Machine Learning and Inductive Inference

Course text, 2014-2015

Hendrik Blockeel

Contents

Preface

This text accompanies the course "Machine learning and inductive inference" that is taught at the Katholieke Universiteit Leuven to students from several study programmes, including Master in Informatics, Master of Engineering (Computer Science), and the Master of Artificial Intelligence program.

Until 2007, the textbook *Machine Learning* by Tom Mitchell was recommended to students following the course. Due to an increasing divergence of the lectures from what was covered in the book, the book has not been used anymore from 2008 onwards. This course text is meant to replace it.

The text provides detailed background information on most of the subjects discussed during the lectures, and is mostly structured in the same way as the lectures.

Besides this course text, copies of the slides are available separately. These offer a more precise account of what has been discussed during the lectures, and form in many cases a concise summary of what is discussed in this text. The text and slides can be used in a complementary manner when studying the course.

This text has benefited from discussions, suggestions and proofreading by Christophe Costa Florencio, Kurt Driessens, Elisa Fromont, Fabian Güiza, Pieter Hiele, Eirik Jensen Opland, Siegfried Nijssen, Leander Schietgat, Stijn Storms, Jan Struyf, Anneleen Van Assche, Guy Van den Broeck, Celine Vens, and several students. The chapter on Reinforcement Learning was contributed by Kurt Driessens. My sincere thanks to all of them.

Hendrik Blockeel
Leuven, January 2010

Chapter 1

Introduction

1.1 Machine Learning

Machine learning is the subfield of artificial intelligence in which artificial learning behavior is studied.

We say that a computer program has the **ability to learn** if it has the capacity to improve its own performance at solving certain problems after receiving additional information about the problem. This additional information often consists of observations of how particular instances of the problem were solved in the past (it is then called *experience*).

To define learning more precisely, we need to be precise about the task that has to be performed, how we measure performance on that task, and what kind of information is used by the system. Tom Mitchell, in his textbook *Machine Learning* [26], considers a learning problem *well-posed* if this information is clearly specified. This leads to the following more formal definition of learning behavior [26]:

> Given a task T, a performance criterion C, and experience E, a system *learns from* E if it becomes better at solving task T, as measured by criterion C, by exploiting the information in E.

Machine learning is quite central in artificial intelligence, as the ability to learn is usually considered an essential part of intelligence. But it is also playing an increasingly important role in today's software, including software that is not traditionally associated with artificial intelligence. Machine learning provides supporting technology in areas such as information extraction, information retrieval, adaptive user interfaces, intelligent agents, robotics, etc.

Example 1.1 Recommender systems are programs that, based on a user's visible interests, try to recommend products or services that the user may be interested in. The classical examples are music or book shops on the Internet, where users who search for certain items receive a number of suggestions for other items they may be interested in. These suggestions are often of the type "users who bought this item also bought ...", but they can also be based on more complicated reasoning. Through analysis of the activity of this user and other users, the recommender system tries to learn about the interests of this user, and to learn about how certain interests relate to certain items.

Example 1.2 In computer games, computer-controlled characters usually behave according to some pre-programmed artificial intelligence. Their strategies and tactics, while clever, are often fixed, and once the player finds out about them, he or she can easily exploit this knowledge; the adversaries become predictable. This can

be turned around as well, however: poker bots, for instance, try to learn about the psychology of their adversaries (is it an aggressive or rather cautious player, etc.). They build a model of each individual player and try to exploit that model to better guess when, for instance, a player is bluffing.

In a more advanced type of learning, the computer continuously adapts his strategy or tactics to a particular user. When a user finds out about particular tactics employed by the computer, the user may adapt his behavior to counter these tactics; noticing this, the computer may again adapt its tactics to the user's adapted tactics, and so on.

On the other side, machine learning draws heavily on techniques and theory from artificial intelligence, statistics, optimization theory, and in some cases also biology and psychology (in which natural learning behavior is studied). It is also closely related to the field of data mining, with which it shares many techniques for data analysis. These properties make machine learning an inherently multidisciplinary field.

1.2 Structure of the text

The strong multidisciplinarity of the field of machine learning is clearly visible in the structure of this text. An attempt has been made to cover the broad variety of approaches that exist in machine learning, with the aim of providing a structured overview of the field. A direct consequence of this broad orientation is that not all subjects can be covered in depth.

The approach chosen in this text is to focus more on those approaches that are not treated in depth in separate courses at the K.U.Leuven. In particular, since separate courses are available that are devoted entirely to, for instance, artificial neural networks or support vector machines, these subjects are not covered in much depth; the chapters on these can be seen as high-level introductions to those other courses. Other subjects are covered in a bit more depth.

Even with the limited depth that this text provides, some chapters require a fair amount of background knowledge in certain areas. Unfortunately these areas differ strongly from one chapter to another: some chapters rely more on statistics, mathematics, or linear algebra; others rely more on informatics, construction of algorithms, or computational logic. The approach taken in this text is to always try to provide first some level of intuition to the reader. In some cases the intuition is backed up by references to other fields. Such references are mainly intended for those readers who are already familiar with those fields, and aim at facilitating the interpretation of this text for them; readers not familiar with those fields can simply ignore these references. It is hoped that in this way, the text is maximally accessible to a broad audience.

The text starts with a discussion of the inputs and outputs of machine learning systems (Chapter 2). This provides a context in which many learning algorithms can later be situated. Further chapters will discuss in more detail specific learning tasks, algorithms, or approaches, with the exception of Chapters 6 and 9, which discuss topics that are more globally relevant for machine learning.

Chapter 2

Input and output knowledge

From the information processing point of view, machine learning is just a process that takes a certain kind of knowledge as input, and produces another kind of knowledge as output. Depending on the context in which it is used, a machine learning tool may be confronted with different types of input knowledge, and with different types of requested output knowledge. The concrete machine learning algorithms that can be used largely depend on this.

Generally, we can see a machine learning system as a function L that, given some input knowledge $I \in \mathcal{I}$ and parameter settings $p \in \mathcal{P}$, produces an output (called a **model** or **hypothesis**) $O \in \mathcal{O}$:

$$L : \mathcal{I} \times \mathcal{P} \to \mathcal{O}$$

In practice, we will often make abstraction of the fact that such a system has parameters that control its behavior, and we will assume, when referring to a machine learning system, that its parameters have been filled in in a particular way (possibly simply using default settings). Thus we get a simpler mathematical description of a machine learning system as a function mapping inputs I to outputs O:

$$L : \mathcal{I} \to \mathcal{O}$$

2.1 Types of learning: predictive, descriptive, transductive, inductive

We can distinguish predictive and descriptive learning settings. We will discuss this distinction later; but here, it is relevant to remark that in the **predictive learning** setting, the ultimate goal is to be able to make predictions for *new cases*, different from what we have observed previously, whereas in the **descriptive learning** setting, the output should describe (not trivially visible) properties of what we *have* observed.

In the predictive learning setting, the predictive model O is often a function mapping instances from some input space $\mathcal{X}$ to some output space $\mathcal{Y}$, i.e., $\mathcal{O} = \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} \to \mathcal{Y}$ represents the set of all functions from $\mathcal{X}$ to $\mathcal{Y}$. In descriptive learning, $\mathcal{O}$ is less constrained: descriptive models can have almost any format.

The above setting for predictive learning, where the output is a function, is called the **inductive learning** setting. There is another setting for predictive learning, called **transductive learning**, where the goal is a bit less ambitious. In transductive learning, the learner is given a dataset together with one or more instances for which a prediction needs to be made, and the task is to make these predictions. It is not required, in this case, that a generally applicable predictive

model (a function that could make a prediction for any instance) is learned; we simply want predictions for the given instances. Transductive learning is thus a weaker version of predictive learning.

To understand better the relationship between transductive and inductive learning, let us look at the typical signature of transductive and inductive learners. An inductive learner L takes as input a dataset $I \in \mathcal{I}$ and produces as output a function from $\mathcal{X}$ to $\mathcal{Y}$, i.e., $\mathcal{O} = \mathcal{X} \to \mathcal{Y}$; hence we have

$$L : \mathcal{I} \to (\mathcal{X} \to \mathcal{Y})$$

For a transductive learner $L' : \mathcal{I}' \to \mathcal{O}'$, the instance $\mathbf{x}$ for which a prediction is to be made is part of the input given to the learner (so we have $\mathcal{I}' = \mathcal{I} \times \mathcal{X}$), and its output is simply a prediction y for $\mathbf{x}$, i.e., $\mathcal{O}' = \mathcal{Y}$; so we have:

$$L : \mathcal{I} \times \mathcal{X} \to \mathcal{Y}$$

where $\mathcal{I}$ is the input space for the corresponding inductive learner.

(The above signature assumes that a prediction is to be made for only one instance; it can easily be extended to allow for a sequence of $\mathbf{x}$ instances to be given, and a sequence of y values output.)

In the remainder of this chapter, we will mostly discuss the inductive learning setting. When a remark is relevant for the transductive setting, this will be explicitly mentioned.

In the following, we first sketch the different kinds of input knowledge $\mathcal{I}$ that learners may use; next, we discuss the different kinds of output knowledge $\mathcal{O}$ that the learning process may result in.

2.2 Format of input knowledge (data)

The most often used setting is when the input knowledge is described as a data set T containing elements $\mathbf{u}$ from a universe $\mathcal{U}$, and $\mathcal{U}$ is the cartesian product of a number of attribute domains A_i. This is the so-called **attribute-value learning setting**. We will discuss this setting first. Next, we will extend the setting towards cases where $\mathcal{U}$ has a more complicated structure, i.e., elements of $\mathcal{U}$ may be sets, trees, graphs, etc.

A note on notation: here, and in the remainder of the text, we use the convention that scalar values are written in italics (e.g., y), while vectors, tuples, or other composite values are written in boldface (e.g., $\mathbf{x}$, when $\mathbf{x}$ is a vector).

2.2.1 Learning from Attribute-value data

In the attribute-value learning setting, the following assumptions are made.

- The dataset is a set T of objects $\mathbf{u}_i \in \mathcal{U}$. $\mathcal{U}$ is called the **instance space**, the universe, or (as in statistics) the population. T is called the **training set**. We often assume a finite T, and denote the number of objects in T as N. The elements of T are called **examples**.

- Each object $\mathbf{u}_i$ is an element of the same domain $\mathcal{U}$. $\mathcal{U}$ is the product set of a number of attribute domains A_i: $\mathcal{U} = A_1 \times A_2 \times \ldots \times A_D$. We call an element of such a product set a **tuple**. An **attribute** is a function mapping $\mathbf{u}$ on one of its components. We will here denote the function that maps a tupel $\mathbf{u}$ onto its i'th component as A_i. (Thus, the notation A_i will be used to refer both to the attribute itself and to its domain.)

An additional assumption made by many learning methods is that the objects are **independent and identically distributed** (usually abbreviated as **i.i.d.**). This means that the dataset has been generated by randomly drawing N elements from $\mathcal{U}$ according to some fixed probability distribution $\mathcal{D}_\mathcal{U}$ over $\mathcal{U}$. Note that this assumption does not relate to the format of the data, only to the way the data set is generated.

The term **attribute-value learning** refers to any learning method that learns from data represented in this format. It is sometimes also referred to as **propositional learning**. This type of learning is probably the most widely used approach in machine learning and data mining.

Predictive learning

In attribute-value learning, learning a predictive model typically means that a model is learned that predicts, for a given object, the value of one specific attribute A_t (called the target attribute) from the values of all other attributes. Without loss of generality, we here assume that the target attribute A_t is listed last, that is, $t = D$. Thus, a **predictive model** represents a function

$$f : A_1 \times \ldots \times A_{D-1} \to A_D.$$

Earlier we said that in the inductive learning setting, we learn a function from $\mathcal{X}$ to $\mathcal{Y}$. Clearly, in this case, we have $\mathcal{X} = A_1 \times \ldots \times A_{D-1}$ and $\mathcal{Y} = A_D$.

When the target attribute is nominal,[1] the learning process is usually called **classification**. Sometimes the term classification is used for the specific case of binary classification, where A_t has only two values (typically called *positive* and *negative*); the term **multiclassification** is then used when A_t has more than two values. When the target attribute is boolean, the term **concept learning** is sometimes used. For ordinal target variables, the term **ordinal classification** is sometimes used. When the target variable is numerical, learning a model for predicting it is called **regression**.

We can generalize this predictive learning setting to the case where not one, but multiple attributes are predicted. Assuming that we predict the last k attributes, this means we learn a function

$$f : A_1 \times \ldots \times A_{D-k-1} \to A_{D-k} \times \ldots \times A_D.$$

The prediction space $\mathcal{Y} = A_{D-k} \times \ldots \times A_D$ now no longer contains scalar values, but k-dimensional tuples. The components of these tuples may be reals, booleans, nominal or ordinal variables, or a combination of these. This setting is often called **multi-target prediction**. In the special case where all targets are boolean, this setting is also called **multi-label classification**; it is then equivalent to predicting a set of nominal values ("labels") for each instance (there is one target attribute for each possible label, which takes the value *true* if that label is in the set and *false* otherwise).

Example 2.1 Table 2.1 shows an example of a data set in tabular (or "attribute-value") format. A number of animals are described. For each animal the table lists the name of a particular animal, the species it belongs to, its "type", size, covering, and the way it moves.

In predictive learning, the task might be to predict how an animal moves, given information about its other properties. Assuming that the name of an individual

[1] Terminology: a variable is **nominal** if it takes values from an unordered set of values (e.g., {`red,green,blue`}). A variable is **ordinal** if it takes values from a totally ordered set of values (e.g., {`small, medium, large, extra large`}). A variable is **numerical** if it takes values from (a subset of) the set of real numbers.

Name	Species	Type	Size	Covering	Movement
Tweety	canary	bird	small	feathers	flies
Pluto	dog	mammal	medium	hair	walks
Jumper	horse	mammal	large	hair	walks
Pingo	penguin	bird	small	feathers	walks
Tsjilp	sparrow	bird	small	feathers	flies
Jerry	mouse	mammal	small	hair	walks
Tom	cat	mammal	medium	hair	walks
Dumbo	elephant	mammal	large	hair	flies
Snowy	dog	mammal	medium	hair	walks

Table 2.1: The "animals" dataset.

animal is not an inherent property of it and should not be used to make the prediction, we can say that what we want to learn is a function f: Species $\times$ Type $\times$ Size $\times$ Covering $\rightarrow$ Movement. An example of such a function is the rule

if Covering = feathers **then** Movement = flies **else** Movement = walks

This function maps any instance for which the attribute Covering has the value *feathers* to *flies*, and any other instances to *walks*. Comparing this function to the dataset, we can observe that the function is mostly, but not entirely, consistent with the data.

Besides the above rule, we could think of other representations of functions, for instance:

if $3 * I_{Covering=feathers} + 2 * I_{Size=small} > 3$ **then** Movement = flies **else** Movement = walks

where the I_C are so-called Indicator variables, which are 1 if C is true and 0 otherwise. This rule, for Tweety, would give a result of 5 for the computation, which is greater than 3, and thus predict that Tweety flies.

An example of a multi-target prediction problem would be: learn a function f: Species $\times$ Type $\times$ Size $\rightarrow$ Covering $\times$ Movement that predicts both the Covering and the Movement of an animal from its other properties. An example of such a function is:

if Type = bird **then** Size = small **and** Movement = flies
else if Species =elephant **then** Size = large **and** Movement = flies
else Size = medium **and** Movement = walks

The above example shows a few simple cases of predictive learning in the attribute-value context. We will see many more examples in the remainder of this course.

Descriptive learning

One can also learn descriptive models. A descriptive model is not necessarily a function that maps an element **x** onto some other element **y** (although it may be representable in that form as well).

Clustering is an example of a descriptive task. In clustering, the objects are grouped into clusters of similar examples. This is in a sense similar to classification, but now the "target variable" that we predict is not one of the A_i but an "external", unobserved, variable, which indicates the cluster the example belongs to. For this reason, clustering is also called *unsupervised learning*, as opposed to classification which is then considered *supervised learning*.

Given a data set T, a clustering may be represented as a partitioning of T: $\{C_1, C_2, \ldots, C_k\}$ where each C_i is itself a subset of T. We could represent the clustering by listing the C_i, but we can also, equivalently, represent it as a function mapping each element of T (or, by extension, each element of $\mathcal{U}$) onto a number from 1 to k that indicates the cluster the element belongs to:

$$\mathcal{O} = \mathcal{U} \to \{1, 2, \ldots, k\}$$

In this format, the link between clustering and predictive learning is more clear: clustering can be seen as learning a predictive function where the variable being predicted was not part of the original dataset (hence, unsupervised learning).

Note that when the domain of the function is $\mathcal{U}$ rather than T, we have learned cluster descriptions that do not just tell us which elements of T are in which cluster, but more generally, which elements of $\mathcal{U}$ are in which cluster. The clustering then generalizes towards the population.

An other example of a descriptive setting where the output cannot be described as a function from $\mathcal{X}$ to $\mathcal{Y}$, with $\mathcal{X}$ and $\mathcal{Y}$ corresponding to one or more attributes, is **frequent itemset discovery**. In this setting, all attributes are assumed boolean (and they are usually called *items* in this context). An itemset is a set of attributes, and the frequency of an itemset is the number of instances in T for which all the attributes in the itemset have the value true. The task is to compute for each itemset with a frequency above a given threshold, that frequency. The output space can now be described as

$$\mathcal{O} = 2^{\mathcal{A}} \to \mathbb{N},$$

where $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ is the set of all attributes, and $2^{\mathcal{A}}$ is the power set (the set of all subsets) of $\mathcal{A}$, i.e., the set of all itemsets.

Summarizing this formally

Let us place the above in the context of our definition of a learning system as a function $L : \mathcal{I} \to \mathcal{O}$.

In attribute-value learning, the input space $\mathcal{I}$ is defined as follows:

- $\mathcal{I} = 2^{\mathcal{U}}$, with $2^{\mathcal{U}}$ the power set of $\mathcal{U}$ (indeed, the input is not a single element $\mathbf{u}$ but a set of such elements)
- $\mathcal{U} = A_1 \times A_2 \times \ldots \times A_D$
- each A_i is assumed atomic (i.e., not composed of sub-attributes) and is nominal, ordinal, or numerical

In words: a learner takes as input a set of instances (elements of $\mathcal{U}$) that all have the same format, namely, a D-tuple (a combination of values for D attributes).

Further, depending on the learning task, we can characterize $\mathcal{O}$. For predictive learning, assuming the last attribute is the target attribute, we have

$$\mathcal{O} = A_1 \times \cdots \times A_{D-1} \to A_D$$

where

- for classification: $A_D = \{c_1, c_2, \ldots, c_k\}$ (there are k classes)
- for ordinal classification: $A_D = \{c_1, c_2, \ldots, c_k\}$ where $c_i \leq c_j$ if $i \leq j$ (classes are totally ordered)
- for regression: $A_D = \mathbb{R}$, with $\mathbb{R}$ the set of real numbers

In the case of multi-target learning, where the last k elements are predicted, we have similarly,

$$\mathcal{O} = A_1 \times \cdots \times A_{D-k-1} \rightarrow A_{D-k} \times \cdots \times A_D$$

i.e., $\mathcal{O}$ is a space of functions mapping $D-k$-dimensional inputs on k-dimensional outputs.

For clustering, a descriptive learning task, we have, with $\mathbb{N}$ the set of natural numbers:

- $\mathcal{O} = T \rightarrow \mathbb{N}$ (if the clustering just defines clusters on T, not on $\mathcal{U}$)
- $\mathcal{O} = \mathcal{U} \rightarrow \mathbb{N}$ (if the clustering generalizes to $\mathcal{U}$)

Other types of descriptive learning exist, which we do not go into here. Generally, for descriptive learning, we do not learn a function that takes a single element as input, but a more general kind of characterization of the data set. That renders the type of the output more diverse.

2.2.2 Learning from sequences

While attribute value learning is a very common setting, it is not expressive enough to handle every possible learning task. Learning from sequences is one example of a strictly more expressive setting. Here, we make the following assumptions:

- The dataset is a set T of sequences $\mathbf{s}_i$, or labeled sequences $(\mathbf{s}_i, l_i)$ with $l \in L$. L is called the set of sequence labels.
- Each $\mathbf{s}_i$ is a sequence of objects $\mathbf{u}_{ij}$ with $\mathbf{u}_{ij} \in \mathcal{U}$. We write this as $\mathbf{s}_i \in \mathcal{U}^*$, with $\mathcal{U}^* = \cup_{k=0}^{\infty} \mathcal{U}^k$. Note that each object is of the same type (an element of the same universe $\mathcal{U}$).

The output space may take varying shapes. Considering only predictive learning, we can have:

- $\mathcal{O} = \mathcal{U}^* \rightarrow L$: in this case a function is learned that predicts, given a sequence, the label l of that sequence. (We can thus classify sequences, cluster sequences, or associate numerical values with them.)
- $\mathcal{O} = \mathcal{U}^* \rightarrow \mathcal{U}$: given a sequence of objects in $\mathcal{U}$, we learn a function that predicts what the following object will be.
- When $\mathcal{U} = \mathcal{X} \times \mathcal{Y}$, we may have $\mathcal{O} = \mathcal{U}^* \times \mathcal{X} \rightarrow \mathcal{Y}$. Objects in the sequences contain two parts: a description $\mathbf{x}$ and a target value $\mathbf{y}$, and the task is to predict the current $\mathbf{y}$ from the combination of the current $\mathbf{x}$ and the preceding $(\mathbf{x}, \mathbf{y})$ values.

Note that there is an essential difference between the first setting on the one hand, and the second and third on the other hand. In the first setting, we learn to predict the label of a given sequence (i.e., we will have *one prediction per sequence*); in the other settings, we learn to predict values for the elements of a sequence (we will have *one prediction per element* in the sequence).

In the former case, the prediction task is quite similar to that in attribute value learning. Here, too, learners typically assume that the sequences are i.i.d. The difference is that the predictive model takes as input a sequence (of variable length, but where each object in the sequence has the same type), rather than a tuple.

In the case of predicting values for elements of the sequence, the task is quite different. In each sequence we may need to make multiple predictions. The prediction for a single element may depend on, for instance, the element preceding it.

Note that in the attribute-value learning setting there was no notion of an element "preceding" another element, because we had an unordered set of data elements. The fact that there is a relationship between the different data elements $\mathbf{u}$ (namely, the "precedes" relation) implies that the elements are not necessarily independent anymore; in other words, the data are no longer assumed to be i.i.d.

A consequence of this difference is that in the first setting, we typically learn from multiple sequences. In the other settings, it is possible to learn from a single sequence, since the elements of that sequence, rather than the sequence itself, play the role of "examples".

Among the descriptive tasks, we again consider clustering. Just like for prediction, there are two different kinds of clustering tasks we can consider: clustering of elements in a sequence, or clustering of entire sequences. In terms of our formal description, this gives

- $\mathcal{O} = \mathcal{U}^* \to \mathbb{N}$: in this case the sequences are clustered.
- $\mathcal{O} = \mathcal{U}^* \times \mathcal{U} \times \mathcal{U}^* \to \mathbb{N}$: individual elements are clustered in the context of their sequence; that is, to cluster an element, we also look at the sequence of elements that precede it, and the sequence of elements that succeed it.

The general setting of sequence learning covers a number of more specific learning settings, which have been studied in more detail in the machine learning literature.

Time series When $U = \mathcal{X} \times \mathcal{Y}$ with $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$, the sequence can be considered a time series. A time series describes the evolution of a numerical variable over time. The blood pressure of a person, or the value of a particular share on the stockmarket, are examples of such time series. Many methods have been developed specifically for learning from time series.

Learning from strings U may equal a finite set of symbols Σ, called an alphabet. The sequences are then (usually finite) strings over the alphabet. *Grammar learning* or *induction of finite state automata* is one approach to learning from such data. *Hidden Markov Models* are another formalism that can be used; they can be seen as a probabilistic extension of finite state automata.

2.2.3 Learning from graphs

The setting of learning from sequences can be generalized to learning from graphs. This setting is defined as follows:

- The dataset is a set T of graphs G_i, or labeled graphs (G_i, l_i), with $l_i \in L$.
- Each graph $G_i = (V_i, E_i)$ consists of a set of nodes V_i and a set of edges $E_i \subseteq V_i \times V_i$.[2]
- Each node v may be labeled by an element $l_V(v)$ of some set of node labels L_V.
- Each edge e may be labeled by an element $l_E(e)$ of some set of edge labels L_E.

[2]We consider only *directed graphs* here, for the sake of simplicity, but the definitions can be rewritten trivially for undirected graphs or other variants.

Similarly to what we did for sequences, we can distinguish methods that make one prediction per graph from methods that make a prediction for each node or edge in the graph; and for clustering, we can distinguish methods that cluster whole graphs from methods that cluster nodes or edges in a graph. The output space can be described as follows.

With V a node alphabet (the set of all possible nodes we can denote) and $\mathcal{G}$ the set of all graphs over V, we have:

- $\mathcal{O} = \mathcal{G} \to L$, when labeling graphs
- $\mathcal{O} = \mathcal{G} \times V \to L_V$, when labeling nodes
- $\mathcal{O} = \mathcal{G} \times (V \times V) \to L_E$, when labeling edges

($\mathcal{G}$ is part of the input in the last two cases because the prediction for a node or edge is made in the context of the graph it occurs in; this context needs to be taken into account.)

In the above settings, we assume that the whole graph is given, and only labels need to be predicted. A more difficult problem setting, quite relevant in practice however, is that of predicting the existence of nodes and/or edges in a graph, given a partial description of the graph. The output space is then:

$$\mathcal{O} = \mathcal{G} \to \mathcal{G}.$$

In practice, functions that take graphs as inputs are difficult to describe. For that reason, many graph learners work in the transductive setting. They do not learn a function that maps graphs onto labels or onto other graphs; their output consists simply of a predicted label, a predicted graph, etc.

Example 2.2 Consider the following task: we are given a set of molecules together with an indication of how well they work as, say, a drug against some disease. The molecular structure of each molecule can be described as a graph; its activity is the label of the graph. We want to learn a function that, given a new molecule, can predict how well it will work. This is an example of a *graph labeling task*.

Example 2.3 The World Wide Web consists of many web pages that are connected by links. As such, it forms a graph, with the pages as nodes and links as edges. A task such as classifying web pages into different types (say, we want to distinguish "home pages of people" from "other pages") is an example of a *node labeling task*.

Example 2.4 Software for social networks often tries to predict possible friends of a user, so the user can connect to those people directly. Assume G_1 is the graph indicating who is connected to whom, and G_2 is the graph indicating who is a friend of whom. The task is then to predict G_2 from G_1. This is also called *link prediction* or *edge prediction*: a graph G_1 is given and we want to predict edges so that it better resembles G_2.

Note that this last example is a case of transductive learning. We do not want to learn a function that in general takes a function G_1 and predicts the corresponding G_2 from it; we simply want to fill in the edge predictions.

2.2.4 Learning from trees

Learning from trees is a special case of learning from graphs. Indeed, a (rooted) tree is just a directed graph with the property that for each node $v \in V$ there is exactly one node $u \in V$ such that $(u, v) \in E$ (u is called the parent of v), with the exception of one node, called the root, which has no parent. Similarly as with

graphs, the direction of the edges can be dropped; the tree is then called an *unrooted tree*. From one unrooted tree with k nodes, k rooted trees can be derived, each one having one node as the root and directing all edges away from this root.

All the learning tasks for graphs carry over to learning tasks for trees (labeling nodes, labeling entire trees, etc.)

While learning from trees can be considered a special case of learning from graphs, many methods have been developed that work specifically in this setting, so that it can be considered an area on its own. An important reason for considering trees separately is that many operations can be performed much more efficiently on trees than on graphs in general.

Example 2.5 Consider the task of learning to extract certain kinds of information from web pages (for instance, learning to extract email addresses.) A web page is typically internally structured as an HTML or XML tree. The information extraction task therefore corresponds to the task of labeling nodes in a tree (where a node is labeled as "relevant" or "not relevant" for the extraction).

Learning tasks such as the one in the above example have been addressed using, for instance, *induction of tree automata* [24, 35]. Tree automata are finite state automata that process trees instead of strings.

2.2.5 Multiple instance learning

In multiple instance learning, the problem setting is the following. We will denote the set of boolean values as $\mathbb{B}$; $\mathbb{B} = \{true, false\}$.

- The dataset is a set T of objects $(\mathbf{s}_i, l_i)$ where $l_i \in \mathbb{B}$ (labels are boolean).
- Each $\mathbf{s}_i$ is a set of objects $\mathbf{x}_{ij}$ with $\mathbf{x}_{ij} \in \mathcal{X}$.
- Each $\mathbf{x}_{ij}$ has a boolean label l_{ij}, but this label is not given in the data set. We do know that $l_i = l_{i1} \vee \cdots \vee l_{in}$, that is: the label of a set is *true* if and only if at least one of its elements is labeled *true*.

The learning task for this setting can be described in two ways:

- $\mathcal{O} = 2^{\mathcal{X}} \to \mathbb{B}$: we want to learn a function f that predicts for a set $\mathbf{s}$ its label l
- $\mathcal{O} = \mathcal{X} \to \mathbb{B}$: we want to learn a function g that predicts for a single element $\mathbf{x}_{ij}$ its label l_{ij}

Note that the first task is subsumed by the second: if we have a function g that correctly predicts the label of a single element, then the function $f(\mathbf{s}) = \bigvee_{\mathbf{x} \in \mathbf{s}} g(\mathbf{x})$ correctly predicts the labels of sets. The converse is also true: if we know f, we can derive g, because the result of f on a singleton must be equal to the result of g for its element: $g(\mathbf{x}) = f(\{\mathbf{x}\})$. In this sense, the tasks of learning f or $g$are equivalent.

Example 2.6 The prototypical multiple instance learning problem is the following [14]: we are given a set of molecules, together with, for each molecule, an indication of whether it is active or not. The activity of a molecule is related not so much to its molecular structure, but to its spatial configuration. But a single molecule can adopt many spatial configurations, and when a molecule is active, we do not know which of its configurations is the one that is active; we just know that one of them is. So, for each molecule we are given a set of configurations it can take, and a label that is positive if at least one of these configurations fulfills some unknown condition. From these data, we need to learn a function that can predict for a new molecule (represented as a set of configurations) whether it will be positive or not.

Example 2.7 Multiple instance learning problems have also been defined in the context of image classification, where we may want to learn a classifier from examples of pictures that are labeled positive whenever any part of the picture shows a certain object (but where it is not indicated for a positive picture exactly where that object is).

2.2.6 Learning from knowledge

The most general learning setting is that where the input knowledge can be any kind of knowledge expressible in some knowledge representation language. As an example, take the following statements in a language based on first order predicate logic:

```
father(bob,ann).
father(bob,billy).
father(bob,ken).
mother(mary,ann).
mother(mary,billy).
mother(mary,ken).
father(X,Y) -> parent(X,Y).
mother(X,Y) -> parent(X,Y).
parent(X,Y) -> father(X,Y) v mother(X,Y).
child(X,Y) <-> parent(Y,X).
```

where the `->` and `<->` symbols denote logical implication and equivalence, respectively; the `v` symbol denotes the disjunction operator ("or"); capitalized characters (X,Y) denote logical variables; and all logical variables are universally quantified. (E.g., the clause `father(X,Y) -> parent(X,Y)` says: for all x and y, if x is the father of y, then x is a parent of y.)

A system could learn from this information hypotheses such as:

```
father(bob,X) <-> mother(mary,X).
```

which states: if Bob is the father of some person X, then Mary is the mother of X, and vice versa; or:

```
child(X,bob) -> father(bob,X).
```

which states that if X is a child of Bob then Bob is the father of X. These formulas are statements that are not part of the original set of knowledge, but have been constructed by analysing that knowledge, just like sets of data structures (tuples, sequences, trees or graphs) were analysed in the other settings. Note that these formulas are descriptive rather than predictive: they describe a pattern that holds in the given knowledge, but may not necessarily be useful for making predictions.

Learning from any kind of knowledge (representable in some knowledge representation language) can be considered the most general setting possible, because any knowledge representation language with a reasonable expressivity will allow us to represent sets, graphs, sequences, etc., simply by stating that something is an element of something else, that something is a node that is connected to some other node, etc. For instance,

```
node(a).
node(b).
node(c).
edge(a,b).
edge(a,c).
```

represents a graph with nodes $\{a, b, c\}$ and edges $\{(a, b), (a, c)\}$.

Knowledge representation languages typically allow us to state not just factual knowledge, such as the description of a specific graph. They also allow us to provide definitions for concepts, rules that allow one to deduce knowledge from other knowledge, constraints that must hold in the domain that is being described, etc. In this respect, knowledge representation languages are more powerful than languages that only allow us to write down, for instance, sets or graphs.

In its most general form, this setting uses $\mathcal{I} = \mathcal{O} = \mathcal{K}$ with $\mathcal{K}$ the set of everything that can be represented in the knowledge representation language. If this language is powerful enough to represent objects and functions (which it typically is), this setting encompasses all the previous ones.

Many knowledge representation languages are based on (a restricted version of) first order logic. The Prolog programming language has been commonly used for representing knowledge to learn from, for instance in the subfield of machine learning known as inductive logic programming (Chapter 15). One could certainly argue that Prolog is not ideal as a knowledge representation language: logic-based languages exist that are better suited for this (an example are the *description logics* that have become popular in the context of, for instance, the semantic web). Yet, in the context of machine learning, Prolog has been the more popular representation formalism, possibly simply because it is more widely known among computer scientists.

2.2.7 Structured prediction

In the preceding discussion, we have usually assumed that the labels to be predicted were scalar values, i.e., values without an internal structure (such as vectors, sets, graphs, ...). Predicting values with an internal structure can be much more difficult. This problem is sometimes called **structured prediction**, or **prediction in structured output spaces**.

Fixed-structure prediction. We first consider the case where the label to be predicted always has the same internal structure, for instance: we need to learn functions that can predict vectors of the form (y_1, y_2, y_3). This is similar to the multi-target prediction setting mentioned under learning from attribute-value data. This case can easily be reduced to the case of learning functions that predict a scalar: we can simply learn a function f_1 that will predict y_1, another function f_2 for y_2, and a function f_3 for y_3. The three functions together define a function $f(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x})]$ predicting the vector.

Thus, if we have learning algorithms that learn functions that predict scalar values (single-prediction functions), we can solve multi-target prediction problems. But that does not mean that it is not useful to try to develop methods that *directly* learn multiple-prediction functions. Indeed, some learning methods will exhibit better performance than others, and perhaps methods that rely on a reduction from the multiple prediction to the single prediction case may suffer a decline in performance. Moreover, in cases where there are many targets (in some applications there may be hundreds or thousands), learning a multiple-prediction function once may be much more efficient than learning a single-prediction for each separate target.

Variable-structure prediction. When the label to be predicted may have a variable structure, the prediction problem becomes a lot more difficult. Take for instance the task of predicting a graph. If the structure of the graph is given and

we only need to predict labels for its nodes or edges, this is similar to the fixed-structure prediction problem. There is a finite number of scalar variables that we need to predict.

If the nodes of the graph are given but not (all of) its edges, and we need to predict which edges it contains, this becomes more difficult. In principle we could make a boolean prediction for each couple of nodes, stating whether there is an edge between these nodes or not. But in a graph with $|V|$ nodes, there are $|V|^2$ possible edges. This number may be huge, and much larger than the actual number of edges that the graph is likely to contain. Take the world wide web for example: it contains billions of nodes, which are very sparsely connected (each node could in principle link to billions of other nodes, but in practice it links to only a few). In such a graph, predicting the edges by considering all possible edges and making a boolean prediction for each of them is unrealistic in practice.

An even more difficult problem is when also the nodes of the graph are not known. If there is no bound on the number of nodes, we cannot map the problem to that of predicting a scalar value for a finite number of variables.

The large majority of machine learning methods focuses on single prediction. Fixed-structure prediction has been considered in a number of cases, while variable-structure prediction is a difficult problem that has not received much attention up till now.

2.3 Format of output knowledge (hypothesis)

The output of an inductive learning process is a representation of a predictive or descriptive model. It is useful to distinguish two levels at which the output can be described: the syntactic level (what language is used to describe the output?) and the semantic level (what does a given output object really mean, what does it represent?). For instance, any classification system yields as output a classifier, i.e., a function that maps individuals onto their classes. From this point of view, it is difficult to distinguish a neural network and a decision tree; indeed, we might be able to represent exactly the same function $f : \mathcal{X} \rightarrow \mathcal{Y}$ as a decision tree, or as a neural net. To describe the outputs of different kinds of learners, it is therefore useful to look at how they represent the functions that they output, rather than those functions themselves.

2.3.1 Predictive models

A predictive model typically represents a mathematical function that maps an element of some input space $\mathcal{X}$ to an element of some output space $\mathcal{Y}$. Such a function can be represented in many different ways. A learner uses a certain language $\mathcal{L}$ to represent its outputs; each member of the language is a syntactical description of a predictive model.

For instance, a *rule set learner* learns predictive models in the form of a set of rules, typically if-then-rules that link certain conditions on the input to predictions for the output. We have already seen an example of such a rule set in the context of the Animals dataset:

if Covering = feathers **then** Movement = flies **else** Movement = walks

represents a function from $\mathcal{X}$ = Species×Type×Size×Covering to $\mathcal{Y}$ = Movement = $\{flies, walks\}$. Another such function, written in another representation language, is the thresholded linear function with indicator variables that we saw before:

if $3 * I_{Covering=feathers} + 2 * I_{Size=small} > 3$ **then** Movement = flies **else** Movement = walks

Decision trees are another language for representing models; yet other languages are neural networks, bayesian networks, mathematical equations, etc. Many of these will be discussed in later chapters.

2.3.2 Descriptive models

A descriptive model will describe properties of the dataset as a whole. It is not necessarily in the form of a function mapping elements from an input space onto some output space, although such functions can often be seen as special cases of descriptive models as well.

Clustering is an example of a descriptive learning task; the idea is that a certain structure in the data set is discovered by identifying clusters of elements that somehow belong together. In the case of *extensional clustering*, the clusters are simply defined as subsets of the original data set. Thus, the language in which the model is described should allow us to represent a set of clusters, where each cluster is again a set defined by enumerating its elements. A possible representation for such a clustering is $\{\{\mathbf{u}_1, \mathbf{u}_4\}, \{\mathbf{u}_2\}, \{\mathbf{u}_3, \mathbf{u}_5, \mathbf{u}_6\}\}$, with the $\mathbf{u}_i$ the examples in the dataset that was to be clustered. Another representation, more in line with our previous observation that clustering can be seen as learning a function mapping elements onto natural numbers, is $\{\mathbf{u}_1 \to 1, \mathbf{u}_4 \to 1, \mathbf{u}_2 \to 2, \mathbf{u}_3 \to 3, \mathbf{u}_5 \to 3, \mathbf{u}_6 \to 3\}$, which uses numbers (1,2,3) to identify each cluster and assigns objects to each number (hence, to each cluster).

Another well known example of a descriptive learning task is frequent itemset discovery. Here, each individual data element $\mathbf{x}_i$, $i = 1 \ldots N$, is a subset of some fixed set of items I, and the task is to describe the dataset by listing for a number of itemsets how often these itemsets occur in the dataset, where "occur" is defined as being a subset of a data element. E.g., the dataset might look as follows:

```
{milk,eggs,pizza,coke}
{milk,bread,chocolate}
{eggs,water,flour,sugar}
```

and the output of a frequent itemset discovery algorithm might look like this:

```
{milk}:2
{eggs}:2
{water}:1
{milk,eggs}:1
{milk,pizza}:1
{milk,bread,chocolate}:1
...
```

The output language, in this case, contains lines of the form *itemset:frequency* where *frequency* is a number and *itemset* is a set described by enumerating its elements.

2.3.3 The representation matters

Sometimes, when we perform learning, the answer to our questions is in the representation of the learned function, rather than in the function itself. For instance, the rule

if Covering = feathers **then** Movement = flies **else** Movement = walks

explicitly states that all feathered animals fly and all other animals walk. This is a piece of information that may be valuable in itself. Similarly, in the case of frequent itemset discovery, we are usually less interested in the function mapping itemsets

to frequencies as a whole, than in certain implications of this function. The fact that milk, bread and chocolate are often bought together might be an interesting piece of information, while the fact that exactly 102 people bought this combination today in a particular supermarket might not.

It is clear now that when we say that a predictive learner learns a function from $\mathcal{X}$ to $\mathcal{Y}$, this is in fact a simplification. The learner really learns an expression M in some language $\mathcal{L}$, and this expression *represents* a function f from $\mathcal{X}$ to $\mathcal{Y}$. This function can be used to predict a value $\mathbf{y}$ when given $\mathbf{x}$, but the expression M itself may be valuable as well. In the context of machine learning and data mining, we are often interested not only in f but also in M itself.

2.4 Supervised, semi-supervised, and unsupervised learning

We have already briefly mentioned the concepts of supervised and unsupervised learning. **Supervised learning** is the standard predictive learning setting: a function $f : \mathcal{X} \to \mathcal{Y}$ that predicts the value of some target attribute Y from other values X, is learned from examples of the form $(\mathbf{x}, \mathbf{y})$ with $\mathbf{y} = f(\mathbf{x})$). When the $\mathbf{y}$ values are not given, we have an **unsupervised learning** setting.

There is a setting that can be considered to be in between supervised and unsupervised learning, and which is called **semi-supervised learning**. In semi-supervised learning, we want to learn a function $f : \mathcal{X} \to \mathcal{Y}$ from a set of examples, but only some of these examples have a known value for $\mathcal{Y}$: for the large majority of instances in the training set, this value is not known. Instances for which $\mathcal{Y}$ is known are often called labeled instances; the others are then called unlabeled.

One might wonder whether such unlabeled instances can be useful at all. If the goal is to learn a predictive function from $\mathcal{X}$ to $\mathcal{Y}$, can examples that have values for $\mathcal{X}$ but not for $\mathcal{Y}$ teach us anything on the connection between $\mathcal{X}$ and $\mathcal{Y}$?

They cannot do so directly, but indirectly, through their connection with labeled examples, they can provide useful information. Figure 2.1 illustrates this. To the left we see only the labeled examples, to the right we see labeled and unlabeled examples. The unlabeled examples provide more information on the distribution of all examples. Intuitively, looking at the figures, we would come up with quite different predictive models in the left case and in the right case, even though the labeled examples are exactly the same.

The figure also illustrates that, often, very few labeled examples may be sufficient to build a relatively good predictive model, if enough unlabeled examples are available. This is one reason why there has been a surge of interest in semi-supervised learning lately. In many practical domains, it is easy to collect many unlabeled examples, but the labeling itself has to be done manually and is therefore expensive; in such cases it is advantageous to be able to learn from few labeled and many unlabeled examples.

Example 2.8 Think of the task of classifying web pages into, for instance, "student home page", "professor home page", or "other". It is easy to collect thousands, even millions of web pages; but they typically do not come with a label assigning them to one of the three above classes. Labeling thousands of such pages is very labor-intensive. However, labeling a few dozen of them might be sufficient to learn a good classifier.

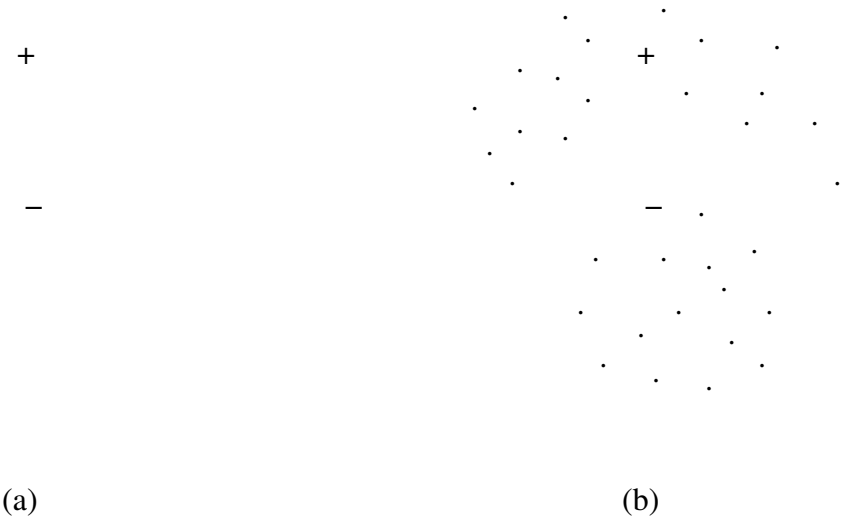

Figure 2.1: An illustration of how the availability of unlabeled examples can help us find the correct "border" between classes. A + in the drawing denotes a positive example, a − a negative one, and a dot is an unlabeled instance. If you need to draw a line that separates positive (+) examples from negative (−) ones, which line would seem to be the most natural one in case (a), and which in case (b)?

Multiple-instance learning as semi-supervised learning

The multiple-instance learning setting can also be considered as some kind of semi-supervised learning. Here, too, limited information on the labels of the instances is available. It is a bit more complicated than the above described setting, though, because above we assumed that some instances (possibly a minority) are labeled, whereas in the multiple-instance learning setting, generally, none of the positive ("true") examples are labeled. There is only a constraint stating that in a positive bag, at least one of the examples must be positive. Thus the input knowledge contains information of the form $f(\mathbf{x}_{i1}) = true \vee f(\mathbf{x}_{i2}) = true \vee \ldots \vee f(\mathbf{x}_{i|B_i|}) = true$, with $\mathbf{x}_{ij}$ the instances of a positive bag B_i. In other words, we have some information on the classes of these instances, but there is no single instance for which we know its class with certainty.

2.5 Situating machine learning methods in this context

There exists a large variety of machine learning methods. Many of these specifically address the task of *supervised*, *predictive* learning in the *attribute-value* framework. These methods vary mostly in the type of output they produce (their output representation), and not so much in their input representation. When we look at the machine learning approaches that we will see in this text: decision trees (Chapter 4), instance based learning (Chapter 7), clustering (Chapter 8), artificial neural networks (Chapter 10), support vector machines (Chapter 11), probabilistic learning (Chapter 12), ensemble methods (Chapter 13), reinforcement learning (Chapter 14), and inductive logic programming (Chapter 15), it is clear that most of these approaches derive their name from the output representation formalism that they use. Clustering refers more to the task that is being performed, rather than the methods, whereas probabilistic learning refers to a collection of methods that rely on prob-

ability theory (but most of these methods have names that, again, refer to their output representation). Reinforcement learning refers to a learning setting that is quite different altogether.

Inductive logic programming (Chapter 15) distinguishes itself from the other methods by the use of logic-based input and output representations. While for the output representation one could say that we simply have "yet another output representation" (namely, logical rules), the input representation is essentially different. Because first order logic is expressive enough to represent not only tuples, but also sets, sequences, trees, and graphs, inductive logic programming has a much broader application potential than the attribute-value learning methods. In a sense it subsumes the other approaches in this respect.

Seeing attribute-value learning as the least expressive of the settings we can consider, and inductive logic programming as the most expressive one, we can define a spectrum, in which multiple-instance learning, learning from sequences, learning from trees, and learning from graphs, are somewhere in the middle. In recent years, more and more research has been devoted to learning in those "intermediate" settings. In this text, we focus mostly on the two extremes, with the majority of the text being set in the context of attribute-value learning.

Chapter 3

Conjunctive concepts and version spaces

In this chapter we consider a first, relatively simple, learning task: learning conjunctive concepts. This is a special case of concept learning where the concept to be learned is assumed to be expressible as a conjunction of conditions. We will see how the learning task can be expressed as a search through a search space, how a generality relationship over this search space can help make this search more efficient, and how the set of all solutions to the problem, called the version space, can be represented and computed efficiently by exploiting the generality relation.

Many of the ideas and concepts introduced in this chapter are applicable in a broader context than learning conjunctive concepts, but not all of them are equally practical in that broader context. They are still interesting from a theoretical point of view, however.

3.1 Learning conjunctive concepts

3.1.1 Concepts

Given an instance space $\mathcal{X}$, a **concept** is defined as a subset of $\mathcal{X}$. We say that an instance $\mathbf{x}$ belongs to the concept C (synonyms: $\mathbf{x}$ is an instance of C, or: C **covers** $\mathbf{x}$) if and only if $\mathbf{x} \in C$, and it does not belong to C otherwise. Equivalently, we can see a concept as a function c from $\mathcal{X}$ to $\mathbb{B}$, with $\mathbb{B} = \{true, false\}$ the set of boolean values. Both are equivalent because for any subset C, a boolean function c exists, and vice versa, such that

$$\mathbf{x} \in C \Leftrightarrow c(\mathbf{x}) = true.$$

Depending on the context, we will sometimes use the function or subset interpretation, whichever is more suitable. In this chapter, we will usually denote concepts with lowercase letters, even when we treat them as sets.

