimal monotone functions

The original SMAL algorithm considers only the minimal and the maximal optimal monotone functions by assigning the label of each vector in Q the minimum and the maximum value of the intervals returned by the relabelling-algorithm. For each vector the the final score is the average of the results of equation 4.7 using the minimal and maximal assignment.

$$H_m m(\mathbf{x}_i) = H_u(\mathbf{x}_i) + H_l(\mathbf{x}_i), \tag{5.11}$$

where H_u is the score with the maximal assignment of labels and H_l the score with the minimal assignment of labels.

5.3.2 Average optimal monotone functions

The relabelling-algorithm calculates all optimal monotone functions before determining the intervals and we can use this information in our algorithm at no additional cost. The score resulting from equation 4.7 is calculated for each optimal monotone function and the results are averaged for each vector. If the distribution of labels assigned by each function is not uniform (some function might assign the same label to some vectors), the average scores should be different from the original approach. For the average H_a we have:

$$H_a(\mathbf{x}_i) = \sum_{f \in \mathcal{F}_o} [H_f(\mathbf{x}_i)]/|\mathcal{F}|, \tag{5.12}$$

where H_f is the score with function f and $\mathcal{F}_o$ is the set of all optimal monotone functions. As the relabelling-algorithm can be shown to approach the true function this average should give a better indication of the true label of the vectors. This should lead to better picks when x^* is determined.

5.3.3 Weighted monotone functions

In addition to all optimal functions, the relabelling-algorithm also computes functions that have losses that are greater than optimal. We consider the effect of using a weighted average of scores and investigate its effect. Each function has a weight factor w_f that is inversely proportional to the distance to the optimal solution d_i, where distance is expressed as the absolute error: $w_f = \frac{1}{d_i}$. The average score H_w is then given by:

$$H_w(\mathbf{x}_i) = \sum_{f \in \mathcal{F}} [w_f H_f(\mathbf{x}_i)]/|\mathcal{F}|, \tag{5.13}$$

where H_f is the score with function f and $\mathcal{F}$ is the set of all monotone functions. While the non-optimal solutions might steer the average away from the true label, we do increase the number of functions and therefore the potential of finding conflicts. As we have more conflicts this approach should yield less errors, but as there is more dissension among the functions there might also be fewer intervals that converge.

Chapter 6

Experiments

In this section we report on the evaluation of our proposed query-strategies by performing experiments on artificial data sets. Each of the four query-strategies that focus on error prevention (equations 5.7, 5.8, 5.9 and 5.10) was implemented with each of the three alternative score heuristics (equations 5.11, 5.12 and 5.13). The data sets are generated from a d-dimensional normal distribution, orders will be decided based on relative value of the d attributes. To simulate the stochastic oracle we generate a random monotone function using the Propp-Wilson algorithm (see Soons [10]) and randomly change the labels of some of the vectors. The randomly generated base-function will also provide the true labels that are needed for the analyses of the performance. Each dataset has its own ordering and therefore also need its own oracle. To eliminate potentially favorable or unfavorable assignments of true labels a new monotone function is generated for each run and hence also a new oracle. In addition we add a parameter e to the SMAL+ algorithm that controls the fraction of mislabeled vectors that the oracle returns on average. We did 100 runs with a data set of 100 vectors and averaged the results for each of the twelve combinations. During each run we store a the value of performance measure after each iteration, the final results can then be plotted against the number of iterations, giving an insight into the solution progression over time.

6.1 Artificial data

The experiment consist of the proposed 100 runs of the the possible query-strategy and score heuristics on 5 different datasets. As the number of attributes will only influence the number of comparable vectors in the dataset we limit our selves to only a bi-normal distribution (that is,

Dataset	Comparability	n.o. labels	n.o. datapoints
Chain	1	2	100
High	0.85	2	100
Middle	0.5	2	100
Low	0.14	2	100
Chain 3	1	3	100

Table 6.1: Summary of the dataset used in the experiments. The left column marked *comparability* shows the fraction of vectors in the dataset that are comparable. The columns marked $\#k$ and *#datapoints* denote respectively the number of possible labels in K and the number of datapoints in the datasets n

$d = 2$) to save some time. Each set of 100 datapoints is drawn i.d.d from a normal distribution with a correlation of c which controls the comparability of the vectors. For the first 4 datasets we choose covariance values of $c = \{1, 0.9, 0, -0.9\}$ resulting in comparabilities as show in table **??**, for convenience we named these datasets *chain*, *High*, *Middle* and *Low* for obvious reasons. The final set named *Chain3* has the same comparability as the set *chain* but will feature an oracle that can assign three different labels, instead of two. The order is determined based on the relative values of the attributes where we use that $\mathbf{x}_i$ precedes $\mathbf{x}_j$ iff all attributes (x_i^1, x_i^2) are smaller or equal to (x_j^1, x_j^2), which also satisfies the monotonicity assumption in equation 2.2. The data is stored as a 0/1-matrix (i, j), where the relative position of the vectors is stored as a 1 if $i < j$ and as a 0 if $i > j$. For each ordering we generate m different monotone functions f_i, one for each run of the algorithm. The naive approach would be to select $k - 1$ vectors and assign to each of the segments one of the labels in such a way that the result is a monotone functions. Unfortunately this approach will not yield a uniform sample, so to ensure this we use an adaptation of the Propp-Wilson algorithm for monotone functions (see Soons [10]). This algorithm ensures we get a uniform random sample from the entire distribution of valid monotone functions.

6.2 Oracle

The oracle will be simulated by a classifier that is generated beforehand, the oracle will have chance to mislabel a vector with a rate of e. This classifier should have two important properties: It should match real-world expert behaviour, in that it must make mistakes in a believable manner. The expert should at least perform better than a random draw from the available labels, we also believe that an expert should make more severe mistakes less often than small ones. In addition for our strategies to be valid in the stochastic framework, it is should also be a function that satisfies the stochastic order constraint in equation 4.9. To this end a randomly selected

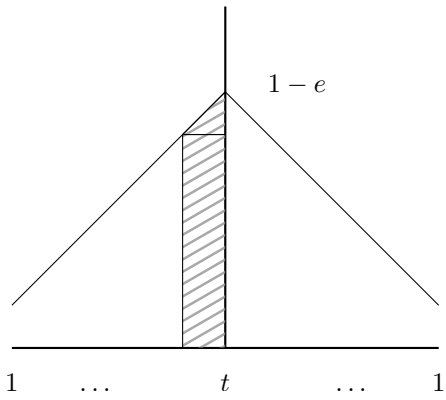

Figure 6.1: An example of a tent function centred around true label t. The area underneath the tent should sum to 1 and each label has an associated value that is great than zero, so in fact the tent describes a probability distribution. The shaded area represent the fraction of the error that is assigned to the label $t-1$, this is the a in equations 6.1, 6.3 and 6.4

monotone function that fits the data order is generated and altered by some *perturbing function*. The monotone function will serve as a *basis* giving the oracle a monotone "shape" as it were. The *perturbing function* should have both the stated properties to make the oracle match expert behaviour while maintaining the stochastic order constraint. We choose a tent-function (see figure 6.1) that resembles absolute loss to serve as our *perturbing function*. The tent-shape nicely captures the required real-world behaviour, were graver errors are less common, and has some useful mathematical properties. By combining this function with the basis we can determine the entire probability distribution $P(y|x)$, we will drop the second parameter if the meaning is clear. Say we have an error rate e, that is the oracle returns the *correct* answer $(1-e)*100\%$ of the time. For some vector $\mathbf{x}$ let t denote the true label assigned by the basis and let L denote the set of all labels except the true label. The oracle should yield the true label $1-e*100\%$ of the time, so the remaining $e*100\%$ the oracle should yield on of the labels from L, in such a way that label that are farther away from t are less likely. If we define a to be some constant whose unit is $\frac{error}{distance}$ we can express the probability of observing label $y=l$ as:

$$P(y=l)=\frac{a}{d_l}, \tag{6.1}$$

where d_l is the distance between label l and the true label t. Note that we have

$$\sum_{l\in L}[P(y=l)]+P(y=t)=1, \tag{6.2}$$

as each label must have a probability greater than zero of being selected. We find that since $P(y = t) = 1 - e$ by definition, $\sum_{l \in L} P(y = l) = e$. We can assume without any loss of generality that our labels range from 1 to k, so the sum in equation 6.1 can be split into two intervals: $1 \dots t-1$ and $t+1 \dots k$. If we measure the distance d_l as the absolute distance relative to the true label, that is $d_l = |t - l|$ we find that:

$$e = a(\sum_{i=1}^{t-1} \frac{1}{i} + \sum_{i=1}^{k-t} \frac{1}{i}). \tag{6.3}$$

