

Algorithms for Machine Learning and Inference

Summary Report

of the course given by Peter Damaschke at Chalmers TH
in Quarter III of Session 2003/04

Urs Enke

791121-0000, no programme

March 2004

Contents

1	Introduction	3
2	Concept Learning	4
2.1	The Conjunctive Case	4
2.1.1	Further limiting the Hypothesis Space	4
2.1.2	Find-S Algorithm	5
2.1.3	Candidate-Elimination Algorithm	5
2.2	Less Specific Hypothesis Space Restrictions	6
2.2.1	Decision Trees	6
2.3	Asking Learners	8
3	Neural Networks	9
3.1	Perceptrons	9
3.2	Learning Process	9
3.3	Capabilities and Extensions	10
3.3.1	Modular ANNs	10
3.4	Backpropagation Algorithm	11
3.4.1	Problems	11
4	Bayesian Learning	13
4.1	Quality Assessments using MAP Hypotheses	13
4.1.1	Consistent Learners	14
4.1.2	Sum of Squared Errors	14
4.1.3	Minimum Description Length Principle	14
4.2	Bayesian Classifiers	15
4.2.1	Bayes Optimal Classifier	15
4.2.2	Gibbs Algorithm	15
4.2.3	Naive Bayes Classifier	15
4.3	Probabilistic Expert Systems	16
4.3.1	Learning	17
4.3.2	Efficient Inference	17
4.3.3	The Use of Bayesian Networks	17
5	Various Topics	18
5.1	Probably Almost Correct Learners	18
5.1.1	Bounds on Training Set Size	18
5.1.2	Bounds on Mistakes During Learning	18
5.2	Instance-Based Learning	19
5.2.1	Nearest-Neighbour Learning	19
5.2.2	Unification with Hypothesis Induction	20
5.3	Determining Relevant Attributes	20
5.3.1	Learning from Queries	20
5.3.2	Learning from Random Samples	21
5.4	Learning as Search: ADATE	21
5.4.1	Inductive Logic Programming	22
5.5	Learning Phylogenetic Trees	22
5.6	Unsupervised Learning	23
5.7	Contrast Sets	23
6	Conclusion	24

1 Introduction

Quite often, we want a computer to perform a task that is difficult or even, at the time of the system's design, impossible to program in a way that yields an adequate performance. Sometimes such problems can be approached by letting the computer 'learn' concepts by example or experience instead of being initially supplied with them. The recognition of a face is an example where it seems obviously easier to let the software find common traits in various pictures of the same person than having its programmer code all details.

Whereas playing games, interacting with the 'real world' or similar tasks involving a rather active role on the computer's side also provide large areas where learning can be beneficial (in the games example, this would be learning from an opponent's actions), the course which this report is to summarize concentrates on the classification of given and prediction of new data (hence 'inference' in the course's name).

The spectrum of actual uses for the methodologies presented is still huge and ranges from 'theoretical' topics like the learning of formal grammars from examples of words that are or are not in the respective language (cf. [1]) to very utilitarian ones like the classification of changed web pages as interesting or not, according to past such classifications by the user in question (cf. [2]).

As required, this report summarizes the lecture *Algorithms for Machine Learning and Inference* in a way suitable for readers new to the topics. Algorithms are thus not explained in mathematical detail, but only their major ideas mentioned. Extensive examples are also largely omitted, instead the attempt is made to give a practical hint now and then in the text that could arouse interest in the reader to selectively seek further information on the respective matter.

The order of topics has been re-arranged a little and additionally structured in order to provide said 'outsider' with a thread along which to read – and optionally omit – the various sections. Where deemed fitting and interesting by the author, aspects and ideas are looked at that have not been part of the lecture, in these cases references to further sources are given.

2 Concept Learning

In case the task assigned to our system is to answer questions on whether an object, situation etc. belongs to a certain class of such *instances*, one approach is to look for a (boolean) function that gets an instance and returns whether it belongs to the class in question or not, i.e. is a *positive* or a *negative example*.

To approximate (or even determine) this function, known as *target concept*, the methods presented in this section evaluate training examples for which the concept function's value is known. The approximated function is called *hypothesis* and after the learning process used to infer classification for previously unknown instances.¹

In order to be able to handle all this algorithmically, we need a formal way to describe instances. Usually, a tuple is used, in which each component represents a certain attribute's value, and that is mapped to *true* if and only if it belongs to the concept we want to learn.

Note that here, we assume that the training examples can not be influenced by the learning algorithm. There are, however, algorithms which *do* require querying of example classifications and are intended for practical settings where this can easily be realized.

2.1 The Conjunctive Case

The first algorithms we are going to introduce assume that among each set of attribute values, there is a group of 'good' ones and that exactly those instances are positive that are composed of (= form a conjunction of) such good values. Obviously, this is a significant reduction of our set of possible hypotheses: Every element of this *hypothesis space* can be described using a tuple of attribute value sets.

2.1.1 Further limiting the Hypothesis Space

Actually we only have to consider hypotheses without empty sets, as this would essentially declare all possible values of the respective attribute bad and mean that there are no positive instances at all. With our common-sense, scientific or other knowledge about the actual circumstances of the subject in question, we can further narrow the hypothesis space by disallowing certain subsets. Whatever remains we call the respective attribute's *constraints*.

It is important to see that limiting the hypothesis space is no 'dirty trick', but the essence of intelligent inference: If every detail (i.e. value of an attribute) could change the outcome (i.e. the belonging to the target concept) independently, then no conclusion at all could be drawn for previously unseen examples, i.e. such that have not been part of the training. The particular set of assumptions underlying a learning algorithm is called its *inductive bias*.²

¹This, of course, requires a certain representativeness on behalf of the training examples: Training a face recognition programme on a person with swollen cheeks and a clown's nose obviously makes little sense.

²In special cases, there may be no inductive bias: In what is called *explanation-based* learning, the aim is to organize deduction (from an exhaustive representation of all there is to know about the subject) in a way as efficient as possible, especially by remembering intermediate proofs for future deductions (cf. [3, p.175+]).

2.1.2 Find-S Algorithm

A first approach that we may choose to take when thinking about how to evaluate training examples is the Find-S algorithm. It basically develops an hypothesis just general enough to yield *true* for all positive training examples.³ Take the hypothesis space's most specific element and then for each positive training example replace the current hypothesis with the next more general which classifies that example correctly.

This description does not limit the algorithm to the conjunctive case. If we take that limitation into account, however, we can re-formulate the algorithm based on attribute constraints: The evolving hypothesis can then at each stage be described as a tuple of attribute constraints: Start with an hypothesis that has the most specific constraint (e.g. \emptyset) set on each attribute, then for each positive training example add the occurring attribute values to the respective constraint in the current hypothesis.

Unfortunately, Find-S is only well-defined if there are no choices in each step. In the general case, for this condition to be true it is sufficient for H to be closed under intersection, which in the conjunctive case means that all attribute constraints are closed under intersection. This is usually not the case when we disallow empty sets, yet those do not add any complexity to Find-S, anyway, as only positive examples are considered. (The ignoring of negative examples in itself being a notable fact.)

As we can see, Find-S is rather efficient, at least in the described conjunctive case. Yet we have also seen disadvantages like lack of applicability in the case of hypothesis spaces that are not closed under intersection, and of course the far from satisfactory behaviour of the resulting hypothesis when applied to new instances of unknown classification.

2.1.3 Candidate-Elimination Algorithm

Especially the latter reason gives rise to the wish to have an algorithm that also takes into account negative training examples. For example, we might want to find *all* hypotheses that classify the training examples correctly (we say that they are *consistent* with the set of training examples), not just the one that classifies as few as possible positively.

As the set of hypotheses consistent with the set of training examples is usually very large, it is a bad idea to get an algorithm to enumerate all elements of this set, which is called the *version space* and which, if we call the set of allowed hypotheses H and the set of training examples D (in the form $(x, c(x))$), can be denoted by $VS_{H,D}$.