3.1.2 Conjunctive concepts

We assume that instances of the instance space $\mathcal{X}$ are described using a fixed number of features, or attributes: $\mathcal{X} = A_1 \times \cdots \times A_D$, with D the dimensionality of $\mathcal{X}$. This is the attribute value setting discussed earlier. For now, we will further assume that each attribute domain A_i is nominal and finite.

A **conjunctive concept** is a concept of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

with each C_i a condition of the form $A_j = v_j$ with v_j a value from the domain of A_j. In other words, a conjunctive concept can always be defined as the set of all instances that fulfill one particular conjunction of conditions.

Example 3.1 In the Animals dataset we saw before, an example of a conjunctive concept is `Covering=feathers` $\wedge$ `Type=bird`. The concept `Covering=feathers` $\vee$ `Type=bird` is not a conjunctive concept, since it contains a logical or in its definition.

For simplicity of notation, we will often denote a concept as a D-tuple $(t_1, t_2, \ldots, t_D)$ where the t_i are either '?', indicating that the corresponding attribute A_i can have any value, or a particular value v_i from the domain of A_i, indicating that the corresponding attribute has the value v_i for all instances that belong to the concept. This formalism allows us to represent any conjunctive concept except one: the empty set, which we will denote as $\emptyset$.

Example 3.2 If we take $\mathcal{X}$ = Species $\times$ Type $\times$ Size $\times$ Covering, the concept `Covering=feathers` $\wedge$ `Type=bird` can be represented as `(?, bird, ?, feathers)`. The question marks indicate that the attributes Species and Size do not matter here.

We consider supervised concept learning here: the dataset contains examples of the form $(\mathbf{x}, y)$ where $y \in \mathbb{B}$. Examples with $y = \textit{true}$ are called **positive examples**; examples with $y = \textit{false}$ are called **negative examples**.

The task of learning conjunctive concepts boils down to learning, from a set of positive and negative examples of a target concept, a conjunctive concept description for that target concept. This description may be in the form of a conjunction of conditions, or, equivalently, a D-tuple of terms t_i or the empty set symbol $\emptyset$).

Definition 3.1 (Task: Learning conjunctive concepts) Given*: a set of data* $T = \{(\mathbf{x}_i, y_i) | i = 1, \ldots, N\}$ *where* $\mathbf{x}_i \in \mathcal{X}$ *and* $y_i \in \mathbb{B}$ *;* **Find***: a conjunctive concept* c *over* $\mathcal{X}$ *such that* $\forall (\mathbf{x}, y) \in T : c(x) = y$.

3.2 Generality-based exhaustive search

3.2.1 The generality relationship

Since concepts are sets, it is possible for one concept to be a subset, or superset, of another concept. The subset relationship imposes a partial order on these sets. This ordering is called the generality ordering: we say that a concept c_1 is more general than another concept c_2 if $c_2 \subseteq c_1$, that is: every element of c_2 belongs to c_1 as well. The inverse relation of "is more general than" is the relation "is more specific than".

Definition 3.2 (Generality, specificity) *Consider concepts* c_1 *and* c_2*, both subsets of the instance space* $\mathcal{X}$*.* c_1 *is* **more general than or equal to** *a concept* c_2*, denoted* $c_1 \geq c_2$*, if and only if* $c_1 \supseteq c_2$*.* c_1 *is* **more specific than or equal to** c_2*, denoted* $c_1 \leq c_2$*, if and only if* $c_1 \subseteq c_2$*.*

3.2.2 Generality lattice over conjunctive concepts

The most general concept is $\mathcal{X}$ itself, which in our formalism for conjunctive concepts can also be written as $(?, ?, \ldots, ?)$. It contains every possible instance. The most specific concept is the empty set $\emptyset$. If we focus on conjunctive concepts, then each non-empty concept c can be written as $(t_1, t_2, \ldots, t_n)$ where each t_i is either ? or v_i with $v_i \in A_i$. We then have the following properties.

- For all concepts c, $(?, ?, \ldots, ?) \geq c \geq \emptyset$.
- $(t_1, t_2, \ldots, t_n) \geq (t'_1, t'_2, \ldots, t'_n)$ if and only if for each t_i, $t_i =?$ or $t_i = t'_i$.

The generality relation is a partial order. A partial order is a relation that is reflexive, antisymmetric, and transitive:

$$\forall x : x \geq x$$

$$\forall x, y : x \geq y \wedge y \geq x \Rightarrow x = y$$

$$\forall x : x \geq y \wedge y \geq z \Rightarrow x \geq z$$

The generality relation has a number of interesting properties that we will be able to exploit later on.

Minimal strict specialization / generalization

When $c_1 \leq c_2$, we call c_1 a **specialization** of c_2, and c_2 a **generalization** of c_1. When $c_1 < c_2$, the specialization / generalization is strict. We call c_1 a **minimal strict specialization** of c_2 (and c_2 a **minimal strict generalization** of c_1) if $c_1 < c_2$ and there does not exist a c_3 such that $c_1 < c_3 < c_2$.

For the conjunctive concepts we have considered up till now, a minimal strict generalization is obtained by changing one v_i into ?; a minimal strict specialization is obtained by changing one ? into a v_i. Since this can be done in multiple ways, a conjunctive concept usually has multiple minimal strict specializations / generalizations.

Example 3.3 The concept `(?, bird, ?, feathers)` has as minimal strict generalizations: `(?, ?, ?, feathers)` and `(?, bird, ?, ?)`. It has as minimal strict specializations: `(?, bird, small, feathers)`, `(?, bird, medium, feathers)`, `(?, bird, large, feathers)`, `(penguin, bird, ?, feathers)`, `(sparrow, bird, ?, feathers)`, `(canary, bird, ?, feathers)`, ...

Given a concept c, any specialization of c can be obtained by performing multiple minimal specializations, and similarly for generalizations.

Least general generalization / least specific specialization of two elements

In a partially ordered set, for any two elements x and y, we call z a **least upper bound** of x and y if and only if $z \geq x$, $z \geq y$, and $\nexists z' : z' \geq x, z' \geq y, z \geq z'$. In the context of our generality ordering, we will also call z a **least general generalization**. The conditions just listed imply that: (a) z is more general than both x and y ("z is a generalization of x and y"), and (b) there does not exist another element z' that has the same property and that is strictly less general than z ("z is the *least general* generalization").

Definition 3.3 (Least general generalization) *z is a least general generalization of x and y if and only if $z \geq x$, $z \geq y$, and $\nexists z' : z' \geq x \wedge z' \geq y \wedge z \geq z'$.*

Similarly, we can define the greatest lower bound of two elements in a partial order, which in the context of our generality relatio will be called a least specific specialization.

Definition 3.4 (Least specific specialization) *z is a least specific specialization of x and y if and only if $z \leq x$, $z \leq y$, and $\nexists z' : z' \leq x \wedge z' \leq y \wedge z \leq z'$.*

A partial order that has the property that for every two elements x and y, the least upper bound and greatest lower bound exist, defines a **lattice**. The "is a subset of" relationship, defined overal all subsets of any set S, is known to form a lattice. Since the generality relation $\geq$ over $\mathcal{X}$ is essentially a subset relation, it also forms a lattice. The top (most general) element of the lattice is the whole instance space $\mathcal{X}$, the bottom element is $\emptyset$, and in between we have all other concepts. There is a downward path from a concept c_1 to a concept c_2 in the lattice if $c_1 \geq c_2$.

Given two concepts c_1 and c_2, their least general generalization is obtained by changing each attribute for which they have different values, the value ?. Their least specific specialization is $\emptyset$ if there exists an attribute for which c_1 and c_2 have contradictory values, and otherwise each attribute is ? if it is ? for both c_1 and c_2, and v whenever either c_1 or c_2 has the value v.

Example 3.4 The concepts `(?, bird, ?, feathers,?)` (all feathered birds) and `(?, bird, small, ?)` (all small birds) have as least general generalization the concept `(?, bird, ?, ?)` (all birds) and as least specific specialization the concept `(?, bird, small, feathers,?)` (all small feathered birds).

3.2.3 Generality-based hypothesis search

Consider the following two variants of the concept learning problem: given a hypothesis space H and a data set T, (1) find a hypothesis in H that is consistent with T; (2) find all hypotheses in H that are consistent with T.

In principle these problems are easy to solve: we can simply enumerate all the hypotheses h in H, test whether they are consistent with the data (that is, for all $(\mathbf{x}, y) \in T : h(\mathbf{x}) = y$), and return one by one each hypothese that passes the test (or just the first one).

A practical problem with this approach is that H can be huge; in fact, the number of all conjunctive concepts lies between the size of the instance space $|\mathcal{X}|$ (since with instance corresponds a singleton concept) and the size of the set of all possible concepts, $2^{|\mathcal{X}|}$ (from which the set of conjunctive concepts is a subset).

Exercise 3.1 Show that if each attribute A_i, $i = 1, \ldots, D$, has $|A_i|$ possible values, then the number of conjunctive concepts in a D-dimensional space is $1+\prod_{i=1}^{D}(|A_i|+1)$. Apply this to the Animals example.

For a high-dimensional $\mathcal{X}$, H can be very large, and enumerating all the hypotheses may be infeasible. Even the subset of H it that is consistent with T may itself be huge, so even just *outputting* the answer to variant 2 may be an enormous amount of work.

Fortunately, there is a way to search H, and to represent the answer to variant 2, a lot more efficiently. In this section, we consider the easier problem of returning one hypothesis consistent with the data; the more complicated problem, returning all of them, is discussed in the next section.

When looking for a single hypothesis consistent with the data T, we can traverse the hypothesis space H in an efficient and principled manner that allows us to prune large part of this space. One way to do that is to do a depth-first search through the lattice imposed by the generality ordering.

Consider a general to specific search. We start with the most general concept. As it is likely too general, we can consider minimal specializations of it. We look at these minimal specializations one by one; if a specialization is too general (covers negative examples), we again specialize it. By generating only minimal specializations, we are certain not to "skip" any potentially correct hypotheses. We keep specializing hypotheses until we find one that is consistent with the data.

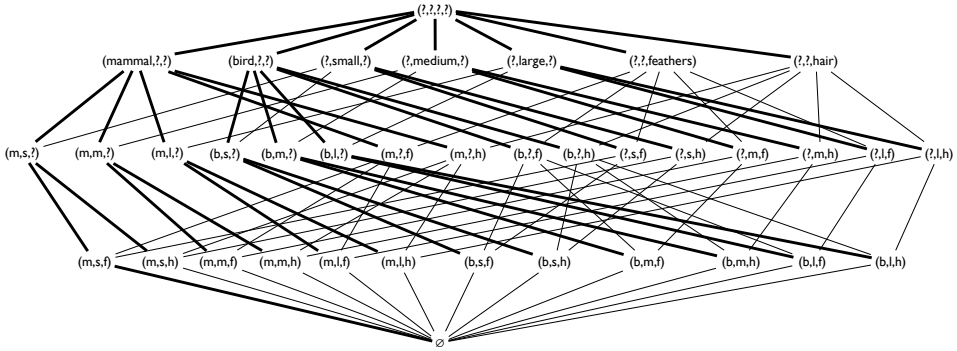

Figure 3.1: A simplified latticed for the Animals example (based on only the attributes Type, Size and Covering; from the second level on, values are abbreviated by a single character). Leftmost paths in the lattice are indicated by heavy lines.

It is possible that after specializing a hypothesis that is too general, we end up with a hypothesis that is too specific. Clearly, further specializing this hypothesis is not useful: if a hypothesis h is too specific (there is at least one positive example that it does not cover), then none of its specializations can be consistent with the data (they cannot cover that positive example either). At that point, we know we have reached a dead end and we can prune everything beneath the current node.

Note that since the space is structured as a lattice, most points in the lattice can be reached via multiple paths. It is sufficient to consider each point only once, though. This can be achieved by ensuring that each point will be reached only through the "leftmost" path in the lattice. This, in turn, can be guaranteed by generating at each point only those specializations that are not specializations of a hypotheses that is more to the left in the lattice. Not generating children that have a more-to-the-left parent corresponds to not instantiating any ? that is to the left of an already instantiated term in the concept description. See Figure 3.1 for an illustration of this "leftmost path search".

An exhaustive search through the full hypothesis space can be performed using the algorithm in Figure 3.2. It traverses the hypothesis space top-down, from general to specific, always following only leftmost paths in the lattice. The algorithm assumes that a consistent hypothesis exists, which is the case if the target concept c is indeed a conjunctive hypothesis. When a hypothesis is reached that is too specific, it is not further specialized; this amounts to pruning away a part of the search space.

3.3 Version spaces

In the above section we discussed how the generality relation can be exploited to efficiently find a single hypothesis that is consistent with the data set. Here we consider the task of finding all such hypotheses.

3.3.1 Version space: definition

Assume that we have a hypothesis space H, consisting of models that might, or might not, be equal to some target model. We want to find those hypotheses in H that are consistent with a data set T. This set is called the **version space** of H with respect to T.

function SEARCHCONSISTENTCONCEPT **returns** hypothesis:
    $Q := [(?, ?, \ldots, ?)]$
    repeat
        remove first element from Q, call this element h
        if h is not consistent with T and $\forall(\mathbf{x}, \mathit{true}) \in T : h(\mathbf{x}) = \mathit{true}$ **then**
            $R :=$ SPECIALIZE(h)
            add all $h' \in R$ to the front of Q *// adding at front yields depth-first search*
    until h is consistent with T
    return h

function SPECIALIZE(hypothesis h) **returns** set of hypotheses:
    let $h = (t_1, t_2, \ldots, t_D)$
    let m be the smallest i such that $\forall j \geq i : t_j = ?$
    $R := \emptyset$
        for $i = m, \ldots, D$:
            for each possible value v of A_i:
                $h' := h$ with t_i replaced by v
                add h' to R
    return R

Figure 3.2: Exhaustive search through the hypothesis space of all conjunctive concepts. The underlined condition makes the algorithm prune away parts of the search space without sacrificing correctness.

Definition 3.5 (Version space) *Given a hypothesis space H and a set of data T, the version space $VS(H,T)$ is the subset of all elements of H consistent with T.*

The terms "hypotheses", "data" and "consistent" are intentionally left vague here. They can in principle take many forms. In the particular context of concept learning, the following more precise definition applies.

Definition 3.6 (Version space) *Given an instance space $\mathcal{X}$, a hypothesis space H containing boolean functions of $\mathcal{X}$, a target concept c which is also a boolean function of $\mathcal{X}$, and a data set T consisting of pairs $(\mathbf{x}, y)$ where $y \in \mathbb{B}$, the version space $VS(H,T)$ is the set of all hypotheses $h \in H$ that are consistent with T, i.e., for which $h(\mathbf{x}) = y$ for each $(\mathbf{x}, y) \in T$. Formally:*

$$VS(H,T) = \{h \in H | \forall(\mathbf{x}, y) \in T : h(\mathbf{x}) = y\}.$$

3.3.2 Computing the version space

It may seem that computing the version space is trivial: one can just enumerate the whole hypothesis space, and for each hypothesis, test whether it is consistent with the data set D, and if yes, store it in the result set. But, as mentioned before, this approach is infeasible for very large hypothesis spaces. Moreover, it is highly redundant: in many cases, there are relationships between the hypotheses in H such that whenever one particular hypothesis is inconsistent with the data set, we can be sure that many other hypotheses are inconsistent as well. If we can devise an algorithm that exploits such relationships, this algorithm may be much more efficient.

In this section we discuss one particular kind of hypothesis space, and show how, given a data set T, the version space can be computed and represented relatively efficiently.

3.3.3 Borders of the version space

We say that a hypothesis h is consistent with a data set T if $\forall(\mathbf{x}, y) \in T : h(\mathbf{x}) = y$. We say that h is consistent with the positive examples if $\forall(\mathbf{x}, \mathit{true}) \in T : h(\mathbf{x}) = \mathit{true}$, and that h is consistent with the negative examples if $\forall(\mathbf{x}, \mathit{false}) \in T : h(\mathbf{x}) = \mathit{false}$.

The following properties then hold:

- If a hypothesis h is consistent with the negative examples, then any hypothesis more specific than h is also consistent with the negative examples.
- If a hypothesis h is consistent with the positive examples, then any hypothesis more general than h is also consistent with the positive examples.
- If a hypothesis h is inconsistent with the negative examples, then any hypothesis more general than h is also inconsistent with the negative examples.
- If a hypothesis h is inconsistent with the positive examples, then any hypothesis more specific than h is also inconsistent with the positive examples.

These properties are interesting because they allow us to prune the search space of hypotheses. We already saw an example of that in the previous section: when a hypothesis is inconsistent with some positive examples in T, then any specialization of that hypothesis is guaranteed to be inconsistent with those examples as well.

These properties can not only be used to prune the search space of hypotheses, however; they can also be used to devise an efficient representation of the version space. To see that, let us first define the *borders* of the version space.

Definition 3.7 (General border) *The general border of a set of hypotheses VS is the set of all the maximally general hypotheses in VS: $G(VS) = \{h \in VS | \nexists h' \in VS : h' \geq h\}$.*

Definition 3.8 (Specific border) *The specific border of a set of hypotheses S is the set of all the maximally specific hypotheses in VS: $S(VS) = \{h \in VS | \nexists h' \in VS : h' \leq h\}$.*

We now have the following important property.

Proposition 3.1 *Let S be the general border and G the specific border of the version space VS of a hypothesis space H. Then for every element h of H it holds that $h \in VS \Leftrightarrow \exists s \in S, \exists g \in G : s \leq h \leq g$.*

Proof: We prove this for finite lattices. We first prove the $\Leftarrow$ direction, then the $\Rightarrow$ direction.

$\Leftarrow$: s and g are in the (borders of the) version space, and therefore consistent with T. Since $s \leq h$ and s is consistent with all positive examples, h must be consistent with all positive examples. Similarly, since $h \leq g$ and g is consistent with all negative examples, h is consistent with all negative examples. This means that h is consistent with all of T, and therefore $h \in VS$.

$\Rightarrow$: $h \in VS$. Consider all the minimal specializations of h. There are two possibilities: either at least one of them is in VS, or none of them is. (1) If none is, h is, by definition, in the specific border of VS, so there indeed exists an $s \in S$ with $s \leq h$, namely h itself. (2) If at least one specialization is in VS, then we can apply the same reasoning on that hypothesis: either it is in S, or at least one specialization of it is in VS. At some point we must reach a hypothesis s in VS that has no specializations in VS (in the worst case we go on until we reach $\emptyset$, which has no specializations at all); s is then in S. Conclusion: in both (1) and (2) there is an $s \in S$ in VS such that $h \geq s$. Similarly, we can show, by considering

generalizations, that there is a $g \in G$ such that $g \leq h$: either h is itself in G and then $g = h$ is a solution, or we can construct a generalization chain from h that is guaranteed to end at the most general concept (?, ?, ..., ?) or sooner, and that ends with a concept $g \in G$. □

The above property is important because it shows that we can always represent the whole version space by just storing its general and specific borders; from these borders, we can tell for each hypothesis h whether it is in VS or not by simply checking whether we find an $s \in S$ and a $g \in G$ such that $s \leq h \leq g$.

3.3.4 The Candidate Elimination algorithm

Consider the following learning task. We are given a hypothesis space H that consists of conjunctive concepts over an instance space $\mathcal{X} = A_1 \times \cdots \times A_D$. Further, a data set T is given; T contains pairs $(\mathbf{x}, y)$ where $\mathbf{x} \in \mathcal{X}$ and $y \in \mathbb{B}$. The task is to compute the version space $VS(H,T)$, that is, the set of all $h \in H$ for which it holds that $\forall(\mathbf{x}, y) \in T : h(\mathbf{x}) = y$.

The Candidate Elimination algorithm can be used to solve this task. The algorithm is incremental: it starts with the complete hypothesis space H as the version space; next, it considers the elements of T one by one, and for each example it adapts the version space by removing all the hypotheses that are contradicted by the example.

Candidate Elimination does not explicitly keep a list of all the hypotheses in the version space; instead, it simply stores the general and specific borders of this version space. As argued before, these borders determine uniquely the full version space.

Thus, Candidate Elimination really computes a pair (G, S), where G is the general border of the version space, and S is the specific border. Given a hypothesis space H and sets of concepts G and S, we define

$$H_S^G = \{h \in H | \exists s \in S, \exists g \in G : s \leq h \leq g\},$$

the set of all hypotheses in H that are "between" S and G.

At the beginning, the version space equals H: each conjunctive hypothesis is in it. The most general conjunctive hypothesis is $(?, ?, \ldots, ?)$, which states that each element of the instance space $\mathcal{X}$ is in the target concept. The most specific conjunctive hypothesis is $\emptyset$, which states that no instances are in the target concept. Thus, G should be initialized to $\{(?, ?, \ldots, ?)\}$, and S to $\{\emptyset\}$.

Candidate Elimination will now incrementally update the version space. That is, at any point in time when a new example $\{(\mathbf{x}, y)\}$ becomes available, if $H_S^G = VS(H,T)$, Candidate Elimination updates G and S such that after the update $H_S^G = VS(H, T \cup \{(\mathbf{x}, y)\})$.

The exact update procedure depends on whether y is *true* or *false*. We consider both cases separately.

Handling a positive example

Consider a hypothesis h that is consistent with a set T of positive and negative examples. Assume that, next, a new positive example $\mathbf{x}$ is encountered. Now there are two possibilities: either $\mathbf{x}$ was indeed predicted positive by h, in which case h is still consistent with all positive and negative examples; or it was not, and then h disappears from the version space at this moment.

The above is true for each $h \in VS(H,T)$, hence it holds also for hypotheses in G and in S. Since Candidate Elimination keeps only this G and S set, it needs to adapt the hypotheses in G and S so that they become the borders of the new version space $VS(H, T \cup \{(\mathbf{x}, y)\})$.

Consider each member g of G separately. If g is consistent with $\mathbf{x}$, it is still in the version space and still a maximally general member of it, therefore it should stay in G. If g is not consistent with $\mathbf{x}$, it is no more in the version space and hence should be removed from G. (One could argue that we could replace it with a generalization of it that does cover $\mathbf{x}$; but g is already maximally general, so none of its generalizations can be in the version space.) Note that when g does not cover $\mathbf{x}$, none of the $h \leq g$ cover $\mathbf{x}$. The effect of removing g from G is that all its specializations will be removed from the version space, and this is indeed exactly what we want.

S contains the most specific hypotheses that are consistent with the data T seen up till now. When a positive example $(\mathbf{x}, \mathit{true})$ is encountered, any hypothesis in S that does not cover $\mathbf{x}$ is apparently too specific, and no longer consistent with $T \cup \{(\mathbf{x}, \mathit{true})\}$. Such a hypothesis needs to be removed, and replaced by one or more new hypotheses that are in VS and maximally specific. Any minimal generalization s' of s that covers $\mathbf{x}$ and for which there exists a $g \in G$ such that $s' \leq g$, has this property. (Indeed, because $s \leq s' \leq g$, s' is consistent with T; because it covers $\mathbf{x}$ it it is also consistent with $T \cup \{(\mathbf{x}, \mathit{true})\}$; and because it is a minimal generalization of s there is no s'' that is more specific and has the same properties.)

For conjunctive concepts, the minimal strict generalization is computed as follows. Let $\mathbf{x} = (v_1, v_2, \ldots, v_D)$. If $s = \emptyset$, it covers no instances whatsoever; the most specific s' that covers $\mathbf{x}$ is then the singleton concept $\{\mathbf{x}\}$, which in our tuple notation for hypotheses is $(v_1, v_2, \ldots, v_D)$ (that is, all attributes must have the same value as $\mathbf{x}$). Otherwise, $s = (t_1, t_2, \ldots, t_D)$, where each t_i is ? or a value from the corresponding domain. Since s does not cover $\mathbf{x}$, at least one of the t_i is not ? and is also not equal to v_i. There may be several such t_i. The most specific hypothesis that is more general than s and covers $\mathbf{x}$ is then obtained by changing each such t_i into ?, and not making any other changes.

The conclusion of all this is the following. When a positive example is encountered, the hypotheses in G should be checked and any $g \in G$ that does not cover $\mathbf{x}$ should be removed; the hypotheses in S should be checked and any $s \in S$ that does not cover $\mathbf{x}$ should be generalized minimally so that it does cover $\mathbf{x}$; and if after these changes there is an $s \in S$ that is not more specific than any $g \in G$, it should be removed from S.

Handling a negative example

The procedure for negative examples mirrors that for positive examples. When a hypothesis in the version space is inconsistent with a negative example $\mathbf{x}$ (it covers $\mathbf{x}$ while it should not), the hypothesis must be made more specific or removed entirely from the border.

Let us start with hypotheses in S. If a hypothesis in S covers $\mathbf{x}$, we would need to specialize it to make it consistent with $\mathbf{x}$; but since every hypothesis in S is already maximally specific with respect to earlier seen examples, we cannot make it more specific without it becoming inconsistent with earlier seen examples; hence, such a hypotheses must simply be removed from S.

Now consider any $g \in G$. If g does not cover $\mathbf{x}$, it need not be changed. If it does cover $\mathbf{x}$, it cannot be in VS, but some of its specializations might. We have to look for minimal specializations of g that do not cover $\mathbf{x}$.

Assume $\mathbf{x} = (v_1, v_2, \ldots, v_D)$, and $g = (t_1, t_2, \ldots, t_D)$ covers $\mathbf{x}$. This implies that for all t_i, $t_i = v_i$ or $t_i =?$. As soon as we change one of the t_i into something that does not match v_i, the resulting hypothesis does not cover $\mathbf{x}$ anymore.

At this point there is an important difference with how we generalized a member of S. When we had a hypothesis that was too specific, we generalized it by changing each t_i that did not match the corresponding v_i into ?. When we have a hypothesis that is too general, we can specialize it by changing just one t_i from ? into any value $v_i' \neq v_i$. (If we would change multiple t_i values, this would no longer constitute a *minimal* specialization.)

As a result, there is usually not one minimal specialization of a hypothesis $g \in G$, but multiple. All of them have the property that they are maximally general elements of the version space, and hence are elements of G.

A consequence of this difference between minimal specialization (which typically gives multiple results) and minimal generalization (which gives only one result) is that the S set will always be a singleton, while G will in general contain multiple elements. *This situation is specific to conjunctive concepts*, however; it is not necessarily true when a Candidate Elimination-like algorithm would be applied to other types of concepts.

The algorithm

The explanation given above naturally leads to an algorithmic description of Candidate Elimination, which is given in Figure 3.3.

3.3.5 Extensions

We have presented the Candidate Elimination algorithm for concepts that can be described as a conjunction of conditions of the form $A_i = v_i$. In fact, it is very easy to extend the algorithm towards conditions of the form $A_i < v_i$ or $A_i \in [l_i, u_i]$ for numerical attributes, or towards other types of conditions.

The main point is that we should have conjunctive concepts of the form $(t_1, \ldots t_D)$ where the t_i are constraints on attribute A_i, and that there should be a method for computing a minimal specialization and generalization of such a constraint, given a new instance. For instance, for a numerical attribute A_i, t_i could be an interval in which the value of A_i should lie. When a new instance is seen, a hypothesis can be generalized to cover that instance by stretching the interval to make it include the instance, or specialized to not cover the instance by narrowing the interval so that it excludes the example.

We have presented Candidate Elimination in a very general form; the algorithm could be used to learn non-conjunctive concepts as well. What changes then are the SPECIALIZE and GENERALIZE procedures; the algorithm itself remains unchanged.

For instance, Candidate Elimination can trivially be used to learn "disjunctive concepts", concepts of the form $C_1 \vee C_2 \vee \cdots \vee C_n$). In this case, the specialize and generalize procedures will be similar to the current ones, but "swapped around": the minimal generalization will now yield multiple results while the minimal specialization will generate a single result; G will be a singleton, while S will typically contain multiple hypotheses.

When we have more complicated concepts than purely conjunctive or disjunctive concepts, the specialization and generalization functions can become much more complex.

function CANDIDATEELIMINATION(dataset T) **returns** pair of sets
    $S := \{\emptyset\}$
    $G := \{(?, ?, \ldots, ?)\}$
    let $T = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$
    for $i = 1$ **to** N:
        if $y_i =$ true **then** *// positive example*
            for all $g \in G$:
                if g does not cover x **then** remove g from G
            for all $s \in S$:
                if s does not cover x **then**
                    $S' :=$ Generalize(s, x)
                    $S := S - \{s\} \cup \{s' \in S' | \exists g \in G : s' \leq g\}$
        else *// y_i = false: negative example*
            for all $g \in G$:
                if g covers x **then**
                    $G' :=$ Specialize(g, x)
                    $G := S - \{g\} \cup \{g' \in G' | \exists s \in S : s \leq g'\}$
            for all $s \in S$:
                if s covers x **then** remove s from S
    return (S, G)

function SPECIALIZE(hypothesis h, instance x) **returns** set of hypotheses
    let $h = (t_1, t_2, \ldots, t_D)$ and $x = (v_1, v_2, \ldots, v_D)$
    R := $\emptyset$
    for $i = 1$ **to** D:
        if $t_i =?$ **then**
            for all $v_i' \in Dom(A_i) - \{v_i\}$:
                $h' :=$ h with t_i changed into v_i'
                add h' to R
    return R

function GENERALIZE(hypothesis h, instance x) **returns** set of hypotheses
    let $h = (t_1, t_2, \ldots, t_D)$ and $x = (v_1, v_2, \ldots, v_D)$
    for $i = 1$ **to** D:
        if $t_i \neq v_i$ **then** $t_i' =?$ **else** $t_i' = t_i$
    return $\{(t_1', t_2', \ldots, t_D')\}$

Figure 3.3: The Candidate Elimination algorithm.

3.3.6 Practical limitations of version space approaches

From a practical point of view, computing the version space is only useful when the learner works in a noiseless setting, that is, when dataset T contains no incorrect values regarding concept membership, nor regarding the description of instances. If there is noise in the data, it may well be that no single concept exists that is 100% consistent with the data. The versions space is then empty. In practice, in such cases, one might be interested in finding the hypothesis that is "least inconsistent" with the data. But that calls for a quite different approach to learning.

Another practical limitation of computing version spaces is that even though, by representing the version space by its borders, we gain a lot of efficiency, these borders themselves can still be very large, to the extent that even computing these borders may not be practically feasible.

For these reasons, the basic version space approaches discussed in this chapter have a limited practical importance. They have a large theoretical importance, however, because they give some idea of how easy or difficult certain learning tasks can be, and also because certain concepts, such as searching in a generality lattice, are foundational for other techniques that we will discuss later.

Chapter 4

Decision trees

Decision tree learning is among the most popular methods for machine learning and data mining. Work in this area started already in the 70's, but it was mostly the work by Breiman et al. on CART [5], and, in parallel, Quinlan's work on ID3 and later C4.5 [34], that made decision trees widely known and tremendously popular as a tool for machine learning and data mining. Part of this appeal comes from the fact that decision trees combine good predictive accuracy with a high interpretability and efficient learning and prediction procedures.

Decision trees are most often used in the context of classification, but they have also been used for regression, as well as for a variety of other tasks. In this chapter, we will first discuss induction of decision trees for classification. In a second part, we extend the method towards other tasks.

4.1 Decision trees: representation and semantics

4.1.1 Definition of decision trees

A decision tree is a tree-shaped structure that represents a function from an input domain X to some output domain Y. The nodes of the tree are called **internal nodes** if they have outgoing edges (children), and **leaves** if they don't. There is a single node that has no incoming edges (i.e., is not a child of any other node); this node is called the **root**.

Each internal node is annotated with a *test*, which is an element from a predefined set of tests $\mathcal{T}$. Each test is a function τ that maps any $\mathbf{x} \in X$ to a finite domain R_τ. The elements of R_τ are called the possible outcomes of the test τ. An internal node annotated with a test τ has for each possible outcome r exactly one outgoing edge labeled with that outcome. Each leaf of a decision tree is annotated with a value $y \in Y$.

A decision tree maps any $\mathbf{x} \in X$ to a single $y \in Y$ as follows. Starting with the root node of the tree, we construct a path from the root to a leaf by computing in each node the outcome of its associated test for $\mathbf{x}$, and following the outgoing edge that is labelled with that outcome, until a leaf is reached. The value y stored in that leaf is the value to which $\mathbf{x}$ is mapped by the tree.

Example 4.1 Leap years are years that have 366 days instead of 365. A year is a leap year if it is a multiple of 4 and not of 100, or if it is a multiple of 400. Using tests of the form "is a multiple of", the procedure to decide whether a year is a leap year or not can be written as the decision tree shown in Figure 4.1.

We next define decision trees and their semantics a bit more formally.

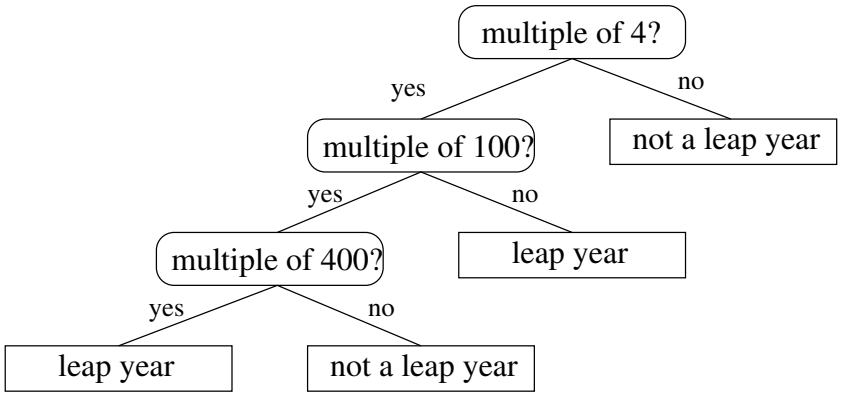

Figure 4.1: A decision tree answering the question whether a given year is a leap year or not.

Definition 4.1 *Given an instance space X, a set of tests $\mathcal{T}$ that contains functions $\tau : X \to R_\tau$, and an output space Y, a decision tree over $(X, Y, \mathcal{T})$ is a structure defined as follows:*

- $\mathbf{leaf}(y)$ *(with $y \in Y$) is a decision tree.*
- $\mathbf{inode}(\tau, S)$ *is a decision tree if and only if S is a set of couples (r_i, t_i) such that: each r_i is an element of R_τ; for each element r of R_τ there is exactly one couple (r_i, t_i) such that $r = r_i$; and each t_i is a decision tree over $(X, Y, \mathcal{T})$.*

We define the *relevant subtree* for $\mathbf{x}$ in t, denoted $rst(\mathbf{x}, t)$, as follows:

$$rst(\mathbf{x}, \mathbf{inode}(\tau, S)) = t' \Leftrightarrow (\tau(\mathbf{x}), t') \in S.$$

We can then define the *relevant leaf* for $\mathbf{x}$ in t", denoted $rl(\mathbf{x}, t)$, as the transitive closure of the relevant subtree function, i.e., as the smallest function that satisfies

- $rl(\mathbf{x}, \mathbf{leaf}(y)) = \mathbf{leaf}(y)$
- $rl(\mathbf{x}, t) = rl(\mathbf{x}, rst(\mathbf{x}, t))$ if t is not a leaf

Given a decision tree t over $(X, Y, \mathcal{T})$, the **function represented by** t is the function that maps any $\mathbf{x}$ onto y if and only if $rl(\mathbf{x}, t) = \mathbf{leaf}(y)$.

4.1.2 Decision trees in the attribute-value setting

In the context of machine learning, decision trees are usually set in the attribute-value learning setting. A test is then typically a function mapping $\mathbf{x}$ onto its value for one of the predefined attributes. As the attributes can themselves be seen as functions, we can say that the tests are simply the attributes themselves.

Decision trees typically predict the value of one particular attribute, called the target attribute, from the other attributes. Let the attributes be $A_1, A_2, \ldots, A_D$ and let A_D be the target attribute; then the output space Y is the set of all values that A_D can take, and the tests τ_i are of the form A_i with $1 \leq i < D$. (Evidently, we are not allowed to use A_D as a test: this is the value that we need to predict.)

Example 4.2 Our "leap year" example could be represented in attribute-value format. To this aim we need to assume that there are boolean attributes available of the form "multiple of n". Assume, for instance, that we have a table with the following attributes:

- A_0: is the year itself. We will write it simply as "Year" instead of A_0.
- for $1 \leq i \leq 999$, A_i is true if and only if the year is a multiple of i.
- A_{1000} is true if and only if the year is a leap year. We will write this attribute as Leap.

The following table represents a few years:

Year	A_1	A_2	A_3	A_4	...	A_{999}	Leap
500	true	true	false	true	...	false	false
1999	true	false	false	false	...	false	false
2000	true	true	false	true	...	false	true

One decision tree that predicts Leap from the attributes A_1 to A_{999} would look exactly as in Figure 4.1, but with the tests A_4, A_{100} and A_{400} filled in in the nodes. The other attributes are not used.

Another decision tree that can predict Leap correctly is the tree that contains just a test on Year. Assuming that we define the domain of Year as simply those numbers that occur in the table, this test would have as many outcomes as there are years in the tables; for each particular outcome the tree would state whether it is a leap year or not. Obviously, such a tree would not be very interesting, in the sense that it is not giving us a generally applicable rule to say whether a year is a leap year or not.

4.2 Learning decision trees from data

Decision tree learning refers to the task of learning from a set of data a decision tree that is consistent with the data. A tree is **consistent** with a data set T if it represents a function f such that $\forall(\mathbf{x}_i, y_i) \in T : f(\mathbf{x}_i) = y_i$.

More precisely, the task can be described as follows.

Task description: decision tree learning

Given: a training set T containing examples $(\mathbf{x}_i, y_i)$, $i = 1 \ldots N$, where $\mathbf{x}_i \in X$ and $y \in Y$, and a set of tests $\mathcal{T}$,
Find: a decision tree t over $(X, Y, \mathcal{T})$ that represents a function $f : X \to Y$ such that $\forall(\mathbf{x}_i, y_i) \in T : f(\mathbf{x}_i) = y_i$.

This is a very general task description. In practice, most algorithms work in a more limited setting. The simplest setting is that of learning classification trees from attribute value data. Even for this setting, a lot of sophistication has been built into decision tree learning algorithms. We therefore start with a basic algorithm for this simple setting, gradually introducing more details and sophistication. By the end of this chapter, we will return to the more general case as described above.

4.3 Learning classification trees from nominal attribute-value data

We start with the simplest case of decision tree learning, which is: learning a decision tree in the attribute-value setting where all attributes, including the class attribute, are nominal. Since we have a nominal class attribute, we are performing classification, therefore the trees we learn here are called **classification trees.**

4.3.1 Task description

Without loss of generality, we assume $\mathbf{x} = (x_1, x_2, \ldots, x_{D-1})$ with each x_i a value for a corresponding attribute A_i, and A_D is the class attribute (i.e., x_D performs the role of y).

We define the set of tests $\mathcal{T}$ as follows. For any attribute A_i, $i = 1 \ldots D-1$, there is a test τ_i that maps $\mathbf{x}$ onto its value for A_i. The result set for τ_i is therefore the domain of A_i, which is finite. We will write the domain of A_i simply as A_i.

This is a rather trivial choice for $\mathcal{T}$: many other, more sophisticated, sets of tests could be defined. For classification trees for nominal data, however, $\mathcal{T}$ is almost always defined as shown here. We will see later that in the case of numerical attributes, the definition of $\mathcal{T}$ becomes less trivial.

Note that finding a tree that is consistent with T is quite easy in this setting: we can just build a tree that keeps splitting examples until each example is in a different leaf, after which we can assign to each leaf the label of the example it contains. Such a tree always exists, since any two examples $(\mathbf{x}_i, y_i)$ and $(\mathbf{x_j}, y_j)$ where $\mathbf{x}_i \neq \mathbf{x}_j$ can always be put in different subsets (by testing one of the attributes where $\mathbf{x}_i$ and $\mathbf{x}_j$ differ). In fact, many such trees usually exist. A more challenging task is to find the *smallest* tree that is consistent with T.

Making all the above assumptions, the task description for classification tree learning from nominal data looks as follows:

Task description: learning classification trees from nominal data

Given: a training set T containing examples $(x_{i1}, x_{i2}, \ldots, x_{i,D-1}, x_{iD})$, $i = 1 \ldots N$, with $x_{ij} \in A_j$,
Find: a decision tree t over $(X, Y, \mathcal{T})$ that represents a function $f : X \to Y$ such that $f((x_{i1}, \ldots, x_{i,D-1})) = x_{iD}$, with $X = A_1 \times \cdots \times A_{D-1}$, $Y = A_D$, and $\mathcal{T} = \{\tau_i | 1 \leq i \leq D-1\}$ with $\tau_i((x_1, x_2, \ldots, x_{D-1})) = x_i$. In addition, try to make t as small as possible.

It is impossible to create an efficient algorithm that always finds the smallest tree consistent with a set T, however: it has been shown that this problem is NP-hard [21, 46]. Therefore, most practical algorithms will employ heuristics to try to find small trees, without guaranteeing that they find the smallest one.

In the following subsections we explain how this particular task can be solved. We first discuss the basic learning algorithm on a high level, then we zoom in on a number of non-trivial details.

4.3.2 The basic learning algorithm

Many algorithms for learning decision trees build these trees top-down, from the root towards the leaves. A generic name for this class of algorithms is TDIDT, which stands for **top-down induction of decision trees**. The approach is also known as **recursive partitioning**, or as **divide-and-conquer**, for reasons that will be clear soon.

We here start with a very simple version of TDIDT that is based on ID3. ID3 was one of the first and most influential algorithms for decision tree learning. With many improvements consecutively added to it, it developed into a system called C4.5, which has been an important benchmark algorithm in machine learning for a long time. A reimplementation of C4.5 in Java has been included in the Weka data mining toolbox under the name J48. J48 is currently probably the most used reference algorithm for decision tree learning.

The way in which a TDIDT algorithm proceeds is as follows. The algorithm starts with the root of the tree, trying to decide which test should be put there. This decision will obviously be based on the training set. Once it has chosen that test, it puts it in the root and creates a child node of the root for each possible outcome of the test. Let us denote with c_r the child node corresponding to outcome r. For each child node c_r, the algorithm next creates a data set containing those examples from the training set for which the outcome of the test is r. The algorithm proceeds with learning for each of these separate subsets a decision tree in exactly the same way: choose a test, divide the set into subsets, repeat the procedure on these subsets. As soon as a subset is found that is "pure", meaning that all the examples in the subset have the same class, the procedure stops. (In the worst case, it stops when all subsets contain one element.)

Example 4.3 We continue our "leap year" example. When building the tree, we can start with the root, asking ourselves what test would be a good test to put here. The test A_1 is obviously not a useful test: it has the value *true* everywhere, so A_1 would "divide" the data set in two sets, one of which is empty and the other is the original data set. That's not helping in any way. The test A_2 is better: this test will divide the dataset into even and odd years, and the set of odd years is pure (all odd years are regular years). The test A_4 is even better: it divides the years into multiples and non-multiples of 4, and the set of non-multiples of 4 will be pure. Since the set of non-multiples of 4 is larger than the set of odd numbers, we get a larger pure subset, which intuitively seems better.

So let us assume we choose A_4 for the root. The set of years is then divided into two subsets. The subset containing non-multiples of 4 is pure, hence we are finished with this set. The subset containing the multiples of 4 is not pure yet (not all multiples of 4 are leap years), so the procedure is repeated on this subset. At this moment the test A_{100} seems a good choice: it will again divide the set into one pure subset (among the multiples of 4, all non-multiples of 100 are leap years) and a non-pure subset containing multiples of 100. The procedure is repeated on the non-pure subset, and the test A_{400} finally gives us two pure subsets.

The general procedure for building a decision tree is now known, but we have left a number of questions unanswered, the most important one being: exactly how do we choose the most suitable test for the root node, and later for lower nodes in the tree? This question is answered next.

4.3.3 Choosing the "best" test

When we include a test somewhere in the tree, we would like it to be as informative as possible about the class of the instance that we are testing. A test is maximally informative if, whatever the outcome of the test is, given this outcome we will have certainty about the class of the instance. This happens when all examples with a given outcome belong to the same class. For instance, if the test has possible outcomes a, b, and c, then all the instances with outcome a must have the same class (if instances with outcome a can belong to two or more classes, then given an outcome a we still cannot be sure which of these two classes a particular instance belongs to), and the same holds for b and c.

We say that a test is *perfect* if for each possible outcome of the test, all instances having that outcome belong to the same class. A perfect test creates a partition in which each subset is pure with respect to the class.

Very often, we will not be able to find a test that is perfect. A non-perfect test will create a partition where not all the subsets are pure, and these subsets will have to be split further.

Our goal is to construct a tree that is as small as possible. More precisely, we want a tree where the average length of a path from root to leaf is minimal. (The length of the path from a root to a leaf corresponds to the number of tests we will have to perform on an instance before we know its class, and this is what we want to be as small as possible.) For that reason, a perfect test is preferable, but if no perfect tests are possible, we want to choose tests that are as close to perfect as possible. Intuitively, a test is closer to being perfect if it creates more subsets that are pure, and if the non-pure subsets it creates are closer to pure.

Attempt 1: maximize purity

One might expect that we can score tests based on the "average purity" that they yield. Let us define the **purity of a set** S, denoted $p(S)$, as the proportion of instances that belong to the most frequent class. E.g., if 20% of the instances are positive and 80% are negative, then the purity is 0.8 (80%). If 30% belong to class A, 30% to class B, and 40% to class C, then the purity is 0.4. Etc. If all instances belong to the same class, the purity is 100%.

We can then define the **purity of a partition** $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ over a set $S = S_1 \cup S_2 \cup \ldots \cup S_k$ as

$$p(\mathcal{S}) = \sum_{S_i \in \mathcal{S}} \frac{|S_i|}{|S|} p(S_i)$$

(note that the average is weighted according to the size of each subset S_i).

Finally, the **purity of a tree** t is the purity of the partition defined by its leaves. Assuming that the prediction stored in each leaf is the majority class of the examples in that leaf, the purity of a tree is also its predictive accuracy, the probability of making a correct prediction for a random instance.

Since we want to maximize the purity of the tree we will eventually get, it seems reasonable to choose that test that leads to the partition with the maximal purity.

Unfortunately, this turns out not to work very well. To see why, consider the following example.

Example 4.4 Assume we have a set S with 80 positives and 20 negatives, and the following two binary partitions of S: $\mathcal{S}_1 = \{S_1, S_2\}$ with S_1 and S_2 each having 40 positives and 10 negatives $\mathcal{S}_2 = \{S_3, S_4\}$ with S_3 having 40 positives and 0 negatives, and S_4 having 40 positives and 20 negatives.

Partition $\mathcal{S}_1$ yields subsets that are just as impure as the original set S. Partition $\mathcal{S}_2$ yields one subset that is pure, and another that is still not pure; in fact this one is more impure than the original set.

Intuitively, it makes sense to prefer $\mathcal{S}_2$ over $\mathcal{S}_1$, since for 40 out of 100 cases, the class will be known after this one test. However, if we compute the purity of $\mathcal{S}_1$ and $\mathcal{S}_2$, we get 0.8 in both cases, which is also the purity of the original set S.

Generally, the purity of a partition will be exactly the same as the purity of the original set, unless the majority class in at least one subset changes. (An analysis of this effect is provided later this chapter.) But even when the majority class does not change, a test may be informative: this was the case in our example. Therefore, purity is not a good criterion for evaluating the usefulness of a test.

Attempt 2: maximize information gain

A better criterion is provided by information theory. In information theory, there is the notion of missing information, which can intuitively be interpreted as "how many binary (yes/no) questions do we need to ask, on average, to find the answer to some question?". This missing information is also called **entropy**.

To understand entropy, assume that you need to guess a number x with $1 \leq x \leq 8$, and you can ask only questions that can be answered by yes or no. How many questions do you need to ask before you know for sure what x is?

It can always be done with three questions. The question "is it even" always leaves us with 4 possible values for x; next asking "is it greater than 4" will always leave us with two possibilities, and finally the question "is it greater than c", with c 2 or 6 depending on the situation, will give us certainty. Therefore, the "missing information" in this problem is 3.

An alternative guessing strategy would be: "Is it one? Is it two? Is it three? ...". If the chosen number was one, we will know after one question. If it was two, we will know after two questions. If it was 7 or 8, we will know which one of these it is after the seventh question. So sometimes this strategy will be better than the previous, sometimes it will be worse. But *on average*, assuming each number is equally likely, we will need $(1+2+3+4+5+6+7+7)/8 = 35/8 = 4.375$ questions: more than in the previous case. So if we play the game multiple times, consistently using the first strategy will be more efficient than consistently using the second strategy.

In fact, the first strategy is optimal: if we need to guess one from 8 equally likely numbers, 3 binary questions is the minimum that any strategy can on average require. Note that 3 is the binary logarithm of 8. Generally, if we start with n possibilities and ask questions that reduce the number of possibilities by half whatever the outcome is, we will always need $\log_2(n)$ questions.