Note that since we changed to absolute distance relative to t the intervals are now also relative to t. Solving this equation for a and plugging the result back into equation 6.1 gives us an expression that determines the fraction of the error that should be assigned to label l to get a perturbing function of the required shape. Equation 6.3 can be solved by noting that the sums are in fact *Harmonic functions* and can be replaced by their proper *Harmonic number* H_n. The sum $\sum_{i=1}^{n} \frac{1}{i}$ can be replaced by the nth *Harmonic number* H_n, since we have already rewritten the intervals to accommodate, this yields: $e = a(H_{t-1} + H_{k-t})$, solving for a and plugging the result into equation 6.1 we find:

$$P(y = l) = \frac{e}{H_{t-1} + H_{k-t}} \times \frac{1}{|t - l|}. \tag{6.4}$$

The oracle can now be simulated by generating a monotone function that assigns to each vector $\mathbf{x}$ its true label $t_{\mathbf{x}}$ and returning a label from $1, \dots, t, \dots, k$ with the probability $P(y = 1), \dots, (1 - e), \dots, P(y = k)$. To prove that the oracle satisfies the stochastic order constraints in equation 4.9 we need to show that the cumulative distribution of a vector dominates the distribution of its predecessor. The probability distribution for the observed label of a vector $\mathbf{x}$ is made up of two parts, the "true" label assigned by the basis function and the "stochastic" part due to the tent-function. Consider that $P(y = t) \geq P(y = l)$ for any $l \in L$ by definition and that for two vectors $\mathbf{x}_i \preceq \mathbf{x}_j$ we have $y_i \leq y_j$ by choice of basis function. For two vectors $\mathbf{x}_i \preceq \mathbf{x}_j$ we have that $P(y = t_i) \leq P(y = t_j)$ and we know that the remainder of the probability distribution is equal, since they are build from the same function. Therefore if a vector succeeds another vector in the order, its cumulative distribution will always dominate that of its predecessor. It follows that $P(y' \leq l|\mathbf{x}') \leq P(y \leq l|\mathbf{x})$ for all $l \in \{1, k\}$ for any $\mathbf{x}' \preceq \mathbf{x}$.

6.3 Performance measures

To determine the performance of the different strategies we define a *precision* and a *recall* measure. A good solution has two important properties, namely it eliminates as many labels as possible and it makes as few mistakes as possible. To this end we compute the fraction of labels that were eliminated from the intervals as recall and the fraction of errors that were made as precision. We use the F-score to combine both of these measures.

The *recall* will measure the quantitative aspect of the solution, a better classifier should eliminate more labels. We define this measure as a fraction of the total number of available labels, that is the number of vectors in the solution times the number of labels in the intervals at the start of the algorithm:

$$recall = \frac{n_{elim}}{m \times (k-1)}, \tag{6.5}$$

where n_{elim} denotes the number of labels that are eliminated by the classifier, that is all the labels that fall outside the predicted intervals. m denotes the number of vectors in the solution (always 100 in our case) and k the number of labels available. The *precision* will measure the qualitative aspect of the solution, a classifier that makes fewer mistakes should score better than one that makes more mistakes. We define this measure as the relative number of errors that the classifier makes:

$$precision = \frac{n_{correct}}{m}, \tag{6.6}$$

where $n_{correct}$ denotes the number of vectors for which the classifier predicted the correct label. Note that we only count vectors for which an unique label is predicted by the classifier. If the predicted interval has a size larger than 1 the vector is considered as predicted uncorrectly.

We can combine both measure using the F_β-score, which is defined as:

$$F_\beta = \frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision + recall}. \tag{6.7}$$

Here β is a constant that allows for increasing the relative importance of either the precision ($\beta > 1$) or the recall ($\beta < 1$). We will use $\beta = 1$ so that both measure contribute equally to the score. Combining equations 6.5, 6.6 and 6.7 we get the following expression:

$$Q_{score} = \frac{1}{m} \times \frac{n_{cor} n_{elim}}{(k-1)n_{cor} + n_{elim}}. \tag{6.8}$$

This score should give high values when both the (relative) number of predicted labels is high and the number of (relative) errors is low. When we make few errors, but also predict eliminate few labels the score is low. The same is true when many labels are eliminated and the number of errors is high. The Q_{score} for a *quick* strategy would converge faster to 1, since its solution would learn the labels faster (making the recall in equation 6.5 go to 1 faster). An *accurate* strategy would aim to avoid errors to maximize precision (equation 6.6, making the Q_{score} converge slower (since fewer labels are learned), but to a higher maximum (since fewer errors are made). The *perfect* strategy would be both *quick* and *accurate* converging quickly to the highest maximum.

6.4 Execution

We gather and analyse results after each iteration of the main loop. This should give a picture of the development of the solution as time progresses, since we get data after each query. The main loop is run until each vector has been queried at least once, due to the nature of the selection mechanism this should be after m iterations. All data is generated beforehand so all strategies can use identical data, eliminating bias. All ties are processed in the following manner: if two vectors score the same during some step of the algorithm, they are put into an increasing order of which the first element will be selected. This ensure that no query-strategy gains an unfair advantage getting lucky with the selection of the next query-point.

We expect the quality scores to converge well before the maximum number of iterations is reached. Due to the experimental setup the learner will keep requesting new labels until the classifier has learned the label for all vectors. This would mean the *recall* will always go to 1 as more and more labels are either observed or inferred. The *precision* on the other hand should on average converge to $1 - e$, if all vectors would be queried at least once. However due to the nature of the problem we set out to investigate some vectors will never be considered (see section 5.1). The proposed query-strategies are expected to converge slower than the SMAL algorithm since they use some iterations on query points that add little new information to the solution. However we would expect our alternative query-strategies to exhibit a lower variance, as they aim to eliminate outliers caused by the misclassification of vectors.

For each of the 100 orders we sample a random monotone function using the algorithm in section 8.5 that will be used to represent the true labels. For each order three oracles will be generated from the set of true labels, one for each setting of parameter $e = 10\% and 20\%$. Each of the twelve algorithms will be trained once on each of the orders allowing for the comparison of the performance on each of the individual sets as well as an averaged performance.

Chapter 7

Results and Discussion

7.1 Results

In this section we will present the results of the experiment run on the 5 different datasets from table **??**.Tables 8.1 and 8.2 show the average and minimum Q-scores for and error-rate of respectively 10% and 20%. Immediately apparent is the relation between the comparability of the vectors in the dataset and the quality of the solution as was also observed by Barile and Feelders in [1]. As expected all alternative heuristics show worse performance in the earlier stages of the runs, after only 5% of the vectors is queried. When more vectors get queried we see that for highly comparable datasets the *cons* and *mid* heuristics both outperform the original strategy. For datasets with a lower fraction of comparable vectors the SMAL algorithm wins out every time. The four right-most columns contain the worst performing run for each heuristic and should give some insight into the worst-case behaviour of the query-strategies. They paint a similar picture as the average Q-scores; For datasets where a large portion of the vectors are comparable the heuristics show better worst-case performance.

The graphs in figure 7.1 shows the precision and recall values for the different heuristics plotted against the time progression of the algorithm for the chain dataset. The recall graphs clearly show how all new heuristics take more time to "get upto speed" than the original strategy. This effect becomes less pronounced when the number of comparable vectors decreases and we see that, in the end, all strategies manage to infer the labels of approximately 95% of the vectors. The precision graphs, somewhat surprisingly, show a similar behaviour to the recall graphs. The aim of the new heuristics was to find solutions with less errors, and indeed this seems to be the case for the *Chain*

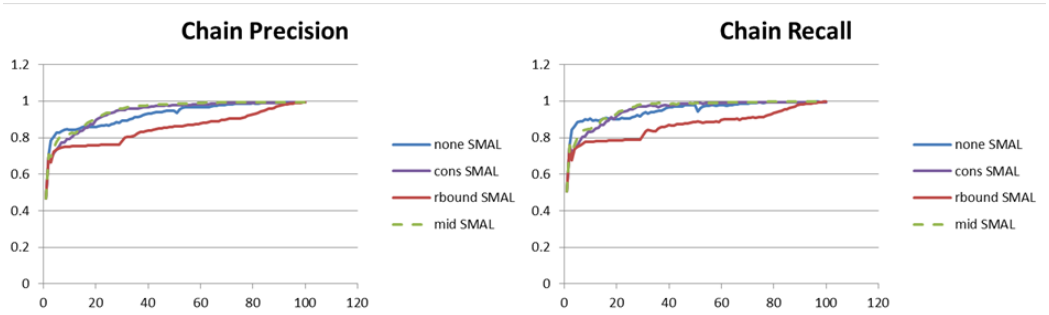

Figure 7.1: Precision and Recall graphs when using an oracle with an error-rate of 10%. The graphs show the results when using the original scoring function with each of the 4 heuristics (including the original SMAL heuristic), on the *chain* dataset.