Instead, we define a partial ordering between hypotheses: $h_1 \geq_g h_2 \Leftrightarrow h_1(x) = 1 \rightarrow h_2(x) = 1$. Intuitively, we can call h_1 *more general* and h_2 *more specific*. We can now define the version space as the set of hypotheses that in terms of specificity lies between two bounding sets, a *general boundary* G and a *specific boundary* S that contain all the most general and most specific hypotheses, respectively, that classify the training examples correctly.

The Candidate-Elimination algorithm initializes these sets and updates them to accommodate each training example that is evaluated.[4, p.33] For it to classify an unseen instance as *true*, it suffices that all hypotheses in S agree on this, likewise all in G have to agree on a classification as *false*.

If the examples are neither contradictory among themselves (*noisy*), nor to our assumptions about the hypothesis space, the resulting version space obviously has to steadily approach the actual target concept. However, this also means that with an empty inductive bias, each positive example will add none but itself to the ones henceforth classified as such, and vice versa with the negative ones.

It also does not help to seek some kind of majority among the hypotheses in the version space: If from the start, all are allowed, exactly those will be removed that classify any of the training examples wrongly.

³Notation and wording of algorithms in this report are usually taken from Mitchell's book.[4]

Among the others, for any hypothesis, there will always be a ‘partner’ that disagrees on exactly one example, thus always leading to a drawn vote.

On a more computational note: In more complex settings, the bounding sets can become quite large and their updates therefore slow. However, algorithmic complication is limited, again, in case we are dealing with the ‘conjunctive case’, where it suffices to treat generate version space elements by looking at constraints on a per-attribute basis.

2.2 Less Specific Hypothesis Space Restrictions

Sometimes, we are not able to sensibly define an inductive bias; we do not know of the special relationships between the attributes that make certain attributes or their values’ combinations more relevant to the outcome. As we have seen, using our previously described methods to learn the target concept would not yield usable results, as previously unseen examples will be classified as ‘don’t know’, and hypotheses will specifically include references to every single attribute.

2.2.1 Decision Trees

In these cases, one reasonable approach seems to be finding out which attributes are most important in determining whether an example belongs to the target concept or not. The attribute that gives us the most information about target concept membership could be found out by looking at the distribution of that membership within the subsets that result when we divide the training examples according to said attribute’s value: In the one most extreme case, knowing this value will already fully determine target concept membership, namely if we encounter only positive training examples for certain values and only negative ones for certain others. The other extreme would be that among each of the subsets, target concept members constitute exactly half the cardinality, meaning that knowing the value does not help much at this stage.

A means of measuring this *information gain* that knowing a certain attribute yields, is by use of Information Theory’s *entropy* concept: The entropy among the subsets belonging to a certain attribute’s values is weighted and summed, and compared to the same calculation’s result for all the other attributes. To be precise, if we call the set of training examples S , we define entropy as

$$Entropy(S) := \sum_{i=1}^c -p_i \log_2 p_i,$$

where p_i is the share of S classified as i .⁴ As ‘most important’ attribute, the A is chosen that maximizes

$$Gain(S, A) := Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v),$$

where $Values(A)$ are the possible values of A and S_v is the subset of S where A has value v . As entropy measures a distribution’s closeness to the equal one, it seems indeed reasonable to consider those attributes important that yield a partition whose elements have a high entropy in comparison to the whole set.

The ID3 Algorithm

Now there’s still the question how we could construct hypotheses (or a single one) using the above idea. An algorithm that tries to find (or approximate) the target concept by help of entropy calculations is ID3, which creates a so-called *decision tree*. [4, p.56] In this tree, nodes represent attributes and branches their values. ID3 puts the attribute with highest information gain as root and creates a branch for each of that attribute’s possible values. For each branch, now, the respective subset of training examples (which have the branch’s value for the chosen attribute) is considered and, again, among the remaining attributes determined which one yields the highest information gain in this setting. This is iterated recursively,

⁴Note that if $i > 2$ we are no longer strictly speaking about concept learning, and indeed the methods described in this section apply to other than boolean-valued functions.

continually partitioning the resulting subsets according to the values of the attribute that was chosen to be the current branch's 'next best'.

Branching ends once such a subset only consists of positive or negative examples (in which case the respective leaf is labelled accordingly), or once there are no more attributes to consider. The latter should, of course, coincide with the former reason unless our examples are noisy – if that is the case, we can label the leaf according to whether positive or negative examples form a majority among the remaining training examples.

It should also be clear that decision tree learning is only applicable when dealing with discrete-valued attributes and a discrete-valued target function. Real values can only be incorporated by dividing them into intervals that are then considered as discrete values in themselves. (These intervals, again, may be chosen to get an attribute with maximal entropy gain.) Another advantage is that the data may not only be noisy, but may also lack some attributes' values: Approaches how to handle this case in both training data and instances to be classified are summarized in [4, p.75].

Classification using Decision Trees

But back to our simple case: Once the tree is completed, each leaf has a label and a conjunction of attribute values assigned to it implicitly by the path leading to it from the root node. Every possible label can thus be seen as having a disjunction of such conjunctions designated to it. We can apply the tree to unknown instances by traversing it according to the given instance's attributes' values. Whatever the label of the leaf that we reach by doing this, this is the most sensible classification.

The question arises with respect to which inductive bias the classification will be 'the most sensible'. As opposed to our previous algorithms, we only get a single hypothesis, which – even when we have no noise in our training data – is bound to make some generalizations that are not necessarily correct: If we only have one training example, we get a 'tree' that classifies all unseen instances as belonging to the target function if and only if that training example also does. Because it stops as soon as it 'thinks' it can make a decision according to the training data, ID3 preferably produces short trees.⁵

Overfitting

However, we might choose to make them even shorter, as by chance, the training examples may lead to tree structures that overemphasize the importance of certain patterns specific to the training data, but not found in that to be classified. Hypotheses that could be changed in a way to suit the complete distribution of instances better at the cost of an increased error among training examples are said to *overfit* them. This problem is common to many learning algorithms. In the case of decision trees, countermeasures include heuristics to stop the tree-growing algorithm earlier or to post-process (*prune*) it, possibly by using an additional set of known instances (a *validation set* that has not been used as training data) in order to benchmark certain aspects of the tree.[4, pp.66-72]

Other Variations

One measure, though, which we can use before reverting to such considerable changes to our algorithm, is to modify our definition of 'the most important' attribute. This is because the use of $Gain(S, A)$ has one potential disadvantage: Certain attributes may divide the data so distinctly (e.g. a different value for each training example) that for the training examples, it yields a very high information gain, yet can be expected to perform poorly once we come to classification. We can counter this problem by using the so-called *gain ratio*, i.e. dividing an attribute's information gain by its entropy with respect to its values, thereby penalizing attributes with many uniformly distributed values:

$$GainRatio(S, A) := \frac{Gain(S, A)}{-\sum_{i=1}^c \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}},$$

⁵This preference, known as *Occam's Razor*, will find some justification in section 4.1.3.

where S_i are the sets of S whose value for attribute A is i . Alternatives to this method, as well as other changes to the formula to penalize attributes whose value may in reality be expensive to determine, are listed in [4, pp.74-76].

As depending on the number of training examples, it can be a very lengthy task to create a decision tree, methods have been developed to cut short the time needed to grow it: Among several such methods described by Dietterich in [5, p.106+] is the idea to use random subsets of the training data for the determination of the first most important attributes and then to continually increase those sets in the tree's lower levels – this should reduce the time required at hardly any cost in terms of accuracy. Another one is to randomly partition the training data, grow a tree for each subset (which can be done in parallel) and have the trees ‘take a vote’ once classification of new instances is needed.