If the numbers are not chosen with equal probability, however, the optimal strategy may change. If, for instance, half of the times the number to be guessed is one; in 1/4 of the times it is 2; and in 1/4 of the cases it is 3 or more (with all those numbers equally likely, i.e., all having a probability of 1/24), then the average number of questions becomes $(1/2*1+1/4*2+1/24*(3+4+5+6+7+7)) = 2.33$. The original strategy would still require exactly three questions.

Thus, the average number of questions required by an optimal strategy is not always $\log_2(n)$: this is only the case when all answers are equally likely. When not all answers are equally likely, it turns out that the optimal strategy requires on average

$$-\sum_i p_i \log_2(p_i)$$

questions, where i ranges over all possible answers and p_i is the probability of each answer being the correct one. This formula defines the entropy of a variable.

Definition 4.2 (Entropy) *Given a variable V that can take values v_i, each with a probability p_i, the entropy of V is defined as*

$$s_V = -\sum_i p_i \log_2(p_i)$$

Exercise 4.1 Show that if each value of V is equally likely, i.e., $p_i = 1/n$ for all p_i, then the entropy of V is $\log_2(n)$. (This is consistent with our earlier example.)

Note now that, when learning a decision tree, we have exactly the same problem: we need to guess the class of an instance, and we want to do that with as few questions as possible. The entropy of the class variable tells us how many binary questions we would need to ask, on average, if we use an optimal strategy.

There is, however, one catch. In the decision tree setting, we are not restricted to binary questions, but there are other restrictions. We cannot ask "any" question; we can only ask questions that correspond to asking the value of one particular attribute. And we cannot ask direct questions about the class of the instance. (If we could, then we would always need only one question to find out what the class

is.) Nevertheless, for those questions that we can ask (the tests in $\mathcal{T}$), we can still measure how good a question is by checking how much it reduces the entropy on average.

Given a set S, the class entropy (i.e., the entropy of the class variable) is

$$s_C(S) = -\sum_c p(c) \log_2(p(c))$$

with c ranging over all possible classes and $p(c)$ the probability that a random element from S has class c.

If we partition S into subsets S_i, then the entropy in each subset is

$$s_C(S_i) = -\sum_c p_i(c) \log_2(p_i(c))$$

with $p_i(c)$ the probability that a random element from S_i has class c.

The average class entropy in a partition $\mathcal{S}$ is the weighted average of the entropies of the S_i:

$$s_C(\mathcal{S}) = -\sum_i \frac{|S_i|}{|S|} \sum_c p_i(c) \log_2(p_i(c))$$

The **information gain** of a test τ, finally, is the difference between the entropy of S and the average entropy of the partition $\mathcal{S}$ that τ induced:

$$IG(\tau, S) = s_C(S) - s_C(\mathcal{S})$$

with $\mathcal{S}$ the partition of S induced by τ.

It turns out that, while purity is not a good criterion for checking whether a test is useful or not, entropy, and the information gain criterion that is derived from it, are good criteria.

Exercise 4.2 For the 80%, 20% example that we used earlier to show that average purity is not a good measure for the usefulness of a test, show that information gain does work well. To show this, compute for the partitions $\mathcal{S}_1$ and $\mathcal{S}_2$ in that example the information gain, and interpret the result.

The conclusion of all this is: the "best" test to be filled in in a node, is the one that yields the highest information gain. By consistently choosing this test, we will on average get trees that are as shallow as possible (i.e., trees where the path from root to leaf is shortest, on average).

Thus, given a node and a corresponding set of data, ID3 will consider all possible tests, compute the partition they induce, compute the information gain for that partition, and finally choose the test for which this information gain is highest.

4.3.4 A basic tree building algorithm

ID3's basic tree building algorithm is shown in Figure 4.2. Given a dataset T with only nominal values, it builds a tree that fits T perfectly. In line with our earlier description of TDIDT algorithms, the algorithm proceeds top-down, starting with the root of the tree. At each node it first checks whether the node is pure. If yes, then the node is turned into a leaf that predicts the one class that all the instances in the node belong to. If the node is not pure, then the algorithm considers all available attributes and finds that attribute that is most informative, according to information gain. It creates a subset S_j for each possible outcome v_j of that attribute. For each non-empty subset thus created, it calls itself recursively on that subset, building a subtree t_j that will be associated with v_j in the tree. If a subset

function ID3(S: set of examples) **returns** decision tree:
    let c be the most frequent class in S
    if S is pure **then return leaf**(c)
    else
        for each attribute A_i:
            for each possible value v_j of A_i:
                $S_{ij} := \{\mathbf{x} \in S : A_i(\mathbf{x}) = v_j\}$
            $IG_i := s_C(S) - \sum_j |S_{ij}|/|S| \cdot s_C(S_{ij})$
        let m be such that $IG_m \geq IG_i$ for all $i \neq m$
        for all S_{mj}:
            if $S_{mj} = \emptyset$ **then** $t_j :=$ **leaf**(c)
            else $t_j :=$ ID3(S_{mj})
        return inode(τ, $\{(j, t_j)\}$)

Figure 4.2: A simplified version of the ID3 algorithm for top-down induction of classification trees.

is empty, it creates a leaf for that subset, using the majority class in the current node as the prediction for that leaf. (This is the best we can do in this case: since there are no examples in that subset, we have no idea what class examples ending up in this subset would have.)

4.3.5 Representation power of decision trees

We have already pointed out that for any training set T with nominal attributes, it is possible to construct a decision tree that is 100% pure on T. This implies that also if the training set consists of the whole instance space X, with for each $\mathbf{x} \in X$ a value $y = f(\mathbf{x})$ associated, we can find a tree that expresses exactly the function f, whatever that function is. In other words, classification trees can represent any function from a nominal input space to a nominal output space. As a corollary, when the input space consists of boolean variables, decision trees can represent any boolean function.

When the output space is boolean, a decision tree can be said to define a concept, namely the set of all instances for which the boolean function represented by the tree is true. It follows from our discussion that decision trees can represent any concept. Contrast this to, for instance, the Versionspaces approach; standard algorithms for Versionspaces make the assumption that the target concept can be represented as a conjunction of conditions. This assumption is quite restrictive. Decision trees make no such assumptions.

4.4 Avoiding overfitting

In our discussion of entropy and information gain, there is one important detail that we have ignored up till now. We will want to use decision trees not just for classifying instances in the training set T from which we learn the tree, but for classifying any instances from X, also outside T. From this point of view, the quantities $p(c)$ used in the definition of class entropy refer to the actual probability that a random example from X has class c, and $p_i(c)$ is the probability that a random example from X that "would have been in subset S_i if it had been in S" (which means that the test τ gave the result corresponding to S_i) has class c. In practice, we do not know these probabilities. We can only estimate them, using

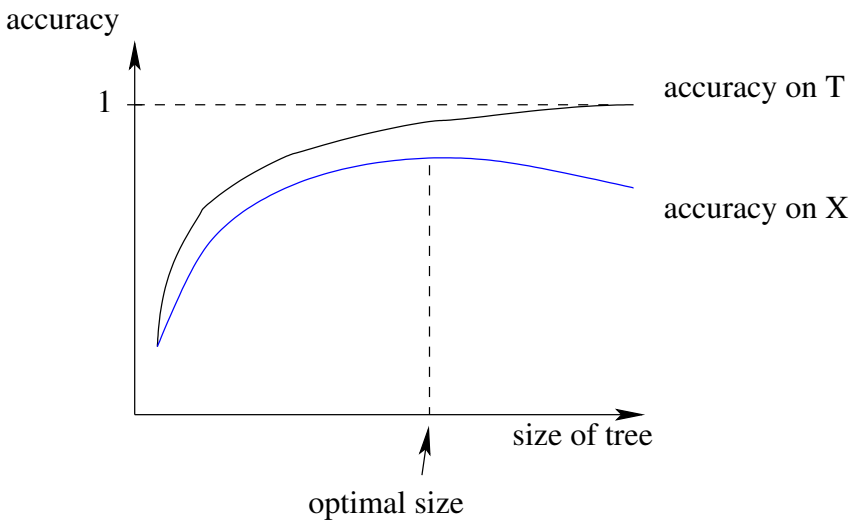

Figure 4.3: Typical evolution of the accuracy of the current tree (on the training set, and on the instance space) as a function of its size, during the tree building process.

the proportion of instances in S that have a certain class as an estimate for the probability that an instance in X has this class.

This is important because when the S_i are small, the estimated probability can deviate significantly from the real probability, and the same holds for the estimated and real entropy. In particular, any subset of size 1 will always have an estimated entropy of 0 (since it contains only one element), and a subset of size 2 will always have an estimated entropy of either 0 or 1. These will usually be bad estimations for the real entropy. Only for large subsets S_i will we generally have a reasonably good estimate.

The difference between estimated and real entropy is an example of a more general problem. Ideally, we should try to learn a tree with pure leaves. But we can measure purity only on the training data T, not on the instance space X; and the measured purity may be far away from the actual purity for small leaves. It is almost always possible to learn a tree that is pure when evaluated on the training set: it is sufficient to keep splitting subsets until subsets of size one are obtained, since these are always pure. But it is not obvious that the tree thus obtained will be pure on the instance space.

Thus, while a tree may seem to become purer when we keep on splitting, there is usually a point where the increased purity of the tree on T does not result in increased purity on X. This phenomenon is called **overfitting**. Overfitting happens when we are building a tree that seems to performs remarkably well when evaluated on T, but that in fact performs less well on X.

Figure 4.3 illustrates the typical evolution, during the tree construction process, of the purity of a tree on T on the one hand, and the accuracy with which it makes predictions for intances not in T on the other hand.

It is clear that it may not be a good idea to keep splitting sets until we get pure subsets, regardless of the size of those subsets. We need to trade off the size of the leaves of the tree against their purity on T, taking into account that an increased purity on T may actually coincide with a decreased purity on X if the leaves of t become too small (i.e., if the tree becomes too large).

It is not obvious, however, how we should do this. Generally, it is difficult to see whether a split that yields some information gain is really a useful split, or just

seems useful because of random effects (i.e., the split accidentally works well on this particular training set T but would not have worked so well on another set).

Two approaches can be taken to avoid overfitting. One is to *stop growing* the tree when there is no guarantee that further splits will be useful. A second one is to grow the tree completely, up till leaves of size one if necessary, but afterwards *prune* the tree, cutting away those branches for which it seems not very likely that they actually improve the purity of the tree.

Of these two, the second has been found to work better in practice. A problem with the first approach is that when a test in itself does not seem very good for some node, this does not necessarily mean that there cannot exist a good subtree (containing multiple tests) with this test in its root.

Example 4.5 Consider the example of a dataset with boolean attributes $X_1, \ldots, X_{10}$, and where the target function to be learned is X_1 **xor** X_2. Suppose we have a set of 100 examples where each combination of true and false for X_1 and X_2 occurs 25 times. Then neither X_1 nor X_2 will yield any information gain (check this). Yet, the tree containing X_1 in the root and X_2 on the second level will exhibit perfect prediction.

For this reason, most practical algorithms work in two phases: a growing phase where they keep splitting sets until sets of the minimal size are obtained, and a pruning phase where they turn subtrees into leaves when it turns out that these subtrees have not yielded any significant improvement. This "improvement" can be interpreted in two ways: improved purity on T, or on X.

No improved purity on T

First, we could say that a subtree does not yield an improvement if its purity on the training set is the same as the purity of its root. Remember that the heuristic used to split a node is not based on purity but on information gain, and that tests may yield information gain without immediately leading to improved purity. The idea is that the purity can still improve later on. But when a maximal subtree has been grown, and there is still no purity improvement, then we know that the whole endeavor of growing this subtree has been in vain, and we can again remove this subtree.

Example 4.6 Suppose we start with a set with 10 positives and 90 negatives (denote this 10/90). A first split might give rise to two sets with 10/60 and 0/30 distributions, respectively; this split yields a clear information gain. The left subset might then be split further into sets 10/40 and 0/20, and the left set again into 10/25 and 0/15. Assume that at this point no further splits of the impure 10/25 set are possible. The majority class in this leaf is the negative class. The tree will therefore predict negative in all leaves. Such a tree evidently does not yield better prediction than just predicting the majority class in the root; that would also consistently give negative predictions.

No improved purity on X expected

A second possible reason for pruning is that, even if the tree does have an increased purity on T, it may not be obvious that its purity on X is also increased. Typically, when growing a tree, its purity on T consistently goes up, but there is a point where the purity on X goes down again. From this point onwards, the tree overfits.

Example 4.7 Figure 4.4 shows an example of a dataset where one single instance is a bit atypical. There exists a small tree that is consistent with all the examples

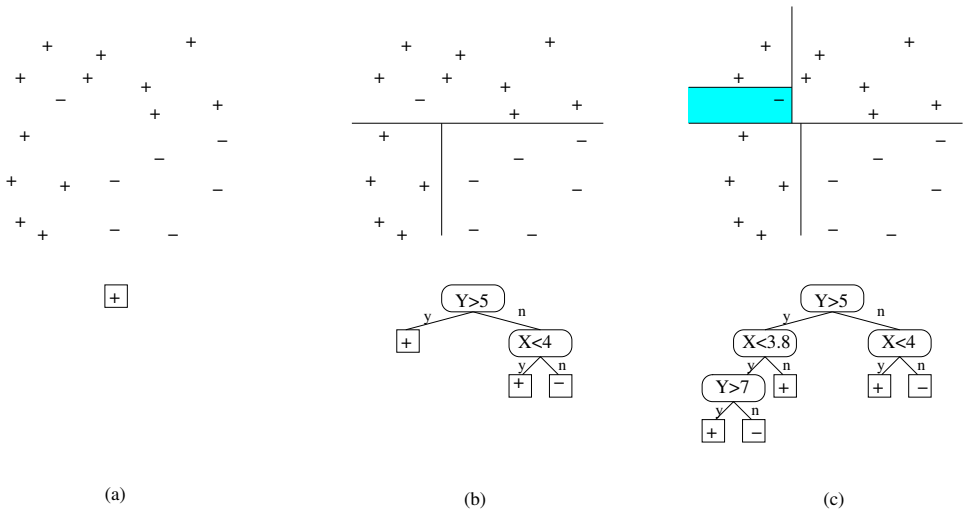

Figure 4.4: Decision trees and their corresponding decision surfaces. The left "tree" (a) is simply a leaf; by always predicting positive, it underfits the data and is likely to have low predictive accuracy. The middle tree (b) fits the data well, but not perfectly. The right tree (c) fits the data perfectly but probably overfits the data: the whole shaded area will be predicted negative, though visually it seems likely that it will contain mostly positive instances.

except this one, and there is a larger tree that is consistent with all examples. However, the large tree will predict all the examples in a relatively large area to be negative, without there being much reason for that. By trying to fit the one exceptional example the tree will probably make more mistakes, instead of fewer, on unseen examples.

Of course we would like to find the tree with maximal purity on X, rather than on T. But we cannot directly measure purity on X. It needs to be estimated. We already know that the purity on T is not a suitable estimate. Several more sophisticated methods have been proposed.

Estimating the purity on X

Using a validation set While the purity on T is a too optimistic estimate for the actual purity of the tree on X, its purity on a set of data *that has not been used for training the tree* is an unbiased estimate. If we have not been adapting the tree to fit the data in that so-called validation set, then there is no reason to think that the tree will fit that validation set any better than a random set from the instance space.

So, when learning a tree, instead of using all the available data (call this set D) to learn the tree, we might choose to set apart a validation set V. We then use as training set $T = D - V$. After the tree has been grown on T, we can test its predictive accuracy on V, and this will be a good estimate of the true accuracy of the tree on unseen data.

The use of this estimate for pruning trees is known as **reduced error pruning**. This method works well, but has the disadvantage that it requires a separate validation set V, which means that less data will be available for training. It is therefore mostly useful when data are available in abundance.

Heuristic methods If we want to avoid the use of a separate validation set for pruning, we need to somehow estimate the accuracy of a tree t on the instance space, given only t and its accuracy on T. Many different methods for this have been proposed. Unfortunately, all of them are somewhat heuristic in nature, and it is not generally accepted which one works best.

Most methods somehow trade off the complexity of the tree against its accuracy on T. That is, a slight decrease of accuracy on T may be considered acceptable if we can obtain a large reduction of tree size in this way. The idea is that smaller trees are less likely to overfit, and hence the apparent reduction of accuracy on the training set may not correspond to a reduction of accuracy on unseen data.

One method that has been proposed is to use a constant penalty α for each node added to the tree. The "cost" of a tree is then its size multiplied with α, plus its error on T, and it is this cost that is minimized. A slightly higher error (lower accuracy) is then acceptable if the tree becomes so much smaller that $n\alpha > \delta$, with δ the increase in error and n the number of nodes that have been pruned away.

In the CART system for decision tree learning, this method is used. The method relies, of course, on having a suitable value for α: with a high value very small trees will be obtained, and with $\alpha = 0$ that tree is learned that has maximal accuracy on the training set. The optimal value for α is determined empirically by CART.

Another method is error based pruning (EBP). This method, used in the C4.5 system, relies on estimating the error of a tree not directly from the observed errors (on T) in its leaves, but from upper bounds on those errors. Generally, for a small leaf, the error on T will be a less good approximation for the actual error than for a large leaf. In EBP, based on the error in a leaf and the size of the leaf, an upper bound on the true error for that leaf is constructed. This upper bound is further away from the leaf's error on T if the leaf is smaller. The estimated error of the tree on unseen data is the weighted average of the estimated errors of the leaves.

Finally, there are methods based on the MDL (minimal description length) principle. These methods again trade off the complexity of the tree against its error, but now using methods from information theory. Essentially, one could say that a possibly imperfect tree (or more generally a hypothesis in whatever format), together with corrections for the errors its makes, forms a perfect description of the data. We want to make this description as small as possible. Assuming we have some encoding scheme that allows us to represent both trees and their exceptions, we can for any tree t and list of corrections c, encode them as a bitstring. A larger tree will lead to a longer bitstring, but fewer errors lead to fewer corrections. Thus, the encoding length of the tree and its exceptions give us a more or less principled way of trading off the size of the tree and its accuracy.

Pruning trees

Algorithms for pruning trees typically work bottom-up. Nodes are considered one at a time, and it is checked for each node whether pruning the tree at that node (i.e., turning the node into a leaf) yields a "better" tree than keeping the subtree that is currently there. The word "better" refers of course to the estimate of the tree quality that one is using.

4.5 Improvements and extensions to the algorithm

In the above section we have discussed the basic procedures for growing a tree and pruning it afterwards. In this section we discuss some details that lead to improvements to these basic procedures.

4.5.1 Information gain ratio

The information gain heuristic has a particular shortcoming that needs to be addressed. In a sense, it is not fair to compare the information gain of tests that can yield many outcomes to that of tests with only a few possible outcomes. The information gain (reduction of entropy) that is obtainable by a binary test can be at most 1. But a test that can have multiple outcomes can yield an information gain of at most $\log_2 k$ with k the number of outcomes. So tests with many outcomes have a much better chance of scoring higher than binary tests.

It is not obvious whether this is desirable or not. We really want to take the most informative test. That tests with many outcomes are more likely to be more informative is not a problem from this point of view.

However, tests with many outcomes will necessarily lead to small subsets in the partition, and thus they more easily give rise to overfitting. As an extreme illustration of this, recall the "leap year" example we saw earlier: choosing the attribute Year as first test (which indeed has maximal information gain in this case) leads to a tree that is correct on the data, but useless for prediction purposes.

For this reason, it has been proposed not to use information gain itself, but to compare it to the maximal information gain that could be obtained by any test with the same number of outcomes and the same subset sizes. This maximal information gain is called the *split information*, and is defined as

$$SI(S,\tau) = -\sum_i \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

A test can never yield a higher information gain than its split information, and hence the quantity

$$GR(S,\tau) = IG(S,\tau)/SI(S,\tau)$$

with $IG(S,\tau)$ the information gain of τ on S as defined before, will always be within $[0,1]$. A gain ratio near 1 implies that a test is optimal, given the number of outcomes that it has.

While this changed heuristic was experimentally shown to work better in some cases, it suffers from some instability for tests with a very low SI value. Therefore this heuristic is usually combined with an additional test that SI is not too close to zero. In C4.5, tests are automatically disregarded if their information gain is less than average. Thus, C4.5 uses a combination of information gain and gain ratio: tests are automatically disregarded if they have a low information gain, but among those with high enough information gain, the winner is the one with a higher gain ratio.

A particularity of the gain ratio heuristic is that it tends to yield less balanced splits: for binary splits, for instance, it is more likely to create one small (and almost pure) subset and a large other subset, while the information gain heuristic has a tendency to create more balanced splits.

4.5.2 Learning from numerical data

When we have numerical input data, it may not be feasible to have a different branch for each possible value. This holds in particular when we have continuous attributes, which can have an infinite number of values.

An obvious way to deal with continuous attributes is to have a slightly different format for tests. Instead of having for each attribute A a test τ_A that maps an instance on its value for A, for continuous attributes we create a test of the form $A < c$, with c a constant. This test has two possible outcomes, true or false; and for each outcome two subtrees are created.

Clearly, there is now no one-to-one correspondence between attributes and tests anymore. For a given attribute A, multiple tests can be defined, all of them differing in the constant that they use. Thus, on a single branch from root to leaf, it may be that the same attribute is tested multiple times.

The decision which test is best is now also more complicated: we need to decide not only what the best attribute is, but also what the best threshold value c is for that attribute. How can we decide that, given that there are infinitely many possible values?

One useful observation is the following. Let's say that the series $a_1 < a_2 < \ldots < a_k$ lists all the values for attribute A that occur in the training set T. It is sufficient to consider as possible thresholds only one value between each couple of consecutive a_i, for instance, $c_i = (a_i + a_{i+1})/2$ with $i = 1, \ldots, k-1$. Indeed, if there are no values between a_i and a_{i+1} in the training set, then any test $A < c$ with $a_i < c < a_{i+1}$ will split T in exactly the same subsets.

But we can do even better. Generally, it also does not make sense to consider a threshold between a_i and a_{i+1} if the class of all the examples having a_i and a_{i+1} is the same. Suppose we have a set S and three disjoint subsets S_1, S_2 and E such that $S_1 \cup S_2 \cup E = S$ and all examples in E have the same class. Then either the partition $\{S_1 \cup E, S_2\}$ or the partition $\{S_1, S_2 \cup E\}$ yields a better information gain than any partition of the form $\{S_1 \cup E_1, S_2 \cup E_2\}$ with $E_1 \cup E_2 = E$. In words: it can never be advantageous to spread examples of the same class over different subsets, compared to putting them in the same subset. This can be proven mathematically.

A consequence of this is that we only need to consider thresholds $c_i = (a_i + a_{i+1})/2$ where there is at least one example with value a_i or a_{i+1} that has a different class than the others.

Even then, it may seem that considering all these thresholds c is computationally expensive. The number of possible thresholds (and hence, tests) may increase linearly in the number of examples in T, and determining the information gain of each test is itself linear in the number of examples; so it seems we end up with something quadratic in $|T|$. Fortunately, this can be avoided. If we first sort the a_i, then the information gain of $A < c_{i+1}$ can be computed incrementally from that of $A < c_i$. Thus, determining the best test of the form $A < c$ is only slightly slower than linear: its complexity is lowerbounded by the complexity of sorting the values, which can be done with time complexity $O(|T| \log |T|)$.

The above approach for finding thresholds is the basic one that all tree learners can use. Some learners allow more complicated tests on numerical attributes in the nodes. They may allow to test, for instance, on linear combinations of multiple numerical attributes, such as $A_1 + A_2 > 1$. Trees that can use such linear combinations of multiple attributes in one test are called **oblique decision trees**. Oblique decision trees significantly extend the representation power of decision trees in numerical spaces, because they allow for decision surfaces that are not axis-parallel, as shown in Figure 4.5; but they are also a lot harder to learn.

4.5.3 Turning trees into rules

When interpreting the theory that a decision tree represents, humans find it useful to resort to a kind of if-then-reasoning: if an instance has this and this property, then it is likely to belong to class c. In a sense, we are then mentally translating the decision tree into a set of **if-then-rules**.

It is the case that any tree can be interpreted as a set of if-then-rules, and in fact it is easy to automatically translate a tree to such a set. The method works as follows: for each leaf l of a tree t, construct an if-then-rule of the form

if c_1 **and** c_2 **and** $\ldots$ **and** c_k **then** C

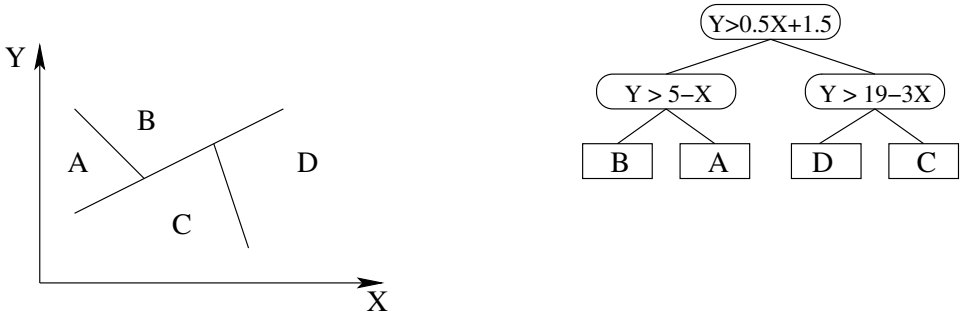

Figure 4.5: Decision surface corresponding to an oblique decision trees in a 2-D input space. Compare this with Figure 4.4: there, all lines are axis-parallel.

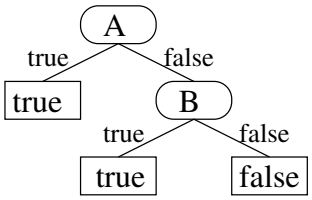

Figure 4.6: A tree representing A **or** B.

where the c_i are all the conditions that are true for any example in the leaf (based on the tests occurring in the nodes) and C is the class that the leaf predicts. The resulting set of rules is equivalent to the tree.

Example 4.8 Consider the tree in Figure 4.6, which represents the logical **or** function applied to two boolean attributes A and B. This tree gives rise to three rules:

if A=true **then** true
if A=false **and** B=true **then** true
if A=false **and** B=false **then** false

This transformation from trees to rules is very simple, and one may wonder why there is any interest in machine learning methods that learn rules directly. But there are quite important differences between the rule sets that we typically get from a tree, and the rule sets that are learned using direct methods.

One point worth noting is that the rules obtained from the tree using our simple translation method are unnecessarily complex. In the example above, we see that the second rule could be simplified by dropping the A=false condition. The resulting set of rules would be equivalent.

The reason why the tree leads to more complex rules is that all the leaves of the tree are mutually exclusive: if an example belongs to one leaf, it cannot belong to another leaf. This follows directly from the fact that decision trees *partition* the data. In a set of if-then-rules, however, there is no problem with multiple rules making a prediction for the same example, as long as the predictions are not contradictory.

Seeing that rules derived from trees are unnecessarily complex, it makes sense to try to further simplify such a set of rules. This leads to a second pruning phase: first the decision tree was pruned to avoid overfitting, now the rule set derived from the tree will be pruned to obtain a simpler and more understandable set of rules. Such pruning can be done as follows: if in a rule

if c_1 **and** c_2 **and** ... **and** c_k **then** C

a condition c_i can be dropped without this affecting the accuracy of the rule, then drop c_i.

As a result of this pruning process, however, it is possible that an instance is covered by multiple rules, and, in the case that the original rules were imperfect (i.e., the tree was not 100% pure), this may mean that some rules make contradictory predictions. To solve this, rules obtained through this process are usually sorted according to their accuracy, putting the most accurate rules in front and the least accurate ones at the end; and the convention is followed that to make a prediction for an instance, the rules are considered in order, and we take the prediction made by the first rule that applies for an instance. That way, if we may have contradictory rules, we are taking the prediction of the most accurate rule and ignoring all other predictions.

In a rule set, we can make clear that the order of the rules matters by writing them as **if-then-else rules**. Once rules have been converted into the if-then-else format, rules can sometimes be simplified even further. The following if-then-else ruleset is equivalent to the previous if-then-ruleset:

if A=true **then** true
else if B=true **then** true
else false

Thus, if-then-else-rules can be pruned more aggressively than if-then-rules, which can themselves be pruned more aggressively than trees.

4.5.4 Handling missing values

In practical applications, it may occur that some attribute values are missing (unknown). This may happen for examples in the training set T, as well as for new examples for which we need to make a prediction. Let us look at how decision tree learning can handle such situations.

Missing values in T

Consider first the case of the training set T containing examples where the outcome of a valid test τ cannot be computed because of a missing value. When computing the quality of that test, we can simply ignore those examples where the outcome of the test is unknown. The estimate of, for instance, the information gain of the test is then slightly less accurate, because it has been estimated using a slightly smaller data set (namely that subset of T for which the value is known). But as long as only a small percentage of the values are missing for any attribute, this is typically not a problem.

Missing values during prediction

Now consider the prediction phase. Assume we are sorting a new instance down the tree to determine its class, and suddenly we hit a test for which we cannot determine the outcome for this instance, because of a missing value. As a result, we cannot determine the subtree into which the instance should go.

There are multiple ways in which this can be handled. One option would be to simply consider the current internal node as a leaf, and predict the majority class in this root for the new instance. This is a possible solution, but it is likely to lead to less good predictive performance.

Another way of handling it is the following: find out for each subtree what the tree would predict for the instance if it belonged to that subtree. If all the subtrees

make the same prediction, then this means that the value of the tested attribute did not really matter for the prediction, at least not for this particular example.

Of course, some of the leaves the example ends up in may contradict each other. In that case it seems useful to compute the "most likely" prediction by letting the different branches "vote" on the class. In this process, we could give each branch an equal weight, but it makes sense to give more weight to a branch if it is more likely that the example would have gone into that branch. More specifically, the weight of a subset will be equal to the proportion of examples in the training set (more specifically, in that part of the training set that is covered by the current node) that was sorted into the subset.

Example 4.9 Assume that for a particular test there are three possible outcomes, a, b, and c; and that if we look at the training set, we see that 60% of the instances had outcome a there, while 20% had outcome b and 20% outcome c. Seeing that the value a occurs more often, it seems reasonable to assume that, among the instances that have missing values, 60% will have value a. It then makes sense to weight the votes of the different branches with the same proportions. Thus, if the a subtree predicts positive while the b and c subtrees predict negative, it seems best to predict positive; this will probably be the correct prediction in 60% of the cases.

4.6 Different kinds of decision trees

Up till now we mostly focused on one particular type of decision trees: so-called **classification trees**, trees that predict to which class (from a finite set of classes) an instances belongs.

Decision trees can also be used for other tasks. In general, they can be used to make any kind of decision or prediction about some case, not only predicting its class. We will first have a look at different kinds of decision trees that have been considered in machine learning; next, we will see how all these different kinds of trees can be learned with variations of one single generic TDIDT algorithm.

Figure 4.7 contains a general description of the task of learning a decision tree. From a training set T of labelled examples (x_i, y_i), we try to induce a decision tree t of which we hope that it approximates some target function f. The quality of the tree is measured by some cost function c: a smaller cost $c(t)$ means that t is of better quality.

The different types of decision trees that we will consider differ mostly with respect to the type of the labels Y.

4.6.1 Classification trees

With classification trees, Y is a set of nominal values (the classes). The task of learning such trees can be described informally as follows. Given a set of examples T, where each example is an instance together with the class it belongs to, we want to learn a decision tree t that predicts for any possible instance the class it belongs to.

Figure 4.8 shows how the general task description of Figure 4.7 is filled in for classification trees. The cost function c allows us to express whatever preferences we have about t (our "preference bias"). There are many ways in which c can be defined. We could define it as the error of t on the training set T (the "training error"), but that is usually not a good idea because we are interested in the error of t on the whole instance space X, rather than on T, and as explained earlier these two can be very different. We could define it as the error on a separate validation set V that does not overlap with T — that usually gives a better estimate of the

Task description: Learning a decision tree

Given:

- an instance space X (the set of all possible objects that we will consider)
- a set of labels Y
- the existence of an unknown target function $f : X \to Y$
- a set of examples $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\} \subseteq X \times Y$ (each example is an instance together with its label); we call T the training set
- a set of possible trees $\mathcal{H}$; each tree contains in each internal node a *test*, a function that maps any instance $\mathbf{x} \in X$ onto one of the children of the tree; and in each leaf, a function mapping any $\mathbf{x} \in X$ onto some $y \in Y$
- a cost function $c : \mathcal{H} \to \mathbb{R}$

Find:

- a decision tree $t \in \mathcal{H}$ that approximately minimizes $c(t)$

Figure 4.7: General task description: learning a decision tree

error on X. We could also define it as some combination of the error of t on the training set and syntactic properties of t such as its complexity.

Example 4.10 One possible cost function is

$$c(t) = |\{\mathbf{x}_i \in T : t(\mathbf{x}_i) \neq y_i\}|/|T| + \alpha|t|,$$

the proportion of training examples predicted incorrectly plus a penalty linear in the size of the tree. If $\alpha = 0$, we typically get large trees that fit T perfectly (which probably means they overfit the data). By increasing α, we give smaller trees an increasing preference, even if they make some errors on the training data.

4.6.2 Regression trees

The setting of regression trees is the same, but now the value to be predicted is not a class (from a finite set of classes) but a real number. A structured description is shown in Figure 4.9. Again c may refer to error on the training set, on some separate validation set, or on the whole population; and it may reflect a preference bias with respect to the format, complexity etc. of the trees. One possible cost function, similar to the previous example, is

$$c(t) = \sum_{\mathbf{x}_i \in T} (t(\mathbf{x}_i) - y_i)^2/|T| + \alpha|t|.$$

4.6.3 Model trees

Model trees predict a real number, just like regression trees. They differ with respect to the format of the tree. When we learn regression trees, $\mathcal{H}$ contains trees that make a constant prediction in their leaves (that is, each instance arriving in that leaf gets

Task description: Learning a classification tree.

In the general task description of Figure 4.7:

- Y is a finite set C of nominal values, called classes:

$$Y = C = \{c_1, c_2, \ldots, c_n\}$$

- $\mathcal{H}$ is a set of tree structures where each leaf of the tree contains a constant function mapping any x onto one specific class c_i
- the cost function c may vary, but $c(t)$ typically depends on $Pr(f(x) = t(x))$ (the probability that the predicted class $t(x)$ is equal to the true class f), and on the size of the tree t.

Figure 4.8: Task description: learning a classification tree

Task description: Learning a regression tree

As in Figure 4.7, with

- $Y = \mathbb{R}$: the labels are numerical values
- the functions in the leaves of a tree t map any $\mathbf{x} \in X$ onto one specific value $y \in Y$
- $c(t)$ typically depends on $\mathbf{E}((t(\mathbf{x}) - f(\mathbf{x}))^2)$ (the expected value of the squared difference between a prediction $t(\mathbf{x})$ and the actual label $f(\mathbf{x})$), and on the size of the tree t

Figure 4.9: Task description: learning a regression tree.

Task description: Learning a model tree

As in Figure 4.9 (regression trees), with this difference:

- the functions in the leaves of a tree t can be any function from some given class of functions, for instance, any function mapping $\mathbf{x} \in X$ onto a linear combination of its attributes

Figure 4.10: Task description: learning a model tree.

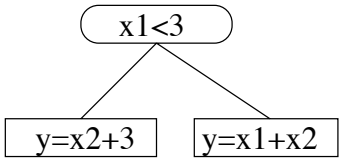

Figure 4.11: Example of a model tree.

the same prediction). When learning model trees, $\mathcal{H}$ contains trees that may have any kind of function in their leaves (for instance, a linear function predicting the target variable from selected attributes). This is formalized in Figure 4.10. Model trees are typically used when the instance descriptions $\mathbf{x}$ contain many numerical attributes.

Example 4.11 Figure 4.11 shows an example of a very simple model tree. It has one internal node and two leaves. In the leaves, we see not a single number, but a linear equation that predicts the target y from a number of other variables.

A prediction for a new instance is made by first finding the leaf that the instance belongs to, then using the function in that leaf to make a prediction.

4.6.4 Multiple prediction trees

Both classification and regression trees can be extended towards trees that predict not one variable, but several. For regression trees, the task definition is changed by associating with each example a vector of n numbers, instead of a single number, see Figure 4.12. The "error" will now be expressed as the norm of a vector. We call such trees **multiple regression trees**.

Similarly, classification trees can be extended by associating with each example a tuple of n nominal values. Not all the components of the tuple need to have the same domain: the set of possible classes for each component may differ. We might even have a mix of nominal and numerical values, in this case we talk about **multiple prediction trees**. Figure 4.13 presents the task description for multiple prediction trees.

With multiple prediction trees, there is a complication with respect to defining c. The error measure (how far off is a prediction from the correct value) is more difficult to define, because errors along different dimensions need to be combined. We need to define a *dissimilarity measure* d. We will return to this when discussing methods for learning decision trees.

Task description: learning a multiple regression tree.

As in Figure 4.7, with

- $Y = \mathbb{R}^n$: the labels are n-dimensional vectors
- the functions in the leaves of a tree t map any $\mathbf{x} \in X$ onto one specific value $y \in Y$
- $c(t)$ typically depends on $\mathbf{E}(||t(\mathbf{x}) - f(\mathbf{x})||^2)$ (the expected value of the squared norm of the difference between a prediction $t(\mathbf{x})$ and the actual label $f(\mathbf{x})$), and on the size of the tree t

Figure 4.12: Task description: learning a multiple regression tree.

Task description: learning a multiple prediction tree.

As in Figure 4.7, with

- $Y = Y_1 \times Y_2 \times \ldots \times Y_n$: the labels are n-dimensional tuples, each component may be from a different domain Y_i
- the functions in the leaves of a tree t map any $\mathbf{x} \in X$ onto one specific value $y \in Y$
- $c(t)$ typically depends on the expected value of the dissimilarity between the predicted and actual values, $\mathbf{E}(d(t(\mathbf{x}), f(\mathbf{x})))$, and on the size of the tree t

Figure 4.13: Task description: learning a multiple prediction tree.

Task description: learning a multilabel classification tree.

As in Figure 4.7, with:

- $Y = 2^C$ with $C = \{c_1, \ldots, c_n\}$ (with 2^C the power set of C)
- the functions in the leaves of a tree t map any $\mathbf{x} \in X$ onto one specific value $y \in Y$ (i.e., one specific set of classes)
- $c(t)$ typically depends on the expected value of the difference between the predicted and actual sets, $\mathbf{E}(|t(\mathbf{x})\Delta f(\mathbf{x})|)$ (with Δ the symmetric set difference operator: $A\Delta B = (A - B) \cup (B - A)$), and on the size of the tree t.

Figure 4.14: Task description: learning a multilabel classification tree.

Multilabel classification trees

A special kind of multiple prediction tree is the so-called **multilabel classification tree**. In multilabel classification, there is one given set of classes C, but each example can be associated with a subset of C, instead of only one element. An example application of this is learning to assign a text to a newsgroup: a text may be relevant for different newsgroups. Figure 4.14 presents the task description, and Figure 4.15 shows an example of a multilabel classification tree.

Multilabel classification trees are a special case of multiple prediction trees because when C is finite, subsets of C can be represented as a vector with as many components as there are classes, where the i'th component is 1 or 0 depending on whether the example belongs to the i'th class or not.

(Dis)advantages of multiple prediction trees

One might wonder what the use of multiple prediction trees is. Clearly, if we are to predict multiple variables, we can also learn a separate tree (or other model) for each of them. The advantages of having one multiple prediction tree instead of several single prediction trees are:

- Trees have a certain explanatory power. Learning a multiple prediction tree has important advantages in this respect:
  - Obviously, a single tree is *easier to interpret* than a set of trees.
  - Decision trees identify the variables that have the most important influence on the target variable. By learning separate trees for each target variable, we identify for each target variable the input variables that are most important for that one variable. By learning a single tree for multiple outputs, we identify the variables that have the *most important overall influence*.
  - *Dependencies between the target variables* can be explicitated by multiple prediction trees. For instance, in the multilabel classification case, classes that often co-occur will tend to co-occur in the leaves of the tree as well.
- A multiple prediction tree may even have *higher accuracy* than a set of single prediction trees. There are several reasons for this. One is that it overfits less easily. This is directly related to the complexity of the learned model: a single decision tree is less complex than a set of many decision trees, and

A	B	C	Class
0	0	0	{a,c}
0	0	1	{a,c}
0	1	0	{b}
0	1	1	{b}
1	0	0	{c}
1	0	1	{c,e}
1	1	0	{c}
1	1	1	{b,e}

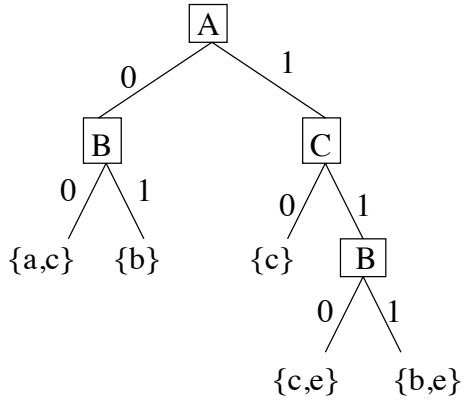

Figure 4.15: Example of a multilabeled dataset with three attributes and a target attribute that has sets of classes as values, and a multilabel classification tree consistent with the dataset.

Given:

- an instance space X
- a set of examples $T = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subseteq X$ (each example is an unlabeled instance)
- a set of possible trees $\mathcal{H}$
- a cost function $c : \mathcal{H} \to \mathbb{R}$

Find:

- the decision tree $t \in \mathcal{H}$ that minimizes $c(t)$

Figure 4.16: Task description: learning a clustering tree.

less complex models have less opportunity for overfitting. Another reason is that when target variables are not independent from each other, they carry information about each other, and looking at values of all these variables instead of just one amounts to using more information while building the model.

- Learning one multiple prediction tree may be a bit slower than learning one classification or regression tree, but it is much faster than learning many of them.
- Making a prediction for all the targets at the same time is just as fast as making a prediction for a single target, and much faster than making a prediction for each target separately.

The above advantages hold under the assumption that the different target variables are dependent. If that is not the case, the multiple prediction tree may be much larger than the set of single prediction trees, and its predictive accuracy much lower.

Exercise 4.3 Assume we have boolean attributes A, B, C, D and two boolean target attributes X and Y, where X is a function of A and B and Y is a function of C and D; take for instance the **xor** function in both cases. Write down single prediction trees for X and Y, and a multiple prediction tree for (X, Y). What is your conclusion? In general, how do you think the size of the multiple prediction tree depends on the sizes of the single prediction trees, when target attributes are independent?

4.6.5 Clustering trees

Clustering trees are a kind of trees where the nodes and leaves in the tree represent clusters of examples. Such trees do not reflect any predictive function; they just identify structure in the dataset, more specifically, they explicitate a hierarchical clustering. The task description is shown in Figure 4.16.

The cost function c now cannot refer to any error in the predictions, because clustering is unsupervised. Instead, it reflects some quality criterion regarding the clustering that has been formed. This criterion typically measures the extent to which examples in the same cluster are similar and the extent to which examples in different clusters are dissimilar.

Example 4.12 An example of a cost function for clustering is

$$c(t) = \sum_k \sum_{\mathbf{x}_i, \mathbf{x}_j \in L_k} d(\mathbf{x}_i, \mathbf{x}_j) + \alpha|t|.$$

where L_k is the k'th leaf of the tree and d is a dissimilarity measure. That is, for each leaf L_k we sum the dissimilarities among all instances of this leaf, and we add up all these sums: this gives us a measure of how coherent the leaves are. Again a term $\alpha|t|$ is added to prevent obtaining a trivial clustering: if $\alpha = 0$, the cost is minimized by a tree that contains one element per leaf.

4.7 A generic algorithm for decision tree induction

We now describe a generic TDIDT algorithm for learning decision trees. This generic algorithm has a number of *parameter procedures*, which are not yet filled in. They can be instantiated in different ways, and that will allow us to learn the different kinds of decision trees that we have seen. Many existing decision tree learning algorithms are instantiations of this algorithm, but not all; the algorithm does not cover, for instance, incremental decision tree learners.[1]

Our generic non-incremental TDIDT-algorithm is shown in Figure 4.17. We use the convention that procedures that are parameters of the algorithm are written in italic capitals (e.g. *OPTIMAL_SPLIT*), while real procedure names are written in small capitals (e.g. LEARN_TREE). Our LEARN_TREE algorithm consists of two phases, a growing phase and a pruning phase.

The tree growing algorithm works as follows. Given a set of examples E, it constructs a set of tests $\mathcal{T}$ using some procedure (*GENERATE_TESTS*). This set usually depends on the form of the example descriptions, but may also depend on specific values occurring in the data (as we saw, for instance, when we discussed how to handle numerical attributes). The ID3 algorithm we saw before has one specific way of constructing tests from attributes, but variations of this exist.

Example 4.13 Suppose example descriptions contain an attribute `Color` that takes only the values `red`, `blue`, and `green` in the observed data. In ID3, `Color` itself would be a test, with three possible outcomes: `red`, `blue`, and `green`. But $\mathcal{T}$ might also contain tests such as `Color=red`, `Color=blue`, or `Color` $\in$ {`red,green`}. These are boolean tests (they return true or false), yielding binary splits.

Generally, a test $\tau \in \mathcal{T}$ is a function from the instance space to a finite result space $\{r_1, \ldots, r_k\}$. It maps each possible instance onto one r_j, and thus, it defines a partition $\mathcal{E}$ on E:

$$\mathcal{E} = \{E_j | E_j = \{\mathbf{x} \in E | \tau(\mathbf{x}) = r_j\}\}. \tag{4.1}$$

In other words, E_j consists of all the elements of E for which the test yielded result r_j.

Example 4.14 The test `Color=blue` partitions the set in two subsets E_{true} and E_{false}. The test `Color` partitions E in three subsets E_{blue}, E_{red} and E_{green}.

The algorithm calls the function *OPTIMAL_SPLIT* to find the test $\tau \in \mathcal{T}$ that partitions E in some optimal way. The meaning of "optimal" is defined inside *OPTIMAL_SPLIT*.

[1] An incremental learner can learn a model from a given dataset, just like other learners, but in addition, when new data become available, it can adapt the model to incorporate the new data without starting all over. Non-incremental learners do not have this property.

```
function LEARN_TREE(E: set of examples) returns decision tree:
    t' := GROW_TREE(E)
    t := PRUNE_TREE(t')
    return t

function GROW_TREE(E: set of examples) returns decision tree:
    T := GENERATE_TESTS(E)
    τ := OPTIMAL_SPLIT(T, E)
    E := partition induced on E by τ
    if STOP_CRIT(E, E)
    then return leaf(INFO(E))
    else
        for all E_j in E:
            if E_j = ∅ then t_j := leaf(INFO(E))
            else t_j := GROW_TREE(E_j)
        return inode(τ, {(j, t_j)})
```

Figure 4.17: A generic decision tree induction algorithm.

The algorithm next calls a function *STOP_CRIT* to check whether the optimal partition $\mathcal{E}$ that was found is sufficiently good to justify the creation of a subtree in the current node. If it is not, a leaf is constructed containing some relevant information about E (this relevant information is computed by the function *INFO*). If $\mathcal{E}$ is sufficiently good, then GROW_TREE is called recursively on all the $E_j \in \mathcal{E}$, and the returned trees t_j become subtrees of the current node.

In most systems the GROW_TREE procedure grows an overly large tree that may overfit the training data. Therefore, after the tree-growing phase these systems have a post-pruning phase in which branches are pruned from the tree, in the hope to obtain a better tree. The pruning algorithm *PRUNE* performs this.

The functions *OPTIMAL_SPLIT, STOP_CRIT, INFO* and *PRUNE* are parameters of LEARN_TREE that will be instantiated according to the specific task that is at hand (e.g., classification requires different functions than regression). The function *GENERATE_TESTS* is also a parameter, but this function does not differ depending on the prediction task, and hence we will not discuss it here.

Most (classification or regression) tree induction systems that exist today are instantiations of this generic algorithm. The most popular ones are C4.5, CART, and J48, a reimplementation of C4.5 included in the Weka data mining tool [44]. These systems vary mostly with respect to how they fill in the parameter procedures just described. We discuss the various options in the following sections. Table 4.1 gives a compact overview of the instantiations that have been proposed for applying decision tree learning to different types of learning tasks.

4.7.1 Splitting Heuristics: the *OPTIMAL_SPLIT* function

Given a set of tests $\mathcal{T}$, the function *OPTIMAL_SPLIT* computes for each $\tau \in \mathcal{T}$ the partition induced by τ on the set of examples E. It evaluates these partitions and chooses the test τ that is optimal with respect to the task that is to be performed.

Classification

For classification, many quality criteria have been proposed. We mention a few of them; for each one it is the case that a split is considered optimal if it maximizes the criterion.