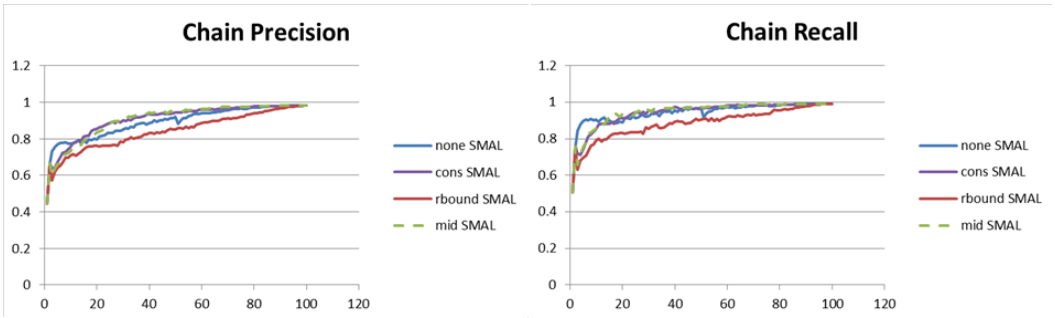

Figure 7.2: Precision and Recall graphs, this time when using an oracle with an error-rate of 20%. The graphs show the results when using the original scoring function with each of the 4 heuristics (including the original SMAL heuristic), on the *chain* dataset.

dataset. However for the other datasets the precision graph seems to closely follow the course of the recall graph. Additional graphs can be found in the appendix. The effects of the new heuristics for the datasets with fewer comparable vectors (*Low*, *Mid* and *High*) are less pronounced and in most cases the new approaches do worse than the original strategy.

The graphs showing results for higher error rates (figure 7.2) clearly exhibit worse performance at the start of the algorithm. However both recall and precision tend to approach the same maximum after a decent number of queries has been performed. As expected the datasets with higher comparability tend to converge to their maximum values after fewer queries than is the case for datasets with lower numbers of comparable vectors. Additional graphs can be found in the appendix (figures 8.1, 8.2, 8.3 and 8.4). For the *Low* datasets we see convergence only after about 80 queries, that is after 80% of the data has been queried. For the *Mid* datasets the graphs reach their maximum at around 60 queries, whereas the *High* and *Chain* datasets peak after 40 and 20 queries respectively.

7.2 Conclusions

The proposed heuristics were expected to perform at least as well as the SMAL algorithm, especially for cases with a higher oracle error-rate. All heuristics were designed with the general idea in mind that by sometimes picking a non optimal vector to query, the number of errors can be decreased. By sacrificing some performance in the early stages of the algorithm's run the average quality of the solution is be increased. Indeed for the *Chain* dataset we see that after an initial setup two of the three heuristics give better results than the SMAL algorithm. Tables8.1 and 8.2 clearly show that the heuristics *Cons* and *Mid* both outperform the original algorithm both in general as well as in worst-case behavior. It must be noted however that the performance of the new heuristics quickly drops off as the number of comparable vectors decreases. On only one occasion does the *Mid* algorithm edge out the original strategy and only after about half of the available vectors have been processed by the oracle. We expect that this behavior stems from the root of the problem: The problem of extremely bad performance only really occurs when a mis-classification is made of a vector that infers the labels of a large number of additional vectors (namely one near the center of the ordering). Hence the problem is expected to occur more frequently on orders that include a large number of comparable vectors (say a chain). In addition the number of relabellings we find through the relabelling-algorithm in the two-label case can be expected to be quite low as it usually only involves flipping the label of one or two vectors. This point is further reinforced by the absence of influence of changing the score-function: There was little difference in using only the minimum and maximum labels compared to using all available monotone functions. This is either caused by having very few possible relabellings, as the choices of changes to make for the relabeling-algorithm are limited. Or it could be that the former approach gives a good estimate of the entire distribution of proper monotone functions. While the latter would be convenient, the lack of options to form different functions for the relabeling-algorithm suggests we are simply dealing with just a small number of monotone functions.

Combined these two properties make it that our approach is less suited for orders that have fewer comparable vectors. This however need not be detrimental to the heuristics, as the problem we set out to explore is most heavily felt when the number of comparable vectors is high. We have shown that the new heuristics lead, on average, to better performance on a dataset with a high number of comparable vectors and can therefore be assumed to avoid worst-case behavior more often than the SMAL algorithm.

Unfortunately our approach is not able to provide any guarantees on the outcome of the

algorithm. Ideally one would like to be able to guarantee the end-user that the solution that is found is always usable and near optimal in most cases. However even though the improvements it is still possible that the algorithm provides a solution that mis-classifies significantly more vectors than simply labeling all vectors by hand. At this point it is questionable if a solution with such a guarantee can ever be found as, due to the stochastic nature of the problem, one can sometimes arrive quite far from the optimal solution, simply due to a streak of bad luck.

To further investigate the problem it might be worth it to look into specifically crafted datasets that aim to cause problems for the original approach. By analysing the algorithms behaviour in these cases it might be possible to identify solutions that are on the wrong track and simply restart the entire process if such is the case.

Another point of interest is the way the algorithm decides when it needs to employ one of the alternative approaches. The focus of this thesis has been on designing and testing new heuristics to approach the problem and the decision whether to employ these has been made in a straightforward manner. It might for example be interesting to examine the algorithm's performance when we only employ the alternative query strategies when a large number of vectors have just had their labels inferred the last step. Such a step contributes large amounts of information to the solution and can therefore also contribute large errors if the initially observed label was incorrect.

Chapter 8

Appendix

8.1 Graphs and Tables

This section contains detailed results of all experiments run while testing the different heuristics. The graphs are ordered in such a way that each "column" represents a single scoring function. Comparing the different graphs does not yield any significant difference in performance between the different scoring-functions.

Out of the 32 different columns in tables 8.1 and 8.2, there are 26 cases where the SMAL algorithm performs best (see bold entries). In $5\frac{1}{2}$ cases the *mid* heuristic performs best and in $\frac{1}{2}$ cases (due to a tie) the *cons* heuristic wins. The *rbound* is never the best performing heuristic, and in the precision and recall graphs this heuristic is always dominated by the other heuristics. For an oracle which mis-classifies 10% of the queries, the new heuristics perform best on the *Chain* dataset, at 20% and 50% of the dataset queried, both *Cons* and *Mid*, show higher average scores than the original. At 20% *Mid* averages scores of $0.919, 0.920$ and 0.922 for the three different scoring functions, whereas the original only achieves averages of $0.878, 0.878$ and 0.878. *Cons* performs slightly worse than *Mid*, but still outperforms SMAL, with scores of $0.908, 0.916$ and 0.916. At 50% we see a similar trend, the *Mid* averages the best scores with $0.988, 0.988$ and 0.986, over $0.962, 0.962$ and 0.962 for SMAL. *Cons* again performs slightly worse with scores of $0.983, 0.977$ and 0.982. For datasets with fewer comparable vectors this performance quickly degrades and we see that the SMAL algorithm is only beaten once in the case of the *High* dataset at 50% of the available vectors queried with *Mid* scoring $0.974, 0.974$ and 0.973 over $0.973, 0.973$ and 0.973 for SMAL. These results are echoed in the results of the experiments done with an