In [6], Pazzani et al. note that sometimes, decision trees may erroneously classify according to rules that are incomprehensible or even contrary to existing knowledge about the topic. They outline methods to improve coherence with such knowledge and show that those are not detrimental to success in classification.

Very interesting from a practical point of view is the aspect Pazzani et al. point out in [7]: Decision trees (and practically all the other algorithms mentioned in this report) are usually designed in a way that minimizes the number of misclassified examples. However, instead of their sheer number, we might rather be interested in avoiding certain kinds of misclassifications as they cause higher costs than others. For example, a system that infers a medical diagnosis from symptoms in order to determine which department a patient should be sent to may be allowed to make mistakes that involve departments that are present at the respective hospital (if the diagnosis was wrong, another department can easily be consulted), but should be correct when it involves sending people to a specialized institution dozens of km away. In the cited paper, Pazzani et al. exemplarily describe approaches how for certain learning methods such costs could be taken into consideration.

2.3 Asking Learners

To now, we have considered cases where a set of training examples is given that the learner can not influence. If, however, it can ask queries, the process can be significantly sped up: Theoretically, the version space can be about halved with a fitting query, so the number of such steps required would only be logarithmic in its cardinality.

Determining the most useful queries may be difficult in general, but is quite easy in the following *group testing* example: Given n objects and the possibility to test groups of them for a certain property, we are to determine which objects, known to number at most r , have this property. A good strategy is to first test all: Either we are finished because the test turned out negative, so no objects have the property, or we go on and test both halves. This halving goes on recursively, and we stop this ‘descent’ once we get a negative test result or have determined a single object to have the property in question – instead of being linear in n , this method only requires a number of tests that is linear in r and logarithmic in $\frac{n}{r}$.

3 Neural Networks

Another approach¹ to develop a system that rather correctly classifies instances represented by attribute-value pairs sacrifices the demand for an ‘understandable’ hypothesis: *Artificial Neural Networks (ANNs)*. Inspired by the human brain, which consists of a web of interconnected neurons that ‘fire’ in case the sum of the positive or negative input gotten from other firing neurons exceeds a certain threshold, an ANN is, mathematically speaking, a graph whose vertices have thresholds and whose edges have weights assigned to them. Just as the brain is not self-contained, also an ANN has certain nodes whose ‘firing’ depends on external inputs, and others whose firing does not further propagate along edges but is considered an output, i.e. that yield a classification, trigger an action etc.

3.1 Perceptrons

As an example, let us take the boolean functions AND and OR, these can already be represented by just one such (computing) ‘artificial neuron’, called *perceptron*. Obviously, in both cases we need two input nodes, one for each of the boolean argument values – their representation be -1 for *false* and $+1$ for *true*. AND would then require the inputs’ weights to be chosen in such a way that both inputs need to ‘fire’ (i.e. be $+1$) in order to exceed its threshold: Weights $+1$ and $+1$, together with threshold $+1$, will do exactly that, as all other cases but $+1 + 1$ will not exceed the threshold. For OR, we simply change the threshold to -0.5 : Now, only the case $-1 - 1$ is ‘too bad’ to exceed it.

The XOR function, however, can not be represented by a perceptron: There is no combination of weights that could make the linear combination of two inputs exceed a threshold in the case of inputs -1 and $+1$ that is not reached by both inputs being identical. Mathematically speaking, with any set of instances of dimension n , there must exist a hyperplane that separates those instances into positive and negative examples – only then, i.e. when they are *linearly separable*, is a perceptron sufficient to represent the concept.

Therefore, we need more complex structures. Instead of these discrete-valued, thresholded perceptrons, ANNs are usually built using continuous functions like $(1 + e^{-y})^{-1}$ (where y is a linear combination of inputs²) that approximate thresholding (at 0, in this case), yet are differentiable, a fact whose importance will be shown further on in this chapter. Nodes with the above, the *sigmoid*, function are named after it and called *sigmoid units*.

3.2 Learning Process

The setting which we are going to treat in this chapter is that we start with an acyclic, directed graph (thus, input and output nodes are implicitly defined) with some almost arbitrary start values for the weights. We then get training examples, whose attribute values we assign to the input nodes and have them propagate to whatever intermediate (*hidden*) or output nodes they are connected to. For each such node, we first compute a linear combination of its respective inputs according to the initial weights, and then the sigmoid function of the result. This process is repeated until we have calculated the value of each output node: If these match the respective value(s) of our training example, we leave the network as it is,

¹The chapter *Concept Learning* was ended because the following methods, though applicable, are not limited to concept learning, defined by Mitchell as the inference of *boolean-valued* functions from training examples.[4, p.21]

²A constant input of 1 is implicitly assumed to also be there, in order to simulate the perceptron’s variable threshold.

as the weights seem to have correctly predicted the outcome. Otherwise, we change these parameters in a way that appears to be the right direction – what exactly this can mean, we will discuss later.

Assuming that we have a reasonable heuristic for changing the weights, the learning process leads to a continuous refinement of the ANN with each new training example. It now becomes clear why the hypothesis that the resulting network represents can not be necessarily expected to be ‘understandable’ as, for example, the conjunctions of attribute values that belong to leaves of decision trees: In big ANNs, it will usually be hard to understand the rationale behind specific weights. If they nevertheless *can* be understood, however, observed weights may provide interesting, perhaps previously unknown, insights into the treated subject.

3.3 Capabilities and Extensions

It turns out [4, p.105] that ANNs with even very few steps between in- and output units are surprisingly expressive. Boolean functions, for example, require only two: Representing them in disjunctive normal form (DNF), the first layer of hidden nodes³ can compute all the conjunctions, the second (the output layer, which takes the hidden nodes as inputs) the disjunctions. If we leave output values ‘unthresholded’, i.e. simply take the linear combinations of the respective inputs, a structure of two layers also suffices to approximate any bounded, continuous functions – and with an additional hidden layer even any function at all, although, of course, the number of nodes required per layer is bound to be enormous for certain ones.

For arbitrary tasks, we may not know which structure of ANN is suited best. Indeed, there are algorithms that expand an initially small network every time it becomes clear that the current size is not going to suffice, or that get rid of nodes if their absence only has a minor influence on correctness – the latter not only saves (the, compared to other algorithms or classification using ANNs, otherwise lengthy) training time, but can even have a positive impact on classification quality as it may reduce overfitting.[4, pp.121+]

Besides dynamical change to the ANN, another extension to our system would be to allow cycles in the graphs. As Mitchell [4, pp.119-121] explains, this can be utilized to solve certain problems in time series prediction⁴ faced when using ‘feed-forward’ networks: An acyclic ANN that predicts events based on data from the previous day could only take into account this one day, we would need to append a copy of the whole network to incorporate data from days further back in the past – unless we allow a cycle, feeding output data back in. Such approaches, however, need more complicated training rules than those we are going to deal with in this report.

3.3.1 Modular ANNs

Azam [8] emphasizes that from a computational point of view, ANNs can be implemented quite efficiently as their structure supports parallel execution of learning algorithms.[8, p.8] Once again inspired by nature, he explains quite popular extensions to the ANN concept that put an even greater emphasis on parallelism, namely *ensemble-based* and *modular* networks.

The former realizes the idea that several may be trained to solve the same problem – we have already seen this notion when talking about potential reductions to the time needed to grow a decision tree in section 2.2.1, and are going to revisit it in the context of a *weighted majority* vote among several algorithms in section 5.1.2. Here, however, we are not seeking the best-developed network, but attempt some sort of automatic interconnection of their results for the sake of optimization.

Modular ANNs, on the other hand, rely on the famous ‘divide and conquer’ concept and are designed by splitting the learning problem into several smaller tasks and having a web that re-combines the individual

³We speak of *layered* networks in case the nodes can be partitioned so that each subset’s nodes only get inputs from a specific other subset.