	OPTIMAL_SPLIT	STOP_CRIT	INFO	PRUNE
classification	gain (ratio) Gini index	χ^2-test min. coverage MDL	mode	C4.5 valid. set
regression	intra-cluster variance of target variable	F-test, t-test min. coverage MDL	mean	valid. set
clustering	intra-cluster variance	F-test, t-test min. coverage MDL	prototype identity	valid. set

Table 4.1: Overview of the different tasks that can be performed with TDIDT by instantiating its procedure parameters.

- We have already seen *information gain*, which is defined as follows:

$$IG(E) = s_C(E) - \sum_{E_i \in \mathcal{E}} \frac{|E_i|}{|E|} s_C(E_i) \tag{4.2}$$

  where E refers to the set of examples currently looked at and $s_C(E)$ is the entropy of a set of examples.

- the *Gini heuristic* [5]: this is similar to information gain, but instead of class entropy, the Gini index for impurity is used:

$$g(E) = \sum_{i=1}^{k} p(c_i, E)(1 - p(c_i, E)) = 1 - \sum_{i=1}^{k} p(c_i, E)^2 \tag{4.3}$$

  The quality of a split is computed as

$$Q = g(E) - \sum_{E_i \in \mathcal{E}} \frac{|E_i|}{|E|} g(E_i) \tag{4.4}$$

- We also saw the "purity" heuristic, where the idea was to choose the test that leads to a maximal increase of purity. This is equivalent to defining, instead of entropy s or Gini index g, a new impurity criterion e that is equal to the error that we would make when always predicting the majority class:

$$e(E) = 1 - \max(p(c_i, E))$$

  and defining the quality of a split as

$$K = e(E) - \sum_{E_i \in \mathcal{E}} \frac{|E_i|}{|E|} e(E_i) \tag{4.5}$$

  which gives this heuristic the same form as the Gini heuristic Q and the information gain IG. We remarked earlier that K does not work well as a heuristic, while IG does.

Since all the heuristics are in a sense related to some form of impurity, we will refer to the "purity" heuristic as the accuracy heuristic (since this particular form of purity indeed corresponds to accuracy).

Why do heuristic such as Q and IG work well, while K doesn't? Breiman et al. [5] explain in detail why this is. Roughly, it amounts to the fact that K is very often zero, and this may happen for very good tests as well as for very bad tests. K

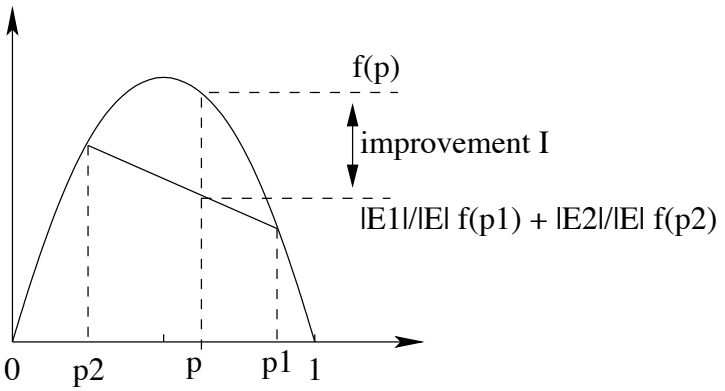

Figure 4.18: A concave impurity function f and the "gain" that a split results in. When a node is split in two child nodes, the quality of the split is measured as the difference between the impurity of the parent node $f(p)$ and the average impurity of the child nodes $|E_1|/|E|f(p_1) + |E_2|/|E|f(p_2)$. The latter is found on the graph as the intersection of the vertical line above p and a straight line connecting the $p1$ and $p2$ points.

simply does not discriminate very well, as we already saw in Section 4.3.3, "attempt 1".

The mathematical explanation, for the case where there are only two classes, goes as follows. The "improvement" that a split causes, is measured by comparing the impurity of the unsplit node with the average impurity of the children after splitting. If the impurity of a node is quantified by $f(p)$ where p is the proportion of positives in the node, and f some function, then the unsplit node has impurity $f(p)$ and the child nodes have an average impurity $\sum_{E_i \in \mathcal{E}} |E_i|/|E|f(p_i)$, and the improvement is

$$I = f(p) - \sum_{E_i \in \mathcal{E}} \frac{|E_i|}{|E|} f(p_i).$$

Note that IG, Q and K all have this form.

Heuristics that work in this way work only well if the function f is strictly concave.[2] Let us look at the case of a split into just two children E_1 and E_2 with proportions of positives p_1 and p_2. The relationship holds that $p = |E_1|/|E|p_1 + |E_2|/|E|p_2$. For a strictly concave function, this implies

$$f(p) > |E_1|/|E|f(p_1) + |E_2|/|E|f(p_2)$$

and for a concave function

$$f(p) \geq |E_1|/|E|f(p_1) + |E_2|/|E|f(p_2).$$

Figure 4.18 illustrates the situation graphically.

The Gini and information gain heuristics use strictly concave functions in p, the proportion of positives. The accuracy heuristic uses a piece-wise linear function, which is concave but not strictly concave. Figure 4.19 shows the curves for entropy, gini, and accuracy. When using accuracy, it holds that $f(p) = |E_1|/|E|f(p_1) + |E_2|/|E|f(p_2)$, unless the class distribution in one of the children changes so much that its majority class differs from that in the parent (i.e., $p_1 < 0.5$ and $p_2 > 0.5$ or

[2]Concave means that a straight line connecting two points on the curve of f always remains below or on the curve. Strictly concave means that such a straight line always remains strictly below the curve, except for its end points.

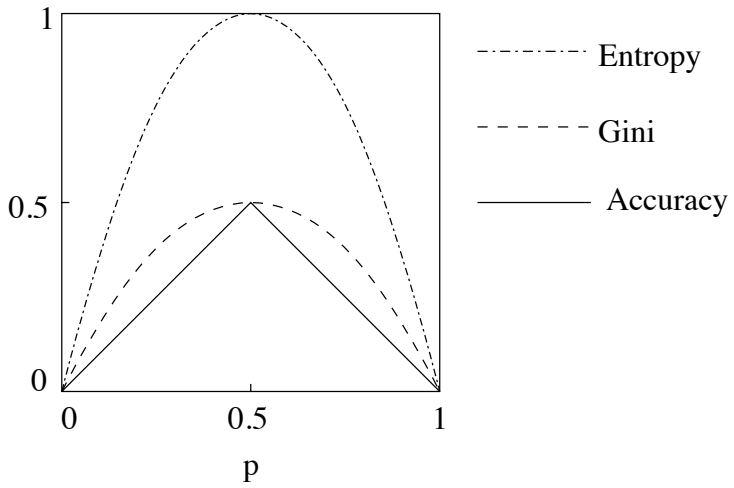

Figure 4.19: Impurity as a function f of the proportion of positives. The function f is plotted for three different heuristics: class entropy, gini index, and accuracy.

the other way around). Especially when starting from skewed distributions (where one class occurs much more often than the other), this is a rather strict condition that none of the possible splits may fulfill. That is the reason why accuracy is usually not used as a splitting heuristic.

Exercise 4.4 Draw a similar figure as in Figure 4.18 but using the curve for accuracy instead of a strictly concave f. Derive from this the conditions under which K will be greater than 0.

With this we end our discussion of heuristics for classification trees.

Regression

Regression systems typically use as quality criterion the intra-subset (or within-subset) variation[3] of the target variable (see e.g. CART [5]):

$$SS_W = \sum_{E_i \in \mathcal{E}} \sum_j (y_{ij} - \bar{y}_i)^2 \tag{4.6}$$

where y_{ij} denotes the j-th observation of the target variable in the set E_i and $\bar{y}_i = \sum_j y_{ij}/|E_i|$. This quality criterion should be minimized, not maximized.

From the statistical technique known as analysis of variance (ANOVA), it is known that if a set of values for a variable is partitioned into subsets, the total variation of the variable (measured as the sum of squares of differences between the values and the mean) can be decomposed into a within-subset and a between-subset variation, as follows:

$$SS_T = SS_B + SS_W \tag{4.7}$$

with

$$SS_T = \sum_{i,j} (y_{ij} - \bar{y})^2 \tag{4.8}$$

$$SS_B = \sum_i n_i (\bar{y}_i - \bar{y})^2 \tag{4.9}$$

[3]Variation is the sum of squared differences between elements and their mean; it is n times the variance, with n the size of the set.

where n_i is the number of elements belonging to subset i, $\bar{y} = \sum_{i,j} y_{ij}/|E|$, SS_T stands for total variation, SS_B is between-subset variation and SS_W is within-subset variation. Equation 4.7 shows that minimizing SS_W is equivalent to maximizing SS_B. This means that two different intuitively appealing criteria, minimizing distances within clusters and maximizing distances between clusters, are in fact equivalent.

Multiple prediction

Multiple prediction trees have not been discussed very often in the literature. For multiple regression, the concept of variance easily generalizes to the n-dimensional case: in the formulas for SS_T and SS_B, the y_{ij}, $\bar{y}_i$ and $\bar{y}$ are now n-dimensional vectors but nothing else needs to be changed.

For multiple classification, the Gini index generalizes in the same way. A closer look at the formula defining the Gini index reveals that in fact it expresses the variance of a variable that is 1 with probability p and 0 with probability $1 - p$. Viewing the Gini index as a variance, we can just generalize this variance in the same way as we did for multiple regression. The same holds for multilabel classification, which is a special case of multiple classification.

Finally, what about multiple prediction when we have a target tuple that is a mix of numbers and symbols? There is no fixed solution that always works best. The reason for this is that there is no universally best way to compare errors in one dimension to errors in another dimension, let alone to compare errors in a numerical dimension to misclassification errors in a symbolic dimension.

The general approach is to use a user-defined distance measure d. In other words, we leave it up to the user to say how errors in different dimensions compare. Assuming that the user can define a suitable d, then we can again easily define SS_T and SS_B:

$$SS_T = \sum_{i,j} d(y_{ij}, \bar{y})^2 \tag{4.10}$$

$$SS_B = \sum_{i} n_i d(\bar{y}_i, \bar{y})^2 \tag{4.11}$$

$$SS_W = \sum_{i} \sum_{j} d(y_{ij}, \bar{y}_i)^2 \tag{4.12}$$

If d is a Euclidean distance, it still holds that $SS_T = SS_B + SS_W$, but this does not hold for every distance. From the regression point of view, it is then better to minimize SS_W.

Clustering

In clustering, the goal is to form subsets (clusters) such that examples inside one cluster are close together and examples of different clusters are far apart. Again, we will assume that there is a user-defined distance that defines how similar two examples are. We now also define the concept of a **prototype** of a cluster. The prototype is an element that is "typical" for the cluster; it fulfills the role of the mean in the regression case.

A generalization towards clustering is then straightforward:

$$SS_T = \sum_{i,j} d(e_{ij}, p(E))^2 \tag{4.13}$$

$$SS_B = \sum_{i} n_i d(p(E_i), p(E))^2 \tag{4.14}$$

$$SS_W = \sum_i \sum_j d(e_{ij}, p(E_i))^2 \tag{4.15}$$

where e_{ij} denotes the j'th example in the set E_i.

Again, if d is a Euclidean distance, it holds that $SS_T = SS_B + SS_W$, but when this is not the case, the clustering system has to choose between maximizing SS_B and minimizing SS_W.

4.7.2 Stopping Criteria: the *STOP_CRIT* function

We saw before that most systems by default keep growing trees until they reach some maximal size, then start pruning them. Nevertheless, several stopping criteria for tree growing have been studied. The reason is twofold. First, stopping early can make the tree learning procedure more efficient. Second, even when the measures discussed here are not used strictly as stopping criteria, they can often be used for pruning.

Some very **simple stopping criteria** are:

- stop splitting when a subset is sufficiently coherent (i.e. its impurity is below an acceptable threshold)
- stop splitting when the number of examples covered by a node is below some threshold

A more complicated criterion is the **MDL principle** [37]. MDL stands for *Minimal Description Length*. The reasoning behind such a criterion is that the correct target values of a set of examples can be encoded in the form of a hypothesis, together with a list of corrections: an exhaustive enumeration of all the values that are predicted incorrectly by the hypothesis. When comparing two hypotheses with different predictive quality and different complexity, one should prefer the one with minimal description length; in other words: only make a hypothesis more complex if the gain in predictive quality is sufficiently large to justify it.

Hence, MDL is some sort of exchange rate that is applied when trading simplicity for accuracy. While the method has theoretical foundations, it is still relatively *ad hoc* in the sense that the best theory is not necessarily the most compact one. Moreover, applying it outside the classification context is quite complicated.

Another family of stopping criteria is based on **significance tests**. In the classification context a χ^2-test is often used to check whether the class distributions in the subtrees differ significantly (the same test is used by some rule-based systems, e.g. CN2 [9]). A χ^2-test was also incorporated in Quinlan's ID3 algorithm [33], the predecessor of C4.5. It was not incorporated in C4.5 because, as Quinlan notes [34], the test is relatively untrustworthy and better results are usually obtained by not using any significance test but pruning the tree afterwards. A similar argument was given for CART [5].

Since regression, multiple prediction and clustering use variance (or variation) as a heuristic for choosing the best split, the F-test is a reasonable heuristic to use as stopping criterion. If a set of examples is split into two subsets, the variance should decrease significantly, i.e.,

$$F = \frac{SS_T/(n-1)}{SS_W/(n-k)} \tag{4.16}$$

should be significantly large (where SS_T and SS_W are defined by Equations 4.13 and 4.15, k is the number of subsets in the partition and n is the number of examples in the whole set). When $k = 2$ the F-test is equivalent to a t-test on the prototypes, which could be used instead.

4.7.3 Information in Leaves: the *INFO* function

The *INFO* function computes the information that is to be stored in a leaf. Typically, one stores only that information that will be necessary for prediction:

- for classification trees: store the mode of the class values observed in the leaf
- for regression trees: store the mean of the values of the target variable that are observed in the leaf
- for model trees: build a (linear or other) regression model for the examples inside the leaf and store that model
- for clustering trees: simply store the set of instances in the leaf.

4.7.4 Post-pruning

As an alternative to using a stopping criterion, one can grow an oversized tree and afterwards prune away those branches that do not seem useful. This procedure is computationally more expensive, but tends to yield better trees [5, 34].

Typically, what happens is that, starting with the original tree, more and more branches are consecutively cut away, yielding a series of ever smaller subtrees of the original tree. The subtree that optimizes some evaluation criterion is selected as the final outcome of the tree learner.

Several methods for post-pruning are in use. C4.5 uses as evaluation criterion an estimate of the tree accuracy on unseen data. This estimate is based on the errors in the training set as well as the size of the leaves of the tree: in a small leaf the training error within that leaf is assumed to deviate more from the actual error of that leaf than in a large leaf.[4] This method is called error-based pruning (EBP). It lacks a firm statistical ground, but works well in practice. It only works for classification trees.

A statistically sound way to estimate the accuracy on unseen data is to simply remove a random sample from the training set, and use it as unseen data to evaluate the tree afterwards. The set that is removed in this way is called a **validation set**, hence we call this pruning method **validation set based pruning**. In the context of classification it is also known as reduced-error pruning (REP). Advantages of this method are that it can be used for any kind of trees, as long as there is some notion of the quality of a tree and a way to compute it, and that it is a statistically sound technique to estimate the quality of a tree on unseen data. A disadvantage of the simple validation set based pruning method is that the training sample becomes smaller, which may affect the quality of the tree.

The CART system avoids this disadvantage as follows. It evaluates trees using a cost function similar to the one in Example 4.10: a combination of training error and tree size, where a parameter α reflects the marginal cost of having an extra node in the tree (i.e., how much a tree's training accuracy should improve per extra node in the tree). The optimal value for α is estimated using a cross-validation: several auxiliary trees are grown, each with a different validation set, and α is chosen so that the average quality of each auxiliary tree on its own validation set is maximized. This α is then used to prune a tree that has been grown on the whole data set. Thus, the final tree does not suffer from the removal of examples to create a separate validation set.

[4] Using the data in the leaf, a confidence interval for the predictive accuracy of the leaf is constructed; the accuracy of the leaf is estimated as the lower bound of this interval, following the reasoning that the training set accuracy in the leaf is probably too optimistic. Smaller leaves yield larger intervals, hence more pessimistic estimates.

4.7.5 Summary

In this section we have given an overview of different kinds of decision trees (classification, regression, multiple prediction, multilabel classification, clustering trees), pointing out the large similarities that exist between them. All of them can be considered special cases of multiple prediction trees. The generic LEARN_TREE algorithm shown in Figure 4.17 further emphasizes these similarities.

In the literature there are many examples of classification and regression trees, but most of the other types of trees discussed here have been barely studied. Yet there are examples where, for instance, multilabel classification trees have been shown to produce interesting results.

4.8 Exercises

Exercise 4.5 Assume that a dataset is given with boolean attributes A,B,C,D,E and a target attribute that is a logical function of them. Show a decision tree that expresses:
a. $A \wedge \neg B$
b. $A \vee (B \wedge C)$
c. $(A \vee B) \wedge (C \vee D \vee \neg E)$
d. $A \wedge ((B \wedge C) \vee D)$

Exercise 4.6 Assume that in a decision tree learner, with numerical inputs, you would allow tests of the form $\sum_i a_i x_i < c$ (instead of just $x_i < c$ for some selected variable x_i), where the a_i and c are parameters to be determined automatically. How would this change the decision surfaces that are created?

Exercise 4.7 Consider the following table of training examples:

Instance	Classification	a_1	a_2
1	+	T	T
2	+	T	T
3	-	T	F
4	+	F	F
5	-	F	T
6	-	F	T

a. What is the entropy with respect to the "classification" attribute ?

b. What is the information gain of a_2 relative to these training examples?

Chapter 5

Learning Sets of Rules

If-then rules are probably the most appealing format for representing knowledge in artificial intelligence systems. An if-then rule is of the form "**if** (conditions) **then** (conclusion)" and represents a piece of knowledge that can be used to make a single step in a possibly complicated reasoning process.

Part of the reason for the appeal of if-then-rules is that such rules are very intuitive and easy to interpret. Domain experts often find it relatively easy to write down their knowledge as a set of if-then-rules (where relatively easy means: compared to other possible formats for expressing knowledge). Conversely, when a learning system discovers a pattern, representing the pattern as an if-then-rule (or a set of such rules) may make it easier to understand and interpret the pattern. As mentioned before, for many applications of learning, interpretability of the results of the learning process is important.

Given the general importance of rule sets in artificial intelligence, it is not surprising that much research has gone into how such rule sets can be learned from data. In this chapter we provide an overview of the different kinds of rule sets that can be learned, issues when learning rule sets, and algorithms for learning them.

Rule learning is often set in the context of concept learning, where a rule set is learned that defines a concept. It is easy to extend this setting to the general classification setting, and we will treat both contexts under the name of *classification rules*. Rule sets for regression have also been proposed, but they are less popular in this context, and we will not focus on them. Finally, a very important kind of rules in the context of data mining are *association rules*. While syntactically similar, association rules differ from classification rules in many ways, and need to be treated differently.

5.1 Learning classification rules

5.1.1 If-then-rules and disjunctive normal form

In concept learning, the task is to find a definition for a concept by observing example instances of it. In the rule learning context, this definition will be expressed as a rule set.

A rule set is a set of rules r_i, $i = 1, \ldots, n$, where each rule r_i is of the form

$$\textbf{if } c_{i1} \textbf{ and } c_{i2} \textbf{ and } \ldots \textbf{and } c_{ik_i} \textbf{ then} \text{ conclusion.}$$

The different conditions c_{ij} within one rule are always connected using *and*. For this reason, these rules are also called *conjunctive rules*.

In the context of concept learning, a condition c_{ij} is typically a condition on a single training example, and the conclusion tells us whether the instance belongs to the target concept or not.

Often, when learning a definition for a concept, only "positive" rules are learned, rules where the conclusion is "yes, the instance belongs to the concept". In addition, it is assumed that the set of rules is complete, that is, there are no other instances of the concept than those described by the rules. Under these conditions, a rule set is equivalent to the so-called **disjunctive normal form (DNF)** definition of the concept.

Before explaining this, we first define some terminology. In propositional logic, a **proposition** is an atomic statement that may be true or false. With atomic we mean that it is not composed of other more basic statements. A **well-formed formula** in propositional logic is defined as follows: any proposition p is a well-formed formula; and if a and b are well-formed formulas, then $a \wedge b$ (the **conjunction** of a and b, pronounced "a and b"), $a \vee b$ (the **disjunction** of a and b, pronounced "a or b") and $\neg a$ (the **negation** of a, pronounced "not a") are also well-formed formulas. A **literal** is an occurrence of a proposition p or its negation $\neg p$ in a well-formed formula.

Any well-formed formula is assigned a truth value *true* or *false* in a way consistent with our intuition: $a \vee b$ is true if and only if a is true or b is true; $a \wedge b$ is true if and only if a and b are both true; $\neg a$ is true if and only if a is false.

Besides the basic logical operators $\vee$, $\wedge$ and $\neg$, we will sometimes also combine formulas using the **implication** operator $\rightarrow$ or the **equivalence** operator $\leftrightarrow$. $a \rightarrow b$ is defined as $\neg a \vee b$ and can be read as "a implies b" or "if a then b". $a \leftrightarrow b$ is defined as $(a \wedge b) \vee (\neg a \wedge \neg b)$ and can be read as "a is true if and only if b is true" or "a is equivalent to b". $a \rightarrow b$ can also be written $b \leftarrow a$. Formulas containing these arrows are also well-formed formulas.

We can now define the concept of disjunctive normal form.

Definition 5.1 (Disjunctive normal form, DNF) *A well-formed formula F is in disjunctive normal form (DNF) if and only if it is written as a disjunction of conjunctions, that is, if it is written in the form*

$$C_1 \vee C_2 \vee \cdots \vee C_n$$

where each C_i is of the form

$$c_{i1} \wedge c_{i2} \wedge \ldots \wedge c_{ik_i}$$

with each c_{ij} a literal.

To see the equivalence of a set of rules defining a single concept, and a DNF formula, note that each rule r_i is of the form

$$r_i = \textbf{if } c_{i1} \wedge c_{i2} \wedge \ldots \wedge c_{ik_i} \textbf{ then } \mathit{positive}$$

which we can rewrite, using

$$C_i = c_{i1} \wedge c_{i2} \wedge \ldots \wedge c_{ik_i},$$

as

$$r_i = \textbf{if } C_i \textbf{ then } \mathit{positive},$$

or, in logic notation,

$$r_i = C_i \rightarrow \mathit{positive},$$

C_i is a *sufficient condition* for the concept: if an instance fulfills C_i it is certainly positive. If it does not, however, we cannot be sure that it is negative: there might be

another sufficient condition that it fulfills. Because each C_i is a sufficient condition, the concept as a whole is really defined as the disjunction of all the C_i, that is, the rule set is equivalent to

$$\text{positive} \leftrightarrow C_1 \vee C_2 \vee \ldots \vee C_n$$

or, writing out the C_i as the conjunctions that they are,

$$\begin{aligned} \textit{positive} \quad \leftrightarrow \quad & (c_{11} \wedge c_{12} \wedge \cdots \wedge c_{1k_1}) \vee \\ & (c_{21} \wedge c_{22} \wedge \cdots \wedge c_{2k_2}) \vee \\ & \vdots \\ & (c_{n1} \wedge c_{n2} \wedge \cdots \wedge c_{nk_n}) \end{aligned}$$

If-then rules for non-boolean data

Propositions are boolean variables: they can only be true or false. If we have a data set where all the attributes are boolean, then each attribute can be used as a proposition. Many datasets are not boolean, however. The propositions are then conditions on the value of an attribute.

For a nominal attribute A with possible values $a_1, a_2, \ldots, a_n$, propositions are typically of the form $A = a_i$. (Sometimes they can be of the form $A \in S$ with S a set of a_i values, but we will not consider this variant here.) For an ordinal or numerical attribute A, propositions are typically of the form $A \leq a_i$ with a_i a value in the range of A.

5.1.2 The learning task

The learning task that we consider here can now be described formally as follows:

Task description: learning a set of rules (concept learning version)

Given: a set of positive examples E^+, a set of negative examples E^-, and a set of propositions $\mathcal{P}$ such that each example assigns a truth value to each proposition,
Find: a set of rules $\{r_1, \ldots, r_n\}$ where each r_i is of the form

if c_{i1} **and** c_{i2} **and ... and** c_{ik_i} **then** positive

with c_{ij} is of the form p or $\neg p$ with $p \in \mathcal{P}$, such that:

- for each $e \in E^+$ it holds that there exists a rule r_i such that $c_{i1} \wedge c_{i2} \wedge \ldots \wedge c_{ik_i}$ is true for e, and
- for each $e \in E^-$, it holds that there does not exist a rule r_i such that $c_{i1} \wedge c_{i2} \wedge \ldots \wedge c_{ik_i}$ is true for e

---

The definition above is typically given in the setting of concept learning; it is then assumed that E^+ contains the examples of some concept, and E^- the counterexamples; the rule set then forms a definition of the concept.

In the attribute-value learning setting, the distinction between positive and negative examples is typically made, not by including them in different sets, but by having one attribute (the class attribute) that indicates whether some example is positive or negative. More generally, the class attribute might have more than two values. One can then learn for each possible class c a set of rules that predicts

positive for each example belonging to class c, and negative for each example not belonging to it. Thus, the rule sets together form a function predicting the class of each instance. We call this the classification version of the rule learning task.

Task description: learning a set of rules (classification version)

Given:

- a data set T containing elements $\mathbf{x}$ with attributes $A_1, \ldots, A_{n-1}$ and a nominal target attribute A_n (called the class attribute)
- a class value c that is in the domain of A_n, and a set of propositions $\mathcal{P}$ where each proposition expresses some conditions on an individual attribute

Find: a set of rules $\{r_1, \ldots, r_n\}$ where each r_i is of the form

$$\textbf{if } c_{i1} \textbf{ and } c_{i2} \textbf{ and } \ldots \textbf{and } c_{ik_i} \textbf{ then } \text{positive}$$

with $c_{ij} \in \mathcal{P}$, such that:

- for each $\mathbf{x}$ where $A_n(\mathbf{x}) = c$, there exists a rule r_i such that $c_{i1} \wedge c_{i2} \wedge \ldots \wedge c_{ik_i}$ is true for $\mathbf{x}$, and
- for each $\mathbf{x}$ where $A_n(\mathbf{x}) \neq c$, it holds that there does not exist a rule r_i such that $c_{i1} \wedge c_{i2} \wedge \ldots \wedge c_{ik_i}$ is true for $\mathbf{x}$

Essentially the same learning algorithms can be used for the two settings (although there are some subtle differences that make the second task a bit more difficult than the first; these are related to the fact that we are not defining a single class, but multiple classes). In the following we discuss what is probably the most popular approach to learning rule sets: the so-called covering approach.

5.1.3 The covering approach

The covering approach is also known as the *separate-and-conquer* strategy, as opposed to decision tree learning, which is then called *divide-and-conquer*. These names reflect a certain symmetry that is present (divide-and-conquer) or absent (separate-and-conquer) in both approaches.

During decision tree learning, the dataset is repeatedly split (divided) into several subsets, all of which are afterwards treated in exactly the same way. For instance, in a binary decision tree, the heuristic indicating the quality of a test depends on the left branch in exactly the same way as on the right branch, and the way in which the subtrees are next developed is exactly the same for both branches.

This contrasts with rule set learning, where a rule set is typically learned one rule at a time. We say that a rule **covers** an example if the condition part of the rule is true for that example (i.e., the rule applies to the example). When learning a single rule, the learner is focusing on a (often small) subset of examples, namely those examples covered by the rule. It tries to make the rule as accurate as possible for the examples it covers. Once the rule cannot be improved any further, it is added to the rule set. The examples handled correctly by the rule are then removed from the dataset (or at least marked as "already taken care of"). Next, the learner will focus on the remaining examples, trying to find good rules for those.

We first discuss the basic algorithm underlying the covering approach. In subsequent sections we will further refine it and mention variants.

```
function LearnRules(T: data set; c: class):
    let T+ be the elements of T belonging to class c
    let T− be the elements of T not belonging to class c
    R := ∅
    P := T+        // P is the set of not-yet-covered positives
    while P is not empty:
        r := LearnOneRule(P, T−, c)
        add r to R
        remove all examples covered by r from P
    return R
```

Figure 5.1: Algorithm for learning a set of rules.

On a high level, the covering algorithm works as follows: a single rule is learned; the positive examples covered by the rule are removed from the dataset; the process is repeated on the remaining examples; this continues until no examples remain or until no good rules can be found anymore.

Pseudocode for this algorithm is given in Figure 5.1. The algorithm learns rules one at a time using an auxiliary procedure LearnOneRule, which we will discuss in a moment. It keeps a set P of examples that are yet to be covered by some rule. In the beginning, P is the set of all positive examples. After having learned a rule, the examples covered by the rule are removed from P. The algorithm continues until no more examples are in P. (In practice, the loop can usually end prematurely if the last attempt to learn a rule did not give an acceptable result; this detail is not shown in the code.)

Let us now look at how a single rule is learned (the LearnOneRule procedure). This can actually be done in a top-down or bottom-up way (or yet other ways); we here discuss the top-down approach.

In the top-down approach, the learner starts with an if-then-rule without any conditions, i.e., with an empty if-part, and with one particular class, e.g., "positive", in the head part:

if *true* **then** positive

Note that we denote the empty if-part with *true*. Indeed, the conditions part is trivially fulfilled if there simply are no conditions.

This rule predicts that all training examples belong to the positive class. For most applications this will obviously be a bad rule. We can try to improve it by including a condition in the if-part, preferably a condition that is strongly correlated with the positive class. That is, from a set of possible conditions $C = \{c_1, \ldots, c_L\}$, we can choose that condition c_i that corresponds most closely to the positive class. This can be measured in different ways. If T^{c_i} is the set of training examples fulfilling c_i and T^+ is the set of positive training examples, then the ideal c_i is such that $T^{c_i} = T^+$. We can measure how well c_i approximates this ideal by computing the symmetric difference of T^{c_i} and T^+: [1]

$$T^{c_i} \Delta T^+ = (T^{c_i} - T^+) \cup (T^+ - T^{c_i})$$

The rule predicts the class perfectly if this symmetric difference is empty. Elements in $T^{c_i} - T^+$ are negative instances that are predicted positive by the rule (so-called "false positives"); elements in $T^+ - T^{c_i}$ are positive instances that are not predicted positive by the rule ("false negatives").

[1] The symmetric difference of two sets A and B, denoted $A \Delta B$, is defined as the set of elements that are in one of these two sets but not in the other: $A \Delta B = (A - B) \cup (B - A)$.

function LearnOneRule(P: positives; N: negatives; c: class) **returns** rule:
    $C :=$ true
    for any set S, let S_C denote the elements of S for which C is true
    let $acc(C) = |P_C|/|(P \cup N)_C|$ denote the accuracy of the rule "**if** C **then** c"
    let τ be the set of all possible literals in a condition C
    stop := false
    while $N_C \neq \emptyset$ and not *stop*:
        // determine the best refinement C^* and its accuracy a^*
        $C^* := \arg\max_{t \in \tau} acc(C \wedge t)$
        $a^* := \text{acc}(C^*)$
        // check if the best refinement is better than the current rule;
        // if yes, make it the current rule and continue
        if $a^* > a$ **then**
            $C := C^*$
            $a := a^*$
        else *stop* := true
    return rule "**if** C **then** c"

Figure 5.2: Top-down algorithm for learning a single rule for a class c from positive and negative examples of this class.

Usually we will not be able to find a single condition c_i such that $T^{c_i} = T^+$. We simply go for the c_i that seems best, and add it to the condition part C of the rule. Next, we repeat the procedure: we look for a new c_i that in combination with the current C approximates T^+ as well as possible. We keep adding conditions until the condition part C is such that $T^C = T^+$.

How should we select the consecutive c_i that are to be added to C? Should we, at each time, select the c_i that makes $T^C \Delta T^+$ as small as possible? While this may seem a sensible criterion at first sight, it turns out it does not work very well.

$T^C \Delta T^+$ consists of false positives and false negatives. While both are undesirable, false positives are worse than false negatives. The reason for this is simple: we are currently only considering a single rule, but other rules may be added to the rule set later on. An example not covered by this rule may yet be covered by another rule, and thus be predicted correctly by the rule set as a whole. Thus, *a false negative may be taken care of later on by another rule.* However, if the current rule predicts an example to be positive, then later rules cannot make this undone; at best, there may be other rules that make a different prediction, thus causing a contradiction.

For this reason, it makes sense to consider false positives worse than false negatives, and our main efforts when learning a single rule should be invested in avoiding false positives. What is typically done, then, is that conditions are added to the rule until all false positives are removed, i.e., the rule does not cover any instances other than the positive class. A secondary goal is to have the rule cover as many true positives as possible.

Figure 5.2 shows an algorithm for learning a single rule top-down. Top-down in this case means: from general to specific, or: starting from a rule covering all examples, moving towards rules that cover fewer and fewer examples, until a rule is found that covers no negatives.

Note that the algorithm consecutively adds to the rule that condition that increases the rule's accuracy the most. Generally, when we add a condition t to C, the accuracy of C goes up if t *proportionally* excludes more negatives than posi-

tives. That is: if a rule covers for instance 20% negatives and 80% positives, then any condition t that excludes examples in such a way that, among the excluded examples, the proportion of negatives is higher than 20%, will result in an increase in accuracy.

The algorithm stops when the accuracy can no longer be increased, which happens when there are no more negatives covered (the accuracy is then 100%), or when no condition t can be found that, when added to the current conjunctive condition C, causes the accuracy of the rule to increase.

5.1.4 Evaluating the quality of a single rule

The rule learning procedure uses a heuristic to evaluate the refinements of a rule and to decide which one to continue the search with. In the basic algorithm, we used the predictive accuracy of the rule as a heuristic: a rule is considered more promising if a larger percentage of its predictions is correct.

In practice, this heuristic is not satisfactory. Consider two candidate rules r_1 and r_2, where r_1 covers 99 positive examples and one negative, while r_2 covers 2 positive examples and no negatives. r_1 has an accuracy of 0.99, r_2 has an accuracy of 1. Is r_2 then the better rule? That may not be the case, for several reasons.

A first point is that, while we want rules to be highly accurate, we also prefer to have simple rule sets with not too many rules. For this reason, we may sometimes accept a slightly lower accuracy if the coverage of the rule becomes much larger because of this.

Second, when there is noise in the data, it is actually quite possible that the one negative example covered by r_1 is simply noisy, that it is really positive but incorrectly listed as negative in the database.

Third, even in the absence of noise, it is debatable whether r_2 is really a rule with an accuracy of 1. For any random rule that covers only two examples in the dataset, there is a relatively high probabilty that the two examples it covers have the same class, and hence, that it has an estimated accuracy of 1. This does not guarantee that its real accuracy, when tested on a separate test set, will be 1.

Finally, and perhaps most importantly: when using a heuristic to determine the best refinement of the current rule, we need a heuristic that does not necessarily tell us how good the rule itself is, but rather, one that tells us how good further refinements of the rule might become. A rule that covers 80 positives and 20 negatives might have an accuracy of only 0.8, but leaves much room for improvement: perhaps, after adding more conditions to the rule, we will end up with a rule covering 50 positives and no negatives. That rule would be much better than a rule covering, say, 5 positives and no negatives.

We already saw for decision trees that accuracy is not a good heuristic, that the accuracy of a candidate tree may not tell us much of how much better the tree might become when we keep refining. Clearly, a similar phenomenon occurs for rules. We need something else than accuracy to evaluate how promising a rule is.

The m-estimate

In particular, we want to have a heuristic that combines the current accuracy of the rule with its coverage. When a rule covers many examples, this has two advantages: (1) its estimated accuracy is likely to be closer to its real accuracy, and therefore more trustworthy; and (2) there is a larger potential for the refinements of the rule to contain a rule with high accuracy that still covers many instances. For this reason, when evaluating the potential of a (partial) rule to lead to a good rule, we will prefer rules that have a high accuracy but also a high coverage.

One particular way in which coverage and accuracy can be combined is the so-called **m-estimate**, which is defined as

$$a = \frac{P + mp}{P + N + m}$$

where a is the estimated accuracy of the rule on unseen cases, P and N are the number of positive and negative examples covered by the rule, and m and p are parameters.

To understand what this rule does, let us consider a number of special cases of the parameters. For $m = 0$, the formula becomes:

$$a = \frac{P}{P + N},$$

that is, a is the proportion of positives among the examples covered by the rule. In other words, a is simply the training set accuracy.

For $m > 0$, the formula can be seen as an interpolation between $P/(P+N)$ and p, where more weight is given to p if m becomes larger. (As m goes to infinity, a approaches p.) The formula behaves exactly as if we had already tested the rule on m examples, and found an accuracy of p, and we are now adding new evidence where the rule has an accuracy of $P/(P+N)$ on $P+N$ examples. The formula gives us the weighted average of the original accuracy p and the new accuracy $P/(P+N)$, where the original accuracy has a relative weight m and the new accuracy has a relative weight $P + N$. (Since the weights must add up to 1, the absolute weights become $m/(P + N + m)$ and $(P + N)/(P + N + m)$.)

Mathematically, we have:

$$a = \frac{P + mp}{P + N + m} = \alpha \frac{P}{(P + N)} + (1 - \alpha)p$$

with

$$\alpha = \frac{P + N}{P + N + m}$$

a weighing factor that determines the importance of the observed accuracy $P/(P+N)$ versus that of the prior estimate p.

It is important to see that the m-estimate is simply an estimate of the accuracy of a rule. It does not give an explicit bonus to rules covering many cases. Nevertheless, if the prior estimate p is low (possibly 0), it will have the effect that among two rules with similar training set accuracies, the one with the highest coverage will be preferred, which is exactly what we want.

5.1.5 Top-down versus bottom-up learning

The algorithm for learning a single rule that we just saw, is a top-down algorithm: it starts with a most general rule (predicting everything as positive), then continuously refines that rule by adding more and more conditions (thus making it more specific), until it covers no negatives anymore. We could think of an algorithm that works in the opposite direction: starting with a maximally specific rule, which contains a maximal set of conditions, it could refine the rule by removing conditions from the body of the rule, until a rule is obtained that cannot be generalized further without covering negative examples.

The top-down procedure has a clear starting point: there is only one maximally general rule, namely the rule "**if** true **then** positive". But is there also a single maximally specific rule?

Just like the maximally general rule covers all examples, the maximally specific rule covers no examples. Any rule that contains contradictory conditions will cover no examples. Hence, there are multiple maximally specific rules, all of which are in fact *syntactic variants* of each other (different ways of writing the same rule).

Example 5.1 Suppose we have two boolean input attributes A and B. The conditions that we can have in the rule are then A, $\neg A$, B, $\neg B$. Any rule that contains as conditions both A and $\neg A$ cannot cover any examples, as these conditions can never be satisfied at the same time. A similar argument holds of course for B and $\neg B$. The following rules are therefore equivalent:

if A and $\neg A$ **then** positive
if B and $\neg B$ **then** positive
if A and $\neg A$ and B **then** positive
if A and $\neg A$ and $\neg B$ **then** positive
if B and $\neg B$ and A **then** positive
if B and $\neg B$ and $\neg A$ **then** positive
if A and $\neg A$ and B and $\neg B$ **then** positive

Clearly, there are very many ways in which we could start the search with a most specific rule. If we make abstraction of the semantics, and use a purely syntactic notion of specificness, we could say that a rule R_1 is more specific than another rule R_2 if the conditions in R_2 are a strict subset of those in R_1. From that point of view, the rule

if A and $\neg A$ and B and $\neg B$ **then** positive

is the most specific one. Each imaginable rule will have as conditions a subset of the conditions in this rule, and can therefore be obtained from this rule by removing conditions. Note that the other rules do not have this property.

We could start the bottom-up search with this maximal rule, but this is not very efficient. By removing one condition at a time, it will take some time before all contradictory conditions are removed, i.e., before the rule starts covering even a single example. If we have D boolean attributes, we have $2D$ possible conditions, among which at least D must be removed before the rule can start covering any examples. Even then, some contradictions may remain. This is illustrated in Figure 5.3.

To see how much smaller the search space becomes if we eliminate the obvious contradictions, note the following. With D boolean attributes, the bottom rule contains $2D$ conditions. Any subset of this set of $2D$ conditions corresponds to a possible rule. There are $2^{2D} = 4^D$ such subsets. In practice we are only interested in rules that do not contain for any attribute A both A and $\neg A$. Thus, for each attribute A, there are 3 options: A is included as a condition, $\neg A$ is included, or the attribute does not occur in the rule. With D attributes, this gives us 3^D possible rules. Finally, a maximally specific rule is a rule that has as many conditions as possible without containing contradictions. That implies that for each attribute A, either A or $\neg A$ is included. There are 2^D such rules. (Thus, in Figure 5.3, the whole lattice contains $4^3 = 64$ elements, from which only $3^3 = 27$ are non-contradictory; the set of non-contradictory rules has $2^3 = 8$ maximally specific elements.)

Clearly, starting from the maximal rule, it is inefficient to search through such a large space of rules, only to arrive at the first "useful" rules after a very long search. It is better to immediately start from one of the 2^D rules that do not necessarily contain a contradiction.

Even among these 2^D rules, it is still possible that many of them cover no examples. Note that, for boolean attributes, each of these rules imposes one particular value (true or false) on each attribute, and therefore, the rule can cover only one example (unless multiple examples with exactly the same values for all the attributes

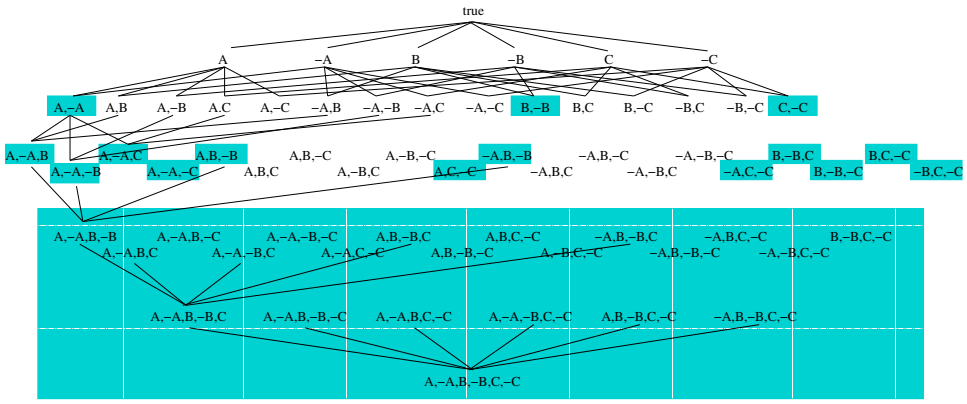

Figure 5.3: A lattice in which conjunctive conditions over three boolean variables A, B and C are ordered according to generality. (For readability reasons not all connections in the lattice are shown.) Some conjunctive conditions contain logical contradictions (e.g., $A \wedge \neg A$); they are indicated with a shaded background in the figure. Each conjunction that contains a logical contradiction is equivalent to the condition *false*. The bottom half of the lattice contains only such conjunctions.

occur). The 2^D rules are equivalent to the 2^D instances that the instance space contains. Since the dataset we learn from typically contains only a relatively small sample from the instance space, the large majority of the 2^D rules, while not containing any logical contradictions, will still cover not a single example from the training set.

Given this equivalence between rules with D non-contradictory conditions and individual examples, there is an obvious solution to the problem of possibly generating many rules that cover nothing: *we should start with a rule that is equivalent to an example that occurs in the training set.* That rule will then automatically cover that one example. Further refinements (generalizations) of that rule will then cover that example and possibly other examples as well.

In case the attributes are not boolean, conditions of the form $A = a$ or $A \leq a$ (with a a value for the attribute A) will typically be used. If only equality conditions are used, the one-to-one correspondence between maximally specific rules and instances will still hold. When inequality conditions are used, there will still be as many bottom rules as there are examples, but one rule may cover multiple examples.

We can now formulate a bottom-up algorithm for rule learning. Informally, it works as follows: choose a positive example at random, construct the most specific rule that covers this example, gradually generalize the rule by removing conditions from it (in such a way that it covers more positive examples but still no negative examples), keep generalizing until it is impossible to further generalize the rule without making it cover negative examples. At that point, store the rule in the rule set; mark the examples it covers as "covered"; and repeat the process, taking care however that the random positive example that is now chosen is a yet uncovered one (choosing an already covered example may lead to a rule that covers only examples that we had already covered, which is not useful).

This gives us the algorithm for bottom-up rule learning shown in Figure 5.4. Just like its top-down counterpart, it follows a covering approach. This algorithm stops when it is not possible anymore to drop conditions without covering negative examples. Again, this criterion may be softened to allow for noise in the data.

function LearnRulesBottomUp(T: data set; c: class):
    let T^+ be the elements of T belonging to class c
    let T^- be the elements of T not belonging to class c
    $R := \emptyset$
    $P := T^+$
// P is the set of not-yet-covered positives
    while P is not empty:
        x := random element from T
        r := LearnOneRuleBottomUp(**x**, P, T^-, c)
        add r to R
        remove all examples covered by r from P
    return R

function LearnOneRuleBottomUp(**x**: seed example, P: positives; N: negatives; c: class):
    let $\mathbf{x} = (a_1, a_2, \ldots, a_D)$
    $C :=$ "$A_1 = a_1 \wedge A_2 = a_2 \wedge \cdots \wedge A_D = a_D$"
    for any set S, let S_C denote the elements of S for which C is true
    stop := false
    while $N_C = \emptyset$ and not *stop*:
        // determine the generalization C^* that covers as many positives as possible
        // without covering any negatives
        $C^* := C$
        for all $t \in C$:
            $C' := C - \{t\}$
            if $N_{C'} = \emptyset$ **and** $|P_{C'}| > |P_{C^*}|$ **then**
                $C^* := C'$
        if $C^* = C$ **then** *stop* := true
        else $C := C^*$
    return rule "**if** C **then** c"

Figure 5.4: Algorithm for learning a set of rules bottom-up.

5.1.6 A top-down search using seeds: the AQ approach

The bottom-up approach always starts from a single example, called a seed. The rule we end up with is guaranteed to cover that one example. This trick can be used also in top-down rule learning, and in fact it can make it much more efficient.

Consider again the case of learning from data with boolean attributes. In top-down rule learning, for each attribute A we will consider both A and $\neg A$ as possible tests to be added to the rule. With D attributes, that gives us $2D$ choices for the first attribute, $2D-2$ for the second (if we have chosen A for the first attribute, then both A and $\neg A$ are not considered anymore for the second one), and generally $2D-2(i-1)$ for the i'th attribute.

Now suppose we have chosen a seed example $\mathbf{x}$, and we want to find a rule that certainly covers $\mathbf{x}$. If $\mathbf{x}$ has the value *true* for A, then we should not add the condition $\neg A$ to the rule, because we know for sure that a rule containing $\neg A$ will not cover $\mathbf{x}$. Similarly, if $\mathbf{x}$ has the value *false* for A, we need not consider adding A: any rule imposing condition A will not cover $\mathbf{x}$.

Thus, for each attribute, we need to add only one test (instead of two) for that attribute, namely the one that corresponds to $\mathbf{x}$. The number of tests considered for addition to the rule is then D for the first step, $D-1$ for the second step, etc. The hypothesis space that is searched in this way is much smaller, and the greedy search through it becomes 2 times faster.

When we have non-boolean attributes, of course, the factor 2 may be larger. For an attribute A with n values a_i, among all the possible tests $A = a_1$, $A = a_2$, ..., $A = a_n$, we need to consider only that test where A is compared to the value that occurs in the seed example.

While the above algorithms for rule learning are greedy, one may also consider exhaustive searches. Such exhaustive searches may be sped up much more than the greedy search. In the case of boolean attributes, the total number of rules in the search space is 3^D: each attribute A may occur as A, as $\neg A$, or may be absent from the conditions in the rule. By using a seed, the size of the search space is reduced to 2^D, since one of the options (either A or $\neg A$, for any attribute A) is not considered anymore.

The idea of using a single example as a seed and then performing a top-down search through a hypothesis space consisting of only hypotheses that cover the seed, was first implemented in a series of rule learning algorithms referred to as the AQ algorithms. Since then the idea has been reused in several other learning algorithms, including the ILP system PROGOL.

5.1.7 Learning rule sets for multiple classes

We have described rule learning in the context of learning rules for one class (called the positive class). This is the typical setting encountered in concept learning. However, in some cases, one may be interested in finding a definition for the negative class as well as for the positive. There may also be more than 2 classes, in which case we may want a definition for each class. Earlier on we called this the classification variant of rule learning.

The above algorithms can simply be used in such a setting as follows: for each class c, run the algorithm treating the examples of class c as positive examples and all the rest as negative examples. The algorithms we have seen are then directly applicable.