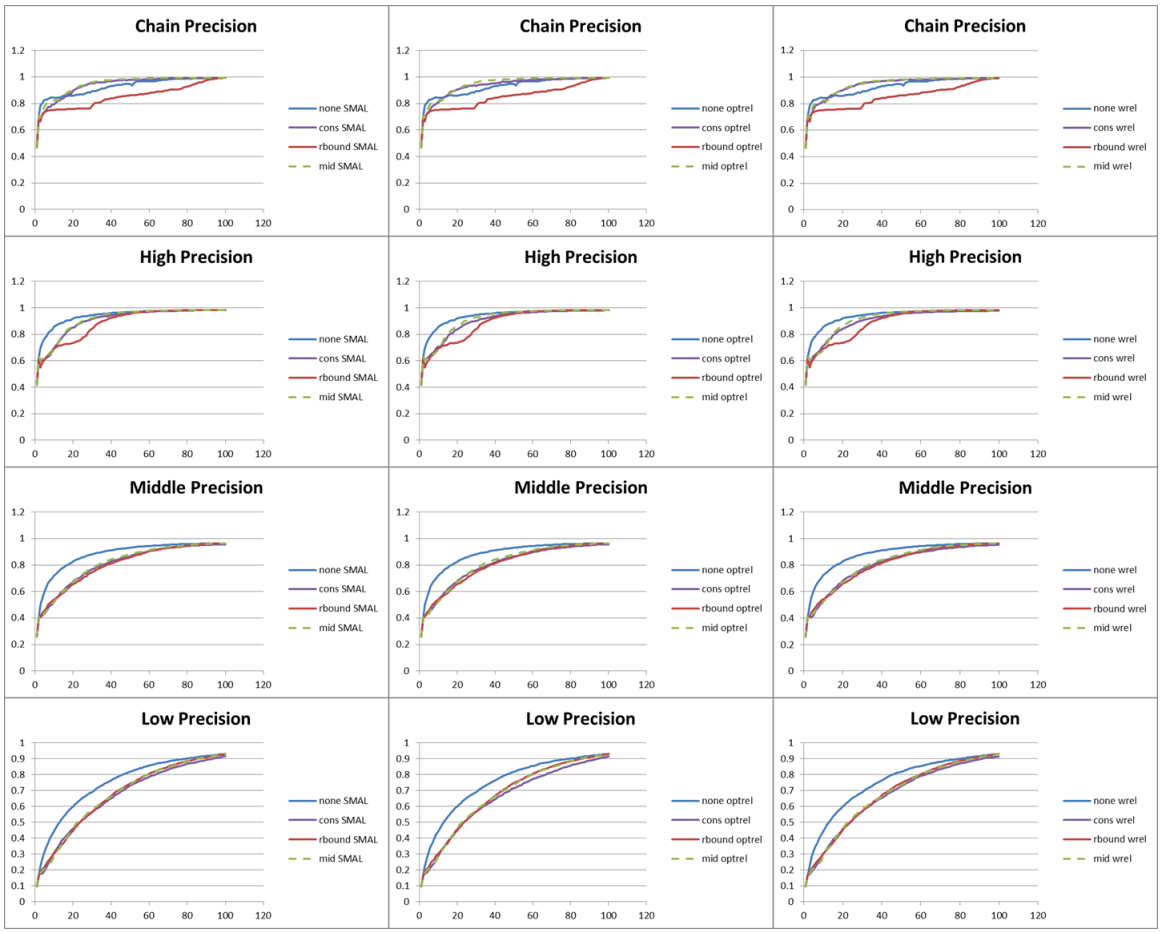

Figure 8.1: Precision graphs of averaged scores for an oracle with an mis-classification rate of **10%**. The results are plotted against the number of queries performed (with a max of 100). Each row shows graphs for a certain data set. From top-to-bottom the data sets are: Chain, High, Middle, Low.

oracle that mis-classifies 20% of the queries on average. The right most columns in tables 8.1 and 8.2 indicate the worst performing experiments out of the 100 runs. Counting again the number of best performances over the 32 different cases we see that: The original approach performs best in 22 cases, in 5 cases the *Rbound* heuristic performs best, and the *Mid* and *Cons* heuristics edge out the others in 3 and 2 cases respectively.

8.2 MAL

The algorithm maintains a list of unlabelled vectors U and a list of labelled vectors T. There can be multiple instances of each vector x in the dataset X, the number of times x occurs is marked by $n(x)$, which is also maintained in T. For each vector x_i we also maintain an interval of potential

	Average Q-score								Minimum Q-score			
Chain	*5%*		*10%*		*20%*		*50%*		*5%*	*10%*	*20%*	*50%*
none SMAL	**0.855**	0.124	**0.873**	0.143	0.878	0.141	0.962	0.069	**0.398**	0.398	0.398	0.675
none optrel	**0.855**	0.124	**0.873**	0.143	0.878	0.141	0.962	0.069	**0.398**	0.398	0.398	0.675
none wrel	**0.855**	0.124	**0.873**	0.143	0.878	0.141	0.962	0.069	**0.398**	0.398	0.398	0.675
cons SMAL	0.745	0.184	0.812	0.164	0.908	0.115	0.983	0.031	0.260	0.280	0.380	**0.860**
cons optrel	0.756	0.146	0.826	0.141	0.916	0.108	0.977	0.048	0.260	0.270	0.490	0.750
cons wrel	0.774	0.123	0.835	0.101	0.919	0.083	0.982	0.043	0.360	0.582	**0.684**	0.750
rbound SMAL	0.742	**0.060**	0.764	**0.065**	0.772	**0.074**	0.873	0.104	0.389	**0.610**	0.610	0.658
rbound optrel	0.742	**0.060**	0.764	**0.065**	0.772	**0.074**	0.876	0.104	0.389	**0.610**	0.610	0.658
rbound wrel	0.742	**0.060**	0.764	**0.065**	0.772	**0.074**	0.874	0.104	0.389	**0.610**	0.610	0.658
mid SMAL	0.788	0.121	0.834	0.110	0.919	0.093	**0.988**	**0.036**	0.353	0.498	0.575	0.700
mid optrel	0.788	0.121	0.834	0.110	0.920	0.093	**0.988**	**0.036**	0.353	0.498	0.575	0.700
mid wrel	0.798	0.100	0.835	0.104	**0.922**	0.091	0.986	0.042	0.353	0.498	0.575	0.700

	Average Q-score								Minimum Q-score			
High	*5%*		*10%*		*20%*		*50%*		*5%*	*10%*	*20%*	*50%*
none SMAL	**0.784**	**0.091**	**0.866**	0.093	0.925	0.045	0.973	0.029	**0.440**	0.370	0.780	0.860
none optrel	**0.784**	**0.091**	**0.866**	0.093	**0.926**	0.045	0.973	0.029	**0.440**	0.370	0.780	0.860
none wrel	**0.784**	**0.091**	**0.866**	0.093	**0.926**	**0.043**	0.973	0.029	**0.440**	0.370	**0.790**	0.860
cons SMAL	0.630	0.117	0.710	0.133	0.857	0.073	0.963	0.038	0.110	0.030	0.630	0.785
cons optrel	0.646	0.106	0.727	0.116	0.842	0.088	0.961	0.047	0.200	0.270	0.540	0.715
cons wrel	0.649	0.102	0.726	0.104	0.849	0.081	0.959	0.039	0.340	0.230	0.490	0.780
rbound SMAL	0.615	0.095	0.701	**0.083**	0.743	0.109	0.960	0.027	0.190	**0.500**	0.570	0.870
rbound optrel	0.615	0.095	0.701	**0.083**	0.746	0.108	0.960	0.027	0.190	**0.500**	0.610	0.870
rbound wrel	0.620	0.092	0.702	**0.083**	0.744	0.107	0.959	0.030	0.190	**0.500**	0.610	0.820
mid SMAL	0.655	0.102	0.700	0.109	0.870	0.069	**0.974**	**0.025**	0.320	0.185	0.530	**0.890**
mid optrel	0.655	0.102	0.700	0.109	0.872	0.063	**0.974**	**0.025**	0.320	0.185	0.580	**0.890**
mid wrel	0.655	0.102	0.696	0.106	0.874	0.061	0.973	0.025	0.320	0.185	0.580	0.880

	Average Q-score								Minimum Q-score			
Middle	*5%*		*10%*		*20%*		*50%*		*5%*	*10%*	*20%*	*50%*
none SMAL	**0.607**	**0.062**	0.726	0.069	0.835	0.059	**0.941**	**0.034**	**0.370**	**0.490**	0.665	**0.815**
none optrel	**0.607**	**0.062**	0.726	0.069	0.835	0.060	0.940	**0.034**	**0.370**	**0.490**	0.665	**0.815**
none wrel	**0.607**	**0.062**	**0.730**	**0.064**	**0.838**	**0.056**	0.939	**0.034**	**0.370**	**0.490**	**0.694**	**0.815**
cons SMAL	0.440	0.069	0.527	0.105	0.682	0.087	0.876	0.067	0.290	0.170	0.430	0.590
cons optrel	0.446	0.059	0.541	0.090	0.686	0.094	0.874	0.053	0.330	0.320	0.380	0.725
cons wrel	0.421	0.087	0.539	0.096	0.689	0.086	0.881	0.062	0.070	0.210	0.450	0.680
rbound SMAL	0.460	0.080	0.542	0.110	0.663	0.113	0.867	0.053	0.260	0.360	0.330	0.725
rbound optrel	0.460	0.080	0.542	0.110	0.664	0.109	0.870	0.052	0.260	0.360	0.480	0.725
rbound wrel	0.461	0.078	0.546	0.112	0.665	0.106	0.873	0.054	0.260	0.360	0.489	0.725
mid SMAL	0.440	0.074	0.530	0.088	0.680	0.084	0.887	0.052	0.250	0.280	0.460	0.770
mid optrel	0.440	0.074	0.530	0.088	0.681	0.082	0.887	0.052	0.250	0.280	0.480	0.770
mid wrel	0.440	0.077	0.530	0.091	0.682	0.081	0.888	0.052	0.250	0.280	0.480	0.770