⁴A general introduction to *temporal-difference learning* is given in [3, 145+].

results, just as certain human cognitive processes also feature sub-operations. Apart from efficiency gains through easier parallelization and smaller partial networks, robustness is supposed to increase as weights from the different parts do not interfere with each other. A network's structural design can also be enhanced if for a certain sub-task, useful structures are already known.

3.4 Backpropagation Algorithm

Now let us come back to 'simple' ANNs. The above description of the learning process lacked any information on how the nodes' input weights could be changed in case a training example shows the network's output for a certain combination of inputs to be erroneous. The most intuitive way to do this is to, for each weight, determine the direction in which it would have to be changed to let the error be reduced. 'The error' on a training example d be defined as a function of the (vector of) weights \vec{w} :

$$E_d(\vec{w}) := \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2,$$

where outputs is the set of output nodes, t_k and o_k are the target and actual values of node k , respectively.

Obviously, trying to minimize the sum of errors over all training examples is infeasible for networks of useful size. The basic idea around this is, as indicated above, to, for every training example, compute the partial derivative of E_d with respect to every weight in the network and then to change the weight accordingly to minimize it – with a certain dampening, of course, in order not to have single training examples induce dramatic changes. Instead, it is rather important to let changes at one place influence others, which is why the described step is usually repeated many (e.g. thousands of) times per training example.

The derivatives' computation for this *gradient descent* requires knowledge of such derivatives further 'downstream', i.e. we start at the output nodes and then work our way back toward the input, hence the algorithm bears the name **Backpropagation**. As differentiation is involved, we can now also see that it was important to take differentiable functions instead of discrete ones. For a formal description of the algorithm and a derivation of its rules, consult [4, pp.97-103].

3.4.1 Problems

Before heading toward the next topic, however, we should take a quick look at certain risks involved with the **Backpropagation** algorithm. For example, although it is biased to smoothly interpolating the given data (which in itself can be a problem when intending to learn concepts that lack a certain continuity), overfitting can also occur here. Methods to counter it include the weights' initialization to small values and increased dampening of changes to them (in order to bias towards 'smoother' decision functions), and, as already described for decision trees, the use of a validation set: While the training examples are used to change the network, it is tested on the validation examples, and the update iteration is stopped once the validation set error has apparently reached its minimum. That minimum should, of course, not be a local one.

The problem of local minima also exists in the algorithm itself: Theoretically, the gradient that we compute to reach a lesser error may be pointing toward a local minimization of that error. Mitchell [4, p.104] describes two reasons why this may not be as big a problem as it sounds: First of all, remember that we defined the error as a function of all weights. Their sheer number will result in a high-dimensional error function that is likely not to have local minima for all weights in the same 'area', thus always providing some 'way out'. Also, local minima may only be expected in areas of the error function where high weight values represent quite complex functions – once the weights, previously initialized to small values, reach that area, local minima might even be acceptable.

Nevertheless, one may want to actively avoid local minima. Heuristic tools for this include the incorporation of the weights' past updates into the current one (keeping 'momentum'), or the training of multiple networks with different initial weights in order to later be able to either take the one performing best on a validation set, or combine their outputs by means of averaging. Also, the described version

of the Backpropagation algorithm is already less vulnerable to getting stuck in local minima because it dynamically updates the weights for each training example.

4 Bayesian Learning

To now, we have considered algorithms that lead to a specific classification for previously unseen instances once a number of training examples had been processed. As could be seen, these classifications were neither always right, nor was the preceding processing of training examples a procedure without certain choices. It seems obvious that sometimes it may be helpful to get several answers from an algorithm, along with an estimated probability with which they are judged to be correct. Which brings up the question in how far the algorithms described have, at all, produced the most probable hypothesis in the first place.

Bayesian learning is named after and based on *Bayes' Theorem*, which – in machine learning terms – expresses the probability $P(h|D)$ that hypothesis h has when we observe data D as a function of the probability $P(D|h)$ that we observe this data given said hypothesis holds:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)},$$

where $P(h)$ is called *prior probability* of h and may incorporate any reasons that lead us to judge its probability independent from any seen data, and where $P(D)$ is the prior probability of D .

For example, if we know a disease's probability ($P(h)$), the probability of the occurrence of a certain symptom in the whole population ($P(D)$) and in that part of the population that we know to suffer from that disease ($P(D|h)$), we can calculate the probability with which a person that features said symptom actually has the disease ($P(h|D)$). When we have a set H of hypotheses to choose from and all the required figures so that we can compute $P(h|D)$ for all hypotheses, there is at least one most probable, called *maximum a-posteriori (MAP) hypothesis*

$$h_{MAP} := \operatorname{argmax}_{h \in H} P(h|D) = \operatorname{argmax}_{h \in H} P(D|h)P(h).$$

(Here, $P(D)$ is dropped because it is constant. If we assume that a priori all hypotheses have the same probability, even $P(h)$ could be omitted; in this case we speak of the *maximum likelihood (ML) hypothesis* h_{ML} .)

An algorithm that does exactly these computations and this pick is called *Brute-Force MAP Learner*. As Mitchell [4, pp.168-170] shows, this could theoretically also be used find hypotheses that output the probability of a certain classification instead of a classification itself: He develops a formula expressing $P(D|h)$ for such hypotheses, whose maximum (assuming equal prior $P(h)$) then leads to the ML hypothesis.

In terms of quality, the *Brute-Force MAP Learner* can be seen as a 'perfect' benchmark, yet due to complexity issues is rather suited to prove other algorithms' quality than to be used itself. Examples for this will be shown in the following section.

Note, however, that ML hypotheses can also be calculated directly when certain restrictions to the hypothesis space are made, i.e. an inductive bias is given: This was shown in the lecture by means of an example where the given boolean attributes were risk factors and said restriction took place by limiting hypotheses to a certain formula, namely the product of parameters raised to the power of the attribute values. Due to $P(D|h)$ being differentiable, its derivative's roots could be calculated to yield those parameter values that form the ML hypothesis.

4.1 Quality Assessments using MAP Hypotheses

As 'brute force' MAP learning makes prior 'knowledge about the world' explicit by assigning probabilities, they do not have the 'hidden' inductive biases of other algorithms and are thus an objective means to

judge these others' quality – sensible probabilities provided, bias toward likelihood seems about the best one can imagine. In the following sub-sections, we will employ MAP and ML notions to determine certain supplementary rationales to learning paradigms.

4.1.1 Consistent Learners

Assuming that a priori, all hypotheses have the same probability, H is finite and contains the target concept, and our data is free of noise (i.e. $P(D|h) = 1$ if D is consistent with h , else 0), then the a-posteriori probability $P(h|D)$ is zero for all h inconsistent with D and $\frac{1}{|S_{H,D}|}$ – the latter should not surprise, as we initially considered all hypotheses equally probable and thus also each member of the version space now is, though with a higher probability as certain hypotheses turned out to be inconsistent with the data.

The point of the above reasoning was to see that with the given assumptions, every learning algorithm that outputs a hypothesis consistent with all training examples (a *consistent learner*) always outputs a MAP hypothesis. Though not explicitly dealing with probabilities at all, Find-S is such a one; in its case the given assumptions require that more specific hypotheses are more probable.

4.1.2 Sum of Squared Errors

If, however, our data is noisy, we like to use algorithms that try to minimize the error, which is usually defined as the sum over all training examples of the squared differences between a hypothesis' predictions and the actually observed values as part of the training data. Mitchell [4, pp.165+] shows that as long as the noise, i.e. the training data's deviation from the target function's values, obeys the standard normal distribution (and the examples are both fixed and their error terms mutually independent), the maximum likelihood hypothesis is the one minimizing said error, and vice versa. Note that for this justification of the squared errors approach to be correct, only the target values may be noisy, the attributes describing the instances must not.