A second approach, however, is to learn rules for the multiple classes in parallel. In the approaches that use seeds, instead of choosing a random positive example as seed, we simply choose a random example as seed, whatever class c it has; the rule that we learn will then be added to the rule set for class c. In the top-down approach

that does not use a seed, we can simply let the class be the most frequent class among the examples covered by the rule. As we look for a rule with high accuracy (i.e., a rule covering mostly examples from the same class), the first condition added to the rule will be chosen such that the rule is as accurate as possible. Assuming the rule covers, for instance, 60% examples of class c_1, 30% of c_2 and 10% of c_3, it is likely that further refinements of the rule will tend towards covering mostly examples of class c_1; however, it is not impossible that after adding an extra condition, the majority class switches. This is not a problem: we are simply looking for a rule with high accuracy, and we choose at each step the rule that is most likely to lead to high accuracy in the end, no matter what class it predicts.

5.1.8 Learning ordered rule sets

The algorithms we have seen up till now learn an unordered rule set. The result is simply a set of rules, and it does not matter in which order we write those rules.

As an alternative, we could consider learning **ordered rule sets**. In an ordered rule set, rules will be evaluated in a particular order, and for any instance, only the first rule that applies to it will be considered relevant. Such rules could also be called *if-then-else* rules. Ordered rule sets are also called **decision lists**.

Decision lists can be learned with basically the same algorithm as unordered rule sets, with one change. When we remove the covered examples from a data set, we can remove the covered negative examples just as well as the covered positives. Normally, it is useful to leave in the negatives, because they remain relevant for the next rules we will learn: those rules should not cover these negatives. But in a decision list, since these examples are covered by the first rule, none of the later rules will be applied to them.

Why would one want to use such an adapted rule learner that learns ordered rule sets? There is an important advantage to this, which is: ordered rule sets can sometimes represent the same function in a more compact way. This is because they can contain rules that have exceptions to them. In an unordered rule set, such rules would give rise to incorrect predictions, but in an ordered rule set, they do not, as long as the exceptions are covered by an earlier rule.

Example 5.2 Consider the definition of a leap year. A year is a leap year if it is a multiple of 4, except when it is a multiple of 100 that is not a multiple of 400.

Using a decision list, we can write this as

if multiple of 400 **then** leap
else if multiple of 100 **then** no leap
else if multiple of 4 **then** leap
else no leap

Each rule formulates a kind of default rule, which is "usually correct" but has exceptions; the rules preceding it take care of those exceptions.

If we use an unordered rule set, a correct definition is

if multiple of 400 **then** leap
if multiple of 100 and not multiple of 400 **then** no leap
if multiple of 4 and not multiple of 100 **then** leap
if not multiple of 4 **then** no leap

Since the rules are not ordered, we can just as well write

if multiple of 400 **then** leap
if multiple of 4 and not multiple of 100 **then** leap

if multiple of 100 and not multiple of 400 **then** no leap
if not multiple of 4 **then** no leap

and thus obtain two separate definitions, one for leap years, and one for non-leap years.

The advantage of an unordered rule set is that each rule is correct independently of the other rules, and thus can be interpreted in isolation. In a decision list, a single rule may not be correct by itself, it may have exceptions, but we can only find out about any exceptions by looking at the rule set as a whole.

On the other hand, decision lists are interpretable for humans in the sense that humans, too, tend to use default reasoning. Humans use relatively simple, more-or-less correct rules, rather than complex entirely-correct rules. (E.g., most people know the rule that "multiples of four are leap years", while not all of them may know the exact rule.) From this point of view, one can argue that decision lists are indeed interpretable.

Note that learning decision lists is only useful in the classification setting, when rules for multiple classes are learned in parallel. When we learn a set of rules for one class, the order of the rules never matters, because each rule leads to the same prediction.

5.1.9 Classifying instances using rule sets

Assume we are learning unordered rule sets. The above algorithms try to return rules that are 100% correct on the training set, that is, a rule never makes an incorrect prediction (rules cover no negatives). As with trees, 100% correctness on the training set does not guarantee us that the correctness on data from outside the training set will also be 100%. We may be predicting class c_1 for an instance that actually belongs to class c_2.

A consequence of this is that multiple rules may in fact contradict each other. An instance $\mathbf{x}$ might be covered by a rule from the rule set for c_1, but also by a rule from the rule set for c_2. Which rule should we use, then?

In such cases, it makes sense to use the rule that we trust most. This would typically be the rule with the highest estimated accuracy, according to for instance the m-estimate. For this reason, the rules in unordered rule sets are sometimes sorted from high to low estimated accuracy. In such a "sorted unordered rule set", the first rule that covers an example is the one that will be used (since we know that later rules that might contradict this rule will be less trustworthy).

Alternatively, we could predict the class of an instance by applying all the rules that cover the instance, and considering their predictions as votes. The votes could be weighted according to the estimated correctness of the rules. This prediction method has the disadvantage that the interpretation of the rule set becomes again a bit more complex.

Note that with ordered rule sets, the issue of which rule to choose in case of contradictory predictions does not occur: in an ordered rule set it is always the case that only one rule applies.

5.1.10 Rule Pruning

In practice, rule learners often work in two phases: in a first phase, a set of rules is grown, and in the second phase the rules are pruned. This is similar to what decision tree learners do, and the motivation is the same: in the first phase, overly complex rules are often learned, with a high risk of overfitting. The pruning phase alleviates this problem.

Rule pruning can be done in the same way as it is done in the context of decision trees. For pruning rules, the *reduced error pruning* (REP) approach is often used. In this approach, a separate dataset is set aside during the rule learning process; this dataset, the so-called pruning set, will be used in the pruning phase but not in the rule growing phase. In the pruning phase, the ruleset is simplified by applying operators such as dropping a single condition in a rule, or dropping an entire rule. These operators are applied repeatedly to the ruleset, as long as the resulting smaller ruleset does not yield an increased error on the pruning set; when no further simplification is possible without increasing the error of the ruleset on the pruning set, the pruning phase stops.

5.2 Association rules

We now turn to another kind of rules, called association rules. Association rules are syntactically quite similar to classification rules: they are also in the "**if** ... **then** ..." format. However, they serve a different purpose: while classification rules can be considered to form a definition of a concept (the concept corresponding to the class that is being predicted), this is usually not the case for association rules. The latter simply express certain dependencies or associations between attributes, without necessarily aiming at accurate prediction of the value of a given attribute.

5.2.1 Terminology, definitions and problem statement

Association rules are typically introduced in the context of market basket analysis. A supermarket nowadays can easily keep track of the items that customers buy: this information is available at the cashier, who scans the products and computes the total amount due. One customer might for instance buy bread, milk, and peanut butter, while a second customer buys cheese, wine, and fruit. We call the set of items that a customer buys an **itemset**. The act of making a purchase is called a **transaction**.

We can represent transactions in the attribute-value format by introducing a boolean attribute for each item that can possibly be bought; for a given customer the attribute takes the value *true* if the customer buys that item and *false* if the customer doesn't. A transaction is then a single tuple in a table with these attributes. An itemset is a subset of the attributes. We say that a transaction contains an itemset if all the attributes in the itemset have the value *true* in the tuple corresponding to the transaction.

Example 5.3 Assume that we have the following items: bread, cheese, wine, milk, fruit. We might have the following data about what customers have bought in a supermarket:

t1	{ bread, cheese, fruit}
t2	{ bread, milk}
t3	{ bread, cheese, wine, fruit}
t4	{ milk, fruit}
t5	{ cheese, wine}

These data can also be represented in tabular format, by introducing a boolean attribute for each item, and having for each transaction a tuple where the attribute is true if the customer bought that item:

ID	Bread	Cheese	Wine	Milk	Fruit
t1	T	T	F	F	T
t2	T	F	F	T	F
t3	T	T	T	F	T
t4	F	F	F	T	T
t5	F	T	T	F	F

An example of an itemset is $\{bread, cheese\}$. This itemset is contained in transactions $t1$ and $t3$.

An association rule is of the form $X \rightarrow Y$ (also written as **if** X **then** Y), with X and Y itemsets. In the context of market basket analysis, the rule is usually interpreted as saying "people who buy X usually also buy Y". More generally, we can interpret it as "transactions that contain X usually also contain Y".

This may seem similar to the interpretation of classification rules, but note the following differences:

1. Y is a set of attributes, not a single attribute;
2. In classification rules the single attribute Y is predefined: we know what we need to predict. In association rules, on the other hand, Y may contain any attributes, and it is part of the learning task to find an interesting set of attributes Y;
3. In "transactions that contain X usually also contain Y", the adverb "usually" weakens the interpretation of the rule. For a classification rule we typically want the rule to predict the target Y with as high as possible accuracy, preferably 100%. In the context of association rules, this would correspond to finding association rules where transactions that contain X "almost always" contain Y. Our actual condition (which we will quantify in a moment) is much weaker.

With itemsets and association rules are associated the following properties:

Definition 5.2 *The* **frequency** *of an itemset X in a set of transactions T, denoted $freq(X,T)$, is the number of transactions that contain X.*

Definition 5.3 *The* **support** *of an association rule $X \rightarrow Y$ in a set of transactions T, denoted $sup(X \rightarrow Y,T)$, is the frequency of $X \cup Y$ in T, relative to the total number of transactions. That is, $sup(X \rightarrow Y,T) = freq(X \cup Y,T)/|T|$.*

Definition 5.4 *The* **confidence** *of an assocation rule $X \rightarrow Y$ in a set of transactions T, denoted $conf(X \rightarrow Y,T)$, is defined as $freq(X \cup Y,T)/freq(X,T)$.*

The support of an association rule is an indication of how important the rule is, in terms of to how many cases it is applicable. Its confidence is an indication of the certainty with which we can conclude that a transaction contains Y, given that we know it contains X.

Example 5.4 In the example given above, the frequency of the itemset {Bread, Cheese} is 2. The association rule $\{Bread \rightarrow Cheese\}$ has a support equal to $freq(\{Bread\} \cup \{Cheese\}) = freq(\{Bread, Cheese\}) = 2$ and a confidence of $freq(\{Bread, Cheese\})/freq(\{Bread\}) = 2/3$.

We can now formulate the problem that association rule learners try to solve.

Task description: assocation rule discovery.

Given:

- a set of transactions T, where each transaction is a tuple $\mathbf{x}$ with boolean components (attributes) $x_1, \ldots, x_D$;
- a parameter s called minimal support;
- a parameter c called minimal confidence;

Find: all association rules $X \rightarrow Y$, with X and Y subsets of the attributes, for which $sup(X \rightarrow Y, T) \geq s$ and $conf(X \rightarrow Y, T) \geq c$.

It is not obvious how this problem could be solved by means of the standard rule learning methods that we have seen before. Given a specific target attribute Y, a rule learner L would typically return a set of rules r_i where each r_i is of the form $X_i \rightarrow Y$, and such that each individual rule has a high accuracy (in terms of association rules, its confidence should be close to 100%) and all the rules together are complete, i.e., for each case where Y is true there is at least one rule that covers that case.

Even if we would change the learner so that it does not refine rules until their accuracy is close to 100% but can return rules with a lower accuracy, this would not be sufficient. Note that the classification learner L does not return *all* rules of the form $X \rightarrow Y$: it finds a minimal set of rules such that (almost) all the cases where Y is true are covered by at least one rule. If, say, two rules are sufficient to cover all the cases where Y is true, the learner will not look for more rules, even if many more rules with sufficiently high confidence and support exist.

We could change the learner such that, instead of using heuristics to efficiently find a minimal set of rules that predict Y as accurately as possible, it simply finds *all* rules $X \rightarrow Y$ with sufficiently high confidence. Note that this will be much more work, and hence the challenge is how to do this efficiently.

Even if we solve that problem, we're not there yet. Up till now we assumed that Y was given, but in the context of association rules Y may be any attribute, and even any subset of attributes. If we have D attributes, we have 2^D such subsets. We could run our adapted rule learner once for each of these subsets, but this is clearly going to be very inefficient.

So the question is: how can we solve the problem stated above in a reasonably efficient way?

Association rule discovery algorithms will do this by performing one integrated search for rules with any attributes in the body and any attributes in the head. The key to solving the problem efficiently is the following. From the definitions of support and confidence of association rules it is already clear that these properties are related to the frequency of the itemsets occurring in the rule. Efficient association rule discovery algorithms will therefore first, in a first phase, compute the frequency of many itemsets. In a second phase, they will use the results from the first phase to compute all the association rules with sufficiently high support and confidence.

The archetypical association rule discovery algorithm is called Apriori. While many new and improved versions have been proposed, some of which work in a quite different way, it is still instructive to see how Apriori works. We next describe the two phases of Apriori in detail.

5.2.2 Finding frequent itemsets

In its first phase, Apriori computes all the itemsets of which the frequency is above a given minimal frequency s. We simply call such sets **frequent itemsets**. We will see later that once all itemsets with frequency at least s are found, we can derive

from this the support and confidence of all association rules with support at least s.

To understand how APRIORI computes all the frequent itemsets, first note the following property of the frequency of itemsets.

Proposition 5.1 *Given an itemset S, for every $S' \subseteq S$ and for every database T it holds that $freq(S', T) \geq freq(S, T)$.*

Proof: Let T_S be the set of transactions that contain S, and $T_{S'}$ the set of transactions that contain S'. By definition, $freq(S, T) = |T_S|$ and $freq(S', T) = |T_{S'}|$. Now, if $S' \subseteq S$, any $\mathbf{x}$ that contains S automatically contains S', and therefore each element of T_S must also be in $T_{S'}$. That means $T_S \subseteq T_{S'}$, which implies $|T_S| \leq |T_{S'}|$ and hence $freq(S, T) \leq freq(S', T)$. □

This property is also called the anti-monotonicity property of the frequency function.

Definition 5.5 *A function f defined over sets S is* **monotonic** *if and only if $S \subseteq S' \Rightarrow f(S) \leq f(S')$. It is* **anti-monotonic** *if and only if $S \subseteq S' \Rightarrow f(S) \geq f(S')$. It is* **non-monotonic** *if it is neither monotonic nor antimonotonic.*

Roughly, when we say that the frequency of itemsets is antimonotonic, this means that when a set grows its frequency goes down (or stays the same), and when it shrinks its frequency goes up (or stays the same).

If I is the set of all itemsets, for any itemset S it holds that $\emptyset \subseteq S \subseteq I$. As the subset relation is reflexive, antisymmetric and transitive, it imposes a partial order on all itemsets, with a single smallest element $\emptyset$ and a single largest element I. This imposes a lattice structure on the itemset space.

APRIORI searches this lattice top-down for frequent itemsets, exploiting the anti-monotonicity of frequency to prune the itemset space. It proceeds in a level-wise manner: it first computes all frequent itemsets of size 1, then those of size 2, 3, etc., until no frequent itemsets remain. In principle this could be done by just enumerating all itemsets of size 1 and checking whether they are frequent, then enumerating all sets of size 2 and checking their frequency, etc. But this is very inefficient: if there are n itemsets, there are $\frac{n!}{(n-i)!i!}$ itemsets of size i. It is useful to prune this search space as much as possible. Significant pruning can be achieved by exploiting the anti-monotonicity property, as follows.

Let F_i be the set of all frequent itemsets of size i. The only itemset of size 0 is $\emptyset$, and the frequency of $\emptyset$ equals the total number of transactions N (since the empty set is a subset of each transaction). Hence, $F_0 = \{\emptyset\}$ if $N \geq s$, and $F_0 = \emptyset$ otherwise. All the sets F_i, $i \geq 1$, can then be computed from F_{i-1} as follows.

Since no itemset can be in F_i unless all its subsets of size $i-1$ are in F_{i-1}, it suffices to extend each itemset $S \in F_{i-1}$ in each possible way with one item x, and check whether $S \cup \{x\}$ is frequent. If it is, it is added to F_i, otherwise it is discarded.

This method improves the efficiency of the process because a set for which none of its subsets of size $i-1$ are frequent (and which therefore cannot be frequent) will never be checked. But we can do better: we only need to check the frequency of any set for which *all* its subsets of size $i-1$ are frequent. Indeed, as soon as one subset is not frequent, the set cannot be frequent itself. Thus, while the above procedure for generating candidate itemsets of size i is correct, it can be made more efficient in the following way: for each itemset S that is constructed by extending a set from F_{i-1} with one item, first check that *all* its subsets of size $i-1$ are frequent; as soon as one of them is not in F_{i-1}, S can be discarded without even checking its frequency. Since we are storing all the frequent sets anyway, this check can be performed quite efficiently, using appropriate data structures.

```
function FrequentItemsets(T, s) returns set of itemset frequencies
    if |T| ≥ s then F_0 := {∅}
    else F_0 := ∅
    i := 0
    while F_i ≠ ∅:
        for each S in F_i:
            f := freq(S,T)
            if f ≥ s then
                for each x ∉ S:
                    S' := S ∪ {x}
                    if for all y ∈ S': S' − {y} ∈ F_i
                    then add S' to Q_{i+1}
        for each S ∈ Q_{i+1}:
            f := freq(S, T)
            if f ≥ s
            then add (S, f) to F_{i+1}
        i := i + 1
    return ∪_i F_i
```

Figure 5.5: Apriori's frequent itemset finding algorithm.

```
function AssocRules(IS, s, c) returns set of association rules
    for each (X, f_X) ∈ IS such that f_X ≥ s:
        for each Y ⊂ X:
            find (Y, f_Y) in S
            conf := f_X / f_Y
            if conf > c then AR := AR ∪ {Y → X − Y}
    return AR
```

Figure 5.6: Deriving association rules from a set of frequent itemsets.

5.2.3 Combining itemsets into association rules

Recall from the problem statement that we are looking for association rules with a support of at least s and a confidence of at least c. The support of an association rule $X \rightarrow Y$ is the frequency of the itemset $X \cup Y$, and its confidence is that same frequency divided by $freq(X)$, which is at least as high as $freq(X \cup Y)$.

To generate all association rules with a support $\geq s$ and confidence $\geq c$, it is sufficient to look at all itemsets X with support $\geq s$ and, for each such itemset, generate all subsets $Y \subset X$ for which $freq(X)/freq(Y) \geq c$. The association rule $Y \rightarrow X - Y$ has a support $\geq s$ and a confidence $\geq c$. The algorithm is described in Figure 5.6.

5.2.4 Imposing more constraints on the output

A problem with the basic algorithms for generating frequent itemsets and association rules, as described above, is that they may produce so much output that interpreting this output becomes a problem in itself. Moreover, many of the rules that are output may in fact be very similar.

To understand the extent of the problem, consider the case where we have D items, each of which occurs randomly in 50% of the cases. Assume that we look

for all itemsets with a frequency of at least 1%. Any singleton itemset will occur in 1 out of 2 cases; any pair of itemsets will occur in approximately 1/4 cases; and generally any itemset of size x will occur in approximately $1/2^x$ data elements. That implies that, with a minimal frequency of 1%, most itemsets up to size 6 will be frequent (itemsets of size 6 occur on average in 1/64 of the data elements, which is 1.5625%, so most will still be frequent). That means that the number of frequent itemsets we find is $C_D^6 = \frac{D!}{6!(D-6)!}$, which can easily run into millions. (For $D = 10$, it is 10*9*8*7*6*5 / 720 = 210; for $D = 100$ it is 100*99*98*97*96*95 / 720 = 1,192,052,400: over a billion!)

There has been a lot of research on how to postprocess the resulting association rules. One option is to sort the rules according to some additional interestingness criterion, so that the most interesting rules are listed first in the output. Another option is to filter the produced rules afterwards: throw away those association rules that do not fulfill additional criteria.

From an efficiency point of view, filtering is of course not optimal: it is often much more efficient to search directly for rules that fulfill a certain criterion (besides confidence and support), rather than look for all rules and filter them afterwards according to the criterion. As a consequence, there has been much research on how additional conditions can be used inside the association rule discovery process. For conditions on itemsets that observe the antimonotonicity property, this is easy; but for other conditions it is not. For many kinds of conditions the question remains open whether, and how, one can efficiently mine for itemsets or rules fulfilling these conditions.

Chapter 6

Evaluation of hypotheses

The question we consider here is how to assess the quality of a model (also called hypothesis or pattern).

There is no single definition for the quality of a model. What constitutes a good model depends on the context. For instance, for predictive models, sometimes the *accuracy* of the predictions may be the only thing that matters. At other times, it may be important to be able to compute predictions *fast*, even if they are slightly less accurate; or it may be important that the model can *explain* its predictions.

In the context of knowledge discovery, there may not be any predictions at all, but the *interpretability* of the model or patterns that are found may be important. Interpretability is a subjective concept, but one can measure related things such as the *size* of a model.

The most studied and best understood dimension is the "quality of the predictions" of predictive models. We will start with listing a number of indicators for that. Other aspects of a model's quality will be discussed next. These first two sections define the measurements, but do not yet discuss how one can actually compute or estimate them. That is the subject of the third section. Finally, we will see how we can test and compare models and learning algorithms based on these estimates.

In this chapter we assume familiarity with probability theory, more specifically the concepts of probability, conditional probability, probability distributions, independence, expected value of a variable, mean and variance.

6.1 Quality of predictions

How we measure the quality of predictions first of all depends on what kind of predictions these are. Classifiers, where a prediction is typically either right or wrong, are evaluated in a different way than numerical predictors, where we are interested in how far off a prediction is, rather than just saying it's right or wrong.

We start with a discussion on how to evaluate classifiers. This problem has been studied in much detail and we will devote a large part of this text on it. Next comes a shorter part on evaluating numerical predictors. A third part discusses "complex" predictors, which are neither classifiers nor numerical predictors.

6.1.1 Evaluation of Classifiers

A lot has been written on how to evaluate classifiers. The most straightforward way to evaluate a classifier is just counting how often it makes an incorrect prediction: obviously, a good classifier should make few mistakes. The notions of *accuracy* and

error reflect this. These are by far the most common evaluation criteria. However, they have shortcomings that make them unsuitable for certain kinds of problems, and therefore alternative evaluation measures have been proposed. We will briefly discuss *correlation* as one possible alternative.

Next, we move on to so-called *cost-based evaluation*. This is a more general kind of evaluation measure that has many advantages over accuracy. In particular, it distinguishes different kinds of mistakes, allowing us to state that some kinds of mistakes are worse than other. (Think of organising a barbecue: if the weather forecast predicted rain and it turns out to be sunny, you have unnecessarily postponed your barbecue; but if sunny wheather was predicted and it's raining, your barbecue is ruined. You may find one kind of mistake worse than the other.)

There are a number of specific evaluation methods related to cost-based evaluation. The best known are "precision-recall" diagrams and "receiver operating characteristics" diagrams. These evaluation methods are gaining popularity, and therefore deserve a detailed discussion as well.

Accuracy

A classifier predicts for each instance a "class", a member from a predefined and usually finite set of classes. In other words, it maps the instance space X onto a discrete set of classes $C = \{c_1, \ldots, c_n\}$. Given a true target function f and a hypothesis h, we say that a prediction for an instance x is correct if $h(x) = f(x)$, otherwise it is incorrect.

The **accuracy** of a model is then defined as the probability that it makes a correct prediction. To be more precise, it is the probability that when we pick an instance $x \in X$ randomly according to some distribution D over X, the predicted class $h(x)$ equals the actual class $f(x)$. The distribution D is often left implicit when talking about accuracy, but strictly speaking the probability is not well-defined without specifying D.

Definition 6.1 (accuracy) *The accuracy of a hypothesis h with respect to a target function f and an instance distribution D is*

$$acc_{D,f}(h) = Pr_{x \sim D}(h(x) = f(x)).$$

The **error** of a model is defined as the probability that it makes an incorrect prediction.

Definition 6.2 (error) *The error of a hypothesis h with respect to a target function f and an instance distribution D is*

$$err_{D,f}(h) = Pr_{x \sim D}(h(x) \neq f(x)).$$

Often D and f are clear from the context, and we just write $acc(h)$ and $err(h)$. Clearly, the following relationship holds:

$$acc(h) = 1 - err(h).$$

Note that computing the true accuracy or error of a model on the whole instance space (or "population") requires knowledge of the population distribution. In practice, we usually do not know this distribution, but we can sample it. The accuracy or error can then be estimated by counting the number of correct and incorrect predictions in the random sample. This is discussed in detail later in this chapter.

Shortcomings of accuracy

While accuracy is an often used criterion for evaluating classifiers, it has a number of shortcomings.

First, it is difficult to say whether a classifier performs well or not, just by looking at its accuracy. An accuracy of 95% may seem good, and an accuracy of 60% may seem poor. But this depends on the class distribution! If you have a dataset where 95% of the instances are negative, a model that always predicts negative also achieves 95%. That does not mean it is a good classifier.

We could conclude from this that a good classifier should at least have a higher accuracy than a trivial model that always predicts the majority class. But even this is not necessarily true. In fact, a higher accuracy does *not* automatically mean a "better" model. The following example illustrates this.

Example 6.1 Assume that 5% of a population are positive and 95% negative. Now consider two classifiers. Classifier 1 just predicts everything to be negative. Classifier 2 predicts 15% of the instances to be positive, and all of the 5% instances that are really positive are among them. Then Classifier 1 clearly has an error of 5% (it misses all the positives), whereas classifier 2 has an error of 10% (it correctly identifies the 5% positives, but also incorrectly predicts 10% of the population as positive when they are are really negative).

So Classifier 1 has a higher accuracy than Classifier 2. Yet, for Classifier 1 we have no reason to believe that it has identified any properties that correlate with the positive class, whereas for Classifier 2 one could argue that it has indeed identified some relevant properties (since it correctly identifies all the positives). While Classifier 2 makes more mistakes (that cannot be denied), one could still argue that it has done a better job at identifying the positives than Classifier 1.

So it seems that accuracy does not always capture very well how useful a model is, or how informative its predictions are. The latter is sometimes expressed better by another quality criterion, the so-called *correlation coefficient*.

Correlation

Statisticians have come up with a so-called **coefficient of correlation**, which measures the correlation between predictions and actual classes. When there are only two classes, this coefficient is defined as follows:

$$\phi = \frac{Pr(Pp)Pr(Nn) - Pr(Pn)Pr(Np)}{\sqrt{Pr(p)Pr(n)Pr(P)Pr(N)}} \tag{6.1}$$

where $Pr(P)$ ($Pr(N)$) is the probability that a random instance is positive (negative); $Pr(p)$ ($Pr(n)$) is the probability that a random instance is predicted positive (negative); $Pr(Pn)$ is the probability that a random instance is positive but predicted negative, and similar for $Pr(Pp)$, $Pr(Np)$, $Pr(Nn)$. Figure 6.1 illustrates the meaning of the sets P, N, p, n, Pp, Pn, Np and Nn. The probabilities of an instance being in any of these sets are also often written down in a so-called **contingency table**. Table 6.1 shows such a table.

ϕ is always between -1 and 1. It is zero if the predicted classes do not correlate at all with the actual classes; that is, knowing that an instance was predicted positive does not give us any information about whether it really is positive. A positive coefficient implies that instances predicted positive (negative) have an increased chance of indeed belonging to that class, whereas a negative coefficient implies that instances have a decreased chance to belong to the class that was predicted.

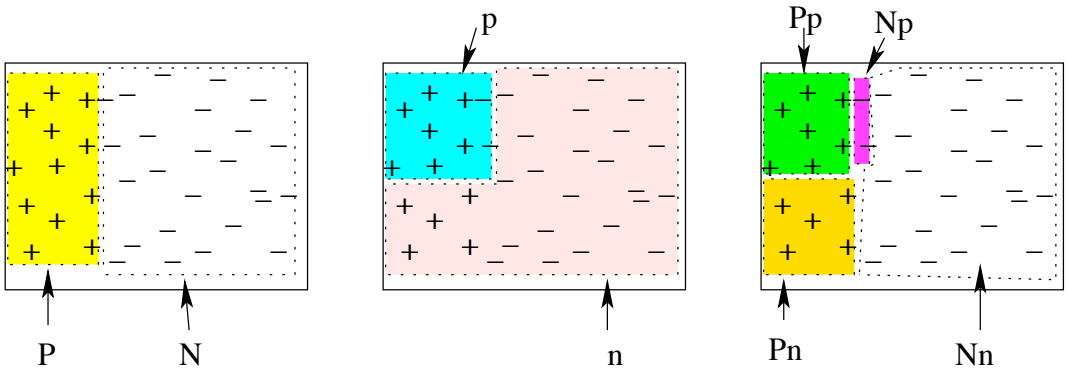

Figure 6.1: Schematic overview of the sets P and N (positive / negative instances), p and n (instances predicted positive / negative by some model), and their intersections Pp, Pn, Np and Nn.

	P	N	
p	$Pr(Pp)$	$Pr(Np)$	$Pr(p)$
n	$Pr(Pn)$	$Pr(Nn)$	$Pr(n)$
	$Pr(P)$	$Pr(N)$	1

	P	N	
p	0.1	0.2	0.3
n	0.3	0.4	0.7
	0.4	0.6	1

Table 6.1: The left table shows how the discussed probabilities are usually represented in a contingency table. Elements in the lowest row (rightmost column) have the property that they are the sum of the elements above (left of) them. To the right, an example contingency table is shown.

Exercise 6.1 Compute the ϕ coefficient for the Classifiers 1 and 2 from Example 6.1.

Exercise 6.2 Show that $\phi > 0$ implies $Pr(P|p) > Pr(P|n)$, that is, an instance that is predicted positive has a higher probability of being positive than an instance predicted negative.

Example 6.1 illustrates that one has to be careful with evaluating classifiers based on their accuracy. Exercise 6.15 illustrates another potential problem with the use of accuracy. Yet another shortcoming of accuracy is that it does not take into account the possibility that some kinds of mistakes may be worse than others. That is what the following section is about.

Cost-based evaluation

Up till now, our formulae for evaluating classifiers had a certain symmetry. We have been counting "mistakes", without paying attention to what kind of mistake was made. But misclassifying a positive as negative is in some situations very different from misclassifying a negative as positive. For instance, when a doctor makes a diagnosis about a patient, diagnosing the patient as healthy when he is ill will have very different consequences, compared to diagnosing illness for a healthy patient. Similarly, when a bank has to decide whether to give a loan to a client, the cost associated with one type of mistake (giving the loan to a client who afterwards turns out to be unable to pay it back) is very different from the cost of another type of mistake (not giving a loan to a client who would have paid it back).

To take such asymmetries into account, we can associate a *cost* to each type of mistake. Generally, when there are n classes, with each misclassification of a class i

example as class j, a cost $c_{i,j}$ is associated. The **expected cost** made by a classifier over a population is then defined as follows.

Definition 6.3 (expected cost) *The expected cost of a hypothesis h with respect to a target function f, cost function c and distribution D, is*

$$EC_{D,f}(h) = \mathbf{E}_{x \sim D}(c_{f(x),h(x)}).$$

Note that the error of a model, as defined before, is a special case of the expected cost where $c_{i,j} = 1$ if $i \neq j$, and 0 otherwise.

We will call a cost function **reasonable** if all costs are positive and the cost of any misclassification is at least as high as the cost of any correct classification. (Unreasonable cost functions would imply it is better to make mistakes than to make correct predictions, which is not useful for our purposes.) Often, correct classifications are assigned a cost of zero, and misclassifications a strictly positive cost.

If we want to evaluate a predictor based on misclassification costs, it is not sufficient to know what the probability of making a mistake is; we need to know for each different kind of mistake what the probability of making that mistake is. That is, we need to know the probability that a random example belongs to class i and is predicted as class j. According to the definition of conditional probabilities, we can decompose this probability:

$$Pr(f(x) = i \wedge h(x) = j) = Pr(h(x) = j | f(x) = i) Pr(f(x) = i)$$

So it is sufficient to know the class distribution $Pr(f(x) = i)$, as well as the distribution of predicted classes for each true class, $Pr(h(x) = j | f(x) = i)$.

In the case of two classes (positive and negative), specific names are given to a number of relevant probabilities:

formula	name(s)
$Pr(h(x) = pos \mid f(x) = pos)$	true positive rate (TP), recall, sensitivity
$Pr(h(x) = pos \mid f(x) = neg)$	false positive rate (FP),
$Pr(h(x) = neg \mid f(x) = neg)$	true negative rate (TN), specificity
$Pr(h(x) = neg \mid f(x) = pos)$	false negative rate (FN)
$Pr(f(x) = pos \mid h(x) = pos)$	precision

Several probabilities have different names: which one is used depends on the context. Note that in the "true/false X rate" terminology, X refers to the predicted class, not to the actual class. Also note that precision is conditioned on the predicted class, while the other probabilities are conditioned on the true class.

We mention all these names here because in certain contexts, evaluation of classifiers relies heavily on these measures. For instance, in information extraction, models are often evaluated using "precision-recall" (PR) curves, and these curves are becoming increasingly popular in machine learning research. In the medical world, the terms "selectivity" and "specificity" are often used. A certain kind of diagrams called "receiver operating characteristics" (ROC) diagrams has been becoming increasingly popular in machine learning in the late nineties; in these diagrams, the true positive rate is plotted against the false positive rate.

In the following we first have a look at precision-recall diagrams, then we will focus on ROC diagrams. ROC diagrams are particularly useful for cost-based evaluation, as we will see in a moment.

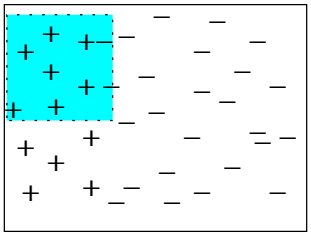

Figure 6.2: Precision and recall illustrated on a finite population. Pluses and minuses indicate positive and negative instances, the shaded area contains the instances predicted positive by some learner. The precision of this result is 7/9 (the shaded area contains 9 instances of which 7 are positive) and the recall is 7/12 (out of 12 positive instances, 7 are predicted positive).

Precision-recall diagrams

To understand the reasoning behind the use of precision-recall curves, consider the context of information retrieval. Say, a user looks on the web for pages on some specific topic. After entering the topic in a search engine, a list of pages is shown to the user. Let us assume that pages are either relevant or not. Ideally, the user wants to see *all* the relevant pages for his topic, and *only* those. Recall is the percentage of all relevant pages that is indeed in the list. Precision is the percentage of pages in the list that are relevant.

Exercise 6.3 Compare the above description of precision and recall to the formal definition of these terms. Map the terminology of "relevant" and "listed" onto predicted/actual positive/negative.

Note that by returning a longer list, the search engine can increase the recall: the more pages are returned, the higher the percentage of the relevant pages that can be included. But a longer list will typically also reduce the precision: there will be more non-relevant pages in the list. Once a recall of 100% has been reached, making the list longer can only increase the percentage of non-relevant pages in this list, i.e., reduce the precision.

Now consider a classifier that predicts instances to be positive or negative. The "list of results returned" corresponds to the set of instances predicted positive. Thus, a single classifier obtains a certain precision and recall. Figure 6.2 illustrates this.

Precision-recall results are typically shown in a two-dimensional diagram, where recall is shown on the horizontal axis and precision on the vertical axis. A classifier that always predicts negative has a recall of 0 and its precision is undefined. A classifier that always predicts positive has a recall of 1 and a precision equal to the proportion of positives in the population. Other classifiers can be anywhere in the diagram. Figure 6.3(a) plots some classifiers.

In some practical situations, such as information retrieval, we might be interested in making the list of returned results longer or shorter. How can we do that? If we want a shorter list, we can just make a random selection from the positive predictions and include those. But sometimes the classifier is more certain about some predictions than about others. It makes sense, then, to include first those instances of which it is most certain that they are positive: that will increase both the recall and the precision of the list. Similarly, if we want to make the list longer, we can add some instances that have been predicted negative, and then it makes

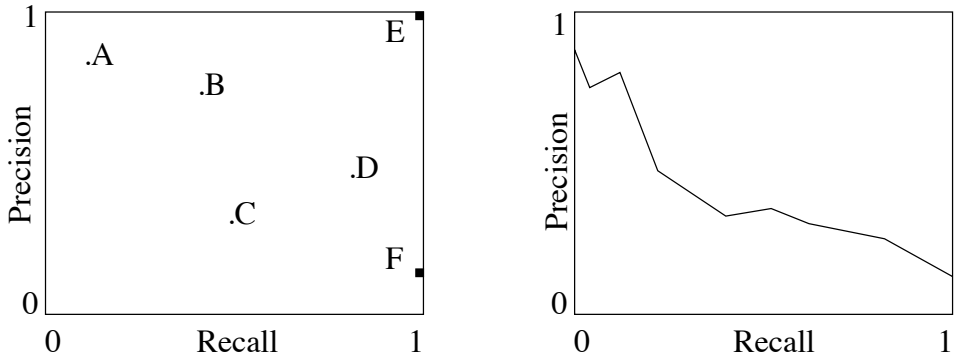

Figure 6.3: Example precision-recall diagrams. (a) A few classifiers plotted in a precision-recall diagram. Classifier A has high precision but low recall. D has much higher recall, but not very high precision. E is a perfect predictor. F is a predictor that always predicts positive. (b) A rank classifier plotted in a precision-recall diagram. Lowering its threshold makes it go from high precision and low recall to low precision and high recall.

sense to first add those instances of which we are least certain that our prediction is correct.

Some classifiers do not only return a prediction but also some kind of certainty about this prediction. They are called **rank classifiers**. They rank their predictions according to how certain they are about it. A special case of rank classifiers are **probabilistic classifiers**: these return together with their prediction an estimate of the probability that the prediction is correct.

Decision trees, for instance, can be seen as rank classifiers, and even probabilistic classifiers. When an instance ends up in a leaf, it is normally assigned the class that occurs most frequently in that leaf. If it is a leaf with only positive instances, then we could say that the tree is quite sure that the instance is positive. But if it is a leaf with 80% positive instances, then it makes sense to say that the instance is probably positive, but not certainly; and more specifically, we could guess that it has a 80% probability of being positive.

Rank classifiers are interesting because we can adapt their precision and recall. We can do this in the following way. Assume the classifier makes k predictions. We can list them from "most likely to be positive" to "least likely to be positive". Then, we can choose any number i between 1 and k and decide that the first i cases are to be considered positive predictions.

By choosing i we choose the length of the list to be returned. By ordering the instances from most likely to least likely to be positive, we try to ensure that our list of i instances has the highest recall and precision among all lists of length i. If we gradually increase i from 0 to k, we are producing ever longer lists. Each of these lists will have its own recall and precision. For each extra result included in the list, either it is a relevant result, in which case both recall and precision go up (we move to the upper right in the diagram), or it is irrelevant, in which case the recall stay the same and the precision goes down (we move down in the diagram). A list with only relevant results always has precision 1, but the recall is between 0 and 1 and is higher if the list is longer.

By following this procedure, a rank classifier gives rise to many different points, which we can all plot in a precision-recall diagram. The rank classifier is then represented by a precision-recall curve instead of a single point. Figure 6.3(b) shows a typical PR-curve for a rank classifier.

Precision-recall curves are typically used in highly skewed class distributions, more specifically, when there are very few positive examples and we want a classifier that returns as many of them as possible without returning too many negatives. It is not always clear in advance how many results we want to see. By learning a rank classifier and plotting its PR curve, we can choose some optimal point on that curve and therefore an optimal "list length" i.

Receiver operating characteristics diagrams.

In ROC diagrams, the false positive rate (FP) is plotted on the horizontal axis, and the true positive rate (TP) on the vertical axis. Like in precision-recall diagrams, a normal classifier is represented by a single point in the diagram, and a rank classifier is represented by a curve.

To understand ROC diagrams better, it is useful to consider a number of specific cases:

- perfect prediction: the classifier predicts everything correctly. That means it has a true positive rate of one (all positives are predicted positive) and a false positive rate of zero (no negatives are predicted positive, i.e., all negatives are predicted negative). This classifier is in the upper left corner in the diagram (TP=1,FP=0).
- maximally incorrect prediction: the classifier is always wrong. That means TP=0 and FP=1. This classifier is in the bottom right corner.
- random prediction: the classifier makes entirely random predictions. Since such predictions are independent of the actual class, all instances have the same probability of being predicted positive, whether they are positive or negative: $Pr(h(x) = pos) = Pr(h(x) = pos|f(x) = pos) = Pr(h(x) = pos|f(x) = neg)$. By definition, then, TP = FP: the classifiers is on the diagonal of the diagram.
- A classifier with precision 1: This corresponds to FP=0. Such a classifier is on the left border of the diagram.
- A classifier with recall 1: This corresponds to TP=1. Such a classifier is on the top border of the diagram.
- A rank classifier gives rise to a curve. If we reduce its positive predictions to zero, then it scores $TP = FP = 0$. If it predicts everything as positive, then $TP = FP = 1$. If one instance is added to the set of positive predictions, then if it really is positive, TP goes up (the curve goes up), otherwise FP goes up (the curve goes to the right). By gradually increasing the number of positive predictions from 0 to all, a curve from (0,0) to (1,1) is obtained.

All this is illustrated in Figure 6.4.

Iso-accuracy lines

If we know that a classifier has accuracy a, can we say where it will end up on the ROC diagram?

To see this, we have to write its accuracy as a function of TP and FP. Note that accuracy is the probability of predicting any instance correctly, TP is the probability of predicting any positive instance correctly, and (since FP is the probability of predicting any negative incorrectly) 1-FP is the probability of predicting any negative instance correctly. Therefore the accuracy is a weighted average of TP and 1-FP, weighted according to the class distribution. Mathematically:

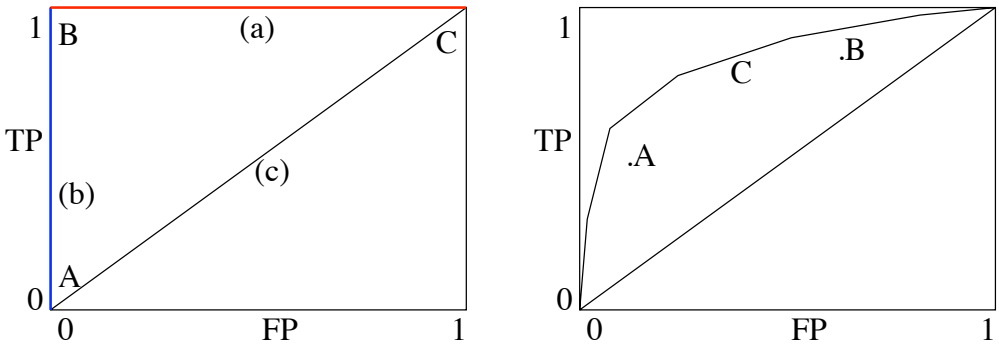

Figure 6.4: Left: A ROC diagram with special areas indicated. Classifiers with recall 1 are on the upper edge of the diagram (a), and classifiers with precision 1 are on the left edge (b). Random classifiers are on the diagonal (c). A is a classifier that always predicts negative, C is one that always predicts positive. B makes only correct predictions. Right: An example ROC diagram showing the performance of two classifiers A and B and a rank classifier C. B has a higher true positive rate than A but also returns more false positives. The rank classifier C can have low TP and FP or high TP and FP depending on its threshold.

$$\begin{aligned} Acc &= Pr(f(x) = h(x)) \\ &= Pr(h(x) = pos \wedge f(x) = pos) + Pr(h(x) = neg \wedge f(x) = neg) \\ &= Pr(h(x) = pos | f(x) = pos) Pr(f(x) = pos) \\ &\quad + Pr(h(x) = neg | f(x) = neg) Pr(f(x) = neg) \\ &= TP \cdot Pr(f(x) = pos) + (1 - FP) \cdot Pr(f(x) = neg). \end{aligned}$$

Choosing a shorter notation: $p^+ = Pr(f(x) = pos)$ and $p^- = Pr(f(x) = neg)$, we have f

$$Acc = TP \cdot p^+ + (1 - FP) \cdot p^-.$$

To see where a classifier with accuracy a ends up, we fill in a for Acc, which gives

$$TP \cdot p^+ + (1 - FP) \cdot p^- = a.$$

Rewriting this to express TP as a function of FP, we get

$$TP = p^-/p^+ \cdot FP + (a - p^-)/p^+$$

This shows that all classifiers with accuracy a are on a straight line with slope p^-/p^+ and intercept $(a - p^-)/p^+$ in the ROC diagram. We call such a straight line an **iso-accuracy** line. Note that given a fixed p^-/p^+, a classifier with higher accuracy a is closer to the upper left corner.

Iso-cost lines

We have said earlier that accuracy, or rather the corresponding error, is a special case of misclassification cost where the cost of misclassifying an instance is always the same, regardless of the true and predicted class of the instance. Similarly to lines of equal accuracy (iso-accuracy lines), we can now look for lines of equal cost, or **iso-cost lines**.

$$Cost = \sum_{i,j} Pr(h(x) = j \text{ and } f(x) = i) \cdot c_{i,j}$$

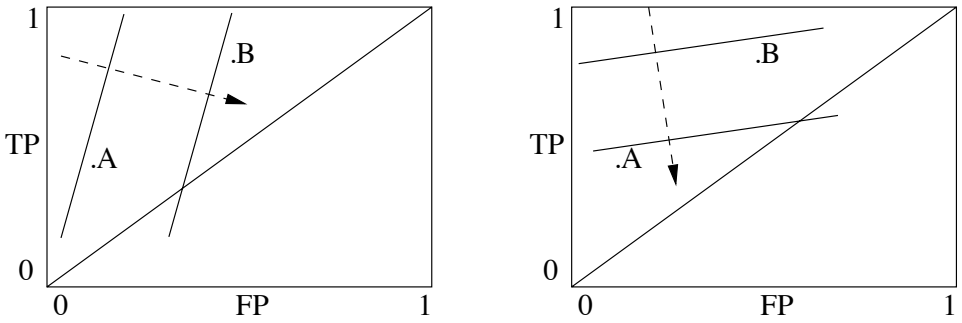

Figure 6.5: An illustration of different classifiers and different iso-cost lines. The arrows indicates the direction in which the cost increases. If the iso-cost lines are steep (left diagram), A has a lower cost. If the iso-cost lines are more level (right diagram), B has a lower cost.

For the two-classes case and assuming that correct classification has cost 0, this formula becomes

$$\begin{aligned} Cost &= Pr(h(x) = neg \text{ and } f(x) = pos) \cdot c_{pos,neg} + Pr(h(x) = pos \text{ and } f(x) = neg) \cdot c_{neg,pos} \\ &= Pr(h(x) = neg | f(x) = pos) \cdot Pr(f(x) = pos) \cdot c_{pos,neg} \\ &\quad + Pr(h(x) = pos | f(x) = neg) \cdot Pr(f(x) = neg) \cdot c_{neg,pos} \\ &= (1 - TP) \cdot Pr(f(x) = pos) \cdot c_{pos,neg} + FP \cdot Pr(f(x) = neg) \cdot c_{neg,pos} \\ &= (1 - TP) \cdot p^+ \cdot c_{pos,neg} + FP \cdot p^- \cdot c_{neg,pos} \end{aligned}$$

A classifier with $Cost = c$ is therefore described by

$$c = (1 - TP) \cdot p^+ \cdot c_{pos,neg} + FP \cdot p^- \cdot c_{neg,pos}$$

which, after rewriting it to express TP as a function of FP, gives

$$TP = \frac{p^-}{p^+} \cdot \frac{c_{neg,pos}}{c_{pos,neg}} \cdot FP + 1 - \frac{c}{p^+ \cdot c_{pos,neg}}$$

Thus, the iso-cost line representing all classifiers with cost c is a straight line with slope $p^-/p^+ \cdot c_{neg,pos}/c_{pos,neg}$ and intercept $1 - c/(p^+ \cdot c_{pos,neg})$.

Note that for any reasonable cost function, a classifier is better if it is on an iso-cost line that is closer to the top left corner. However, these iso-cost lines have different slopes for different cost functions. Given two classifiers A and B, and their position in the ROC diagram, it may be impossible to know which one has the lowest cost unless we know the slope of the iso-cost lines in the diagram. Figure 6.5 illustrates this.

The motivation for using ROC diagrams for evaluation is the following. When we have two classifiers, it may not be clear which one we should use in practice, unless we know some characteristics of the environment in which they will be used. The ROC diagram allows us to see under which circumstances one classifier is better than an other one, and this is one reason why it has become popular.

Consider two classifiers A and B, with corresponding TP_A, FP_A, Acc_A and TP_B, FP_B, Acc_B. Assume $Acc_A > Acc_B$. Does this mean that A is to be preferred under all circumstances? The answer is no. Although A has higher overall accuracy, it might make more mistakes in one direction than B, while making fewer mistakes in the other direction. For instance, $FP_A > FP_B$ (A predicts more negatives as positive) and $TP_A > TP_B$ (A predicts fewer positives as negative). There are two kinds of circumstances where A then actually becomes worse than B.

- The cost of a false positive is much higher than the cost of a false negative. Then the higher FP of A will not be compensated by its better TP.
- There are many more negatives than positives. Note that when we say that A has a higher accuracy than B, this is for a given class distribution. When the class distribution changes, A may become *less* accurate than B. More precisely, since $Acc = TPp^+ + (1 - FP)p^-$, an increase of p^- (and corresponding decrease of p^+) will give FP more weight in the accuracy.