	Average Q-score								Minimum Q-score			
Low	*5%*		*10%*		*20%*		*50%*		*5%*	*10%*	*20%*	*50%*
none SMAL	**0.320**	**0.043**	**0.452**	0.058	**0.613**	**0.053**	**0.839**	0.053	0.160	**0.300**	**0.470**	**0.720**
none optrel	**0.320**	**0.043**	**0.452**	0.058	**0.613**	**0.053**	**0.839**	0.052	0.160	**0.300**	**0.470**	**0.720**
none wrel	**0.320**	**0.043**	**0.452**	**0.057**	0.612	0.055	0.838	**0.048**	**0.180**	**0.300**	0.420	**0.720**
cons SMAL	0.199	0.046	0.313	0.079	0.473	0.093	0.748	0.077	0.080	0.150	0.290	0.565
cons optrel	0.205	0.055	0.294	0.074	0.468	0.081	0.732	0.076	0.090	0.150	0.280	0.555
cons wrel	0.207	0.048	0.310	0.064	0.469	0.076	0.749	0.068	0.110	0.155	0.320	0.570
rbound SMAL	0.221	0.045	0.307	0.066	0.457	0.093	0.760	0.068	0.120	0.200	0.305	0.620
rbound optrel	0.221	0.045	0.307	0.066	0.457	0.093	0.762	0.067	0.120	0.200	0.305	0.620
rbound wrel	0.223	0.045	0.305	0.064	0.458	0.094	0.764	0.067	0.120	0.200	0.305	0.620
mid SMAL	0.215	0.052	0.291	0.069	0.478	0.072	0.755	0.066	0.110	0.182	0.333	0.599
mid optrel	0.215	0.052	0.291	0.069	0.478	0.072	0.753	0.061	0.110	0.182	0.333	0.599
mid wrel	0.215	0.052	0.294	0.071	0.476	0.072	0.751	0.059	0.110	0.182	0.333	0.599

Table 8.1: From top to bottom the tables show the average Q score for the datasets *Chain*, *High*, *Middle* and *Low* when using an oracle with an error-rate of 10%. Each pair of columns shows the average (left) and standard deviation (right) after 5%, 10%, 20% and 50% of the data is queried. The bold-faced entries mark the best performing heuristic.

	Average Q-score								Minimum Q-score			
Chain	*5%*		*10%*		*20%*		*50%*		*5%*	*10%*	*20%*	*50%*
none SMAL	**0.828**	**0.130**	**0.834**	0.160	0.840	0.187	0.944	0.063	0.077	0.113	0.220	0.666
none optrel	**0.828**	**0.130**	**0.834**	0.160	0.840	0.187	0.944	0.063	0.077	0.113	0.220	0.666
none wrel	**0.828**	**0.130**	**0.834**	0.160	0.840	0.187	0.944	0.063	0.077	0.113	0.220	0.666
cons SMAL	0.705	0.203	0.800	0.174	0.882	0.159	0.954	**0.073**	0.037	0.205	0.230	0.660
cons optrel	0.689	0.209	0.769	0.204	0.874	0.157	0.955	0.088	**0.128**	0.110	0.290	0.460
cons wrel	0.739	0.187	0.808	0.159	**0.884**	0.118	0.959	0.075	0.000	0.080	0.400	0.480
rbound SMAL	0.668	0.165	0.747	0.138	0.794	0.117	0.877	0.114	0.000	**0.274**	0.361	0.440
rbound optrel	0.668	0.165	0.754	**0.132**	0.798	**0.112**	0.887	0.100	0.000	**0.274**	**0.534**	0.673
rbound wrel	0.668	0.165	0.754	**0.132**	0.797	**0.112**	0.890	0.098	0.000	**0.274**	**0.534**	**0.755**
mid SMAL	0.704	0.208	0.786	0.169	**0.884**	0.125	**0.963**	0.084	0.000	0.077	0.182	0.500
mid optrel	0.704	0.208	0.786	0.169	0.881	0.139	0.962	0.084	0.000	0.077	0.182	0.500
mid wrel	0.740	0.171	0.811	0.131	0.878	0.139	0.957	0.087	0.000	0.077	0.182	0.500

	Average Q-score								Minimum Q-score			
High	*5%*		*10%*		*20%*		*50%*		*5%*	*10%*	*20%*	*50%*
none SMAL	**0.782**	**0.098**	0.830	0.123	**0.903**	**0.069**	0.952	0.052	**0.440**	**0.284**	**0.600**	0.665
none optrel	**0.782**	**0.098**	0.830	0.123	**0.903**	**0.069**	0.952	0.052	**0.440**	**0.284**	**0.600**	0.665
none wrel	**0.782**	**0.098**	**0.834**	**0.121**	0.901	0.073	0.952	0.051	**0.440**	**0.284**	**0.600**	0.665
cons SMAL	0.617	0.149	0.719	0.125	0.830	0.113	0.938	0.050	0.110	0.320	0.395	0.790
cons optrel	0.639	0.137	0.681	0.175	0.828	0.097	0.944	0.040	0.200	0.194	0.529	0.810
cons wrel	0.613	0.157	0.707	0.122	0.823	0.108	0.946	0.040	0.090	0.250	0.365	0.798
rbound SMAL	0.589	0.143	0.689	0.129	0.765	0.110	0.941	0.036	0.190	0.120	0.564	0.850
rbound optrel	0.589	0.143	0.694	0.125	0.769	0.111	0.940	0.039	0.190	0.120	0.564	0.810
rbound wrel	0.592	0.142	0.695	0.125	0.759	0.114	0.942	0.038	0.190	0.120	0.425	0.810
mid SMAL	0.622	0.157	0.666	0.148	0.837	0.109	**0.958**	**0.030**	0.070	0.130	0.210	**0.875**
mid optrel	0.622	0.157	0.661	0.155	0.838	0.094	**0.958**	0.031	0.070	0.130	0.480	0.850
mid wrel	0.624	0.154	0.669	0.143	0.836	0.099	0.957	**0.030**	0.070	0.130	0.422	**0.875**

	Average Q-score								Minimum Q-score			
Middle	*5%*		*10%*		*20%*		*50%*		*5%*	*10%*	*20%*	*50%*
none SMAL	0.592	0.080	0.696	0.093	0.822	0.067	0.908	0.047	**0.280**	0.300	0.565	0.748
none optrel	0.592	0.080	0.699	**0.082**	**0.825**	**0.062**	0.909	0.047	**0.280**	**0.500**	**0.660**	0.748
none wrel	**0.594**	**0.077**	**0.704**	0.085	0.821	0.064	**0.910**	**0.043**	**0.280**	0.300	**0.660**	**0.778**
cons SMAL	0.443	0.115	0.548	0.117	0.668	0.112	0.863	0.055	0.010	0.170	0.290	0.734
cons optrel	0.438	0.119	0.536	0.137	0.680	0.098	0.857	0.057	0.010	0.070	0.300	0.675
cons wrel	0.431	0.109	0.558	0.094	0.696	0.095	0.862	0.058	0.030	0.284	0.445	0.699
rbound SMAL	0.433	0.102	0.549	0.115	0.689	0.110	0.871	0.053	0.120	0.253	0.410	0.735
rbound optrel	0.433	0.102	0.550	0.116	0.688	0.109	0.873	0.053	0.120	0.253	0.410	0.759
rbound wrel	0.434	0.102	0.550	0.114	0.683	0.108	0.872	0.054	0.130	0.253	0.410	0.759
mid SMAL	0.437	0.098	0.536	0.122	0.689	0.094	0.867	0.052	0.050	0.182	0.295	0.700
mid optrel	0.437	0.098	0.539	0.123	0.692	0.086	0.868	0.053	0.050	0.182	0.435	0.670
mid wrel	0.440	0.097	0.547	0.114	0.685	0.092	0.869	0.052	0.050	0.182	0.400	0.675

	Average Q-score								Minimum Q-score			
Low	*5%*		*10%*		*20%*		*50%*		*5%*	*10%*	*20%*	*50%*
none SMAL	**0.313**	0.053	**0.453**	**0.058**	0.596	0.059	0.798	0.057	**0.140**	**0.260**	**0.458**	0.665
none optrel	**0.313**	0.053	**0.453**	**0.058**	**0.597**	0.058	0.797	0.056	**0.140**	**0.260**	**0.458**	**0.669**
none wrel	**0.313**	0.053	0.451	0.059	0.597	**0.058**	**0.807**	**0.050**	**0.140**	**0.260**	0.446	**0.669**
cons SMAL	0.205	0.065	0.327	0.083	0.491	0.078	0.742	0.067	0.064	0.145	0.320	0.583
cons optrel	0.209	0.068	0.316	0.079	0.485	0.084	0.742	0.064	0.075	0.130	0.295	0.594
cons wrel	0.201	0.056	0.315	0.070	0.481	0.076	0.732	0.065	0.060	0.175	0.310	0.594
rbound SMAL	0.214	0.046	0.317	0.074	0.491	0.097	0.758	0.063	0.120	0.147	0.290	0.589
rbound optrel	0.214	0.046	0.317	0.074	0.490	0.096	0.757	0.064	0.120	0.147	0.290	0.589
rbound wrel	0.214	0.046	0.319	0.072	0.489	0.096	0.758	0.062	0.120	0.190	0.290	0.589
mid SMAL	0.213	**0.042**	0.296	0.068	0.487	0.082	0.732	0.071	0.120	0.169	0.320	0.528
mid optrel	0.213	**0.042**	0.296	0.068	0.487	0.082	0.732	0.070	0.120	0.169	0.320	0.551
mid wrel	0.212	**0.042**	0.298	0.070	0.486	0.082	0.738	0.068	0.120	0.169	0.320	0.551

Table 8.2: From top to bottom the tables show the average Q score for the datasets *Chain*, *High*, *Middle* and *Low* when using an oracle with an error-rate of 20%. Each pair of columns shows the average (left) and standard deviation (right) after 5%, 10%, 20% and 50% of the data is queried. The bold-faced entries mark the best performing heuristic.