4.1.3 Minimum Description Length Principle

One interesting way to express the MAP hypothesis is to minimize the negative sum of logarithms of the right-hand-side probabilities instead of maximizing their product. It thus becomes:

$$h_{MAP} = \operatorname{argmin}_{h \in H} -\log_2 P(D|h) - \log_2 P(h)$$

As $-\log_2 P(x)$ is the description length of x in an optimal encoding, h_{MAP} can be seen as minimizing the description length of h and D when optimally encoded together. With *Minimum Description Length Principle* we mean the suggestion to take the hypothesis h_{MDL} that minimizes the description length, as, for optimal encodings for h and $D|h$, $h_{MDL} = h_{MAP}$.

' $D|h$ ' can be seen as the information that a) tells about the attribute values of D and b) specifies in how far the given hypothesis h does not already classify D correctly – this latter piece of information can be empty in case the classification is indeed correct. Mitchell [4, p.174] notes that thus, the MDL principle “provides a way of trading off hypothesis complexity for the number of errors committed by the hypothesis” and names attempts to use this property in order to alleviate problems with overfitting.

The justification for Occam's Razor promised in section 2.2.1 and in a way obtained by showing the above relationship between probability and description length is limited in so far as Shannon's 'optimal' encoding usually remains a quite theoretical object.

4.2 Bayesian Classifiers

4.2.1 Bayes Optimal Classifier

In case we are rather interested in knowing a new instance's most probable classification than to know the most probable single hypothesis, using the MAP hypothesis for classification may be wrong as several less likely hypotheses may all come to a different conclusion and have a combined probability higher than that of the hypotheses that share the MAP hypothesis' classification. The, according to the described intent, 'best' classification value is called the *Bayes Optimal Classifier (BOC)*, defined as

$$v_{BOC} = \operatorname{argmax}_{v \in V} P(v|D) = \operatorname{argmax}_{v \in V} \sum_{h \in H} P(v|h)P(h|D),$$

where V is the set of classification values. Note that the hypothesis defined by all v_{BOC} potentially is not a member of H .

4.2.2 Gibbs Algorithm

A concrete computation of v_{BOC} may be too complex, thus it may be warranted to use the so-called Gibbs algorithm: Here, we only choose one h according to its posterior probability $P(h|D)$ and classify the next instance according to this h . As we shall see for the case of binary classification values, given already fixed observations D , the Gibbs algorithm guesses wrongly at a probability at most twice that for the BOC being in error: Be $p \leq 0.5$ the posterior probability of the less probable one of the two classification values, then BOC's guess would be right with probability $1 - p$, as it chooses the more probable value. Gibbs would choose the more probably value with probability $1 - p$ and the less probably value with p . As these are also their own posterior probabilities, we simply have to square and add them to get Gibbs' probability of success:

$$P_{succ}^{Gibbs} = (1 - p)^2 + p^2 = 1 - 2p(1 - p) \leq 2(1 - p) = P_{succ}^{BOC}$$

The largest difference between both algorithms actually occurs at $p = 0.25$, where $P_{succ}^{BOC} = 0.75$ and $P_{succ}^{Gibbs} = 0.625$. Mitchell [4, p.176] emphasizes that this result once again has an implication for the quality of other learners.

4.2.3 Naive Bayes Classifier

Generally, the most probable classification value for a given new instance described by attribute tuple \vec{a} can, using Bayesian terms, be described as

$$v_{MAP} = \operatorname{argmax}_{v \in V} P(v|\vec{a}) = \operatorname{argmax}_{v \in V} P(\vec{a}|v)P(v).$$

As usual with Bayesian brute-force approaches, the above computation is hardly feasible because, whereas $P(v)$ may be quite easily determined as the frequency of v in the training examples, it would require every instance to occur many times in our training data in order to draw reasonable conclusions about $P(\vec{a}|v)$. If, however, the component attributes of \vec{a} were conditionally independent from one another, the following substitution can be made that results in the *Naive Bayes (NB) Classifier*:

$$v_{NB} = \operatorname{argmax}_{v \in V} P(v) \prod_{i=1}^n P(a_i|v),$$

in which $P(a_i|v)$ can now also be determined quite practically using single attributes' values' frequencies in the training data.

The assumption underlying the naive classifier deserves its name as it is often not justified. Nevertheless, even when it is not, the method may work quite well, as shown in [4, pp.180-184]: Here, the specific and actually violated assumption is that the occurrence (frequency) of different words in a text is conditionally independent.

As was shown in the lecture, unjustified simplifications can sometimes be compensated: When trying to reconstruct an original word w from its 'scrambled' version s , we want to find the w that maximizes

$P(s|w)$. The simplification could then be to set this equal to the product of conditional probabilities $P(s_i|w_i)$, which, in case a Gray code was used for the single letters, may be reasonably estimated by the letters' distance in the alphabet. The information loss caused by only considering single letters may then be partially compensated by penalizing certain subwords that hardly or never occur in the language in question. Incrementally building up the most likely sequences, we then have a good chance to succeed in the reconstruction.

Another aspect we have to be careful about is not to let a certain attribute value's lack of occurrence make the product zero. One possible solution is to assume at least one occurrence. A more elaborate approach is described in [4, pp.179+]: Instead of calculating the occurrence frequency of an attribute's certain value c (their number be n_c) among all samples with a given classification value (their number be n) as $\frac{n_c}{n}$, we incorporate a prior estimate p on the values' probability and get the m -estimate of the wanted probability: $\frac{n_c+mp}{n+m}$, where m is a constant chosen the higher the more weight the estimate p should get in the calculation. p , in turn, is usually chosen to be uniformly distributed in case we have no special reason to use a different distribution.

4.3 Probabilistic Expert Systems

Instead of simply having an instance of an attribute tuple that may need to be classified, we may want to model situations in which we know of complex interdependencies between variables. Not only may many variables be connected in a causal chain or web rather than just having a single set of attributes influencing the probability of a single outcome – it would also constitute an extension to our previous thoughts if we were given such an 'outcome' and should judge on the potentially responsible attribute values.

Implicative networks have traditionally been called *expert systems*, as they usually were databases fed by experts with rules "if X , then Y " that were supposed to infer – taking the example of medical systems – diagnoses fast and without any necessity for their user to do more than insert known symptoms. As causalities are normally not as clear, it makes sense to use rules like "if X , then Y with probability p ". Depicting (random) variables as nodes and 'conditional causalities' as edges, a set of such rules can be represented as a graph. The rules can be inferred either from statistical data or from expert 'beliefs', hence the resulting graphs are often called *Bayesian Belief Networks*.

Setting up the network involves determining a *probability table* (or, more compactly, a function from which the table can be derived) for each variable. In case there are other variables that influence the one in question (i.e. *parent* variables), for all possible value combinations of those parent variables, a probability is given for each of the possible values¹ of the current one. If there are no parents, a general a-priori probability distribution among the values is given.

In a very simple example network, we may have the variables T (Thunder) and R (Rain). Assuming that at any given minute the probability of there being thunder is $P(T) = 1\%$ and that the probabilities for rain are $P(R|T) = 80\%$ and $P(R|-T) = 10\%$, we have a network that we can, in the simplest case, ask for the probability for rainfall given that we heard thunder (80%). But, and this is where the Bayes aspect comes in, we can also infer a revised probability for there being thunder once we know that it is raining:

$$P(T|R) = \frac{P(R|T)P(T)}{P(R)} = \frac{P(R|T)P(T)}{P(R|T)P(T)+P(R|-T)P(-T)} = \frac{0.8 \cdot 0.01}{0.8 \cdot 0.01 + 0.1 \cdot 0.99} \approx 7\%$$

When several 'outcomes' are known, their evaluation and the consequent revision of probabilities can work toward 'explaining away' potential causes.

There are two obvious questions that arise when dealing with Bayesian networks: How to determine the structure and/or probability distributions for the initial set-up (i.e. how to learn), and how to answer queries (i.e. determine certain variables' marginal distributions when given other variables' values).