Example 6.2 Take the example of flu vaccinatins. In most years, only a small percentage of people get the flu. Yet doctors advise vaccinations for many more people: basically for anyone with an increased risk of getting the flu, even though few of them will actually get it. The cost of a non-vaccinated person getting the flu is indeed much higher than the cost of a single vaccin.

Vaccination against smallpox is a good example of changing class distributions. Vaccination against smallpox was common many years ago. But as smallpox is now considered eradicated, no vaccinations against smallpox are administered anymore. The reason for this is not that the cost of vaccines has gone up or the cost of getting smallpox has gone down, but that the probability of getting smallpox (p^+) has gone to zero.

All these circumstances are recognizable in the ROC diagram. Both the cost ratio $c_{pos,neg}/c_{neg,pos}$ and the class distribution p^+/p^- determine the slope of iso-cost lines, and p^+/p^- determines the slope of iso-accuracy lines. So which classifier is on the best iso-cost or iso-accuracy line, A or B, depends on these ratios.

We can conclude that a higher accuracy or lower cost for A under certain circumstances does not necessarily imply that A is better than B under all circumstances. Yet, there exist conditions under which we can conclude that A is always better than B. When $TP_A > TP_B$ and $FP_A < FP_B$, then there exist no reasonable cost functions for which A is worse than B. On the ROC diagram, this means that A is to the upper left of B.

Generally, when A is to the upper left of B, it is always better, for any reasonable cost function and any class distribution. When A is to the lower right of B, it is always worse than B. When A is to the upper right or lower left of B, it depends on the cost function and class distribution whether A is better than B.

The ROC convex hull

Assume now that we have a set of classifiers. When we know in what circumstances the classifiers will be used (the operating characteristics), we can choose one single classifier from the set that we know will perform best. If we don't have this information, there may not be a single classifier that works best under all circumstances, but there may be classifiers that can never be optimal, in whatever circumstances, and these classifiers can be removed. Which classifiers should we remove, then, and which classifiers should we keep?

The classifiers that we should keep, are identifiable in the ROC diagram because they are on the so-called **ROC convex hull**, the convex hull of all the classifiers in the diagram.[1] Figure 6.6 shows a set of classifiers and their convex hull.

It is easy to see that a classifier below the convex hull can never be optimal. One way to see this is the following. Think of any iso-cost line through the (0,1) point (call this line L). For a reasonable cost function, this line will always have a positive slope. Lines parallel to L are also iso-cost lines, and if they are to its

[1] Given a set of points in a 2-D diagram, their convex hull is the smallest polygon that contains all the classifiers and is convex. Convexity means that whenever two points are inside or on the polygon, the entire straight line connecting them is inside or on the polygon.

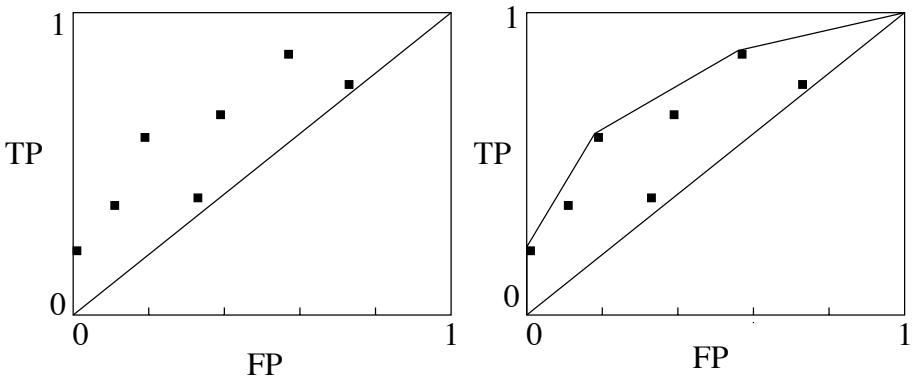

Figure 6.6: A set of classifiers in a ROC diagram, and the convex hull around the set.

lower right they represent a higher cost. So by gradually shifting an iso-cost line to the lower right, starting from L, the line will cover points with gradually increasing cost. The first classifier that the line hits, is then the classifier with lowest cost. Clearly, the moving iso-cost line cannot hit any classifiers below the convex hull without first hitting the convex hull itself, and that means means hitting at least one classifier on the convex hull.

Another way to see that a classifier below the convex hull can never be optimal, is the subject of exercise 6.17.

An important point to note is that when a classifier A can never be optimal, this does not mean that there exists a single classifier that is *always* better than A. It just means that whatever the cost function and class distribution is, you can always find some classifier better than A *under these circumstances*.

6.1.2 Evaluation of numerical predictors

Up till now, we have talked about evaluating classifiers, where predictions are on a nominal scale: they are either correct or not. For numerical predictions, such an evaluation criterion does not make sense: a prediction will rarely be exactly equal to the actual value. Rather, we want the prediction to be close to the target value. Evaluation will be based on "how far off" the predictions are, on average.

This can be measured by the expected absolute error (EAE) and expected squared error (ESE) of predictions. Given a target function f,

$$EAE(h) = \mathbf{E}(|h(x) - f(x)|)$$

$$ESE(h) = \mathbf{E}((h(x) - f(x))^2)$$

The use of a squared error instead of absolute error may seem a bit far-fetched, yet in practice the squared error is used more often than the absolute error. This is mainly for technical reasons: it is easier to manipulate mathematically than the absolute error, and has a number of other desirable properties.

The EAE and ESE depend on the range of the target attribute, or the scale on which it is expressed. As a consequence, we cannot tell whether an ESE of, say, 150, is good or bad. This problem disappears if we compare the ESE to that of a trivial predictor. For instance, a predictor that always predicts the population mean, will have an ESE equal to the variance in the population. The ratio of a predictor's ESE to this variance is called the **relative error**, RE, of the predictor.

The relative error of a hypothesis h, compared to some trivial hypothesis t, is:

$$RE_t(h) = ESE(h)/ESE(t)$$

In practice, one often uses for t the predictor that always predicts the population mean. Then $ESE(t)$ is equal to the population variance σ^2. We will then drop the t subscript.

$$RE(h) = ESE(h)/\sigma^2$$

All the above measurements somehow relate to how close the predictions are to the actual values. This is not the only possible criterion. Sometimes one is less interested in the closeness of the predictions, but more in the correlation between predicted values and actual values. Pearson's correlation coefficient can be used for this. If we define the deviations of f and h from their mean as

$$\begin{aligned} h'(x) &= h(x) - \mathbf{E}(h(x)) \\ f'(x) &= f(x) - \mathbf{E}(f(x)) \end{aligned}$$

then Pearson's correlation coefficient is defined as

$$\rho = \frac{\mathbf{E}[h'(x)f'(x)]}{\sqrt{\mathbf{E}[h'(x)^2]\mathbf{E}[f'(x)^2]}}$$

Finally, like with classifiers, we can consider a cost-based model of evaluation, by explicitly associating a cost with any errors made. Assuming there is a cost function $c(x, y)$ that gives us the cost of predicting y when the actual value is x, the expected cost of a model h is

$$EC(h) = \mathbf{E}(c(f(x), h(x)))$$

The EAE and ESE are special cases of expected cost, with $c(x, y) = |x - y|$ and $c(x, y) = (x-y)^2$, respectively. But cost-based evaluation also allows us to use more exotic cost functions. For instance, overestimating the target value (predicting a value that is too high) might be less costly than underestimating it; in that case, a cost function would be used that is not symmetric, e.g., $c(x, y) = (x - y)^2$ if $x > y$ (underestimate), and $c(x, y) = y - x$ if $x < y$.

Ordinal scales

Sometimes predictions are made on ordinal scales: the values are then usually expressed as numbers, but the numbers only indicate whether a value is greater than an other one, and the difference between two numbers does not have much meaning.

For instance, we could define three temperature levels *high*, *medium*, and *low*. These form an ordinal scale. We could represent them as numbers 1,2,3, which keeps the order among the values, but it is important to realise that the difference between two numbers does not have any real meaning (it doesn't make sense to say that the difference between high and low is twice the difference between high and medium, for instance).

The difference between values may not be meaningful, or it may be well-defined but not very useful. As an example of that, consider the following problem, taken from an actual machine learning application. A number of chemical substances

are given, with a description of their molecular structure and an indication of how fast they degrade in the environment. Degradation is measured using a so-called half-value times (the time until half of the chemicals have disappeared from the environment). These half-value times (HVT's) may range from hours to years.

Now, when using a learner to predict half-value times, the difference between the prediction and the actual value is not a very good measure for the quality of a prediction. If the learner predicts a HVT of 300 days when the actual HVT is 310 days, we would consider that a quite accurate prediction. If the learner predicts 1 hour when it is in fact 1 week, that is quite far off. Yet, the difference between 1 hour and 1 week is about 7 days, while the difference between 310 days and 300 days is 10 days.

In cases like the above, it is better to represent the target values and predictions on a logarithmic scale. Differences on the logarithmic scale are much more meaningful than differences on the original scale.

The trick of using a logarithmic (or perhaps some other) transformation on the target values is an important one that is often useful in practice. For truly ordinal data, such as in the high/medium/low example, no transformation may exist that yields a scale in which differences are meaningful. The quality of predictions then cannot be measured using differences, but alternative measurements such as "rank correlation" exist. We do not discuss the details of this, as ordinal prediction does not seem very common in machine learning. Most standard textbooks on statistics consider ordinal values and how to process them.

6.1.3 Evaluation of complex predictors

Besides classifiers and numerical predictors, we can consider models that make complicated predictions. They might predict a vector, a tree or graph structure, etc. In such cases, it depends on the application what evaluation criterion should be used. The cost-based evaluation approach is quite flexible in this respect, and generalizes easily to this kind of applications.

We can again define a cost function $c(x, y)$ indicating the cost of predicting y when x is the actual value. Sometimes the natural cost function is a distance (this is the case if $c(x, y) \geq 0$, $c(x, y) = 0 \Leftrightarrow x = y$, $c(x, y) = c(y, x)$, and $c(x, y) \leq c(x, z) + c(y, z)$ for all x, y, z), in which case we speak of distance-based evaluation.

When the target is a vector of reals, then the euclidean distance between x and y can be used for $c(x, y)$. When the target is a vector of nominal values, the Hamming distance can be used. Generally, for a vector with n components of possibly different types, a distance can be defined as follows. If x and y are vectors of type $D = D_1 \times D_2 \times \cdots \times D_n$,

$$d(x, y) = (\sum_{i=1}^{n} d_i(x_i, y_i)^k)^{1/k}$$

where each d_i is a distance function over D_i. Note that this reduces to the euclidean and Hamming distances with appropriate choices of k and d_i.

Also predictions that are sets, trees or graphs can be handled using the cost-based or distance-based evaluation. Distance measures for sets, trees and graphs have been proposed by several researchers, but discussing them in detail would lead us too far.

6.2 Other properties of models

With all these different methods for evaluating the quality of predictions of a model, one would almost forget that there is more to machine learning than just building

models that make accurate predictions. In the knowledge discovery context, we are often also interested in gaining insight in the domain, in understanding what the model means or why certain predictions are made.

6.2.1 Interpretability

The notion of interpretability is much more difficult to quantify than predictive performance. When is a model interpretable? This depends on the *format* in which the model is represented, and the extent to which the reader of the model is familiar with that format. It also depends on the *overall size* of the model, and on whether the model can be *decomposed* in smaller chunks that can be interpreted in isolation. In addition, there is the question of how easily a single *prediction* can be interpreted, which is different from interpreting the model.

We discuss three levels of interpretation: full models, partial models, and individual predictions.

Interpreting full models

The interpretability of a model as a whole depends to a large extent on the complexity of the model.

Symbolic models such as rule sets and decision trees are generally considered easy to interpret. This is mostly related to the fact that they can be decomposed into smaller parts that are easier to interpret, and because the predictions they make can easily be explained. But if we ignore this fact and focus on understanding the full model, only relatively small rule sets / decision trees are really easy to interpret. The size of the full model can be measured as, for instance, the number of nodes in a tree, or the total number of conditions in a rule set.

Even relatively large symbolic models are usually considered easier to interpret than subsymbolic or numerical models, such as neural networks and support vector machines. These are often called black-box models. Yet, some interpretation is usually possible. The weights of the connections in a neural network can be interpreted as indicating a positive or negative influence of one node on another one. Support vector machines could be said to identify the "border cases" of different classes, which is also some kind of interpretation.

Pure instance based models offer no interpretation at all at the level of the full model, as the model is simply the training dataset itself. Instance-based models that identify prototypes offer some interpretation, in a similar way that clustering does: they identify "typical" cases that are representative for a whole group of instances.

One technique that is sometimes used to gain some understanding in full predictive models is **what-if analysis**. What-if analysis consists of using the model to simulate certain situations. Inputs are given to the model that reflect some meaningful situation, and the output of the model for these inputs is studied.

For instance, in the context of managing the quality of river water, people have used neural networks to model the effect of certain parameters of rivers (width, depth, flow velocity, ...) on certain quality criteria, such as biodiversity. Even though the network does not explicitly show the relationship between, say, width and biodiversity, one can input a description of the current situation to the network and look at the outcome, then vary the width parameter and see how the outcome changes.

An advantage of what-if analysis is that it can be performed for any predictive model, from simple symbolic models to complex black-box models. Moreover, it allows us to focus on specific situations, and to try to understand the model in

those situations without attempting to grasp the complex model as a whole. In a sense, it provides that local interpretation of the model.

Interpreting parts of models

Whatever format is used, when the full model is very complex, it is difficult to grasp it all at once. It is very useful, then, when the model consists of different parts that can be understood in isolation.

Rule sets Rule sets are the typical example of such models. A model might be very complex and consist of hundreds of rules, but if each rule in itself is simple and interpretable, humans tend to consider the whole model interpretable.

A single rule can still be simple or complex. This depends for instance on its length. Just like the number of conditions is a measurement for how complex a full model is, it can also measure how complex a part of a model (a single rule) is.

An important advantage of rules is that they can easily be transformed into a natural language sentence. Even though a symbolic representation of an if-then rule may seem quite readable, many domain experts find that reading many of them is much easier when they are expressed in natural language. This is especially the case when there are many variables with not very clear names, when single conditions can be relatively complex, or when a first order logic is used (which is inherently more complex than propositional logic).

Example 6.3 A first order logic rule such as

```
pos(X) <- contains(X,A), shape(A, triangle), contains(X,B),
          shape(B, circle), in(A,B)
```

can automatically be transformed into a sentence

"An example is positive if it contains a triangle A and a circle B and A is inside B"

which is much more readable.

Since decision trees can be turned into rule sets, they also have this advantage of consisting of interpretable chunks. A single rule corresponds to a path from the root of the tree to a leaf.

An important point here is that the above remarks hold for rule sets with unordered rules. Decision lists, where a rule is only relevant for some instance if none of the previous rules made a prediction for it (hence the order of the rules matters), do not have the property that a single rule can be interpreted exactly without looking at the other rules. They can at best be interpreted approximately, considering them as rules of thumb that may have exceptions.

Linear regression models Linear regression models also exhibit a certain kind of "decomposability in chunks", but a different kind. A variable's coefficient in such a model indicates the effect that that variable has on the outcome variable. An important property of linear models is that this effect does not depend on the value of other variables.[2]

[2]One has to be a bit careful with this interpretation though. The coefficient of a variable shows the effect of the variable *under the assumption that all the other variables stay the same.* When variables are not independent but are correlated, this is not a realistic assumption. As a result, a positive coefficient (for instance) should not be interpreted as "higher values for this variable co-occur with higher values for the outcome", unless one is sure that the variable is not correlated with other variables in the model.

Note that rules/trees and linear models exhibit different kinds of "local interpretability". For rules or trees, a model chunk (a single rule or leaf) represents a small part of the instance space. In linear models, the model chunk on which we focus represents the influence of a single variable throughout the whole instance space.

Exercise 6.4 How can the "overall influence" of a single attribute (which is obvious in a linear model) be extracted from a decision tree or a set of rules?

Exercise 6.5 Explain how you can perform what-if-analysis with trees, rule sets, and linear models.

Interpreting predictions

Besides understanding the model, domain experts are also often interested in understanding a single prediction.

Take the example of a bank's client whose application for a loan has been rejected. It is useful to be able to explain to that client why he does not get the loan, but for that it is not necessary to explain the full model. There might be many reasons why one does not get a loan, but this client is only interested in the reason that applies to himself.

Rule sets and decision trees have again an advantage here. The prediction made by a rule set can be attributed to one single rule, and the prediction made by a decision tree can be attributed to a single leaf. In both cases, it is possible to list exactly the conditions that have led to the prediction. (For rules, it is of course possible that several rules cover the example. They offer different explanations then, each of which is valid in itself. Mentioning just one of these as an explanation is then sufficient.)

6.2.2 Complexity

We have already discussed how complexity is related to interpretability. However, complexity is also an important property for other reasons.

First, there is the memory complexity of a model. Models such as support vector machines and instance-based models may grow very large. (The instance-based model is roughly as big as the input dataset.) Also decision trees and rule sets can be large; indeed, when there exists no simple decision tree with perfect prediction, and when there are enough variables, the learner typically keeps on expanding the tree until each leaf contains only a few instances. Clearly, the number of nodes is then only a few times smaller than the number of data. Such trees do not differ all that much from instance-based models. The main advantage of the trees is that we can force them to be smaller, if we want to (e.g., by enforcing a minimal coverage for each leaf).

Exercise 6.6 Think about the relationship between large decision trees and instance-based models. Consider the limit case where there are as many examples to learn from as you want, each example is described by a finite set of attributes with symbolic values, and each descriptive attribute (not the class) is allowed to occur in the tree.

One might wonder whether the memory complexity of a model is all that relevant, now that computers have gigabytes of internal memory. It is, for several reasons. First, a learned model might be embedded in another device than a computer, or might be implemented in a single chip. Storage limitations may be more severe in such cases. Second, more and more intelligent and adaptive software is

appearing on our computers. Many of these adaptive programs run in parallel. We don't want each of these programs to use large amounts of memory just to store their models.

Second, there is the computational complexity. Sometimes a model that has been learned will be used very intensively, and it needs to make predictions very fast. Then the computational complexity of the model is relevant. Decision trees and feedforward neural networks, and to a lesser extent rule sets, are all very fast. Instance based models tend to be slower: finding the k nearest neighbours of an instance in a set of millions of instances requires very good indexing mechanisms.

Another reason for keeping the computational complexity of a model low, is that the "computations" involved may in fact boil down to obtaining information from external sources, for instance sensors or even humans. This process may be slow, especially when humans are involved. Moreover, when for instance all the inputs necessary to make a prediction need to be obtained from a human, this imposes a burden on that human.

Example 6.4 Consider again the example of a bank deciding whether to give someone a loan or not. The bank may use a questionnaire, asking questions to the client and using the answers to reach a decision. Surely the bank wants to reach a decision with as few questions as possible.

The use of information gain as a heuristic for building decision trees relates directly to the goal of reaching a decision with as few questions as possible.

6.3 Estimating a model's quality in practice

The previous sections discussed theoretical evaluation measures, many of which are based on some distribution of instances in the instance space. In practice, we usually do not know this distribution. We then have to estimate these measures from a limited data set.

6.3.1 Sample-based estimates of evaluation measures

Classification

For classification, measures based on probabilities are typically estimated by a proportion on a finite data set. For instance, the accuracy of a model (the probability that it makes a correct prediction on any instance) is estimated as the proportion of correct predictions on some finite data set. If the model predicts 170 out of 200 cases correctly, we estimate its accuracy as $170/200 = 0.85$.

Similarly, a model's error, correlation, TP, FP, TN, FN, ... are all obtained by substituting for the probabilities in their definitions estimates based on proportions in a test set.

Exercise 6.7 Assume we have a test set with 100 positives and 200 negatives. A model h predicts 70 out of the 100 positives as positive, and the rest as negative; it also predicts 50 negatives as positive and the rest as negative. Estimate the accuracy, error, correlation, TP, FP, TN, FN of h.

Regression

The expected values EAE and ESE that we saw before are estimated by making predictions on some set of instances and comparing the predicted values with the actual values. Given a set S, the following definitions are commonly used:

- sum of squared errors (SSE): $SSE_S(h) = \sum_{x \in S}(h(x) - f(x))^2$
- mean squared error (MSE): $MSE_S(h) = \sum_{x \in S}(h(x) - f(x))^2/|S|$
- mean absolute error (MAE): $MAE_S(h) = \sum_{x \in S}|h(x) - f(x)|/|S|$
- relative error w.r.t. a trivial model t: $RE_{S,t}(h) = MSE_S(h)/MSE_S(t)$
- relative error (RE): $RE_S(h) = MSE_S(h)/Var(f)$

6.3.2 The importance of independent test sets

How well do sample-based estimates approximate the corresponding measures? Statistics tells us that, under certain conditions, the sample-based estimates have an expected value that is equal to the theoretical measures: $\mathbf{E}(\hat{\theta}) = \theta$ where θ represents some measure and $\hat{\theta}$ its estimate. In other words, and roughly speaking: given some estimated value, there is no reason to believe that the actual measure is probably higher, or that it is probably lower. In those cases that the expected value is not equal to the measure, we say that there the estimator is biased, and its **bias** is defined as $\mathbf{E}(\hat{\theta}) - \theta$. Note that this **statistical bias** is not the same thing as the inductive bias we have seen before.

In addition, again under certain conditions, statistics tells us how far off the estimate may be. The expected squared error of an estimator is

$$ESE(\hat{\theta}) = \mathbf{E}[(\hat{\theta} - \theta)^2].$$

For an unbiased estimator, this is equal to the variance of the estimator.

Example 6.5 The classic example in statistics is estimating a population mean μ using a sample mean $\bar{x}$. The sample mean is unbiased: $\mathbf{E}(\bar{x}) = \mu$. Its variance is $Var(\bar{x}) = \sigma^2/n$ with n the size of the sample and σ^2 the variance of the population.

The formulae from statistics are valid "under certain conditions". What are these conditions? There is one crucial assumption, namely that the sample on which the estimate is based has been taken randomly from the population, using the population's probability distribution.

The notion of randomness is important here. Consider an alternative case where we take 10 random subsets from the population. Some will have a mean that is higher than the population mean, others will have a mean that is lower. We expect about 5 subsets to have a higher mean, and 5 to have a lower mean.

Now what happens if we choose from these 10 subsets the subset with the lowest mean? Clearly, there is a very high probability that this mean is below the population mean. (The opposite would require that all 10 subsets have a mean above the population mean, which is very unlikely.) Obviously, for a subset chosen according to this procedure, the claim that "there is no reason to believe that the population mean is higher or lower" does not hold.

Generally, if we choose a subset not entirely randomly but according to some procedure that somehow takes into account the thing we are estimating, then the assumptions on which the statistical formulae are based are violated, and the formulae may not be valid.

Example 6.6 We continue the example about the mean. Suppose we have ten random samples S_i with sample means $\bar{x}_i$. If we have sampled the distribution correctly, then $\mathbf{E}(\bar{x}_1) = \ldots = \mathbf{E}(\bar{x}_{10}) = \mu$.

Now define $\bar{x}_M = \max\{\bar{x}_1, \ldots, \bar{x}_{10}\}$. $\bar{x}_M$ is also a sample mean, and yet $\mathbf{E}(\bar{x}_M) \geq \mu$ (with equality only on exceptional distributions). The fact that we did not choose the sample entirely randomly but chose it from 10 random samples according to some non-random procedure, destroys the property that it is unbiased.

A similar thing happens when we build a predictive model from some dataset. We use some kind of search or optimisation procedure that actively tries to yield a model with high accuracy on this dataset. If we would pick a *random* model and look at its accuracy on the dataset, this accuracy is a good estimate for the accuracy on the population, but if we take a model learned from this dataset, its accuracy on the set is probably an overestimate of its accuracy on the population.

Note that even if the learning procedure does not optimize accuracy but some other criterion, the randomness assumption is still violated and there is no reason to believe that the returned model's accuracy on the dataset is a good estimate for its population accuracy.

Conclusion: when a performance metric is not entirely independent (directly or indirectly) from the optimization process that goes on during learning, then an estimate of this metric based on the training set is not a good estimate: it may have a statistical bias. The training set accuracy of a model, for instance, will typically be higher than its true accuracy.

How can we estimate a performance metric, then? The key to correct estimation is to use a different dataset than the dataset on which the model was trained, more specifically, a dataset that is independent from the training set. To constrast it with the training set, such a dataset is often called the **test set**.

6.3.3 Using a fixed test set

An often used approach to learning and evaluating models is the following: given some dataset, set aside a part of it (e.g., 1/3) for use as a test set. Learn the model on the remaining 2/3 of the dataset, then evaluate the model on the test set. This procedure allows you to learn a model and estimate its predictive quality correctly.

There is one downside to this procedure. Generally, using more data to learn a model gives a better model. Setting aside part of the dataset as a test set reduces the data available for learning, and when only a limited set of data was available to start with, this may be a problem. A procedure called cross-validation may provide the solution then.

6.3.4 Cross-validation

Cross-validation is an often applied evaluation procedure in cases where limited data are available to learn from. It aims at providing a good estimate of predictive accuracy while using as much of the dataset as possible to learn from.

The problem is the following: given a dataset D, a learner L, and a model h learned from D using L, we want to estimate the predictive accuracy of h, but besides D no other data are available.

The cross-validation procedure solves this problem as follows. We partition D into a number of subsets, say, N subsets $D_1 \ldots D_N$. Next, we learn N separate models h_i, where each h_i is learned from the whole dataset except D_i using the same learner L, and its predictive accuracy p_i is evaluated on D_i. Let p be the mean of all the p_i. p is an estimator of the population accuracy of h.

Pseudocode for the cross-validation procedure is given in Figure 6.7.

The procedure is also called N**-fold cross-validation**, where N refers to the number of subsets in the partition.

A special case of cross-validation is **leave-one-out cross-validation**. Here, $N = |D|$. In other words, $|D|$ models are learned, and each time a single example is left out of D and used as a test example.

Cross-validation has the advantage that it makes optimal use of the data, in a sense. Each p_i is an unbiased estimate of the population accuracy of h_i, because it was obtained from a set of examples not used for learning h_i. The p_i may

procedure cross-validate(learner L, dataset D, integer N):
    partition the dataset D in N equal-sized subsets D_1, ..., D_N
    for i=1 to N:
        $T_i := D - D_i$
        learn a model h_i from T_i using L
        $p_i :=$ evaluate L's performance on D_i
    $p := \sum_{i=1}^{N} p_i/N$
    report p as the expected performance of any model learned from D using L

Figure 6.7: The cross-validation procedure.

be relatively inaccurate, because they are computed on small test sets D_i, but by computing the mean of these p_i a relatively accurate estimate for the average population accuracy of all the h_i is obtained. And because the training sets T_i are almost as large as the whole dataset D, the h_i can be expected to be almost as accurate as h.

Bias of cross-validation

Cross-validation is a popular method for estimating the accuracy of models: it yields a quite accurate estimate of predictive accuracy. However, this estimate is not entirely unbiased. In fact, we have a *pessimistic bias*: the true accuracy of h is probably a bit higher than its estimated accuracy.

There are two reasons for this. First, as we already mentioned, h is learned from the whole dataset, whereas each h_i is learned from a slightly smaller set, and therefore the population accuracies of the h_i, and the mean of these accuracies, can be expected to be slightly lower than the population accuracy of h.

The above effect can of course be minimised by using leave-one-out crossvalidation. There, the training sets T_i are only one element smaller than D: a negligible difference, unless D is very small.

But there is a second cause of pessimistic bias, which is somewhat less easy to see. It relates to the fact that in a cross-validation, the training sets and test sets are not truly independent, given a fixed dataset D. Independence would mean that the training set carries no information about the test set. But, given D, if we know T_i we can deduce D_i.

The T_i and D_i are not only dependent, they are complementary, and behave in opposite ways. For instance, the more positive instances T_i contains, the fewer positives D_i contains. The proportion of positives in T_i depends linearly on the proportion of positives D_i. In a regression setting, the mean target value in T_i and the mean target value in D_i have a negative correlation: the higher the mean of T_i, the lower the mean of D_i.

Exercise 6.8 Compute the linear relationship between the mean value of an attribute within T_i and within D_i, given a fixed dataset $D = D_i \cup T_i$ and $D_i \cap T_i = \emptyset$. Do the same for the proportion of instances belonging to some class in T_i and D_i.

The training sets are in a sense counter-representative for the test sets, and this leads to slightly worse results. More precisely, the population is on average more similar to the training set than the test set, and therefore the predictive accuracy on the population can be expected to be slightly higher than that on the test set.

Exercise 6.9 This exercise illustrates the pessimistic bias that is due to the counter-representativeness of the training sets for the test sets during a cross-validation.

Consider a population in which 50% of the instances is positive, and a data set D of 10 examples randomly sampled from that population, containing 5 positive and 5 negative examples. Consider a trivial learning algorithm L that just looks at which class occurs most frequently in its training set, and outputs a model that predicts that class for every instance. (Note that there are only two models possible: one that always predicts positive, and one that always predicts negative.)

a) Let h be a hypothesis learned with L from D. What is the population accuracy of h? (Note that you can only answer this question because you know the population. In practice, this is usually not the case.)

b) Simulate a leave-one-out cross-validation with the learner L on D. What is the estimated accuracy of h according to this cross-validation?

One way to alleviate the effect of this complementarity between training and test sets, is to perform **stratified cross-validation**. In stratified cross-validation, the D_i are chosen not entirely randomly, but in such a way that the class distribution in each D_i reflects as precisely as possible that in D. For instance, if D contains 40% positive examples, care is taken that the different D_i also have approximately 40% positives. This obviously increases the representativeness of the T_i for the D_i.

6.3.5 Confidence intervals

We already know that the estimated accuracy that we obtain from a dataset is not exactly the same as the true accuracy. If we estimate accuracy using an independent test set, then we know that the estimate is *on average* equal to the true accuracy (that is, its expected value is the same), but it deviates from it.

Theory from statistics tells us how much the estimate may deviate from the true value. Based on this knowledge, we can construct upper and lower bounds for the true accuracy based on our estimate. These are usually not hard bounds, but probabilistic bounds: we can be "almost sure" that the true value lies between these bound, but usually not 100% sure. The interval between the upper and lower bound is called a **confidence interval**. A "95% confidence interval", for instance, is an interval for which we have 95% confidence that the true value lies in it; more precisely, this means that the interval is constructed according to a procedure of which we know that if we always employ this same procedure, 95% of the intervals we construct will be "correct", in the sense that they contain the true value.

Let us look at the specific case of estimating predictive accuracy. Call the true accuracy of some model a. When we take a random set of n instances from the population, the model's accuracy on this set follows a binomial probability distribution, which can be approximated by a Gaussian distribution if n is large enough.[3] That Gaussian distribution has a mean of a and a standard deviation of $\sqrt{a(1-a)/n}$.

For a Gaussian distributed variable, the probability that it deviates at least k standard deviations from the mean is a function of k, and this function has been tabled. Special k values are those for which this probability reaches 10%, 5%, 2%, 1%, ... These are shown in Table 6.2.

Since our accuracy estimator has (approximately) a Gaussian distribution with mean a and standard deviation $\sqrt{a(1-a)/n}$, we know that

$$Pr(|\hat{a} - a| > k\sqrt{a(1-a)/n}) = \alpha$$

[3]Typically, $n > 30$ is considered large enough, but when a is close to 0 or 1, a larger n may be needed. A reasonable rule of thumb is $n > 4/a$ and $n > 4/(1-a)$.

α	0.10	0.05	0.02	0.01
k	1.64	1.96	2.33	2.58

Table 6.2: For selected values, the table lists k such that for a variable X with Gaussian distribution with mean μ and standard deviation σ, $Pr(|X-\mu| > k\sigma) = \alpha$.

where k and α can be filled in according to Table 6.2. For instance,

$$Pr(|\hat{a} - a| > 1.96\sqrt{a(1-a)/n}) = 0.05.$$

If the true accuracy of our model is $a = 0.8$ and $\hat{a}$ was computed on a sample of 100 elements, then

$$Pr(|\hat{a} - 0.8| > 1.96\sqrt{0.8(1-0.8)/100}) = 0.05$$

$$Pr(|\hat{a} - 0.8| > 0.0784) = 0.05$$

which means that there is a 95% probability that $\hat{a}$ deviates less than 0.0784 from 0.8, in other words, that $\hat{a}$ lies between 0.7216 and 0.8784.

So, this formula allows us to construct an interval in which $\hat{a}$ lies with high probability, if we know a and n. But of course, in practice, we want to do exactly the opposite: we know $\hat{a}$ (we compute it from our sample), but we don't know a. Can we use the same formula for constructing an interval for a?

Looking again at the formula

$$Pr(|\hat{a} - a| > k\sqrt{a(1-a)/n}) = \alpha,$$

it is clear that by filling in $\hat{a}$ and not a in the left hand side, the formula expresses an interval for a. One problem, though, is that a occurs also in the formula for the standard deviation. This problem is countered, in practice, by filling in $\hat{a}$ instead of a. The reasoning is that since $\hat{a}$ is usually close to a, this will cause only a small error for the width of the interval.

So, by cheating a little bit and filling in $\hat{a}$ for a in the formula for the standard deviation, we can construct, given $\hat{a}$ and n, a confidence interval for a of which we know that it is "correct" (contains the true a) in approximately $100(1-\alpha)$ percent of the cases (approximately, because $\hat{a}$ was filled in instead of a), where we can choose α as small as we want (though in practice α is usually chosen from the values mentioned in Table 6.2).

Exercise 6.10 Suppose we have evaluated a model on a test set of 400 examples and obtained a predictive accuracy of 0.9. Construct a 95% confidence interval for the true accuracy of this model.

6.4 Evaluating learners

From the application point of view, it is useful to evaluate a model. But from the point of view of machine learning, it is also useful to evaluate learning algorithms.

Some evaluation measures for learning algorithms are inherited from the measures for models. Given an evaluation measure m for hypotheses, let us write as $m(h)$ the score of a hypothesis h on m. Then, given a distribution over datasets, for any learner L we can define a measure m' as follows:

$$m'(L) = \mathbf{E}(m(L(S)))$$

where S is a dataset from the given distribution and $L(S)$ is the hypothesis learned by L from S. In other words: $m'(L)$ is the average m that hypotheses learned with

L have, if we learn them from certain kinds of datasets. Normally, we are interested in datasets that are drawn randomly from the population of instances, and often also in datasets of a specific size. These requirements can be incorporated in the definition of the distribution over S.

In practice, $m'(L)$ is estimated by running L k times on different datasets S_i, $i = 1 \ldots k$, which results in k hypotheses $h_i = L(S_i)$, then taking the mean of the $m(h_i)$.

In this way, measures such as accuracy, ESE, model size, computational complexity of making a prediction with the model, etc. can all be upgraded to the level of the learner. But there are a few additional measures that we can define for a learner:

- The computational complexity of learning. How long does it take, on average, to learn a model using L?
- The inductive bias of the learner: what kind of hypotheses can be learned by the learner?
- The variance of the learner. If we apply L to different datasets, how much do the returned models differ? To answer this question, we need a definition of the difference between models. This could be syntactic, but it can also take into account properties of the models such as their accuracy. The "variance w.r.t. m" of a learner L can be defined as

$$\mathbf{E}[(m(h) - m'(L))^2].$$

6.5 Comparing models and learners

Let us recap the previous sections. We have learned how we can express certain properties of models (such as their predictive accuracy), and how we can estimate values for these properties. We have also briefly discussed properties of learners, instead of single models.

The ultimate goal of the machine learning researcher or practitioner will often be the selection of the best (or most suitable) learner or predictive model from a number of options. This will involve *comparing* the models or learners. Similarly to our derivation of formulae for estimating the quality of a single model or learner, we can derive formulae for comparing them.

6.5.1 Comparing models

Assume model h_i has an estimated accuracy of $\hat{a}_i$ (where i is 1 or 2). If $\hat{a}_1 > \hat{a}_2$ (i.e., h_1 scores better than h_2 on our test set), can we deduce from this that $a_1 > a_2$ (i.e., h_1 is really more accurate on the population)?

Clearly this does not follow automatically: even if a_1 is slightly smaller than a_2, $\hat{a}_1$ might accidentally be a bit higher on our test set, and $\hat{a}_2$ a bit lower, so that $\hat{a}_1 > \hat{a}_2$ even though $a_1 < a_2$. But this is likely to happen only when a_1 and a_2 differ only a little bit.

Can we quantify this? Yes. The trick is to look at the variable $D = \hat{a}_1 - \hat{a}_2$. Statistics tells us that when two variables have a Gaussian distribution, their difference also has a Gaussian distribution, with as mean the difference of the means, and as variance the sum of the variances.[4]

This implies that we can write a confidence interval for the true difference $a_1 - a_2$ in a very similar way as we did for the population accuracy. The table

[4]This actually presupposes that the variables are independent.

with the relationship between k and α remains valid, since it is valid for all Gaussian distributions; only the formula for the mean and for the standard deviation differs. We have a variable $\hat{a}_1 - \hat{a}_2$ with mean $a_1 - a_2$ and standard deviation $\sqrt{[a_1(1-a_1)]/n_1 + [a_2(1-a_2)]/n_2}$, where n_1 and n_2 are the sizes of the test sets on which $\hat{a}_1$ and $\hat{a}_2$ were obtained. Our formula for the confidence interval thus becomes

$$Pr(|(\hat{a}_1 - \hat{a}_2) - (a_1 - a_2)| > k\sqrt{(a_1(1-a_1))/n_1 + (a_2(1-a_2))/n_2}) = \alpha$$

and again, in practice, we will fill in $\hat{a}_i$ instead of a_i in the formula for the standard deviation:

$$Pr(|(\hat{a}_1 - \hat{a}_2) - (a_1 - a_2)| > k\sqrt{(\hat{a}_1(1-\hat{a}_1))/n_1 + (\hat{a}_2(1-\hat{a}_2))/n_2}) = \alpha$$

even though this formula is not entirely correct.

Different models can now be compared by constructing a confidence interval for the difference in accuracy between them. If such an interval contains 0, the possibility cannot be excluded that both models have the same true accuracy. If the interval for $a_1 - a_2$ lies entirely to the right of 0, then it is very likely that $a_1 - a_2 > 0$, in other words, $a_1 > a_2$. If it lies entirely to the left of 0, we are pretty sure that $a_1 < a_2$.

(This approach is similar to "hypothesis testing" as it is known in statistics. Statisticians would say that the hypothesis that $a_1 = a_2$ has been rejected if the interval does not contain 0. Note that the term "hypothesis" in this context has a different meaning than how we use it throughout the course.)

Exercise 6.11 Suppose you have a model h_1 that scored an accuracy of 0.8 on a test set of 100 instances. A model presented in the literature was shown to have an accuracy of 0.75, measured on a set of 200 examples. Is there reason to believe that your model is better than the previous one?

The above formula for comparing two models can be used safely if the test sets on which two models have been evaluated are independent. If this is not the case, e.g., if the two models have been evaluated on the same test set, then $\hat{a}_1$ and $\hat{a}_2$ may not be independent. Then the conditions under which the used formulae are correct are violated, and the procedure may give unreliable results.

Exercise 6.12 *(some background in statistics required)* It is reasonable to assume that $\hat{a}_1$ and $\hat{a}_2$ are positively correlated: on an "easy" dataset both probably perform better than on a "difficult" dataset. Assuming that the correlation is indeed positive, how does this influence the confidence interval: will it be too wide, too narrow, or is it impossible to say?

Fortunately, a different method has been proposed for such cases: it is called McNemar's test. This method requires that h_1 and h_2 have been evaluated on the same set of instances. Moreover, instead of just using statistics about how often h_1 and h_2 were right on this set of instances, it needs statistics about the cases where h_1 was right and h_2 was wrong, and the other way around.

The following table represents the probability of any combination of h_1 and h_2 being correct or incorrect:

	h_1 correct	h_1 incorrect	
h_2 correct	b	c	$b+c$
h_2 incorrect	d	e	$d+e$
	$b+d$	$c+e$	1

so that b is, for a random instance, the probability that h_1 is correct and h_2 is correct, etc. The corresponding proportion of cases in the test set is denoted with

hats: $\hat{b}$ is the proportion of cases in the test set where both models were correct, etc.

Note that $a_1 = b + d$ (and $\hat{a}_1 = \hat{b} + \hat{d}$), and $a_2 = b + c$ ($\hat{a}_2 = \hat{b} + \hat{c}$).

The reasoning for this test goes as follows. If h_1 and h_2 have the same true accuracy, then whenever we take a single random instance and notice that h_1 and h_2 make a different prediction for it, both have an equal probability of being correct. (Indeed, if $a_1 = a_2$, since $a_1 = b + d$ and $a_2 = b + c$, it follows that $d = c$.)

Under this assumption, $\hat{c}$ and $\hat{d}$ should also be approximately equal. More precisely, $\hat{c}$ and $\hat{d}$ have a binomial distribution with mean c and d and standard deviation $\sqrt{c(1-c)/n}$ and $\sqrt{d(1-d)/n}$ (with n the size of the test set), which can be approximated by a Gaussian distribution with the same mean and standard deviation if n is large enough. Using this Gaussian approximation, and using the properties of the difference of Gaussian variables mentioned before,

$$Pr(|(\hat{c} - \hat{d}) - (c - d)| > k\sqrt{(c(1-c))/n + (d(1-d))/n}) = \alpha$$

with k and α again filled in according to Table 6.2.

This formula can be used to estimate a confidence interval for $c - d$. If the interval is entirely to the right of 0, this means it is very likely that $c > d$ and therefore $a_2 > a_1$. Similarly, if it is to the left of 0, it is most likely that $a_2 < a_1$. If the interval contains 0, this means that $\hat{c}$ and $\hat{d}$ are as close together as we would expect if $c = d$, so we cannot draw a clear conclusion.

Note that if we had only $\hat{a}_1$ and $\hat{a}_2$ at our disposal, we would not be able to perform McNemar's test; we need specifically $\hat{c}$ and $\hat{d}$ for this test.

The difference between McNemar's test and the previous one is that McNemar's test focuses on the cases where the two models differ. This is useful, because cases where they make the same prediction cannot tell us much on which one is better (they are both right or both wrong) but they may camouflage the truly relevant results, as the following exercise shows.

Exercise 6.13 Model h_1 has an accuracy of 0.8 on a test set of 100 examples, and h_2 has an accuracy of 0.7 on the same set. Is h_1 truly better than h_2, or does it only seem so? Test this in the case where (a) in 50 out of the 100 cases both h_1 and h_2 were correct; (b) in 70 out of the 100 cases both were correct. Try to predict the outcome of the test intuitively.

6.5.2 Comparing learners

The methods described above allow us to determine, given a set of models, which one has (probably) the highest accuracy. But sometimes this is not exactly what we want. Sometimes we may be interested in determining which *learning method* is best, for a particular application, rather than which specific model is best. In other words, we would like to know in advance, if we're going to learn a model h_1 using a learner L_1 and h_2 using a learner L_2, whether it is likely that h_1 will be better than h_2.

This question is very important for machine learning researchers, who, when developing new learning methods, want to find out whether their method works better than the existing ones, on average. "On average" here means: when averaged over many different datasets.

We know how to determine for one dataset whether h_1, the model learned by L_1, is better than h_2, learned by L_2; we even know how to estimate $a_1 - a_2$, the difference in accuracy of the models. But for different datasets we will get different models and therefore different accuracies. If on one dataset L_1 performed clearly better than L_2, that does not mean this will also be the case for any other dataset.

It is reasonable, then, to do the following: take a large number of datasets S_1, ..., S_k and corresponding test sets T_1, ..., T_k from some population of datasets. For each S_i, learn $h_{1,i}$ and $h_{2,i}$ by running L_1 and L_2 with S_i as input. Estimate $d_i = a_{1,i} - a_{2,i}$ by computing $\hat{d}_i = \hat{a}_{1,i} - \hat{a}_{2,i}$ (using the test set T_i).

We thus get a set of d_i, $i = 1 \ldots k$. Intuitively, if the large majority of these d_i are positive (negative), it seems reasonable to say that L_1 (L_2) performs better, on average. The mean $\bar{d} = \sum_{i=1}^{k} \hat{d}_i/k$ gives an estimate of the difference $d = a_1 - a_2$, and formulae for building confidence intervals for d have been proposed. These are based on the so-called t-distribution or Student-distribution. Defining the sample variance of $\hat{d}_i$ as

$$s_d^2 = \frac{\sum_{i=1}^{k}(\hat{d}_i - \bar{d})^2}{(k-1)},$$

an estimator for the variance of $\bar{d}$ is

$$s_{\bar{d}}^2 = s_d^2/k$$

and the probability that $\bar{d} - d$ exceeds some threshold is

$$Pr(|\bar{d} - d| > t_{k-1,\alpha} s_d) = \alpha.$$

The relationship between $t_{k-1,\alpha}$ and α is now NOT the one tabled in Table 6.2, but another one that is more complex. The constant $t_{k-1,\alpha}$ does not only depend on α but also on k, the number of comparisons performed. Formally, $t_{k-1,\alpha}$ is the threshold that a Student-distributed variable with $k-1$ degrees of freedom exceeds with a probability of α.

The above method has been applied many times in the machine learning literature, and is often referred to as the "paired t-tests" method for comparison. It has an important pitfall, however. The k runs are supposed to be independent. That implies that k independent datasets S_i and T_i must be available. Where do we find all these datasets?

A trick that is often used in practice is to generate from one dataset S many different variations. For instance, from S, the S_i are generated by taking a random sample of 2/3 of the elements from S and $T_i = S - S_i$. Such repeated 2/3 - 1/3 splitting of the same datase gives S_i and T_i that are not independent, however, and that raises the question whether the procedure is trustworthy.

In 1998, Thomas Dietterich showed experimentally that this is not the case [13] : the procedure can lead to large "type 1" errors, implying that there is a large risk of concluding that a learner is better than another when this is really not the case, and it should therefore never be used. Dietterich proposes the "5x2cv paired t-test" which is based on five two-fold cross-validations. This test is shown experimentally to exhibit low type 1 error. We do not discuss this in further detail. The main message of this discussion is that comparing learners using repeated 2/3-1/3 splits of the same dataset, a procedure that can be found in many papers, is unsafe and should not be used.

6.6 Exercises

Exercises marked with an asterisk are examples of questions that have been asked at previous exams.

Exercise 6.14 Consider the following situation. There is a team of policemen checking car drivers for alcohol intoxication. When the traffic is too busy for the police to perform alcohol tests on each driver, the following procedure is adopted.

The police stop all cars that pass by, and try to judge on sight (or smell) whether a driver may be intoxicated. When the drivers seems sober, he or she may continue; but when the police officer is suspicious, he or she submits the driver to an alcohol test.

Assume 3% of the drivers have actually drunk alcohol. Consider two different police officers, Ed and Fred. Ed is in a rather lazy mood tonight: he just lets each driver pass. Fred submits 15% of the drivers to an alcohol test; as it turns out, 1 in 5 of these are indeed intoxicated.

Call the officer's decision correct when either he submits a drunk driver to an alcohol test, or he lets a sober driver continue without testing. Which of the two cops, Ed or Fred, is then the most accurate?

Exercise 6.15 Assume a classifier has an accuracy of 90% on a test set containing 50% positive and 50% negative examples. Now assume this same classifier is used in an environment where 70% of the instances are positive (without other properties of the distribution changing). Will it still have 90% accuracy?

Exercise 6.16 Draw an example ROC convex hull in a ROC diagram. Take two classifiers A and B that are on adjacent points on the convex hull (that is, A and B are connected by a straight line). Now consider a third classifier C which makes predictions as follows: with 50% probability it copies A's prediction, and with 50% probability it copies B's prediction. Indicate C's position on the ROC diagram. Is it also on the convex hull?

Exercise 6.17 Another way to see that classifiers below the convex hull cannot be optimal is the following. We know that a classifier A can never be better than a single classifier B if B is to the upper left of A. Show that for any point A below the convex hull of a set of classifiers S, you can always either find a classifier in S, or construct a new classifier based on the classifiers in S, such that the new classifier is to the upper left of A. (If you find this difficult, first solve exercise 6.16.)

Exercise 6.18 Suppose the true accuracy of a model is 0.9, and on a test set of 100 examples an accuracy of 0.91 is obtained. In the formula for a 95% confidence interval, the experimenter will fill in 0.91, because she does not know that the true accuracy is 0.9. What is the width of the "correct" interval, and of the computed interval? What do you conclude with respect to the interpretation of the percentage that we associate with confidence intervals?

Exercise 6.19* If A has higher accuracy than B, is it possible that B is on the convex hull in a ROC diagram and A is not? Explain your answer.

Exercise 6.20* Is the following statement correct or not: "if A is strictly below the convex hull in a ROC diagram, there must be at least one classifier on the convex hull with strictly higher predictive accuracy than A."