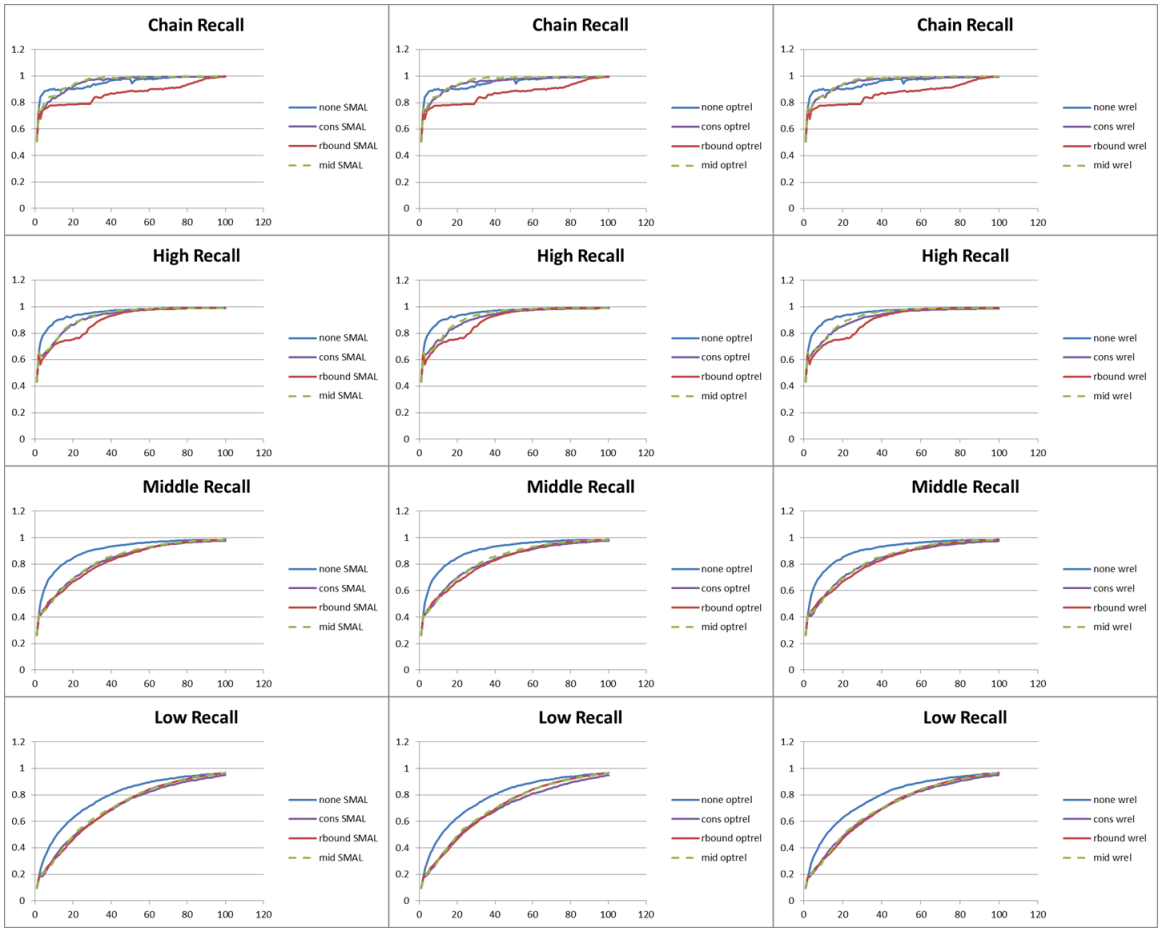

Figure 8.2: Recall graphs of averaged scores for an oracle with an mis-classification rate of **10%**. The results are plotted against the number of queries performed (with a max of 100). Each row shows graphs for a certain data set. From top-to-bottom the data sets are: Chain, High, Middle, Low.

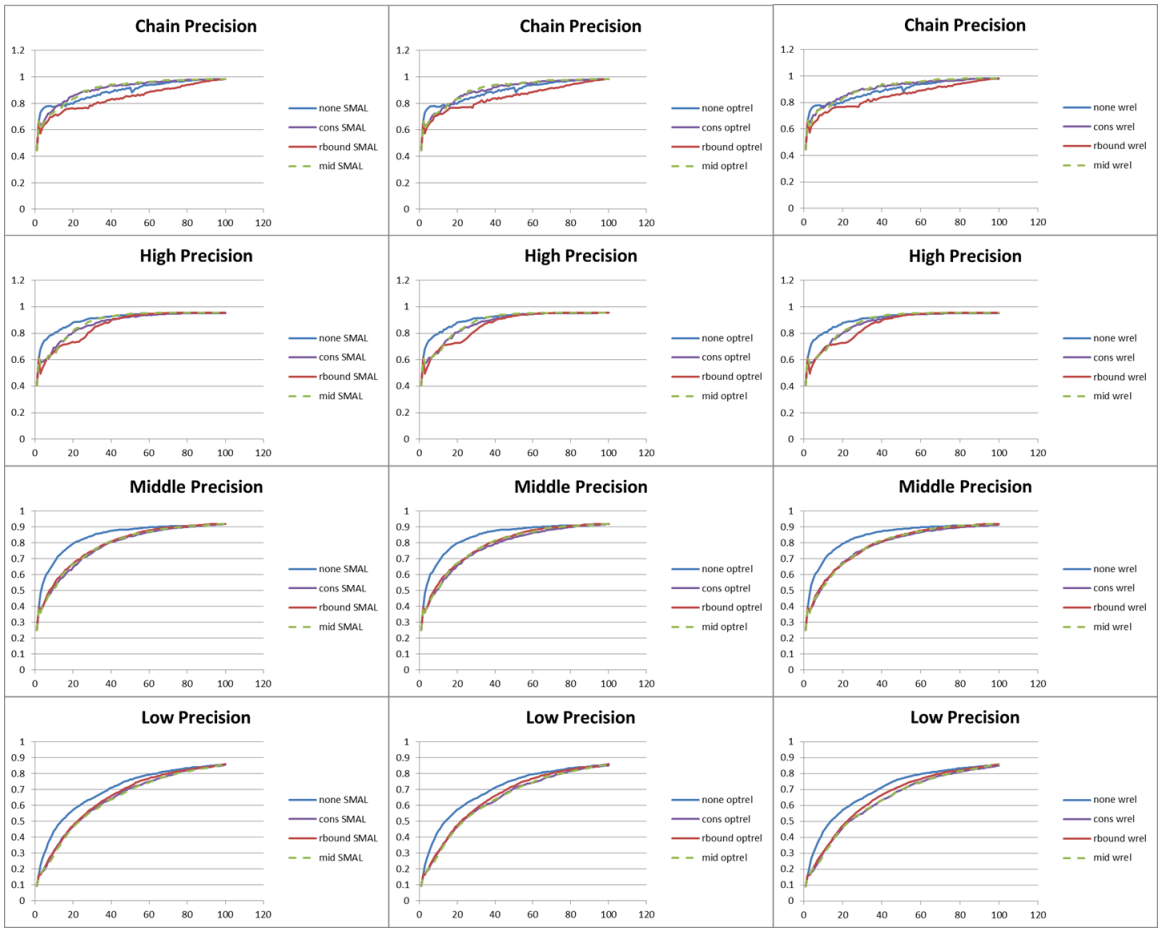

Figure 8.3: Precision graphs of averaged scores for an oracle with an mis-classification rate of **20%**. The results are plotted against the number of queries performed (with a max of 100). Each row shows graphs for a certain data set. From top-to-bottom the data sets are: Chain, High, Middle, Low.