¹In this discussion, we limit ourselves to discrete-valued variables.

4.3.1 Learning

As mentioned above, given a graph structure, the (quantitative) conditional probability distributions can, for example, be estimated using statistical data. Determining the (qualitative) structure itself, however, is more complicated, as a number of possible graphs has to be considered that is exponential in the number of variables. It is even harder to distinguish between cause and effect, which is why algorithms that converge on a most likely structure (like IPS or more efficient or otherwise improved approximative methods proposed by Wedelin [9, pp.315-320]) usually develop an undirected solution. Ibid., Wedelin suggests a method to find the most likely directions afterwards.

For this report, it shall suffice to say that basically, constructing a Bayesian network involves testing variables for mutual independence. The degree of sensitivity at which this takes place is important as, like in other learning models, there is a danger of overfitting the given data by ‘finding’ dependencies where there are none – one way to circumvent this is to restrict the number of parental nodes for each variable.

4.3.2 Efficient Inference

Ignoring computational complexity, the straightforward way to infer answers to queries would be to compute the joint distribution of all variables and then do summations to get the marginal probabilities of the query-specific variables required, e.g. for given $A = a$, $B = b$ and queried X to get

$$p(X|A = a, B = b) = \frac{p(X, A=a, B=b)}{p(A=a, B=b)}.$$

Although by intelligent summation (cf. [10, p.9]) the initial computation of the whole joint distribution (whose size is exponential in the number of attributes) can be avoided, it is preferable to use alternative methods. One could be to create samples with a Monte Carlo simulation that takes into account the known variables’ values and estimating the marginals accordingly.

Another involves converting the graph into its so-called *moral* version by omitting edges’ directions, connecting parental nodes and adding edges where necessary to eliminate cycles of length over three. As Lauritzen and Spiegelhalter [10, pp.9-16] describe in more detail, the distribution can then be expressed as a product of functions over each clique-forming set of variables, which, although depending on the intelligence of certain choices made, usually greatly enhances the otherwise exponential complexity.

4.3.3 The Use of Bayesian Networks

As one can imagine, the described ones are not all the problems faced when constructing a Bayesian network. For example, source data may be incomplete, in which case the up-to-then developed model can be used to infer the missing data, e.g. by employing the Expectation Maximization algorithm, which, as its name vaguely indicates, iteratively estimates the expected values according to the current ML hypothesis and then updates it.[4, pp.191-196] Yet it may also be that the absence of data has a reason (as opposed to being *missing at random*) that can be modelled using new variables in the network.

Once the learning is completed, however, a Bayesian network can be very useful. Inference, for example, can be enriched by an explanation why a certain ‘outcome’ was evaluated to be likely by looking at all variables’ estimated values instead of just one. This causality analysis can be complemented by a sensitivity analysis when certain parameters are marginally changed.

In case the network structure is algorithmically developed using source data, the process might even help discover previously unknown dependencies – remotely reminiscent of Neural Networks, yet certainly more understandable for people, as all nodes have a pre-determined meaning.

5 Various Topics

5.1 Probably Almost Correct Learners

In section 4.1.1, we concluded that consistent learners have a certain quality with respect to the available training examples, namely that they always output a MAP hypothesis. But in addition to knowing how well a certain algorithm does compared to what can be considered possible to infer from the data at hand, it would be good to be able to judge how well it does in ‘absolute terms’. By that we are henceforth going to mean the probability $1 - \delta$ with which a learner (‘probably’) outputs a hypothesis that, with another probability $1 - \varepsilon$ (‘almost’), correctly classifies instances that are drawn according to the same distribution that the training was based on.

A set of target concepts is therefore defined as *Probably Almost Correct*-learnable by a specific learner if it achieves the desired error bounds in time polynomial in δ^{-1} , ε^{-1} , the number of attributes per instance and the description length of the concept. It makes sense to allow marginal errors as with usually limited training examples, perfection can not be expected to be reached, anyway.

5.1.1 Bounds on Training Set Size

As a non-polynomial number of training examples would imply a non-polynomial learning time, a bound on that number exists for any PAC-learner and given probability parameters. In addition to giving an exact definition of PAC, Mitchell [4, pp.206-209] shows that for consistent learners, this bound is linear in ε^{-1} and logarithmic in both the hypothesis space’s cardinality and δ^{-1} .

Perhaps for large, but definitely for infinite hypothesis spaces this bound is useless. There is, however, another, similar bound that, instead of the hypothesis space’s cardinality, uses its *Vapnik-Chervonenkis (VC) dimension*, which is the size of the largest set of possible instances on which every possible concept is induced by some hypothesis.

For example, for learning an interval of real numbers, our hypothesis space may be infinite, yet its VC dimension is only 2: At least two, because when given two distinct numbers, one can always construct an interval containing any subset of them. Yet this is not possible for three, as there can be no interval that contains the extreme values, but not the one in the middle.

5.1.2 Bounds on Mistakes During Learning

Note that it may not only be relevant how many training examples are required to learn a concept reasonably well, but also to know its performance when used for classification during training, which could, for example, be the case with intrusion detection systems in computer networks.

Find-S, for example, when dealing with an hypothesis space made up of conjunctions of n boolean attributes, can make up to $n + 1$ mistakes: Assuming that the target concept is constant at 1, all training examples will be positive, and by definition, each time the algorithm obtains an example from which it can actually learn (i.e. change its current hypothesis), it will classify it wrongly. On [4, pp.220-225], Mitchell goes into more detail about this and other algorithms’ mistake bounds.

One interesting aspect mentioned there and in the lecture is that when we use several algorithms in parallel on the same training data and define their common classification as a *weighted majority* vote, the number of mistakes made by the whole system is linear in the number of mistakes made by the (in this respect) best of the participating algorithms and logarithmic in the number of participants. All we need to do is to initialize the weights equally and halve an algorithm's weight for each mistake it makes.

It should be mentioned, though, that this way of combining several learners to one is not necessarily the best: One example to obtain further improvement specific to the learning algorithm are modular ANNs (cf. section 3.3.1), another is the *Multiple Feature Subsets* for approach proposed by Bay [11] for Nearest-Neighbour Learning, the latter being subject of the following section.

5.2 Instance-Based Learning

Already in connection with artificial neural networks, it became obvious that when interested in classification, one may not necessarily care about having an explicit hypothesis. Even more so, the Bayes classifiers refrain from formulating one. This raises the question how we could directly infer classification of instances by looking at related ones in our training data.

5.2.1 Nearest-Neighbour Learning

In case we are dealing with instances that are characterized by tuples of real-valued attributes, we can apply distance measures and do a majority voting (or averaging, in the case of continuous-valued functions) on the classification among a certain number of training examples nearest to the new instance. This method is called *Nearest-Neighbour (NN) Learning* and obviously biased towards target concepts that hardly change locally, as such changes are easily ignored. If, however, there is few local variation, then this also means that noise will not seriously hamper NN learning.

Obviously, the larger the training set, the more computationally expensive the method becomes compared to classifications take with algorithms that have previously incorporated the training 'experience' in a hypothesis. It is therefore important to choose efficient data structures like sorted tables, hash-tables or (e.g. quad) trees.

There are at least two parameters NN algorithms have, one is the choice of distance measure: Whereas in geometric spaces, we may tend to use the euclidian distance, this prove entirely useless in other contexts. Assume the attributes in question are a geographic location's distance from the equator and elevation above sea level, both given in km, and we are striving to infer mean temperature values – obviously, the influence of elevation is greatly undervalued. To counter this so-called *curse of dimensionality*, it may be necessary to 'stretch' (or shrink) the axes, which can be done either manually or automatically by testing certain stretch factors' influence on the NN algorithm's resulting classification error using a validation set.