Exercise 6.21* Is it possible that for a given set of classifiers, the classifier with the highest accuracy is not on the ROC convex hull of the set?

Exercise 6.22* Classifier A has 90% accuracy on a training set with 50% positives. B has 80% accuracy on a training set with 70% positives. Show where A and B lie in a ROC diagram. (If you can't obtain a single point, indicate the area.)

Exercise 6.23* One hypothesis A is learnt from data with many positives and few negatives; an other hypothesis B from data with few positives and many negatives. Both are learnt by learners that try to achieve high accuracy. Indicate on a ROC diagram where you expect A and B to be. Explain your reasoning.

Exercise 6.24* A classifier A has TP=0 and FP=1. Interpret this. Is it possible to derive a better classifier from A in some principled way?

Exercise 6.25* Plot the hypothesis represented by the following decision tree on a ROC diagram. x/y in a leaf means that y cases are covered by the leaf, of which x are correctly predicted.

```
Sky=?
+--Sunny: Temperature=?
          +--warm: NoTennis (8/10)
          +--medium: Tennis (7/8)
          +--cold: Tennis (4/7)
+--Cloudy:Humidity=?
          +--high: NoTennis (5/5)
          +--low: Tennis (9/13)
+--Rainy: NoTennis (7/7)
```

Chapter 7

Instance based learning

Instance based learning could be considered the simplest possible type of learning. In the most basic version of this approach, there is in a sense no learning at all: the "hypothesis" that is learned is simply the dataset itself. When a prediction is to be made, the new instance is compared with the instances stored in the dataset. If an instance is found with the same (or a very similar) description as the new instance, it is considered likely that the target values are also the same (or very similar). Generally, a prediction is made for the new instance that is inspired by the most similar instances found in the dataset.

This type of methods works surprisingly well in many cases, but can also fail miserably. Some sophistication is typically added to these methods, leading in the end to learning methods that have a complexity not unlike that of other methods.

In this chapter we have a closer look at nearest neighbor methods, starting with the most basic version, and gradually extending it with improvements.

7.1 The basic idea: nearest neighbors

The **nearest neighbor** classification method is based on a simple idea: when you need to classify a new instance, look at the instances you've seen before, select the instance that is most similar to the new one (the "nearest neighbor"), and give the new instance the same class as that instance.

This makes sense. Yet, there is a certain risk that the one neighbor that happens to be nearest is a bit untypical, or that it is noisy (i.e., contains a mistake in its description or class value). In that case this neighbor is perhaps not so relevant. For instance, imagine that there are n instances that are very similar to the new instance, among which $n-1$ are positive, but the one negative example happens to be the nearest one. Intuitively it is not very appealing to rely entirely on that one rather exceptional case.

A somewhat more robust method for prediction is: for some value k, take the k nearest neighbors of the new instance, and predict the class that is most common among these k neighbors. This method is called the **k-nearest neighbors** method. The algorithm is shown in Figure 7.1. Obviously, when $k = 1$, this is the standard nearest neighbor method.

Note that the algorithm discussed here is a prediction algorithm, not a learning algorithm. If we look at the other machine learning approaches, we typically can distinguish a learning algorithm, which involves learning a hypothesis h from a training set T (e.g., induction of a decision tree), and a (often trivial) prediction algorithm, showing how to use h to make a prediction for a new instance (e.g., applying a decision tree to a new instance to make a prediction). In the nearest

function kNN(k, T, $\mathbf{x}$) **returns** class:
    let $S = \{(\mathbf{x}_1, y_i), \ldots, (\mathbf{x}_k, y_i)\}$ be the set of the k nearest neighbors of $\mathbf{x}$ in T
    let $f(c) = |\{(\mathbf{x}_i, y_i) \in S : y_i = c\}|$
    return $\arg\max_c f(c)$

Figure 7.1: The basic k-nearest neighbors classifier.

function kNN(k, T, $\mathbf{x}$) **returns** real:
    let $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_k, y_k)\}$ be the set of the k nearest neighbors of $\mathbf{x}$ in T
    $\hat{y} := \sum_{(\mathbf{x}_i, y_i) \in S} y_i$
    if target attribute is discrete **then** round $\hat{y}$ to nearest target value
    return $\hat{y}$

Figure 7.2: The k-nearest neighbors method for numerical predictions.

neighbor context, the learning algorithm is trivial, even non-existent: there is no other "hypothesis" than the training set T itself. The prediction algorithm, applying this hypothesis to make a prediction for a new instance, is the one that is less trivial here. Nearest neighbor methods are therefore called "lazy learners": they postpone all the work until a prediction is requested. We will discuss this issue in a bit more detail later on.

If we have only two classes, the prediction formula is more easily written using a numerical representation for the classes. With classes $+1$ and -1, we can compute

$$\hat{y} = \sum_{\mathbf{x}_i \in NN} y_i$$

This $\hat{y}$ is the "average class" among the nearest neighbors; it has a value between $+1$ and -1. The actual prediction will be $+1$ if $\hat{y} \geq 0$ and -1 otherwise.

If the target variable is continuous, we can use the same formula but without rounding off the prediction to $+1$ or -1. This gives us **nearest neighbors based regression**. If the target variable is numerical but discrete (e.g., 1,2,3,4,5), the prediction can be rounded to the nearest number in the range.

The modified kNN algorithm is shown in Figure 7.2.

7.1.1 Weighting the examples

Using the k nearest neighbors to make a prediction, with a predetermined k, creates in a sense an arbitrary threshold: all examples among the k nearest ones are considered important for the prediction (and equally important), and all the other examples are ignored completely. In some cases this may not make much sense; Figure 7.3 illustrates this with an example.

Given the motivation for nearest neighbor methods, namely the idea that similar instances are likely to have a similar class, it would make sense to simply give closer examples a higher influence, whether they are among the k nearest neighbors or not. This is easily achieved by changing the average into a weighted average, and including all examples in T (rather than only the k nearest ones):

$$\hat{y} = \frac{\sum_{\mathbf{x}_i \in T} w_i y_i}{\sum_{\mathbf{x}_i \in T} w_i} \tag{7.1}$$

where w_j expresses the similarity of $\mathbf{x}_j$ to $\mathbf{x}$:

$$w_j = sim(\mathbf{x}, \mathbf{x}_j)$$

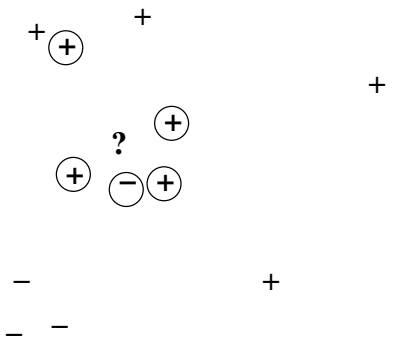

Figure 7.3: An example where considering the k nearest neighbors as relevant for the prediction, and all the others as completely irrelevant, is counterintuitive. With $k = 5$, four very near neigbors and one that seems much less relevant will be taken into account for the prediction.

Similarity, in this context, is to be interpreted as "relevance for the prediction of the target attribute". The similarity function can be the same as the one that was used to decide which are the nearest neighbors, but sometimes it is more sophisticated; for instance, it may be desirable to let this similarity function go to zero more quickly than the "natural" similarity measure. We will return to this later, when we discuss similarity and distance functions.

Note that there is now no notion of k nearest neighbors anymore: we compute the weighted average over the whole dataset. Faraway examples are automatically given a small influence anyway. Nevertheless, it is still possible to combine this weighted average with selection of k nearest neighbors, for instance for efficiency reasons, or because the number of "faraway" instance might be so large that all together they do have an influence that is greater than desired (we will come back to this). To that aim, one may prefer to still use S, the set of k nearest neighbors, instead of T (the whole training set) in Equation 7.1.

7.1.2 Typical decision surfaces of nearest neighbor methods

While nearest neighbor classification methods do not explicitly build a model, other than the training set itself, they still represent a classifier defined over the whole input space, and we can try to visualize this classifier by drawing decision surfaces. Such a visualisation is easiest for a two-dimensional numerical input space. In our visualization, we will further assume that "similarity" is related to Euclidean distance: the closer two points are to each other (i.e., the smaller the distance between them), the more similar they are.

So consider a 2-D input space with a number of positive and negative examples, as illustrated in Figure 7.4. For the one-nearest-neighbor method, all points for which the nearest neighbor is some example $\mathbf{x}$ will be assigned the same class, namely the class of $\mathbf{x}$ (denoted $c(\mathbf{x})$). If the dataset consists of only two examples $\mathbf{x}_1$ and $\mathbf{x}_2$, then the boundary between points nearer to $\mathbf{x}_1$ and points nearer to $\mathbf{x}_2$ is a straight line running right in the middle between the two examples. Similarly, if we want to find the area of all the points that are nearer to some example $\mathbf{x}_i$ than to any other example $\mathbf{x}_j$, we construct this middle line between $\mathbf{x}_i$ and $\mathbf{x}_j$ for each $\mathbf{x}_j$. All these lines together bound the area of points that are closer to $\mathbf{x}_i$ than to any other training example, and that therefore have the class $c(\mathbf{x}_i)$. Call this the "area of influence" of $\mathbf{x}_i$. We can easily construct this area of influence for each

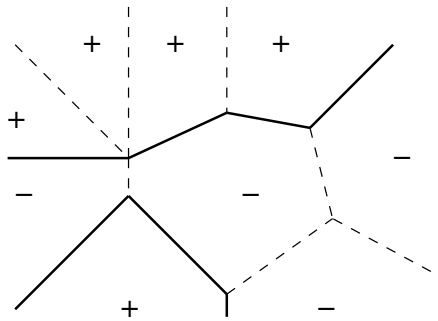

Figure 7.4: The Voronoi diagram of a dataset. The lines separate areas of influence of individual examples. Dashed lines separate areas of the same class, solid lines separate areas of different class. The solid lines are decision boundaries for the nearest neighbor classifier.

example. The result is called a **Voronoi diagram**. Given a set of points $\mathbf{x}_i$ in a 2-D space, the Voronoi diagram shows a number of areas A_i having the property that each point in A_i is closer to $\mathbf{x}_i$ than to any other $\mathbf{x}_j$.

The decision surface, i.e., the lines separating areas of different classes, are then given by those boundaries in the Voronoi diagram that separate areas A_i for which the $\mathbf{x}_i$ have a different class.

Figure 7.4 illustrates that the nearest neighbor method tends to construct decision surfaces that consist of short straight lines but that are globally not very straight; they typically have a rugged, zig-zag shape. This distinguishes nearest neighbor methods from most other methods, such as decision trees or rule sets.

These diagrams are also useful to illustrate another aspect of nearest neighbor methods. Other methods construct a compact model that is in a sense an *approximation* of the dataset. Certain details about the dataset may be lost when we look at the model. The model created by a nearest neighbor method contains all the detail that was in the original dataset.

Consider Figure 7.5, which shows a dataset with a number of "isolated" examples: examples that are surrounded by examples of a different class. The nearest neighbor method constructs "islands" around these examples, hypothesizing that each instance in that island has the same class as the isolated example that created it.

Most other learning methods construct a model from the data that abstracts away the occurrence of a single example with a different class. Because of that, those method are less sensitive to noise in the data.

The k-nearest neighbor method, by using the majority class among the k nearest neighbors for its prediction, has the same effect of abstracting away some details. With increasing k, fewer of those "islands" of examples with a different classes will remain. Clearly, when k becomes very large, too many details may be abstracted away. When $k = N$ with N the number of training examples, the method will simply always predict the majority class in the training set.

7.2 Eager versus lazy learning

As mentioned before, instance based learners deviate from other learning methods in the sense that the "model" that is learned is trivial (it is the dataset itself); the non-trivial work happens during the prediction phase.

More specifically, when an instance based learner is given a set of data, it does

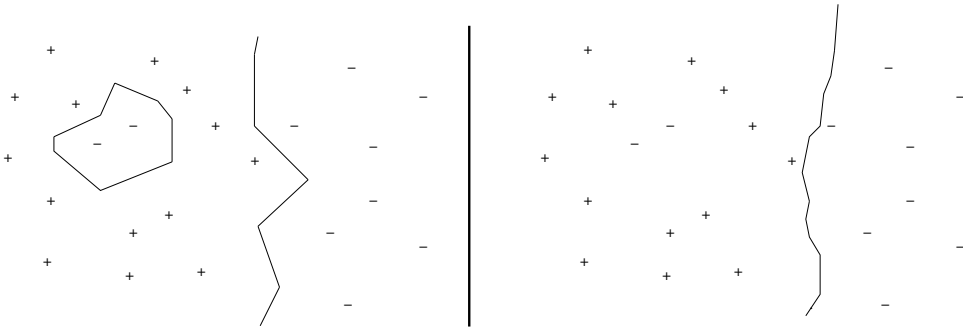

Figure 7.5: Decision surfaces for a dataset with "isolated" cases, for $k = 1$ and $k = 5$. Note that the "island" of negatives constructed for $k = 1$ disappears when $k = 5$.

not do anything except store the data. It essentially defers all computation until a question is asked, i.e., a prediction for a new instance needs to be made. At that time, the learning system starts making computations.

Instance based learners are called **lazy learners** because of this property, as opposed to **eager learners**, which start performing computations as soon as they see a dataset.

Generally, a learning system is a system that has a well-defined task to perform, and becomes better at performing this task with experience. The task of a learner is often defined in one of the following two ways. Given a population U of instances, the task can be to

1. construct a (interpretable) model of the population
2. predict target values for new instances from the population

and the "experience", or input to the learner, is a training set T.

In many cases, the first task subsumes the second: if we have a model of the population, this model can often be used to make predictions for new cases.

In the context of predictive learning, the first task is often referred to as **inductive learning**, and the second as **transductive learning**. We define the tasks of induction and transduction as follows.

Task description: Induction
Given:

- an unknown population $Z = X \times Y$ consisting of couples $(\mathbf{x}, y)$;
- a training set $T \subseteq Z$;

Find: a function $f : X \to y$ such that $\forall(\mathbf{x}, y) \in Z : f(\mathbf{x}) = y$.

Task description: Transduction
Given:

- a unknown population Z consisting of couples $(\mathbf{x}, y)$;
- a training set $T \subseteq Z$;
- an instance $\mathbf{x}$;

Find: the value $y \in Y$ such that $(\mathbf{x}, y) \in Z$.

Thus, the goal of induction is to find a single function f that allows us to make predictions for any new cases, whereas the task of transduction is simply to find a prediction for (a set of) new cases.

Clearly, whenever we solve the first task, we have solved the second task. Further, if we allow the function f to be represented as the training set itself, then the inductive setting is de facto reduced to the transductive one (that is, the only non-trivial work that is being done is related to the actual prediction).

There is in a sense a spectrum between eager and lazy learning: an eager learner invests effort in constructing f in such a way that it will be easy to apply it to new instances; a lazy learner uses a representation for f that is very easy to construct but has to invest effort in applying f to new instances afterwards. The extremes of this spectrum correspond well to the tasks of inductive and transductive learning.

The advantages and disadvantages of eager and lazy learning can be summarized as follows:

For eager learners:

- the learning phase is slow
- prediction is fast
- computations take into account only T, not $\mathbf{x}$; i.e., a single model is learned that is supposed to work well on all data

For lazy learners:

- the learning phase is fast
- prediction is slow
- computations take into account not only T but also $\mathbf{x}$; computations can be more sophisticated because they have access to this extra information

Lazy learners have in a sense some extra flexibility, because given a $\mathbf{x}$, they can construct a model $f_{\mathbf{x}}$ from T and $\mathbf{x}$, and use that f to make a prediction for $\mathbf{x}$. For a different instance $\mathbf{x}'$, they would construct a different function $f_{\mathbf{x}'}$ that is supposed to work well in the neighborhood of $\mathbf{x}'$. Thus, lazy learners could be said to construct **local models**, as opposed to eager learners, which construct **global models**.

This may seem a bit abstract at this point, but the section on Locally Weighted Regression provides a good illustration of what it means in practice.

A link with semi-supervised learning

This discussion on the difference between transduction and induction brings us to the setting of **semi-supervised learning**. In semi-supervised learning, the learner gets as input a set of labelled examples T_l as well as a set of unlabeled examples T_u, and the goal is to construct from this input a function f that allows us to predict the labels (target values) of new cases. While it may seem a bit counterintuitive, the set T_u can help to improve the learning process, even though no information is available about the labels of the examples in T_u. There is additional information on the distribution of the examples, and that may help the learner to construct an accurate model. Figure 7.6 illustrates this.

Semi-supervised learning is somewhat similar to the transductive setting, in the sense that information on unlabeled instances is taken into account during the learning. Nevertheless, we can apply semi-supervised learning in an inductive setting just as well as in a transductive setting. This gives us the following task descriptions.

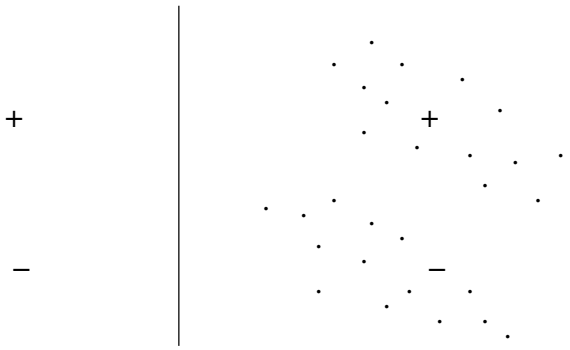

Figure 7.6: Semisupervised learning: exploiting the information in a set of unlabeled examples to construct a more accurate model. Left: a single positive and a single negative example. Right: the same positive and negative example, now in a context of many unlabeled examples. Construct what you think is the most natural separator between the positives and negatives in both cases.

Task description: Semi-supervised Induction
Given:

- a unknown population $Z = X \times Y$ consisting of couples $(\mathbf{x}, y)$;
- a set of labeled examples $T_l \subseteq Z$;
- a set of unlabeled examples $T_u \subseteq X$;

Find: a function $f : X \to y$ such that $\forall(\mathbf{x}, y) \in Z : f(\mathbf{x}) = y$.

Task description: Semi-supervised Transduction
Given:

- a unknown population Z consisting of couples $(\mathbf{x}, y)$;
- a set of labeled examples $T_l \subseteq Z$;
- a set of unlabeled examples $T_u \subseteq X$;
- a set of test instances $Te \subseteq X$

Find: for each $\mathbf{x} \in Te$ the value $y \in Y$ such that $(\mathbf{x}, y) \in Z$.

Note that in the transductive setting, the learner will typically include the test instances Te in the set of unlabeled examples T_u; there is no reason not to do that. We still distinguish Te from T_u because Te is the set of instances for which the target values are asked, whereas we may not be interested in the labels of examples in T_u.

7.3 Similarity measures

Since nearest neighbor prediction is based on the notion of similarity, it is useful to take a closer look at similarity measures here.

Assuming that there is some intuitive notion of when two instances are similar, we call a measure a **similarity measure** if it increases with similarity, and a

dissimilarity measure if it decreases with similarity.

Dissimilarity is often expressed using a distance metric: the greater the distance, the smaller the similarity between two instances. However, similarity or dissimilarity are in fact more flexible measures than distances. A distance metric d has to fulfill a number of properties:

$d(\mathbf{x}, \mathbf{x}) = 0$
$d(\mathbf{x}, \mathbf{y}) > 0$ if $\mathbf{x} \neq \mathbf{y}$
$d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$ (symmetry)
$d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{z}, \mathbf{y})$ (triangle inequality)

A (dis)similarity measure need not necessarily be symmetric or fulfill the triangle inequality. Nevertheless, it is very common to use a distance metric when applying nearest neighbor methods. We will therefore mostly focus on distances in the following.

7.3.1 Numerical input spaces

When examples are described as vectors in a Euclidean space, the Euclidean distance is often the most obvious choice of a distance measure, and then the similarity measure can for instance be one of the following:

$$sim(\mathbf{x}, \mathbf{y}) = 1/d(\mathbf{x}, \mathbf{y})$$

$$sim(\mathbf{x}, \mathbf{y}) = 1/d(\mathbf{x}, \mathbf{y})^2$$

or

$$sim(\mathbf{x}, \mathbf{y}) = \exp(-d(\mathbf{x}, \mathbf{y})^2/\sigma^2)$$

with σ some parameter.

In the unweighted version of the nearest neigbor method, all these similarity measures are equivalent. This is because to find the k nearest neighbors, the actual similarities are not important, only the ranking of the examples (among two examples, which one is nearest?) is important. For any $\mathbf{x}, \mathbf{x}_i, \mathbf{x}_j$ the test $sim(\mathbf{x}, \mathbf{x}_i) < sim(\mathbf{x}, \mathbf{x}_j)$ gives the same result for all three the above similarity measures, because sim is monotonically decreasing in d: $sim(\mathbf{x}, \mathbf{x}_1) < sim(\mathbf{x}, \mathbf{x}_2) \Leftrightarrow d(\mathbf{x}, \mathbf{x}_1) > d(\mathbf{x}, \mathbf{x}_2)$.

For the weighted version of nearest neighbors, it does make a difference which similarity measure is used. The first two, $1/d(\mathbf{x}, \mathbf{y})$ and $1/d(\mathbf{x}, \mathbf{y})^2$, can give a very high weight to examples that are very close, since they go to infinity if $d(\mathbf{x}, \mathbf{y})$ goes to zero. Thus, if the instance for which a prediction has to be made happens to be very close to a single training example, that one training example can dominate the prediction. The third measure, $sim(\mathbf{x}, \mathbf{y}) = \exp(-d(\mathbf{x}, \mathbf{y})^2/\sigma^2)$, does not have this property.

At the other end of the spectrum, we can look at the influence of faraway examples. All similarity measures asymptotically go to zero with increasing $d(\mathbf{x}, \mathbf{y})$, but the first similarity measure approaches zero more slowly than the second one, which in turn is slower than the third one. As it turns out, with the first approach, the similarity goes down so slowly with distance that faraway examples may actually dominate the prediction, for the simple reason that there are more of them. This effect is less severe with the second approach, and much less with the third one.

We can easily illustrate how problematic slow convergence to zero can be. Assume instances have a uniform distribution in a 2-dimensional space. For a random point in this space, there will typically be more examples at high distance than at low distance. Figure 7.7 illustrates why this is. With a uniform distribution, the

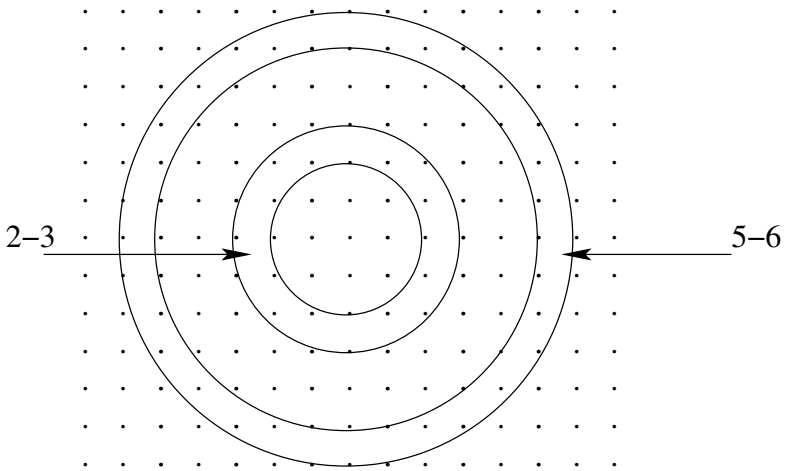

Figure 7.7: When instances are uniformly distributed in the instance space, there are more instances at a distance of approximately 5.5 (i.e. between 5 and 6), than at a distance of approximately 2.5 (between 2 and 3). In the picture, we can count about 16 dots in the 2-3 area, and about 30 in the 5-6 area.

number of instances expected to be in some area is proportional to the size of that area (an area twice as big on average contains twice as many instances). So the number of examples at a distance between 2 and 3 from some instance $\mathbf{x}$ is, in this two-dimensional space, proportional to the area between two concentric circles with radius 2 and 3, which is $3^2\pi - 2^2\pi = 5\pi$. The number of examples at a distance between 5 and 6 is proportional to $6^2\pi - 5^2\pi = 11\pi$. Hence, we can expect more than twice as many examples at a distance between 5 and 6 than between 2 and 3.

Generally, the number of instances at a distance between r and $r + \epsilon$ is proportional to $(r + \epsilon)^2\pi - r^2\pi = (r^2 + 2r\epsilon + \epsilon^2)\pi - r^2\pi = (2r\epsilon + \epsilon^2)\pi$, which for a fixed and small ϵ is proportional to r. So in a two-dimensional input space with instance distributed uniformly, we have the situation that the number of instances at (approximately) distance r is proportional to r. With the first similarity metric, these instances have an influence that is inversely proportional to r. Hence, *the first similarity metric globally does not give more influence to nearer examples than to faraway examples*. An individual example that is near has a higher influence, but because there are more examples at larger distance, these all together compensate for the nearer example. For each example with a weight of w, there are 10 examples with a weight of $w/10$, and all together these have just as much weight.

This discussion is a simple version of a more general reasoning, backed up by mathematics, that shows that especially in higher-dimensional spaces, the faraway examples form an overwhelming majority. More precisely, in a D-dimensional space, and if the examples are distributed uniformly, the number of examples at distance r is proportional to r^{D-1}. So in a three-dimensional space, for each example at distance x there are 100 examples at distance $10x$; in a four-dimensional space it would be 1000; etc.

Clearly the first two similarity measures will allow faraway examples to dominate the prediction, which goes totally against the principle of making a prediction based on the nearest neigbors. This is to some extent an argument for artificially limiting the examples that influence the prediction to the k nearest neighbors (although among these k nearest instances, the same problem remains).

The third similarity metric does not have this problem: whatever D is, the number of examples at distance r increases less fast with r than the influence per

example decreases, at least when r is large. Mathematically, for any D it holds that

$$\lim_{r \to \infty} r^{D-1} \cdot \exp(-r^2/\sigma^2) = 0,$$

i.e., the summed up influence of all examples at distance r asymptotically approaches zero, whatever D is.

This asymptotic approach to zero only kicks in when r is large enough. How large that is, depends on the dimensionality D, and on the parameter σ: the smaller σ is, the closer an instance must be to have a noticeable influence on the prediction.

The conclusion of this discussion is: when using a distance-weighted version of nearest neighbors, best use a similarity measure that goes to zero exponentially fast.

7.3.2 Non-numerical input spaces

In non-numerical input spaces, i.e., spaces where the input examples have nominal attributes (possibly mixed with numerical attributes), or have some more complex structure (e.g., a graph structure), choosing a suitable distance metric or similarity measure is less obvious.

In a vector space, the Euclidean distance is defined as follows:

$$d(\mathbf{x}, \mathbf{y})^2 = \sum_i (x_i - y_i)^2$$

We can see this formula as expressing that the squared Euclidean distance is the sum of squared distances of the components, and rewrite the formula as

$$d(\mathbf{x}, \mathbf{y})^2 = \sum_i d_i(x_i, y_i)^2$$

where the distance function d_i reflects the distance in one particular dimension; for numerical components it is simply the difference $|x_i - y_i|$.

By rewriting the distance metric in terms of component distance functions d_i, we can handle any example space where examples are described as a tuple of attribute values. For attributes of different types, a different distance function can be defined.

For nominal attributes, an often used similarity measure is the identity function: the similarity of two values is 1 if they are equal, and 0 otherwise. This translates to a distance function

$$d(x, y) = \delta(x, y)$$

where $\delta(x, y) = 0$ if $x = y$ and 1 otherwise.

It is not very clear how to best handle ordinal attributes. Ordinal attributes have values that are ordered but are not numerical (e.g., values *excellent*, *good*, *average*, *bad*) and hence the difference between the values is not defined. Because the values are ordered, there is a notion of small and large differences: it makes sense to say that the difference between *excellent* and *good* is smaller than that between *excellent* and *average*. Treating such attributes as nominal ignores the ordering of the attributes, and considers each difference as equally important. On the other hand, we could convert the values to a numerical scale, translating k ordinal values to the numbers $1, 2, \ldots, k$, but that introduces an assumption that the difference between *excellent* and *average*, for instance, is "twice" that between *excellent* and *good*, which is not a very meaningful statement.

A more general way to express the difference between nominal or ordinal values is to assign a weight to each couple of values (x, y) that expresses how different x is from y. This gives us a weight matrix M. These weights are often also called costs, in the sense that they can be interpreting as reflecting the cost of predicting value

x instead of value y, if we would be predicting this attribute. M is called a **cost matrix**.

For three values a, b, c of a nominal attribute, the δ distance function corresponds to the cost matrix

	a	b	c
a	0	1	1
b	1	0	1
c	1	1	0

and if the values are ordered, $a < b < c$, then mapping them onto $1, 2, 3$ corresponds to using the matrix

	a	b	c
a	0	1	2
b	1	0	1
c	2	1	0

Using cost matrices, however, we are not limited to the above options. We can express that the difference between between two values depends on what these values are, even if the values are not ordered; also asymmetric "distances" (which strictly speaking should not be called distance metrics anymore) are possible. For instance

	a	b	c
a	0	0.2	2
b	0.1	0	1
c	2	1	0

reflects that a and b are quite similar but c is quite dissimilar to b and very dissimilar to a. It also states that b is more similar to a than a is to b, which may seem strange if we view them as distances, but which can in some cases (which we do not consider here) be useful.

7.3.3 The Curse of Dimensionality

The discussion about distance based methods in numerical vector spaces, a few sections earlier, illustrates a problem that is more widely known as the **curse of dimensionality**. Learning becomes more difficult as the dimensionality of the input space increases. This is visible in many different ways. In the case of instance-based learning, the fact that the majority of the neighbors tend to be far away (and the higher the dimensionality is, the larger this majority is) is one way in which high dimensionality makes learning more difficult.

Another way in which this can be seen is the following. Assume that the class of an instance is determined by just a few attributes, and the other attributes are irrelevant for the classification. Distances or similarities are computed based on all attributes together. In a high-dimensional space, the irrelevant attributes can easily dominate the relevant attributes. It can easily happen that two instances are quite similar, but differ in just those few attributes that matter. They would get a similar prediction, when in fact it should be different.

Example 7.1 Consider a ten-dimensional space where all attributes x_i $(i = 1-10)$ are boolean, and assume that the target concept is $x_1 \wedge x_2$, i.e., only the first two attributes are relevant. Assume that we have an instance $\mathbf{x}$ that is compared with two training examples $\mathbf{x}_1$ and $\mathbf{x}_2$:

$$\mathbf{x} = (T, T, \ldots)$$
$$\mathbf{x}_1 = (T, T, \ldots)$$
$$\mathbf{x}_2 = (T, F, \ldots)$$

Since the class of $\mathbf{x}$ depends only on the first two attributes, it must have the same classification as $\mathbf{x}_1$. However, a 1-nearest neighbor method would assign it the class of $\mathbf{x}_2$ as soon as, among the remaining 8 attributes, $\mathbf{x}_2$ agrees with $\mathbf{x}$ on at least two more attributes than $\mathbf{x}_1$.

If we assume that the remaining attributes are totally random, then the probability that this happens turns out to be approximately 21%. In a 6-dimensional space it would have been some 10%, in an 8-dimensional space approximately 17%.

The example illustrates that with higher dimensionality, there is an increasing probability that the nearest neighbor classifier is confused by irrelevant attributes. For this reason it makes sense to try to extend nearest neighbor methods with some means of "tuning" their distance metric to the task at hand, more specifically, to make them adapt the distance metric so that it focuses on relevant attributes. This is the subject of the next section.

7.4 Rescaling the dimensions

There are mainly two reasons why we may want to rescale dimensions. The first one is that the natural ranges of different attributes may be very different, giving some attributes unduly more importance than others. Rescaling is then useful to give different attributes comparable chances of influencing the outcome. The second reason is related to what we discussed above: we may want to artificially reduce the influence of irrelevant attributes.

In the following we discuss both issues and how to handle them.

7.4.1 Incomparable attributes

Numerical attributes may have largely different scales; for instance, attribute A might be a number from 0 to 1, and B a number from 0 to 100. From the point of view of distance-based learning, it seems reasonable to say that a value of 80 for B is close to 90 in exactly the same way that a value of 0.8 for A is close to 0.9. However, this principle is not reflected by the Euclidean distance measure. This measure will consider two values close if their difference is small in absolute terms. This is not always what we want, as the following example illustrates.

Example 7.2 Consider the following three 2-dimensional datapoints, which describe the age (in years) and height (in meters) of persons.

Name	Age	Height
A	37	1.87
B	35	1.88
C	37	1.57

The Euclidean distances between the individuals are:

$$d(A,B) = \sqrt{4 + 0.0001} = 2.00002$$
$$d(A,C) = \sqrt{0 + 0.09} = 0.3$$
$$d(B,C) = \sqrt{4 + 0.0963} = 2.02$$

According to these distances, we can conclude that A and C are very similar persons, whereas B is quite different from these two.

It is clear, however, that this result is mainly due to the fact that any difference in the Age attribute automatically leads to a large distance, while the Height attribute has only a small influence on the distance. This is counterintuitive. Intuitively, we would consider two people with an age difference of two years "almost equally

old", while two people whose height differs with 0.3m would not be considered "almost equally tall"; but the Euclidean distance measure considers a difference of 2 (whether it is years or meters) much greater than a difference of 0.3.

This is purely a problem of representation. Interestingly, if we would indicate people's heights in centimeters instead of meters, we would get:

Name	Age	Height
A	37	187
B	35	188
C	37	157

and the distances become

$$d(A,B) = \sqrt{4+1} = 2.24$$
$$d(A,C) = \sqrt{0+900} = 30$$
$$d(B,C) = \sqrt{4+963} = 31.10$$

Now we notice that A and B are considered very similar, while C is considered quite different from the other two. This coincides much better with our intuition.

It is highly undesirable that a machine learning system will behave totally differently, depending on whether we express a length in meters or centimeters. The meaning of the data stays exactly the same! The choice of measurement unit should not have an influence on the inference that the learning system makes.

The above example is just an illustration of the general principle that we don't want the outcome of the learning process to depend on the scale on which an attribute happens to be expressed. A natural way to avoid this is to rescale all attributes into the $[0,1]$ interval. Any numerical attribute x_i can be rescaled into this interval using the following transformation:

$$x_i' = \frac{x_i - m_i}{M_i - m_i} \tag{7.2}$$

where x_i' is the rescaled version of x_i, and m_i and M_i are the minimal and maximal values of x_i, respectively. This may be known for the population, or they can simply be taken from the data (i.e., m_i is the smallest value of x_i observed in the dataset).

Alternative transformations may be used for rescaling. For instance, for attributes with a Gaussian distribution, there is no minimal or maximal value (though the smallest and largest observed values could still be used), but normalizing them (subtracting the mean and dividing by the standard deviation) gives them all a standard deviation of 1, which makes them comparable.

For datasets that have a mixture of nominal and numerical values, a distance measure is often defined that assigns a difference of 1 to different nominal values and a difference of 0 to equal nominal values. Thus, the smallest and largest possible difference between two nominal values is roughly the same as between two numerical values, if the latter have been rescaled to $[0,1]$. This makes nominal attributes comparable to numerical ones, whereas a rescaling of numerical attributes to, say, $[0,10]$, would give nominal attributes a smaller influence than numerical attributes.

7.4.2 Estimating the importance of attributes

In the previous section we tried to make all attributes comparable, to avoid the effect that the unit in which an attribute is expressed influences the outcome of the learning process. In this section we will be doing exactly the opposite: we try to give some attributes more influence on the outcome, if there is good reason to believe they are indeed more important.

```
function DetermineWeightsNN(training set T) returns weight vector:
    // start with all weights equal to one
    w := [1, 1, ..., 1]
    a := accuracy on T of k-NN with attribute weights w
    repeat
        w' := perturb(w) // make small random changes to the weights
        a' := accuracy on T of k-NN with weights w'
        if a' > a then
            w := w'
            a := a'
    until w has remained unchanged for a number of iterations
    return w
```

Figure 7.8: An empirical approach to optimizing the attribute weights for the k-nearest neighbor algorithm.

These two goals may seem contradictory, but they are not. We simply want to remove influences that are unwanted, and introduce (or strengthen) influences that are wanted.

Example 7.3 Let us return to the example from the previous section. Assume that the classification problem is whether the person is allowed to drive a car. The people may be living in a country for which we don't know the rules about who can drive a car, but it is likely that the age of the person is important, while their length is not. If a test instance (a new person) is the same age as someone we know who drives a car, it is likely this test person is also allowed to drive a car; if the test instance is the same length as someone who drives a car, however, this doesn't mean much. In this situation, we would hope that the learning system can discover that age, rather than length, should have a major influence on the outcome.

An attribute can be given more or less importance by using a **weighted Euclidean distance**, giving higher weights to more important attributes:

$$d_W(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{D} w_j (x_j - y_j)^2.$$

In this formula the w_j are the weights that are to be tuned by the learning process for optimal predictive performance.

Note that these w_j have nothing to do with the weights that we saw before, when weighting instances! Weighting instances was done to give a greater influence to instances nearby, and a smaller influence to instances far away. Here, we are weighting attributes, trying to give a greater influence to relevant attributes and a smaller influence to irrelevant attributes.

The question is now: how does the instance based learner determine the optimal weights? One possibility is to determine them empirically, using the "wrapper" approach shown in Figure 7.8. This approach consists of trying out k-nearest neighbors with various weights for the attributes, and keeping those weight settings that work best.

Alternatively, one can use statistical formulas to estimate the importance of weights. One such formula is the following (from Langley, 1996) :

$$w_i = 1 - 1/N \sum_{k=1}^{c} \sum_{j=1}^{N_k} |\bar{x}_{ki} - x_{ji}|$$

This formula assumes that the components have already been scaled to the $[0, 1]$ interval. The way the formula then determines the weight of attribute j can be explained as follows. For each class k, we determine the average value of attribute x_i for that class; call that $\bar{x}_{ki}$. We check how much the individual values x_{ji} within this class vary w.r.t. the class average. If there is much variation, and this is the case for many classes, then this decreases the weight of the component. If for many classes the x_i values are close to each other (hence, close to the average for that class), the sum term is small and the weight w_i will be relatively large.

The reasoning here corresponds quite closely to the reasoning followed in the statistical technique known as analysis of variance (ANOVA): there, a nominal variable is considered to have a significant impact on some target variable if, after dividing the dataset into groups based on their value for the nominal variable, the average intra-group variance is significantly smaller than the overall variance of the target variable. Langley's formula does not use sums of squared differences, as ANOVA does, but the formula is similar in spirit.

7.5 Using prototypes

We have seen earlier that the k-nearest-neighbor method tends to "forget more details" when the number of neighbors k increases, in the sense that, for instance, small islands of examples belonging to a different class than those in the area surrounding them can be generalized away. The use of prototypes is another modification to nearest neighbor methods that causes them to generalize more strongly.

Generally, a **prototype** can be defined as a data element that is a substitute for, or a representative for, multiple other data elements. The prototype may be of the same type as these other data elements, or it may be of a different type. Advantages of the use of prototypes are that it may make the nearest neighbor method more *efficient*, because the number of objects that is stored is reduced (since one prototype typically represents multiple other elements), and that it may make the learner yield a *smoother decision surface*. (As can be seen in several figures, the decision surface of nearest neighbor learners tends to look quite ragged.)

One way to introduce prototypes is the following: when there a set of data elements that are close together and where each element belongs to the same class, we could decide to represent the whole set by one single example that is the mean of those examples (if we have numerical data); or more generally, by a single example that is somehow a good representative for all the examples in the set. Figure 7.9 shows what happens if we have three clusters of examples where all the examples in a cluster always belong to one class. Note that by replacing each cluster by a single element, the decision surface becomes smoother.

In the case of a k-nearest neighbor method, the single new element may be given a weight equal to the number of examples it replaces, to approximate more closely the behavior of the original, non-prototype-using, method.

How do we decide when a set of examples can be changed into a prototype? Usually, there is no "correct" way of doing this: the system with prototypes will generally not be equivalent to the system without prototypes, as the changed decision surface in Figure 7.9 illustrates. By introducing prototypes, the system will behave in a different way, and this may be a good thing or a bad thing, in the same way that using a greater value for k may be good or bad, depending on the context. Replacing examples with a single prototype *removes detail* from the dataset in a similar way that increasing k causes the prediction to *ignore details* in the dataset.

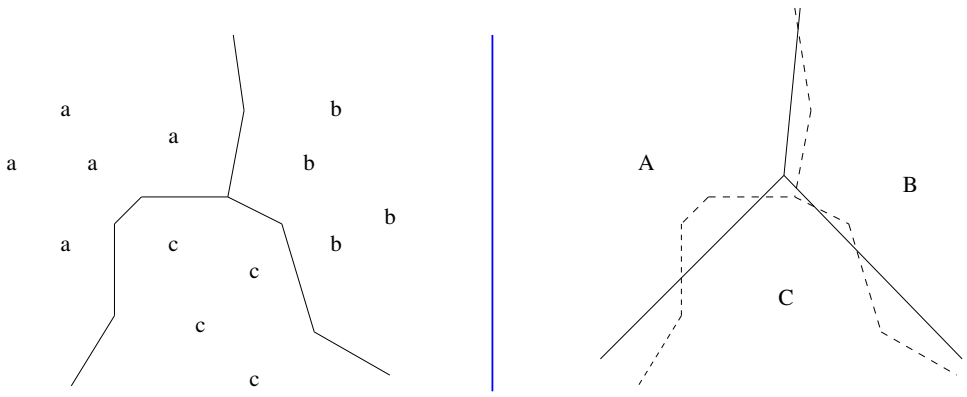

Figure 7.9: Replacing multiple data elements with a single element. In this case, all examples of each class (a, b, c) are replaced by a prototype instance (A, B, C) for that class. The decision surface becomes less ragged, and in this case probably also somewhat less accurate (one c element in the data is quite close to the A-C border).

Moving from lazy to eager learning

Just like the computation of attributes weights, the construction of prototypes comprises a step away from "pure" instance-based learning, where no other model is constructed than the dataset itself. With prototypes, instance-based learning becomes more like inductive learning, in the sense that the dataset itself is replaced by a model of it, for which there is no guarantee that it is entirely accurate.

From this point of view, constructing prototypes can be seen as a way of creating hybrid learners, learners that are somewhere between instance-based and inductive learning. An example of an approach that takes this viewpoint is the RISE algorithm [15]. In RISE, groups of examples with the same class can be replaced by a rectangle, which itself is represented with a conjunctive condition. Thus, the rectangle can be seen as representing an if-then-rule. For instance, in a twodimensional space with coordinates x and y, a rectangle $1 \leq x \leq 3 \wedge 5 \leq y \leq 6$ might be formed as a prototype, if the conditions are right (say, the rectangle contains 12 positive examples and no negatives). Prediction is still based on the k-nearest-neighbor principle, but the k nearest neighbors may be rectangles instead of original data elements. The distance metric needs to be extended to include the concept of distance between a point and a rectangle. (This may be, for instance, defined as the distance between the point and the closest point on the rectangle.)

Note an important difference with regular rule learners. A regular rule learner might learn the rule "**if** $1 \leq x \leq 3$ **and** $5 \leq y \leq 6$ **then** positive", but then any example outside this rectangle would not be influenced by the rule. In the RISE approach, the rule will still have an influence on examples close to (but outside) the rectangle, and this influence will diminish when we move further away from the rectangle.

A rule set learner tries to model the whole dataset with rules. An instance-based learner uses the dataset itself as its model. The hybrid approach, as illustrated by RISE, models as rules those parts of the data set for which it seems advantageous to model them as rules, and keeps the individual examples in those areas where rules seem not appropriate. Thus, it attempts to combine the advantages of both approaches.

Figure 7.10 gives an idea of how the use of rectangles as prototypes influences the decision surface of the learner.

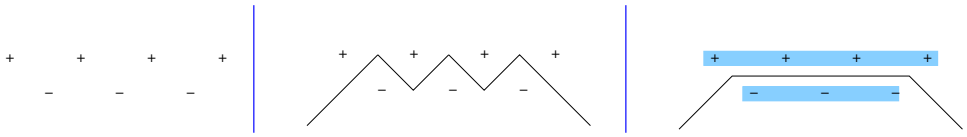

Figure 7.10: Decision surfaces obtained with the original elements and with rectangles as prototypes. Left: the dataset. Middle: 1-nearest neighbor produces a "zigzagging" decision surface. Right: using rectangular prototypes removes the zigzag in the middle, while still exhibiting similar behavior as nearest neighbor in areas farther away from the dataset.

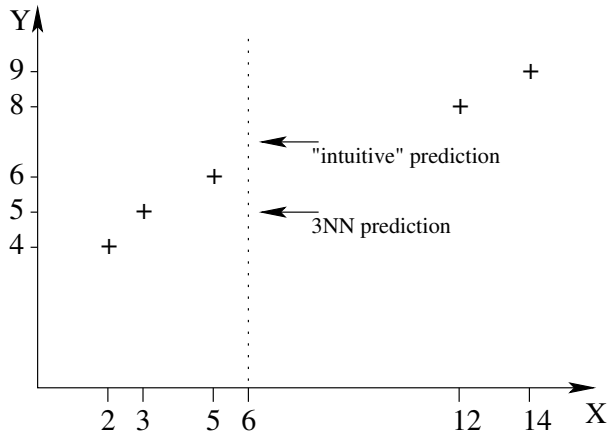

Figure 7.11: Nearest neighbor regression for a monotonically increasing target function. If the dataset contains points for X=2,3,5,12,14 as shown, and a prediction needs to be made for X=6, then 3-nearest neighbor would predict Y=5, whereas something like Y=7 seems intuitively more likely. By predicting the mean of the nearest neighbors, standard nearest neighbor methods do not take any local trends in the data into account.

7.6 Locally weighted regression

In the context of regression, we want to predict numbers rather than nominal values. Earlier we suggested that it makes sense to predict the mean of the k nearest neighbors, which is roughly the equivalent of predicting the most frequently occurring class among the k nearest neighbors in the case of classification. However, it is not difficult to think of cases where we would intuitively expect that predicting the mean is not optimal.

Figure 7.11 shows a one-dimensional input space X with a number of examples (x_i, y_i). Visualizing the training set in this way, we see that there seems to be an upward trend in the y values for increasing x. Assume now that we have a new instance $x = 3.2$ for which we want to predict the y value, and assume we use a 3-nearest neighbor predictor. The 3 nearest neighbors for the new x are to the left of x. Seeing the upward trend, we can intuitively expect y to be greater than the y value of its rightmost neighbor. But taking the mean of the neighboring y values will obviously return a smaller value, as Figure 7.11 illustrates.

Clearly, simply taking the mean of a number of neighbors does not always yield the most "natural" prediction. But what is the most natural prediction for x, given a small set of neighbors? Can we estimate it in a better way than just taking the mean?

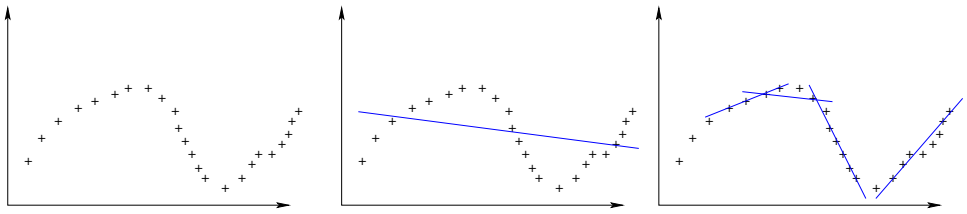

Figure 7.12: A global linear approximation, versus multiple local linear approximations, for a non-linear function.

The answer is yes. Note that we have again a regression problem, but instead of estimating the target value for x from the whole dataset using some regression method, we now want to estimate it from a local neighborhood around x. Such a regression problem can be solved using, for instance, linear regression.

Note the difference between applying linear regression locally and globally. Fitting a linear function through the whole dataset may yield a very inaccurate model of the dataset, if the target function is not linear. But fitting a linear function through a local neighborhood allows us to make relatively accurate predictions within that neighborhood, even for non-linear target functions. The reason is that many non-linear functions can be approximated locally quite well with a linear function, as long as the area within which they are approximated is sufficiently small. Figure 7.12 illustrates this.

Thus, the k-nearest neighbor regression method can be adapted as follows: instead of predicting the mean of the k nearest neighbors, fit a linear function through these nearest neighbors and use that function to predict the target value of the new point.

Fitting a function is of course based on a certain error criterion that needs to be minimized. One can minimize the sum of squared errors among the k nearest neighbors, or, similarly to what has been discussed before, one can minimize a weighted sum of squared errors among neigbors, giving a higher weight to closer neighbors. As discussed before, it is important to make the weights go down fast with the distance of the neighbors, to avoid the curse of dimensionality. One formula that can be used is

$$E = \sum_{x_i \in T} (y_i - f(x_i))^2 K(d(x_i, x))$$

where the K function is a so-called kernel function that decreases quickly as a function of its argument, for instance, $K(x) = \exp(-x^2/\sigma^2)$ with σ some parameter.