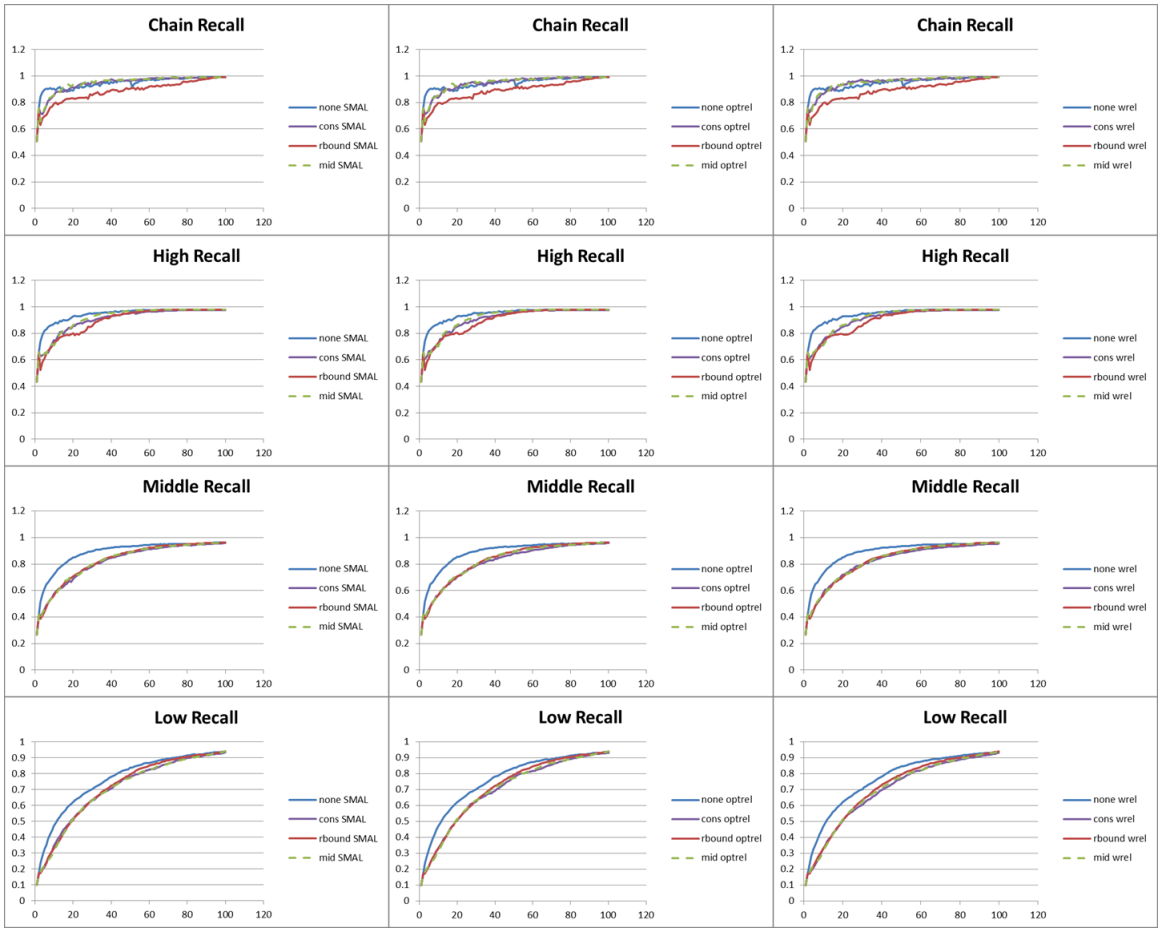

Figure 8.4: Recall graphs of averaged scores for an oracle with an mis-classification rate of **20%**. The results are plotted against the number of queries performed (with a max of 100). Each row shows graphs for a certain data set. From top-to-bottom the data sets are: Chain, High, Middle, Low.

Algorithm 2 MAL(X,max)

1: $T \leftarrow \emptyset$
2: $\mathcal{U} \leftarrow X$
3: **while** $max > 0 \wedge U \neq \emptyset$ **do**
4: $\quad x^* \leftarrow \arg\max H(x)_{x \in \mathcal{U}}$
5: $\quad y^* \leftarrow \mathcal{O}(x^*)$
6: $\quad T \leftarrow T \cup \{(x^*, n(x^*)), y^*\}$
7: $\quad \mathcal{U} \leftarrow \mathcal{U} \setminus \{x^*\}$
8: $\quad$ **for all** $x_i \in\downarrow (x^*)$ **do**
9: $\quad\quad h_i \leftarrow \min(h_i, y^*)$
10: $\quad$ **end for**
11: $\quad$ **for all** $x_i \in\uparrow (x^*)$ **do**
12: $\quad\quad l_i \leftarrow \max(l_i, y^*)$
13: $\quad$ **end for**
14: $\quad$ **for all** $x_I \in \mathcal{U}$ **do**
15: $\quad\quad$ **if** $l_i = h_i$ **then**
16: $\quad\quad\quad T \leftarrow T \cup \{(x_i, n(x_i)), l_i\}$
17: $\quad\quad\quad \mathcal{U} \leftarrow \mathcal{U} \setminus \{x_i\}$
18: $\quad\quad$ **end if**
19: $\quad$ **end for**
20: $\quad max \leftarrow max - 1$
21: **end while**
22: **return** T

labels $[l_i, h_i]$. The algorithm returns the list of labelled vectors T. At initiation this interval is set to $[1, k]$ where k is the largest label. In line 4 the best vector to query is selected using the following heuristic. For each vector x_i number of labels that can be eliminated in the worst-case is computed with:

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

where $z_+ = max(0, z)$. We then select the vector that eliminates the most vectors to be the next to be presented to the oracle in line 5. Lines 8-13 perform the inference step by updating the upset and the downset of the selected vector x^*. Here we use the monotonicity assumption, which states that vectors in the downset (upset) of x^* cannot have a label that is greater (smaller) than y^* Lines 14-19 check whether any vectors have their labels uniquely determined by inference, that is, we check if any vector x_i has $l_i = h_i$. The main loop in line 3 runs until either all unlabelled vectors are labelled or until a predetermined number of iterations has been reached.

8.3 SMAL

The algorithm is supplied a list of unlabelled vectors U and a list of vectors that have been labelled so far Q. The algorithm also maintains a count of the number of times a vector unique x appears

Algorithm 3 SMAL(Q,U,max)

1: **while** $max > 0 \wedge U \neq \emptyset$ **do**
2: $(l, u) \leftarrow Relabel(Q)$
3: **for all** $x \in \mathcal{U}$ **do**
4: $H_l(x) \leftarrow$ query value of x using l
5: $H_u(x) \leftarrow$ query value of x using u
6: **end for**
7: $x^* \leftarrow \arg\max_{x \in \mathcal{U}} (H_l(x) + H_u(x))/2$
8: $y^* \leftarrow \mathcal{O}(x^*)$
9: $Q \leftarrow Q \cup \{(x^*, y^*)\}$
10: $n(x^*) \leftarrow n(x^*) - 1$
11: **if** $n(x^*) = 0$ **then**
12: $\mathcal{U} \leftarrow \mathcal{U} \setminus \{x^*\}$
13: **end if**
14: **end while**
15: $(l, u) \leftarrow Relabel(Q)$
16: **return** (l, u)

in U as $n(x)$. Since a vector can be present in U multiple times, the algorithm uses this count to determine whether or not the vector can still be queried. If the count $n(x)$ reaches 0, the vector x can no longer be asked of the oracle. The algorithm returns a maximal and a minimal relabelling of Q. Line 2 computes the maximal optimal relabelling u and the minimal optimal relabelling l for the vectors in Q, by using the Relabelling algorithm. In line 3-6 the relabellings u and l are each combined with the set of unlabelled vectors (performing inference if necessary). We then apply the same worst-case heuristic as the MAL algorithm to compute the scores H_l and H_u for all vectors in U. Line 7 selects the query point that maximizes the average between these two scores. Line 10 updates the count of the labelled vector, and lines 11-13 checks whether x^* should be removed from U. The main loop runs until either all vectors are labelled or until a predetermined maximum of iterations is reached.

8.4 Generating Monotone Functions

To draw a random sample from the distribution of all monotone functions on some order we use an algorithm by Soons [10]. The algorithm is an implementation of the Propp-Wilson algorithm (see [7]), with a *state space* $\mathcal{S}$ and *update function* ϕ that have been modified to work with monotone functions. The Propp-Wilson algorithm runs two coupled Markov chains starting in points $S_\perp$ and $S^\top$ "in the past" from time T to -1. Each time-step the update function randomly transitions each chain to the next state in a Monte Carlo simulation. The update function will have the form $\phi_t(S, u)$, where $S \in \mathcal{S}$ is an instance of the state space also known as the *initial solution*, and u is

a uniformly distributed random number in the interval $(0, 1)$. It can be shown that, if two chains 'meet' at some point they become 'stuck' forever, and if we run multiple chains with the same values of u each step they all would have coalesced at this state. This last result means that, no matter what initial state we started in, all result in the same 'fixed' state and there is no more bias. It is thus enough to start two chains in different states and run them until they 'meet', their shared state is then an unbiased sample from the entire state space.

Recall the observation that we can build a monotone function by assigning labels to each item of a nested sequence of lower sets (equation 4.6). We can modify an existing function by removing or adding items to lower sets as long as we maintain the monotonicity. A vector $\mathbf{x}$ can be effectively moved up a set in the sequence by removing it from the first lower set that contains it, as long as none the vectors in its upset are in its current set. If we use equation 4.6 to assign labels, this action increases the label of vector $\mathbf{x}$. In a similar way we can decrease the label of a vector, by finding the last lower set that doesn't contain it and adding it.