The other parameter is the number of nearest neighbours to be considered. Its choice should depend on a number of aspects, for example the relation between sample size and the target concept's complexity, which together determine how many neighbours we can actually expect to be representative for our unknown instance.

Optionally, though, the neighbours could be weighted according to their distance before using them to arrive at a classification. This would reduce the seriousness of the decision how many neighbours to consider: Theoretically, we could even take all training examples into account, but this is supposedly computationally impracticable.

5.2.2 Unification with Hypothesis Induction

Domingos [12] suggests a way between the alleged extremes of rough generalizations that may come along with hypothesis induction and NN approaches vulnerable to noise: Noting that instances can be seen as representing a most specific hypothesis, he describes an algorithm that, so he claims, combines 'the best of both worlds' by generalizing these 'training hypotheses'.

5.3 Determining Relevant Attributes

As we have already seen in the last section, it can be important to know which attributes are (most) relevant in determining the target value. Thinking of medical or technical risk factors makes it clear that the question is generally important, not 'just' concerned with algorithms' quality.

By definition, an attribute is *relevant* if there exists a pair of instances that only differ in that attribute and the target value. This definition could be expanded to measuring relevance quantitatively, i.e. according to the difference between the target values in relation to the change of the attribute in question by which the former is induced.

5.3.1 Learning from Queries

If a given boolean function is known to only have one relevant attribute (the so-called *dictator*), this can be found by application of the **Halving** algorithm: Depending on whether changing a certain half of the function's attributes changes the function's value, it can be seen which half the dictator is an element of. Now among the dictator's half, half of the attribute values are changed to see in which of the two quarters the dictator is actually located. This process can be iterated recursively with $\log_2 n$ queries until the single dictator is found.

In case we do not know the number of relevant attributes, this strategy fails, as general boolean functions of n variables are not determined by less than 2^n function values. If we do know the number of relevant attributes to be r , we call them a *junta*.¹

For r -juntas, the **Halving** algorithm fails, because whenever one half's values are changed, yet the function value does not, we can not know whether this was due to no influence on behalf of the changed attributes or whether those influences cancelled each other out. Instead, we need to find attribute tuples with different function values and then apply the **Halving** algorithm only to those attributes that differ in the two tuples, and replacing that one of the current tuples with the changed one which has the same function value.

For example, if $f(010) = 0 \wedge f(101) = 1$ and halving of the second tells us that $f(110) = 0$, we continue with the triples 101 and 110, knowing that one of the latter two attributes must be a member of the junta. The question remains how to efficiently get the initial two tuples. In order to be guaranteed to find all of the junta's members, we need not to consider all possible tuples, but only a set in which all subsets of cardinality r get assigned all possible bit combinations. Such a set is called *universal query set* and difficult to construct explicitly, yet it turns out that we can approximate it arbitrarily close by randomly picking a number of bit vectors that is exponential in r , yet only logarithmic in n , and thus feasible if the junta is small.

¹The analogy implied by this political term is a little flawed as this term is used for arbitrary boolean functions on these r variables, which implies that a 'yes' decision of two members of a junta may be made a 'no' by the additional agreement of a third member! Monotone boolean functions may rather fit the term, but learning such juntas is simpler than the general case described here.

5.3.2 Learning from Random Samples

In case we are not allowed to pose queries but only get a set of bit vectors randomly picked out of a uniform distribution, we have to revert to statistical evaluation: Dictators can be found out about by checking whether certain attributes value influences the probability of the function value being 1. For r -juntas, the same must be done with all r -subsets, which obviously requires an even larger sample size: The more attributes are relevant, the more samples do we require. This peaks when dealing with parity functions: They can only be identified as such when we have seen every possible bit vector.

If the sample distribution is not uniform, we have to normalize the conditional probabilities as

$$P(f = 1|d = b) = \frac{q(1,b)}{q(0,b)+q(1,b)} \text{ for } b \in \{0, 1\},$$

where d is the attribute to be checked for dictatorship. The d for which these probabilities' differences is maximal should be the dictator, independent from how often it actually has a certain value in the sample set.

5.4 Learning as Search: ADATE

Yet another kind of learning can be applied when computer programs, i.e. algorithms, are to be developed. Basically, the task is similar to the ones before in that programs can be seen as a kind of hypothesis that is applied to a training example and then tried to be improved in case it did not produce a satisfactory result. The system that is going to be briefly introduced in this section, *Automatic Design of Algorithms Through Evolution (ADATE)* starts with a practically empty program and keeps applying transformations to it until it has the intended functionality, like sorting a list.

A simple approach could be to consider programs as nothing but bit sequences. As we are essentially looking for a certain bit sequence that will 'do the job', one could naively generate 'all' programs with ever-increasing length until the right one is found. Even if the problems arising from non-terminating programs were ignored, this method would be far too computationally expensive.

ADATE's approach is to incrementally compose programs in a programming language, for reasons of complexity in ML, a functional one, in which mis-typed programs can quickly be dismissed. Given a specification consisting of in- and output types as well as training examples, once an initial framework is constructed, the execution of one of several transformations and the application of an evaluation function to the resulting program's output take turns. Like the naive search, ADATE has a bias toward shorter programs.

The transformations include the *replacement* of one expression with another, the separation of sub-functions (*abstraction*)², the factoring out of case expressions (*case distribution*)³ and what is called *embedding*, the changing of types, addition of arguments etc. Only the first transformation actually semantically changes the program, which is why the others only make sense in combination with an abstraction step.

The evaluation that is applied after the program has semantically changed not only considers the program's output on specific input and how far that deviates from the aim, but also syntactic complexity and computational duration as penalty factors. Also, the evaluation function should propose transformations to be made and rank them. As there may be several ones considered promising to improve on the current program instance, instead of a purely linear evolution, a *beam search* is conducted: Several possibilities are tested in parallel, yet no backtracking is performed. For the sake of complexity, the number of 'active' nodes in the search tree is limited.

This 'Darwinist' aspect is quite extraordinary, as we have not encountered it when dealing with the other topics. But whereas those others usually took some profoundly reasonable step toward optimization, the

²In ML this is done using the `let` expression.

³I.e. a function which has a case expression as argument is made subject to the case distinction.

evaluation function's ability to predict which transformation is most likely to succeed is quite limited, so ADATE has to 'bet on several horses'.

Although ADATE has been shown to master the creation of sorting algorithms (naturally, the handling of lists is a task suited well for functional languages), setting up the specifications required to start the search for an algorithm performing a certain task can be very complex. It hardly needs to be mentioned that a search as general as performed by ADATE takes quite long. Thus, for the moment at least, it seems far-fetched to believe that the development of algorithms may one day be performed automatically, yet the idea is without doubt fascinating and certainly very popular with science fiction writers.

5.4.1 Inductive Logic Programming

A seemingly similar task is taken on in *Inductive Logic Programming*, whose aim it is to learn logical functions, usually in logical programming languages like Prolog. Yet on closer inspection, ADATE is far more ambitious: In its basic form presented by Nilsson [3, p.97+], ILP is not about outputting unification results etc. that, for example, enable prolog to sort lists, but limited to learning functions that simply yield a truth value. This enables approaches similar to version spaces and bounding sets, starting off with two 'programs' that constantly yield 'true' and 'false', respectively, and progressively developing them toward a function 'in between' which only yields 'true' when fed with the right arguments. Quite obviously, this is easier than trying to learn arbitrary algorithms.

5.5 Learning Phylogenetic Trees

In the past chapters, we have dealt with attempts to infer ways to classify new instances with respect to experience gained from training examples. A different kind of task that can be imagined is to find structures in given data and possibly infer data that is, in a way, 'missing'. In biology, for example, it is helpful to reconstruct ancestral structures (*phylogenetic trees*) among given samples of genetic information, thereby fitting in any potentially missing links.