If we choose our state space $\mathcal{S}$ to be all sequences of lower sets, that is $S = L_1^S, \ldots, L_{k-1}^S$, we can define out update functions as follows. The function $increase(S, x)$ will attempt to remove $\mathbf{x}$ from the first lower set L_j that contains it, that is:

$$increase(S, x)S[L_j^S \to L_j^S \setminus \{\mathbf{x}\}] \text{ if } \mathbf{x} \in (L_j^S \setminus L_{j-1}^S) \text{ and } L_j^S \setminus \{\mathbf{x}\} \text{ is a lower set.} \tag{8.1}$$

The notation $S[X \to Y]$ means state S with element X replaced with element Y. The function removes $\mathbf{x}$ from the first lower set in which it's encountered, as long as the resulting set doesn't violate monotonicity. Conversely the function $decrease(S, x)$ will attempt to add $\mathbf{x}$ to the first lower set L_j doesn't contain it, that is:

$$decrease(S, x)S[L_j^S \to L_j^S \cup \{\mathbf{x}\}] \text{ if } \mathbf{x} \in (L_{j+1}^S \setminus L_j^S) \text{ and } L_j^S \cup \{\mathbf{x}\} \text{ is a lower set.} \tag{8.2}$$

We can set up our initial states $S_\perp$ and $S^\top$ as any sequence, but we choose $S_\perp$ to be the sequence of complete sets, that is $L_i^{S_\perp} = X, \forall i \in 1, \ldots, k$. And $S_\top$ to be the sequence of empty sets, that is $L_i^{S^\top} = \emptyset, \forall i \in 1, \ldots, k$.

The algorithm generates for a given order $(X, \preceq)$ a monotone function with k labels. Lines 1 and 2 set up the initial states for the simulation. The loop in lines 3-14 simulates the propagation of both states, the value T should be chosen large enough to allow the states to 'meet'. Line 5 and 6 draw both a value from a uniform distribution (0,1) and a random vectors from X. If $u \geq 0.5$

Algorithm 4 Generate Monotone Function $(X, \preceq, k)$

$S^\top \leftarrow L_i^{S^\top} = \emptyset, \forall i \in 1, \ldots, k$
$S_\perp \leftarrow L_i^{S_\perp} = X, \forall i \in 1, \ldots, k$
for $i = -T$ to -1 **do**
    $u \leftarrow random(0, 1)$
    $x \leftarrow sample(X)$
    if $u \geq 0.5$ **then**
        $S^\top \leftarrow increase(S^\top, x)$
        $S_\perp \leftarrow increase(S_\perp, x)$
    end if
    if $u < 0.5$ **then**
        $S^\top \leftarrow decrease(S^\top, x)$
        $S_\perp \leftarrow decrease(S_\perp, x)$
    end if
end for
return $S^\top$

the subroutine *increase* will attempt to increase the label of vector $\mathbf{x}$ provided that the result is stil a valid monotone function. If $u < 0.5$ the subroutine *decrease* will attempt to decrease the label of vector $\mathbf{x}$. At termination of the loop we know that both $S^\top$ and $S_\perp$ represent the same monotone function and all initialization bias is gone.

8.5 Relabelling algorithm

The Relabelling algorithm by Feelders [4] computes an optimal monotone classification of a data set for convex loss functions. The algoritm *relabels* in the sense that it re-evaluates each label and decides which labels can be changed to construct the optimal monotone function, that is the monotone function with the smallest loss.

Let $f_y = \sum_{j=1}^{k} n(x_i, j)L(y, j)$, denote the loss when vector x_i is assigned the label y. Each vector x_i has assigned to it a weight:

$$w_i(y) = f_i^{'}(y) = f_i(y + 1) - f(y),$$

that indicates the increase in loss when the label of vector x_i is increased from y to $y + 1$. The weight of a subset $S \subset D$ is given by $w_S(y) = \sum_{i \in S} w_i(y)$, where D is the collecting of all attribute vectors. We define a *minimal minimum weight upper set (MMWUS)* $U^*(y)$ as the smallest upper set with minimum weight $w_U^*(y)$. That is, the smallest upper set that would add the least to the loss when the label of all vectors in U^* is increased by one. Feelders [4] shows that if we can find such a MWUS $U^*(y)$ all vectors x_i have an optimal label y_i^* that is greater than y. In addition

it's shown that if we take the complement of such a MWUS, $\bar{U}^*(y)$ there is an optimal solution where all vectors get assigned an optimal label that is at most y. Thus we can find a *relabelling* with minimal loss if we find $\bar{U}^*(y)$ for each label in $\{1, ..., k\}$.

Algorithm 5 Relabel($\mathcal{X}, \preceq$)

```
1: for y = 1 to k − 1 do
2:     for i ∈ X do
3:         w_i ← f_i(y + 1) − f_i(y)
4:     end for
5:     U* ← minimal minimum weight upperset of (X, ⪯, w)
6:     for all j ∈ Ū* do
7:         y*_j ← y
8:     end for
9:     X ← X \ Ū*
10: end for
11: for i ∈ X do
12:     y*_i ← k
13: end for
14: return y*
```

The algorithm is supplied a set of labelled vectors $\mathcal{X}$ and an ordering $\preceq$, and returns an optimal relabelling $\mathbf{y}^*$. In line 3-5 the weights for all vectors are computed given the current value of y. The algorithm computes the MMWUS in line 5 by solving a max-flow problem as shown by Picard [6]. In line 6-8 we use that the vectors in the complement of the MMWUS get a label that is at most y. In line 11-13 the remaining vectors are assigned the greatest label k.

To compute all optimal monotone function the algorithm computes the minimal minimum weight lower set (MMLUS) $L^*(l)$ analogous to the MMWUS. For the attribute vectors in $L^*(l)$ we have that the label van be at most l. Combined with the values obtained earlier we now have, for each attribute vector, an interval of optimal labels. Here we use that we know that the vectors in a MMWUS for a certain value of l should have a label that is greater than l, giving a lower bound. Similarly the vectors in a MMWLS should have a value that is at most l, giving an upper bound.

The algorithm is again supplied a set of labelled vectors $\mathcal{X}$ and an ordering $\preceq$. For each vector i a minimum y_i^{min} and a maximum label y_i^{max} is kept. The algorithm returns an interval $[y^{min}, y^{max}]$ for all vectors in $\mathcal{X}$ In line 5-7 the algorithm computes the weights for the current value of y. In lines 8 and 12 the MMWUS and the MMWLS are computed using the same max-flow strategy as before. In line 9-11 the algorithm assigns the label $y+1$ to y^{min} for all vectors in the MMWUS. Similarly, in line 13-15 the vectors from the MMWLS get their upper bounds.

Algorithm 6 Intervals$(\mathcal{X}, \preceq)$

for $i \in \mathcal{X}$ **do**
    $[y_i^{min}, y_i^{max}] \leftarrow [1, k]$
end for
for $l = 1$ to $k - 1$ **do**
    for $i \in \mathcal{X}$ **do**
        $w_i \leftarrow f_i(l + 1) - f_i(l)$
    end for
    $U^* \leftarrow$ minimal minimum weight upperset of $(\mathcal{X}, \preceq, w)$
    for all $j \in U^*$ **do**
        $y_j^{min} \leftarrow l + 1$
    end for
    $L^* \leftarrow$ minimal maximum weight lowerset of $(\mathcal{X}, \preceq, w)$
    for all $j \in L^*$ **do**
        $y_j^{max} \leftarrow l$
    end for
    $\mathcal{X} \leftarrow \mathcal{X} \setminus L^*$
end for
return y^{min}, y^{max}

Bibliography

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

[2] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[3] Sanjoy Dasgupta. Analysis of a greedy active learning strategy. In *NIPS*, volume 3, page 2, 2004.

[4] Ad Feelders. Monotone relabeling in ordinal classification. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 803–808. IEEE, 2010.

[5] Wojciech Kotłowski and Roman Słowiński. Statistical approach to ordinal classification with monotonicity constraints. In *Preference Learning ECML/PKDD 2008 Workshop*, 2008.

[6] Jean-Claude Picard. Maximal closure of a graph and applications to combinatorial problems. *Management Science*, 22(11):1268–1272, 1976.

[7] James Gary Propp and David Bruce Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics. *Random structures and Algorithms*, 9(1-2):223–252, 1996.

[8] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[9] Victor S Sheng, Foster Provost, and Panagiotis G Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 614–622. ACM, 2008.

[10] Pieter Soons and Ad Feelders. Exploiting monotonicity constraints in active learning for ordinal classification. Technical report, Technical Report UU-CS-2014-001, Department of Information and Computing Sciences, Utrecht University, Utrecht, 2014.

[11] Vetle I Torvik and Evangelos Triantaphyllou. Discovering rules that govern monotone phenomena. In *Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques*, pages 149–192. Springer, 2006.