In the setting we are going to look at it, learning phylogenetic trees means that given a number of bit vectors, we want to infer an ancestral tree of incremental changes that fits the given data, with the potential addition of missing vectors.⁴ To be precise, we are interested in *Perfect Phylogenies (PP)*, i.e. bit vector trees where each bit changes along one edge only.

The lecture presented Gusfield's algorithm, whose idea is to first consider only the first bit component in all the vectors, find a solution for that (i.e. a 'tree' with two nodes labelled 0 and 1) and then continually expand this by considering the next bit, until all are incorporated. 'Considering' means appending the new bit to the respective existing node. If, for example, there are bit vectors starting with 00 and 01, there are two possible continuations for node 0, so we split it in two, connect them with an edge and connect 0's former edges to the continuation that has the same new bit as the node to be connected. In case all continuations are uniquely determined, we may have to artificially create a second continuation at places where neighbouring nodes would otherwise differ by more than two bits following our extension step. This artificial continuation creates the 'missing links' that our data lacks to forming a perfect phylogeny.

It can be shown that two continuation 'collisions' in the same step mean that there is no PP to the given data. If we do not have to insert missing links, the tree is unique, and in any case found by the algorithm in time linear in m and n , where n is the number of instances and m the number of bits per vector.

In case we allow some bits (namely k with $k \ll m$) to flip several times, we can change our agenda to identifying these and reconstructing the PP for the remaining ones. For this task, it helps to write the input as a matrix, whose rows are the vectors. The 'two collisions in one step' mentioned above are then characterized by having a pair of columns in the matrix that contains all combinations 00, 01, 10 and 11,

⁴Note that just any leaf of the resulting tree could be its root, as we are given no information on the samples' age.

i.e. are called *complete*. If we could identify these columns, they would exactly represent the bits that we then have to ignore in the PP construction.

Interpreting the columns as nodes in a graph that are connected by edges if and only if their corresponding pair of columns is complete exposes the problem as that of finding *k-vertex covers* in this graph. By removing all vertices of degree $> k$ we can speed up the otherwise exponential time required to find the cover, because the remaining graph's size only depends polynomially on k .

5.6 Unsupervised Learning

The odd-sounding term *unsupervised learning* refers to cases where at the beginning of the learning process, it is not completely clear what specific classifier should actually result from it. Nilsson [3, p.131+] gives the example of learning hierarchies of clusters: Being given a number of points by their coordinates, the task is to classify them according to which cluster they belong to, but the clusters are not yet defined. In this case, the learning algorithm needs to group the points and later, or – preferably – in parallel, train classifiers for them. An extension to this particular problem would be to allow the algorithm to find a complete hierarchy of clusters, i.e. to find clusters of mutually close sub-clusters.

Clustering seems like a very narrow example, but in fact, grouping according to similarities is the essence of categorization, so the distance metric on which the unsupervised algorithms base their judgement on how to define clusters ‘simply’ needs to be adjusted to the respective problem. For instance, an advertising agency may want to create an hierarchical database of professional speakers’ voices in order to be able to quickly find the voice suited for a specific advertisement. Depending on the subject, however, it may still be a considerable task to define similarity, e.g. in the agency example, to find out what classification by which voice features would later enable the employees to intuitively browse the database.

5.7 Contrast Sets

The last section dealt with automatic clustering of training examples. What if the clustering according to one attribute were clear, but we wanted to infer information on what effect this has on the other attributes, i.e. in how far the two sets contrast besides the attribute by which they were split?

Bay and Pazzani [13] call this inference ‘mining’ of *contrast sets*. In order to emphasize the difference between this approach and that of unsupervised learning, let us refer to a variation on their example of student statistics: Suppose we wanted to infer interesting conclusions from data on a student population, unsupervised learning would be good at classifying ‘sorts’ of students (heavily dependent on the employed distance metric, he may or may not find the stereotypes ‘social biologist’, the ‘crazy physicist’ and the ‘elitist economist’). Mining contrast sets, however, would mean that we pre-partition the students, e.g. according to their subject, and then infer in how far this affects attributes like talkativeness, craze or snobbism.

6 Conclusion

In my mind, the course left three main impressions: First of all, it opened up a huge toolbox with various learning approaches and information on their suitability for certain learning aims. Independent from whether I expect to require all of them in the coming years, it was certainly interesting to, for example, finally get an idea of what Artificial Neural Networks are. They, together with the automatic generation of Bayesian Networks and the ADATE system, constituted the group of the most intriguing approaches because they appear to be very powerful – always assuming that the algorithms used actually finish in one's lifetime.

Talking of which: The second important aspect naturally were the algorithms presented for those respective topics, as they gave an idea of all the intelligent tricks that can be applied to improve learning efficiency. Complexity understandably is the 'arch-enemy' when trying to make the computer learn concepts and structures, and both the book and browsing the web have shown that the researchers' newest weapons certainly go way beyond what was shown in this introductory lecture.

Thirdly, certain problems with almost all the algorithms like only vaguely specified inductive biases or overfitting were such a recurring feature that anyone having attended the course – and written the summary report – will certainly not forget about those when, in the future, trying on their own account to get computers to 'learn'.

I very much appreciate the broad spectrum of topics covered; this and the usually¹ very intuitive way they were presented are certainly worth recommending to fellow students. But unfortunately they will never be allowed to read this text...

¹Some Maths is always bound to spread confusion...

Bibliography

- [1] P. Dupont: *Inductive and Statistical Learning of Formal Grammars*, 2002
www.info.ucl.ac.be/people/pdupont/pdupont/pdf/GrammarInduction2p2p.pdf
- [2] B. Starr, M. Ackerman, M. Pazzani: *Do I Care? – Tell Me What’s Changed on the Web*, AAAI Spring Symposium, 1996
www.ics.uci.edu/~pazzani/Publications/mlia96-bstarr.ps
- [3] N. Nilsson: *Introduction to Machine Learning*, 1996
robotics.stanford.edu/people/nilsson/mlbook.html
- [4] T. Mitchell: *Machine Learning*, McGraw-Hill, 1997
- [5] T. Dietterich: *Machine-Learning Research – Four Current Directions*, AI Magazine 18/4, 1997
www.aaai.org/Library/Magazine/Vol18/18-04/Papers/AIMag18-04-010.pdf
- [6] M. Pazzani, S. Mani, W. Shankle: *Comprehensible Knowledge-Discovery in Databases*, Cognitive Science Conference, 1997
www.ics.uci.edu/~pazzani/Publications/CogSci97.pdf
- [7] M. Pazzani et al.: *Reducing Misclassification Costs*, 11th International Conference of Machine Learning, 1994
www.ics.uci.edu/~pazzani/Publications/MLC94.pdf
- [8] F. Azam: *Biologically Inspired Modular Neural Networks*, Dissertation at Virginia PI&SU, 2000
scholar.lib.vt.edu/theses/available/etd-06092000-12150028/unrestricted/etd.pdf
- [9] D. Wedelin: *Efficient Estimation and Model Selection in Large Graphical Models*, Statistics and Computing 6, 1996
- [10] S. Lauritzen, D. Spiegelhalter: *Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems*, Royal Statistical Society, 1988
- [11] S. Bay: *Combining Nearest Neighbour Classification from Multiple Feature Subsets*, International Conference on Machine Learning, 1998
www.isle.org/~sbay/papers/mfs_icml98.pdf
- [12] P. Domingos: *Rule Induction and Instance-Based Learning: A Unified Approach*, 14th International Joint Conference on Artificial Intelligence, 1995
www.ics.uci.edu/~pedrod/ijcai95.ps.gz
- [13] S. Bay, M. Pazzani: *Detecting Change in Categorical Data: Mining Contrast Sets*, 1999
www.ics.uci.edu/~pazzani/Publications/stucco.pdf

I hereby declare that I have written this report myself, i.e. that its wording is my own unless otherwise stated.

Urs Enke

Göteborg, 14 March 2004