Note that the induction of a local regression model can be seen as somewhat similar to the induction of a prototype. Both are "local models", models that describe a local area within the instance space. There is still some difference though. The prototypes we discussed are typically constructed during the training phase, before we actually want to make any predictions. The local regression function f discussed here is constructed at the time of prediction, not earlier. It uses information about the instance that we will make a prediction for, and hence it is tuned to work optimally for exactly that instance. Afterwards, the function is discarded: we will not be able to reuse it for other instances. In contrast, prototypes do not use information about a single instance and will be re-used each time they turn out to be relevant for a particular prediction.

The difference between these two approaches is not inherent, however. The "construct the prototype at prediction time" principle, as used in locally weighted regression, could just as well be used for the prototype-approach mentioned before.

Instead of constructing rectangles or if-then-rules during the training phase, one could wait until a new instance is given for which a prediction needs to be made, and then try to construct an if-then rule (expressing for instance a rectangle) covering that instance.

Similarly, for numerical prediction, one could construct prototypes, or generally a model describing the full dataset, beforehand. The "radial basis functions" mentioned in the following section are an example of that.

7.7 Radial Basis Functions

The basic idea behind instance-based regression is that in any point of the space a prediction is made that is influenced mostly by the nearby training examples, and the prediction is typically a (linear or non-linear) combination of the target values of those training examples. In the case of local regression, this is achieved by fitting a model f in the local area around the prediction point.

When fitting a model f, the structure of f is usually given (e.g., a linear function), and only its parameters are learned from the data. Besides linear functions, other structures can be used. One possible structure for f is the following: represent f as a linear combination of other functions f_i, so-called **basis functions**:

$$f(\mathbf{x}) = \sum_i w_i f_i(\mathbf{x}).$$

Note that fitting a linear function corresponds to choosing $f_i(\mathbf{x}) = x_i$, the i'th component of $\mathbf{x}$. In general, $\mathbf{x}$ does not need to contain numerical attributes for this approach to work; it is sufficient that the range of each $f_i(\mathbf{x})$ is numerical. Further, the basis functions f_i may be fixed, or their parameters may themselves be unknown, so that the parameters of the basis functions are to be learned together with the parameters w_i.

In the case of *radial* basis functions, $f_i(\mathbf{x})$ depends only on the distance of $\mathbf{x}$ to some specific point μ_i, which is the "center" of the basis function. For instance, f_i may be a Gaussian with mean μ_i and variance σ_i^2:

$$f_i(\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma_i} \exp(-\frac{(\mathbf{x}-\mu_i)^2}{2\sigma_i^2}).$$

The value of $f_i(\mathbf{x})$ then approaches zero quite fast as the distance of $\mathbf{x}$ to μ_i increases.

By choosing as many radial basis functions (RBFs) as there are training instances, and choosing the training instances as their centers, we get a regression method that is very similar to earlier discussed methods for instance-based regression. In particular, if we use Gaussians for the radial basis functions f_i, then the linear combination of these basis functions in any point $\mathbf{x}$ has the same form as the weighted instanced-based regression method mentioned earlier, if weights are chosen that decay exponentially with Euclidean distance. One difference is that with RBFs, a different σ_i can be chosen (learned) for each instance, a possibility that was not provided in the weighted method.

Instead of choosing one RBF per training instance, one can also choose a smaller number of RBFs. Each RBF then serves more or less as a "prototype" in the sense we saw before: it is representative for a number of training instances (not just one) and replaces them in our model. Besides the σ_i parameter, also the μ_i parameter now needs to be determined.

A model based on radial basis functions can be learned using, for instance, neural networks; the term **radial basis function networks** is then used. We do not discuss this in more detail here.

Chapter 8

Clustering

Generally speaking, clustering, or "unsupervised learning", refers to the task of finding structure in a set of data, in the sense that elements of the set may not be distributed entirely randomly but tend to form certain patterns or coherent groups. This task is easily illustrated graphically. Figure 8.1 shows two data sets: the left one has no obvious structure, but in the right one we can clearly see that the instances are not entirely randomly (uniformly) spread over the instance space: there are clearly identifiable groups, or **clusters** of instances. Given a data set, clustering is the task of discovering which clusters exist in it.

There are many different methods for clustering, but also many different variants of the clustering task. We start with an overview of several dimensions along which clustering tasks can be distinguished. Next, we will discuss a number of clustering methods.

8.1 Variants of clustering

8.1.1 Disjoint clusters versus overlapping clusters

In most definitions of clustering it is assumed that a single instance cannot belong to more than one cluster. In other words, the clusters are *disjoint*. There are cases, however, where one may be interested in relaxing this restriction, and allowing *overlapping* clusters. In the latter case the task is often called **subgroup discovery** rather than clustering.

Note that when talking about "overlapping" clusters, there are two natural ways to interpret this. One is that a single instance may belong to more than one cluster. Another is that the areas in the instance space covered by the clusters overlap. In

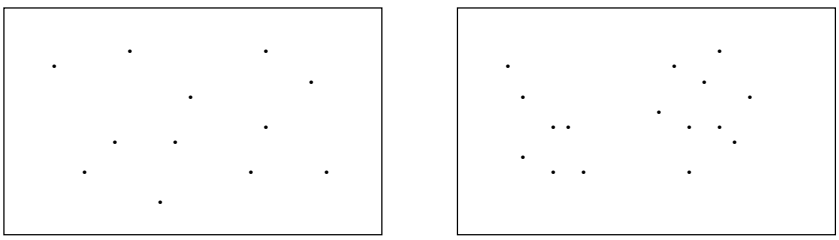

Figure 8.1: Two example datasets. In the left dataset, instances are spread more or less uniformly over the instance space. In the right dataset, there are clearly identifiable groups of instances, or "clusters".

the second case, there are regions where instances from different clusters may occur, but each individual instance still belongs to one cluster. In this text, we follow the first interpretation: when we say that clusters overlap, we mean that an example may effectively belong to multiple clusters at the same time.

8.1.2 Exhaustive versus partial clustering

It is also often assumed that every instance must belong to some cluster. We can call this *exhaustive clustering*. This restriction can be relaxed, allowing instance to belong to no cluster at all (*partial clustering*. In practice, the difference between these two is not so important because an instance that does not belong to any cluster can always be considered a cluster on its own.

In the remainder of this text we will always assume that clusters are exhaustive and disjoint, i.e., all examples belong to exactly one cluster, unless specified otherwise.

8.1.3 Flat clustering versus hierarchical clustering

Often, clusters can be defined at several levels: a cluster may itself have clearly identifiable subclusters. **Flat clustering** refers to the task of identifying clusters on one single level. That is, the data set is partitioned into clusters that are disjoint and exhaustive, and clusters do not have subclusters. Formally, the task is defined as follows:

Task description: Flat clustering

Given: a data set S
Find: a set $\mathcal{C}$ of k clusters $C_i \subseteq S$, $i = 1, \ldots, k$ such that:

- $\forall C_i, C_j : C_i \cap C_j = \emptyset$ (disjoint)
- $C_1 \cup C_2 \cup \ldots \cup C_k = S$ (exhaustive)
- $\mathcal{C}$ optimizes some quality criterion Q.

Note that this task description is equivalent to the following one:

Task description: Flat clustering (2)

Given: a data set S
Find: a total function $h : S \to \{1, 2, \ldots, k\}$ such that:

- h optimizes some quality criterion Q.

Indeed, since h assigns exactly one number from 1 to k to each $x \in S$, we can define C_i as the set of all $x \in S$ for which $h(x) = i$.

The first formulation better explicitates the structure of the clustering (disjoint and exhaustive), but the second formulation is sometimes useful too, as we will see in a moment.

In both formulations, the quality criterion is a function that maps a clustering $\mathcal{C}$ onto its "quality" $Q(\mathcal{C})$. Generally, the quality criterion will reward clusterings that do a better job of keeping similar instances in the same cluster and dissimilar instances in different clusters. This can be done in different ways. For now we do not go in more detail about this; examples of Q functions will be given later on.

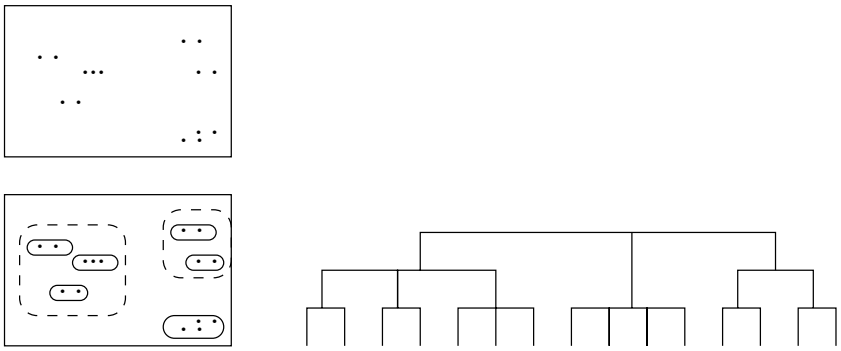

Figure 8.2: A cluster hierarchy. The left image shows a dataset with clusters indicated; the right image shows the tree representation of the cluster hierarchy. The bottom level has one cluster per instance, the top level is the whole dataset, in between are two levels of clustering.

In **hierarchical clustering**, clusters are defined at different levels. A hierarchy of clusters is returned: at the top is the entire data set S, which can be seen as one cluster; this set is divided into clusters $C_{11}, \ldots, C_{1k_1}$: each of the clusters C_{1i} is further divided into subclusters, etc.; and at the lowest level, each instance is a cluster on its own. The formal task description is as follows:

Task description: Hierarchical clustering

Given: a data set S
Find: a set $\mathcal{C}$ of clusters such that:

- $S \in \mathcal{C}$
- $\forall x \in S : \{x\} \in \mathcal{C}$
- $\forall C_i, C_j \in \mathcal{C} : (C_i \cap C_j = \emptyset) \vee (C_i \subseteq C_j) \vee (C_j \subseteq C_i)$
- $\mathcal{C}$ optimizes some quality criterion Q.

A cluster hierarchy is usually represented graphically as a tree structure. Figure 8.2 shows an example.

Note that in a hierarchical clustering, the notion of "disjoint clustering" is a bit more complex. In a flat clustering we simply define a clustering as disjoint if the intersection of two clusters is always empty. In a hierarchical clustering, we call the clustering disjoint if clusters only intersect with their subclusters or superclusters.

8.1.4 Extensional versus intensional clustering

Clusters can be defined extensionally, i.e., by *listing their elements*. Alternatively, one might define it by *listing conditions* that an instance must fulfill in order to belong to the cluster: this is called an intensional definition. According to whether clustering algorithms return an extensional or an intensional definition of clusters, we call them **extensional** or **intensional clustering** algorithms.

Example 8.1 Consider a set of animals occurring in cartoons or comic strips, containing descriptions of Mickey, Donald, Tom, Jerry, Speedy, Garfield, Odie, Snoopy.

function IntensionalClustering(dataset S,
extensional clustering algorithm EC,
supervised concept learning algorithm SCL,
language $\mathcal{L}$)
returns set of cluster definitions:
$\mathcal{C}$:= EC(S)
let C_1, C_2, ..., C_k be the clusters in $\mathcal{C}$
for $i := 1$ **to** k:
$D_i := SCL(C_i, S - C_i, \mathcal{L})$
$\mathcal{D} = \{D_1, \ldots, D_k\}$
return $\mathcal{D}$

Figure 8.3: Intensional clustering as a combination of extensional clustering and supervised concept learning. SCL represents a generic concept learner; its arguments are the set of positive examples, the set of negative examples, and the language in which the concept should be expressed.

The description might include properties such as their size, speed, etc. An extensional clustering might yield clusters {Jerry, Mickey, Speedy}, {Donald}, {Tom, Garfield}, {Odie, Snoopy}. An intensional clustering might yield the same clusters, but describe the first cluster as $\{x|size(x) = small \wedge speed(x) = fast\}$: the set of all creatures that are small and fast.

An obvious difference between the two kinds of definitions is that intensional definitions generalize to the population. An instance that was not in the training set belongs to a cluster when it fulfills the conditions for that cluster. When clusters are defined extensionally, they contain, by definition, only the instances in the training set. However, there is often an implicit method of converting the extensional definition into an intensional one. For instance, there may be an implicit assumption that an instance belongs to a cluster C_i if it "fits" in it better than in any other cluster, where the fit is related to how similar the instance is to the instances in the cluster. Exactly how we measure this similarity, and how we use it to determine the fit, may differ from one clustering algorithm to another.

Some intensional clustering algorithms work in two steps: first, they cluster the instances extensionally, then they attempt to characterize these extensional datasets by providing intensional definitions. Note that this second step is in fact a supervised concept learning (classification) task. Figure 8.3 illustrates this.

8.1.5 Summary: Dimensions of clustering

To summarize the above: We categorize clustering tasks along four dimensions:

- disjoint versus overlapping
- exhaustive versus partial
- flat versus hierarchical
- intensional versus extensional

In this text we always assume disjoint and exhaustive clustering. We will see examples of flat as well as hierarchical clustering systems, and of intensional as well as extensional systems.

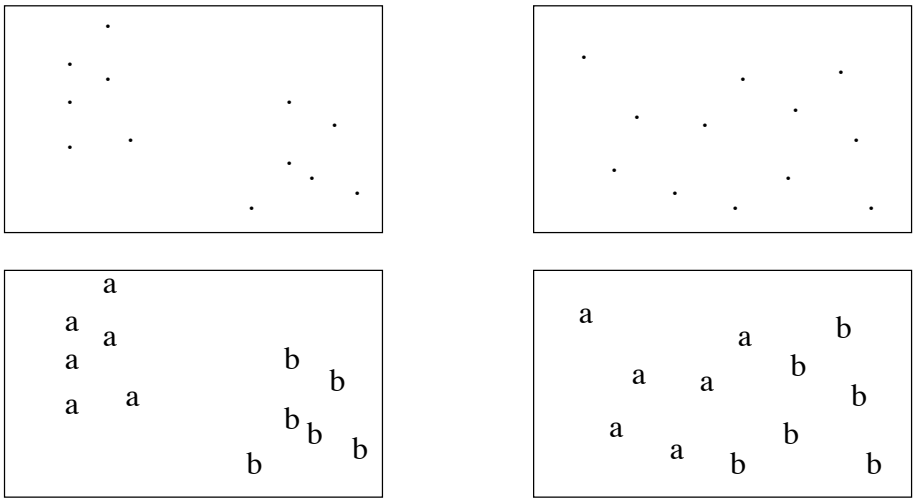

Figure 8.4: Unsupervised versus supervised learning. Top: unlabeled examples (no class information given). Bottom: examples are labeled with their class (a or b); these classes define the "target clusters" that we want to find. To the left is a situation where we might hope that a clustering system will discover clusters that correspond to the classes. To the right is a situation where it is easy to build a classifier that distinguishes class a from class b, but it is unlikely that a clustering system would discover clusters that correspond to the classes.

8.2 Evaluation of clustering solutions

A major problem with clustering is how to evaluate results. For supervised learning, there is an obvious quality criterion: the predictions that the model makes on an instance should correspond to its actual target value. There is a clear notion of "correct" and "wrong". But for clustering, this is not always the case.

We can distinguish two situations:

1. There exists a natural clustering that is known to us, but no information about this is given to the learner. This is like a classification setting where the class attribute is not included in the training set. We cannot hope to find a function mapping instances onto their true classes, because no information on these true classes is known; but we can still hope to find clusters such that each cluster corresponds to one specific class. Figure 8.4 illustrates this.

2. We do not know any natural clustering ourselves. This will almost always be the case in practical clustering tasks.

If it is in practice almost always the case that we don't know the true clustering, why are we considering the first case? The reason is that this first case is perfectly suitable for *testing whether a clustering algorithm can be expected to give good results*. If it yields clusters equivalent to the true classes on a dataset where we know these true classes, then we can consider it likely that also in a case where we don't know the true classes, the clusters returned correspond to them.

We now consecutively consider the two cases.

8.2.1 A target clustering is known

Note that when we say a target clustering is known, this means: known to the human evaluator of the clustering. It is not known to the learner.

The clustering task can in this case be described more precisely:

Task description: find clusters corresponding to classes

Given: a data set S; the existence of an unknown set of classes C and an unknown mapping $f : S \to C$
Find: a hypothesis h (including the number k) such that:

- $\forall x \in S : h(x) \in \{1, \ldots, k\}$
- $h(x_1) = h(x_2)$ if and only if $f(x_1) = f(x_2)$

This is the same task description as we saw before, but with a specific instantiation of the quality function Q: the aim is to find a clustering h such that each cluster i contains all the elements of a class $c_j \in C$. Note that it is not required that $j = i$; it is only important that h makes the same prediction for two examples if and only if they belong to the same true class.

This task description makes it more clear how unsupervised learning compares to supervised learning. In supervised classification we learn a function $h : S \to C$, and we aim for $h(x) = f(x)$, which of course implies (i.e., is a stronger condition than) $h(x_1) = h(x_2) \Leftrightarrow f(x_1) = f(x_2)$. In unsupervised classification, since we do not even know the values that $f(x)$ can take, we have to be happy with fulfilling the weaker condition.

It is now possible to assess the quality of a clustering h by looking at how close h is to the true clustering f. A straightforward quality measure is the probability that two random instances are in the same predicted cluster but have different true class values, or the other way around:

$$Q(h) = Pr(h(x_1) = h(x_2) \wedge f(x_1) \neq f(x_2) \vee h(x_1) \neq h(x_2) \wedge f(x_1) = f(x_2)).$$

If $Q(h) = 0$, we have found a perfect clustering.

8.2.2 No target clustering is known

This situation often arises in practice. Indeed, if we already know the true clustering, there is no need to learn it anymore! The above method is useful from the point of view of evaluating how good a clustering algorithm is: we can run it on a problem where we know the classes, and check how well the clustering algorithm identifies them. But in most real-life clustering tasks, this evaluation method cannot be used.

Yet, it is possible to define quality criteria for clustering that are independent of a target clustering. Generally, instances that are similar should be in the same cluster and instances that are dissimilar in different clusters. Hence, if we have a similarity measure between instances we could use this to quantify, for instance, the average intra-cluster similarity (how similar are instances in the same cluster, on average?) and inter-cluster similarity (how similar are instances that belong to different clusters, on average?). Denoting with $s(x_1, x_2)$ the similarity between two instances x_1 and x_2, we can define:

$$IntraCS = \mathbf{E}_{h(x_1)=h(x_2)}[s(x_1, x_2)] \quad (8.1)$$
$$InterCS = \mathbf{E}_{h(x_1)\neq h(x_2)}[s(x_1, x_2)] \quad (8.2)$$

A clustering is better if it has a higher $IntraCS$ and lower $InterCS$. However, it is difficult to compare two clusterings when one of them has higher $IntraCS$ as

well as higher $InterCS$. Which clustering is best then depends on how one wants to trade-off these two criteria. One simple way to do this might be to maximize the ratio $IntraCS/InterCS$, or the difference $IntraCS - InterCS$. Another way to trade-off the two will be discussed later, in the part about the COBWEB clustering system.

Exercise 8.1 Give an example of two clusterings where one has both higher $IntraCS$ and higher $InterCS$.

Exercise 8.2 How does the optimality criterion for the case where classes are known correspond to the setting where clusters should have high $IntraCS$ and low $InterCS$? In other words, what notion of similarity are we using in the first case?

We will see more sophisticated quality criteria for clustering later on.

8.3 Similarity measures

Since the goal of clustering is to form clusters such that instances in the same cluster are similar and instances from different clusters are dissimilar, we need to agree on the notion of "similarity" before we can define the quality of clusters.

We can define a **similarity measure** $s : X \times X \to \mathbb{R}^+$ that expresses the similarity between two instances x_1 and x_2. The more similar x_1 and x_2 are, the larger the number $s(x_1, x_2)$ is. Alternatively, one can define a **dissimilarity measure** $d : X \times X \to \mathbb{R}^+$ with the property that $d(x_1, x_2)$ is larger if x_1 and x_2 are less similar. Often, $d(x, x) = 0$: the dissimilarity of any instance to itself is 0, and is the smallest dissimilarity possible.

The notions of similarity and dissimilarity are quite general: little is imposed concerning the properties of d and s. For instance, it is not imposed that the functions are symmetric: x_1 might be more similar to x_2 than x_2 is to x_1. While this may seem strange, such asymmetric similarity notions are sometimes the most natural ones.

But in many cases, the dissimilarity function d is naturally defined as a distance metric. A function d is a distance metric if it fulfills the following properties:

- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) = d(y, x)$ (symmetry)
- $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality)

Some clustering algorithms assume that dissimilarity is a distance metric; this allows them to exploit certain properties of distance metrics. Other clustering algorithms work just as well with any similarity or dissimilarity measure.

When instances are described as vectors in a vector space, a popular choice is the Euclidean distance. Assuming that the vector space is D-dimensional, and defining $x_j = (x_{j1}, x_{j2}, \ldots, x_{jD})$, we have

$$d(x_1, x_2) = \sqrt{\sum_{i=1}^{D} (x_{1i} - x_{2i})^2}$$

When instances are described using nominal values, the Hamming distance is often used as a dissimilarity measure:

$$d_H(x_1, x_2) = \sum_{i=1}^{D} \delta_{x_{1i} \neq x_{2i}}$$

where δ_C is 1 if $C = true$ and 0 otherwise.

Exercise 8.3 The quality function Q that we proposed for evaluating a clustering when the true classes are known (see Section 8.2.1) can be interpreted as using a particular similarity measure between instances. Which one is that?

Many other (dis)similarity measures are possible, see also the discussion on instance-based learning.

8.4 Clustering algorithms

Now that we have discussed a number of variants of the clustering tasks, evaluation criteria, and (dis)similarity measures, we can look at a number of specific clustering algorithms.

8.4.1 Flat clustering

Leader

LEADER is one of the oldest clustering algorithms; it is very simple and fast but gives relatively bad results. The ever increasing speed and internal memory of computers have rendered it mostly obsolete.

The LEADER algorithm takes one parameter: a similarity threshold t. It forms clusters that have the property that the "radius" of each cluster is at most t, that is, each cluster contains an element l (called the "leader") such that all members of the cluster are at a distance of at most t from l.

The algorithm makes one single pass through the data set, and each example is processed as follows. If the distance between the example and an already found leader is less than t, the example is added to that leader's cluster. (If there are multiple leaders close enough, the first one encountered is chosen.) Otherwise, the example becomes a new leader. Once all examples in the data set has been processed, the clustering process is finished.

There are several disadvantages of this algorithm.

1. It does not attempt to find an "optimal" clustering, it just finds "some" clustering that satisfies the constraint that all the elements in a cluster are at less than a given distance t from some center point. If t is badly chosen, the whole data set might be returned as one cluster (if t is too large) or one might get a different cluster for just about every example (if t is too small), and this will happen regardless of the true structure in the data.

2. The result of the algorithm depends on the order in which the dataset is processed. Indeed, assume two examples x and y are at a distance less than t from each other, and they are the first two examples in the dataset. Then depending on which of them comes first, one of the two becomes a leader and the other one just joins that cluster. All remaining examples are compared to the leader; if this leader is x, we may get a different clustering than if the leader is y.

3. The algorithm has a tendency to produce unbalanced clusterings; more specifically, larger clusters for leaders that were found early in the process, and smaller clusters for leaders found later.

Exercise 8.4 Take a one-dimensional dataset with examples 1,2,3,4,5 and $t = 2.5$. Show that depending on the order in which the examples appear, the leader algorithm may yield one cluster, or two; and that in the case of two clusters, one may

get a cluster of 3 elements and another one of 2, but also a cluster of 4 and one of 1 element.

The only advantage of LEADER is that it is fast: only one pass through the data set is needed, and for each example only a (usually) small number of distance computations needs to be made.

Most of the other algorithms yield much better results and still run sufficiently fast on today's computers, even when dealing with large datasets. Therefore the LEADER algorithm is barely used nowadays.

k-means clustering

k-means clustering is currently probably the most popular algorithm for flat extensional clustering. The algorithm takes as input, besides the data set to be clustered, one parameter k that indicates the number of clusters that is to be constructed. The fact that this number has to be given in advance is considered a disadvantage: it is usually not known in advance how many clusters the optimal clustering has. Very often, the k-means clustering algorithm is therefore run several times with different k, and the different clusterings that result from this are compared using the quality criterion Q.

The k-means clustering algorithm is shown in Figure 8.5. It starts with a clustering that is determined more or less at random, and is usually rather bad. It then uses a specific procedure to improve that clustering, and repeatedly applies this same procedure to the new clustering until no further improvements are obtained. At that point it stops and outputs the last clustering. We now discuss how the algorithm finds its first clustering and how it improves a clustering.

Initialization: The initial clustering is determined as follows. k instances are drawn randomly; these instances are called seeds. With each seed a set is associated that is initially empty. Then the algorithm looks at all the instances in the data set, and for each instance it finds the seed that is closest to that instance. The instance is put in the set corresponding to that seed. Once all the instances have been assigned to the set of the closest seed, we have our initial clustering. The quality of this clustering depends strongly on how the seeds have been chosen. Since this was randomly, the clustering is likely to be a bad one.

Improvement: The procedure for improving the clustering works as follows. First, for each cluster in the current clustering, the mean of all the vectors in that cluster is computed. This mean will be a new seed. k seeds are determined in this way. Next, instances are re-assigned to the closest seed. Since the new seeds are not the same as the old ones, instances may move from one cluster to another. An instance that was in the middle of its current cluster is likely to stay there, but an instance that was quite far away from the center (which is now a new seed) and is in fact closer to the center of another cluster, will be assigned to that other cluster. This procedure generally yields a better clustering than the previous one.

It may not be obvious that this procedure of iterative improvement of an originally bad clustering works well. And indeed, it is not guaranteed to yield a good clustering; combinations of data sets and initial seeds can be constructed for which the procedure converges to a quite bad clustering. But in general, it tends to work reasonably well. Figure 8.6 illustrates how even with a seemingly very bad choice of initial seeds, the procedure easily converges onto a natural clustering.

function k-means-clustering(data set D, integer k) **returns** clustering:
    // form the initial clustering:
    for $i := 1$ **to** k:
        $s_i :=$ random instance from the instance space
        $C_i := \emptyset$
    for $j = 1$ **to** N:
        let s_c be the seed for which $d(x_j, s_c)$ is minimal
        add x_j to C_c
    // improve the clustering until you no further improvements are possible:
    repeat
        for $i := 1$ **to** k:
            $s_i := \sum_{x_j \in C_i} x_j / |C_i|$
            $C'_i := C_i$
            $C_i := \emptyset$
        for $j = 1$ **to** N:
            let s_c be the seed for which $d(x_j, s_c)$ is minimal
            add x_j to C_c
    until $\{C_1, C_2, \ldots, C_k\} = \{C'_1, C'_2, \ldots, C'_k\}$
    return $\{C_1, C_2, \ldots, C_k\}$

Figure 8.5: The k-means clustering algorithm.

The problem that the result depends on the initial random seeds can be alleviated by running the algorithm several times and choosing among the resulting clusterings the one with the highest quality. Similarly, the problem that k has to be chosen in advance can be alleviated by running it several times with each time a different value for k.

The quality function Q that k-means clustering tries to optimize is the sum of intra-cluster variances.

$$Q(\mathcal{C}) = \sum_{C_i \in \mathcal{C}} \sum_{x_j \in C_i} d(x_j, s_i)^2$$

where s_i is the mean of the vectors in cluster i and the C_i are the clusters that have been formed.

In this discussion we have assumed that the objects to be clustered are represented by numerical vectors, so that a mean can be computed. The procedure can be generalized towards other kinds of data, as long as the following two functions are defined: a distance measure (for assigning objects to clusters), and a *centroid* or *prototype* function that maps a set of objects onto a single object that is maximally representative for the set (for instance, for which the sum of squared distances of all elements to this element is minimal). The generalized procedure is sometimes referred to as *k-centroids clustering*.

EM as a clustering procedure

EM stands for *Expectation Maximization*. It is a general procedure for learning so-called mixture models from data. In a mixture model, the population distribution is assumed to be a superposition of different distributions that each describe a subset of the population. In other words, the population itself is a mixture of different subpopulations. We are interested in learning models for these subpopulations, as well as estimating for each instance which subpopulation it belongs to. That is exactly what the EM algorithm tries to do.

EM consists of the repeated application of two steps:

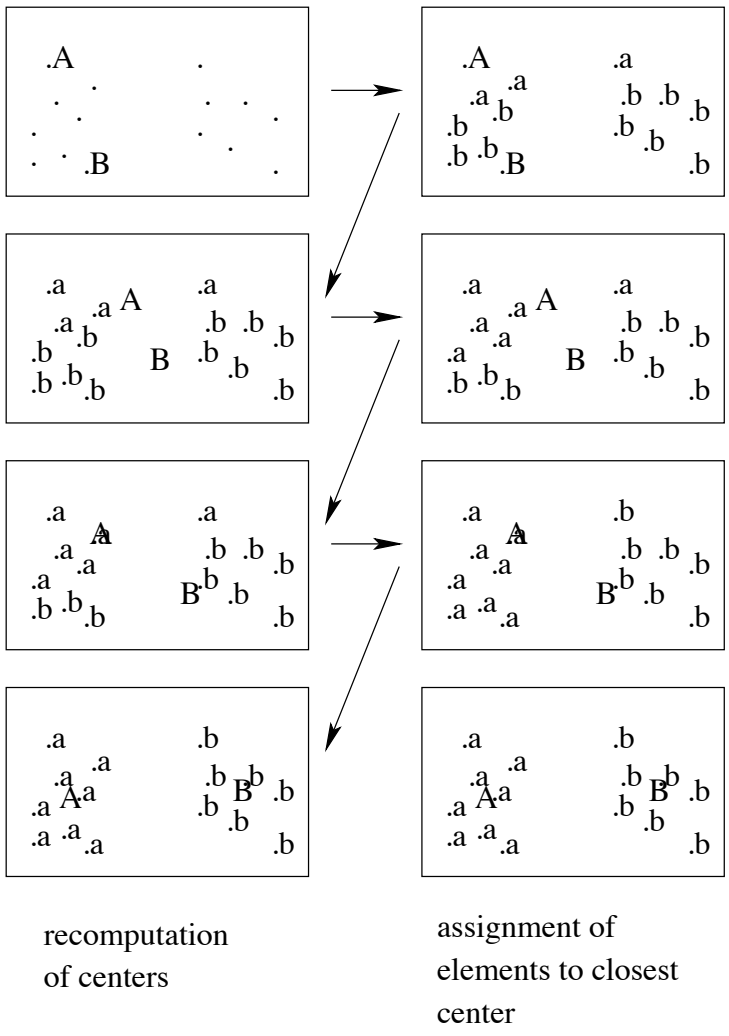

Figure 8.6: An illustration of how the clusters in k-means clustering gradually converge to the true clustering. A and B denote the seeds, which are initially two random elements. Elements assigned to A (B) are labelled a (b). In the fourth assignment of elements to seeds, no changes occurr in the assigment, which means the process has converged.

- the Expectation step: for each instance x, and for $i = 1, \ldots, k$, the probability that x belongs to the i'th subpopulation is estimated
- the Maximization step: the models describing the subpopulations are re-estimated according to the just computed probabilities

After re-estimating the models, the probability that an instance belongs to some subpopulation has changed, so the Expectation step needs to be redone. This necessitates another Maximization step, and so on.

EM can be used as a clustering procedure, and then it behaves much like k-means clustering. The use of EM for clustering is as follows. The dataset is considered to have been generated from a mixture of k Gaussians. That is: we assume that there are k clusters, and for each cluster i, its elements follow a Gaussian distribution with mean μ_i and standard deviation σ_i. (For simplicity reasons, we assume x to be one-dimensional for now.) Given the dataset and the parameter k, we want to estimate the values of μ_i and σ_i, and we want to know for each example the probability that it belongs to cluster i, for all i. The two steps from EM then concretely look as follows.

Expectation step: For each example, the probability that it belongs to cluster i is (re-)computed. Assuming that each cluster is a priori equally likely to contain the example, the probability that an example x belongs to cluster C_i is proportional to $p_i(x)$, the probability density of the cluster's distribution, evaluated in x. More specifically,

$$Pr(x \in C_i) = p_i(x) / \sum_j p_j(x)$$

where

$$p_i(x) = \frac{1}{\sqrt{2\pi}\sigma_i} exp(-(x - \mu_i)^2 / 2\sigma_i)$$

is the density function of a Gaussian distribution with mean μ_i and standard deviation σ_i.

Maximization step: for all clusters, we recompute their mean and variance. The standard formula for mean and variance is used, but each example is weighted according to the current estimate of the probability that it belongs to the cluster. Thus we get, for i from 1 to k:

$$\mu_i := \frac{\sum_j x_j Pr(x_j \in C_i)}{\sum_j Pr(x_j \in C_i)}$$
$$\sigma_i^2 := \frac{\sum_j (x_j - \mu_i)^2 Pr(x_j \in C_i)}{\sum_j Pr(x_j \in C_i)}.$$

A summary of the algorithm is shown in Figure 8.7.

This procedure is very similar to k-means clustering, the main difference being that in k-means clustering an example is always assigned as a whole to a cluster, whereas in EM it is assigned partially to each cluster, more specifically, proportionally to the probability that it belongs to that cluster. To estimate that probability, we need to make some assumption about the model that generated the data. In this case, that assumption is that each cluster has its own Gaussian distribution with a different mean and variance.

Two remarks are in place regarding the above description. First, we have for simplicity assumed that x, μ and σ are scalars. The procedure naturally generalizes towards D-dimensional data. x and μ are then vectors, but σ becomes a covariance matrix Σ, and the probability density function becomes $\frac{1}{\sqrt{(2\pi)^D |\Sigma|}} exp(-(x_j -$

function ClusterEM(data set S, integer k) **returns** set of Gaussians:
    Initialize μ_i and σ_i randomly
    while stop criterion not reached:
        for $j = 1$ **to** N:
            for $i = 1$ **to** k:
                $Pr(x_j \in C_i) := \frac{exp(-(x_j-\mu_i)^2/2\sigma_i)}{\sum_{i'=1}^{k} exp(-(x-\mu_{i'})^2/2\sigma_{i'})}$
        for $i = 1$ **to** k:
            $\mu_i := \frac{\sum_j x_j Pr(x_j \in C_i)}{\sum_j Pr(x_j \in C_i)}$
            $\sigma_i^2 := \frac{\sum_j (x_j-\mu_i)^2 Pr(x_j \in C_i)}{\sum_j Pr(x_j \in C_i)}$
    return $\{(\mu_1, \sigma_1), (\mu_2, \sigma_2), \ldots, (\mu_k, \sigma_k)\}$

Figure 8.7: EM as a clustering algorithm.

$\mu_i)^\top \Sigma^{-1}(x_j-\mu_i)/2)$ where $x^\top$ denotes the transpose of x. Second, we have assumed that the a priori probability of a random instance belonging to a cluster is equal for all clusters; if this is not the case, then the different $Pr(C_i)$ need to be taken into account: $Pr(x \in C_i) = Pr(C_i)p_i(x)/\sum_j Pr(C_j)p_j(x)$.

8.4.2 Hierarchical clustering

In hierarchical clustering, the aim is to find a hierarchy of clusters. The top cluster contains all the data. It is divided into subclusters, which are each divided into their own subclusters, and so on, until we arrive at the lowest level where each data element is a different cluster.

Hierarchical clustering methods may try to identify the clusters from bottom to top, starting with one cluster per element and merging clusters until only one remains; or they may work top-down, starting with the top cluster and dividing clusters into subclusters until one-element clusters are obtained. The first kind are called **agglomerative clustering methods**, the second kind **divisive clustering methods**. Some methods use a mix of the top-down and bottom-up approaches. Below, we discuss a number of well-known hierarchical clustering algorithms.

Agglomerative clustering

Agglomerative clustering typically works as follows. Start with defining each single instance as a cluster on its own. Then repeatedly do the following: take the two clusters that are closest together and merge them into a supercluster. Keep repeating this procedure until you end up with one single cluster, which is the whole data set.

The agglomerative clustering method just described relies on the notion of how close two clusters are. Note that this is a new notion: we already had the notion of similarity, dissimilarity, or distance between two *instances*, but not between two *clusters* (which are sets of instances).

Different variants of agglomerative extensional hierarchical clustering exist, which differ in how they define the "closeness" of two clusters. Often used variants include:

- **single linkage**: two clusters are as close as their closest elements. In other word, the distance between two clusters C_1 and C_2 is the minimum of the distances between any two instances $x_1 \in C_1$ and $x_2 \in C_2$. Defining d_C as the function that measures the distance between clusters, we have

$$d_C(C_1, C_2) = \min\{d(x_1, x_2) | x_1 \in C_1, x_2 \in C_2\}$$

function AGGLEHC(data set D, cluster distance function d_C) **returns** cluster hierarchy:
    for $i := 1$ **to** $|D|$: $C_i = \{x_i\}$
    $L := \{C_1, \ldots, C_{|D|}\}$ *// L will be the current top-level clustering*
    $\mathcal{C} := L$ *// $\mathcal{C}$ will be the final clustering*
    for $i := 1$ **to** $|D| - 1$:
        // merge the closest clusters
        let C and D be clusters such that for all $C', D' \in \mathcal{C} : d_C(C, D) \leq d_C(C', D')$
        $L := L - \{C, D\} \cup \{C \cup D\}$
        $\mathcal{C} := \mathcal{C} \cup \{C \cup D\}$
    return $\mathcal{C}$

Figure 8.8: Agglomerative extensional hierarchical clustering.

- **complete linkage**: two clusters are as close as their most distant elements. In other word, the distance between two clusters C_1 and C_2 is the maximum of the distances between any two instances $x_1 \in C_1$ and $x_2 \in C_2$:

$$d_C(C_1, C_2) = \max\{d(x_1, x_2) | x_1 \in C_1, x_2 \in C_2\}$$

- **average linkage**: The distance between two clusters C_1 and C_2 is the mean of the distances between any two instances $x_1 \in C_1$ and $x_2 \in C_2$:

$$d_C(C_1, C_2) = \frac{\sum_{x_1 \in C_1, x_2 \in C_2} d(x_1, x_2)}{|C_1||C_2|}$$

Note that all these instantiations have the property that when the two clusters have only one element, their distance is the same as the distance between these single elements:

$$d_C(\{x_1\}, \{x_2\}) = d(x_1, x_2).$$

Therefore, d_C uniquely determines d.

The algorithm for agglomerative extensional hierarchical clustering (AgglEHC) is shown in Figure 8.8.

The algorithm behaves quite differently depending on the choice of d_C. The single linkage version tends to form elongated clusters, a kind of serpentines, whereas complete linkage has a tendency to form spherical clusters.

Any hierarchical clustering can of course be turned into a flat clustering by looking at a single level of the hierarchy. As such, these methods can also be used as alternatives to k-means clustering.

Note the computational complexity of this clustering procedure. Whereas k-means clustering is linear in the number of examples, the above procedure is cubic. This makes it suitable only for relatively small data sets.

A mixed approach: COBWEB

The COBWEB algorithm, presented by Douglas Fisher in 1987, is an example of a hierarchical clustering approach that is not purely top-down or bottom-up. It performs a search through the space of cluster hierarchies, employing a set of operators that may lead to division or merging of clusters.

COBWEB is an incremental clustering system: it looks at examples one at a time, and continuously keeps a "current" clustering that reflects the cluster hierarchy that best describes the part of the dataset that the system has seen thus far. When a new data element is processed, the current clustering is adapted based on this new information.

Generally, a good clustering should have the property that elements within a cluster are similar, and elements of different clusters are different. In COBWEB, this principle is instantiated in the concept of **category utility**, which is itself based on the *predictiveness* and *predictability* of attributes values.

An attribute is highly *predictable* in a cluster if most elements in that cluster have the same value for it. For instance, if we would cluster vehicles into bicycles, cars, and trucks, then the attribute "number of wheels" is highly predictable for bicycles and cars, but less so for trucks.

An attribute is highly *predictive* for a cluster if knowing the attribute's value implies you can say with high certainty to which cluster it belongs; in other words, if it does not often occur that elements from different clusters have the same value for that attribute. Returning to the vehicles example, the number of wheels is highly predictive: anything with two wheels is likely to be a bike, with four wheels it's usually a car, with more than four wheels it's very likely to be a truck.

Clearly, high predictability of attributes implies a large intra-cluster similarity, whereas high predictiveness implies a large inter-cluster dissimilarity.

COBWEB defines the quality of a clustering as a combination of the predictiveness and predictability of attributes in the clustering. This goes as follows. We can define the predictiveness of a particular value a_{ij} of some attribute A_i for cluster C_l as $Pr(C_l|A_i = a_{ij})$ (the probability that when an instance has $A_i = a_{ij}$ it indeed belongs to C_l). Similarly, the predictability of the same value given the same cluster is $Pr(A_i = a_{ij}|C_l)$. Fisher proposes the following formula to measure the overall predictability and predictiveness of all attribute values over all (k) clusters:

$$\sum_{l=1}^{k} \sum_{i} \sum_{j} Pr(A_i = a_{ij}) Pr(A_i = a_{ij}|C_l) Pr(C_l|A_i = a_{ij}) \tag{8.3}$$

In words, we compute a weighted average of the product of the predictiveness and predictability of each attribute value, weighted according to the frequency of each value, for each cluster, and sum this over all clusters.

While it may be intuitively clear that the resulting number will be higher if there are many attribute values for which both predictability and predictiveness are high for at least one cluster, the exact meaning of this formula may not seem so clear. It becomes clearer when we rewrite $Pr(A_i = a_{ij})Pr(C_l|A_i = a_{ij})$ as follows:

$$Pr(A_i = a_{ij})Pr(C_l|A_i = a_{ij}) = Pr(A_i = a_{ij} \wedge C_l) = Pr(A_i = a_{ij}|C_l)Pr(C_l)$$

which, substituted in Equation 8.3, gives:

$$\sum_{l=1}^{k} Pr(C_l) \sum_{i} \sum_{j} Pr(A_i = a_{ij}|C_l)^2 \tag{8.4}$$

Defining the predictability of an attribute A_i (as opposed to that of a specific value), within a cluster C_l, as

$$\sum_{j} Pr(A_i = a_{ij}|C_l)^2 \tag{8.5}$$

we see that the above measure can be seen as the average predictability of all attributes in all clusters, weighted according to the size of the clusters. Or, recalling the Gini index that is used by the CART decision tree learner as an impurity measure, and observing that $\sum_j Pr(A_i = a_{ij}|C_l)^2$ is one minus the Gini index of A_i given C_l, we can interpret Formula 8.4 as the average purity of the clusters with respect to all attributes A_i together.

Exercise 8.5 Compare the formula for predictability of an attribute with the Gini index used by the CART decision tree learner in more detail. In particular, try to map the different settings of classification trees (where the Gini index is used to evaluate how predictive an attribute is for the class) and hierarchical clustering (where the Gini index is used to evaluate how predictable the attributes are given the cluster) onto each other.

The measure of **category utility** is now defined as the increase of the predictability of attributes given the clustering, compared to their predictability without clustering, and this divided by the number of clusters. In other words, if we know the cluster that an instance belongs to, how much information does that give us, compared to if we did not know the cluster? The precise definition of the category utility $CU(\mathcal{C})$ of a set of clusters $\mathcal{C} = \{C_1, \ldots, C_n\}$ is

$$\frac{\sum_{l=1}^{k} Pr(C_l) \sum_i \sum_j Pr(A_i = a_{ij}|C_l)^2 - \sum_i \sum_j Pr(A_i = a_{ij})^2}{k} \tag{8.6}$$

Exercise 8.6 What would happen if the division by k is not made?

Now that we have discussed the evaluation measure, we can discuss the COBWEB algorithm itself. As said, COBWEB is an incremental algorithm. It starts with an initial clustering, and adapts its clustering each time it sees a new instance. Adapting the current clustering is done with a number of predefined operators. Each operator takes a clustering of a dataset and a new instances, and returns a modified clustering in which the new instance has been incorporated. COBWEB uses the following operators:

- Include: the new instance is included in an existing cluster
- New: the new instance is added to the clustering as a new cluster
- Merge: two clusters plus the new instance are merged into one cluster; the original clusters remain in the hierarchy as subclusters of this cluster.
- Split: an existing cluster is split into a number of clusters; this happens by replacing it with the subclusters that it has at a lower level in the hierarchy.

Given a clustering and an instance, each of these operators is tentatively applied, and the category utility of the resulting clustering is computed. The operator yielding the clustering with the highest category utility is effectively applied, and COBWEB is recursively applied to the subcluster containing the new element x to optimize its further subclusters.

For the Merge operator, not all possible pairs of clusters need to be tried: only for the two clusters where the new instance fits best, it makes sense to see if merging them improves the category utility. Similarly, the Split operator needs to consider splitting only the cluster in which the new instance fits best.

Divisive clustering using a decision tree algorithm

While decision trees are usually considered to be predictive models, a decision tree can also be interpreted as a hierarchy of clusters. Indeed, by definition, the top node of a decision tree divides the data into disjoint subsets, which can be considered clusters; each subset is then subdivided into smaller subsets, etc., until we arrive at the leaves of the tree, which form the smallest clusters in the hierarchical clustering.

We have seen earlier that in a decision tree, when a node is split into subsets, the split should be such that the subsets are as "pure" as possible with respect to the class attribute, or, more generally, as homogeneous as possible with respect to

function COBWEB(hierarchical clustering $\mathcal{C}$, instance x) **returns** hierarchical clustering:
    let $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$
    // Include operator
    for i := 1 **to** n :
        $\mathcal{C}_{Ii} := \{C_1, \ldots, C_{i-1}, C_i \cup \{x\}, C_{i+1}, \ldots, C_n\}$
    $a :=$ the i for which $UC(\mathcal{C}_{Ii})$ is highest
    $b :=$ the i for which $UC(\mathcal{C}_{Ii})$ is second highest
    $\mathcal{C}_I := \mathcal{C} - \{C_a\} \cup \{\text{COBWEB}(C_a, x)\}$
    // New operator
    $\mathcal{C}_N := \{C_1, \ldots, C_n, \{x\}\}$
    // Merge operator
    $\mathcal{C}_M := \mathcal{C} - \{C_a, C_b\} \cup \{\text{COBWEB}(\{C_a, C_b\}, x)\}$
    // Split operator
    $\mathcal{C}_S := \mathcal{C} - \{C_a\} \cup \text{COBWEB}(C_a, x)$
    // select the best of all the new clusterings
    $\mathcal{C}' :=$ argmax $CU(C)$ with $C \in \{\mathcal{C}_I, \mathcal{C}_N, \mathcal{C}_M, \mathcal{C}_S\}$
    return $\mathcal{C}'$

Figure 8.9: The COBWEB algorithm. Given a clustering $\mathcal{C}$, the operators Include, New, Merge and Split are applied, yielding new clusterings $\mathcal{C}_I$, $\mathcal{C}_N$, $\mathcal{C}_M$ and $\mathcal{C}_S$, among which the best is finally selected. Clusters are here represented not as sets of objects, but as sets of subclusters, so that a hierarchical clustering (as opposed to a flat clustering) is obtained. Pay attention to the use of brackets in the algorithm; compare for instance the formulae for $\mathcal{C}_I$ and $\mathcal{C}_S$.

the target attributes. This homogeneity can be expressed, for instance, using some kind of variance measure: low variance according to the target attributes implies high homogeneity with respect to these target attributes, or: high predictability of the target attributes.

Clustering is different from decision tree learning because it is unsupervised learning: there are no target attributes here. The clusters we need to form should not be homogeneous with respect to the target attributes, but with respect to all attributes. But from the point of view of decision tree learning, this is easily fixed: it just means that the "variance" that is used to indicate homogeneity, should be computed on all attributes instead of just a limited set of attributes (namely the target attributes).

Thus, decision tree induction can easily be seen as a kind of clustering. However, it is important to see that there is a strong restriction on the kind of clusters that can be created. In contrast to the other clustering methods we have seen before, the clusters in a decision tree are clearly defined by means of the values of a few attributes. The top node of a tree might split the data set into three subsets based on the value of attribute A, or in two subsets according to the value of attribute B, etc., but it cannot split the dataset in any possible way (whereas AGGLEHC and COBWEB can). Many possible (extensionally definable) clusterings are not considered, even if they might be good clusterings, because they cannot be defined using a simple condition (e.g., the value of a single attribute) that can be expressed in one node of the tree.

Thus, decision trees can be used for clustering, but one should be aware that they impose quite strong constraints on the clusters. This may be good or bad, depending on the situation. If one is looking for clusters with a simple description, it is clearly an advantage. Another interesting property is that decision tree induction has a computational complexity of $O(Nd)$, with N the number of examples and d

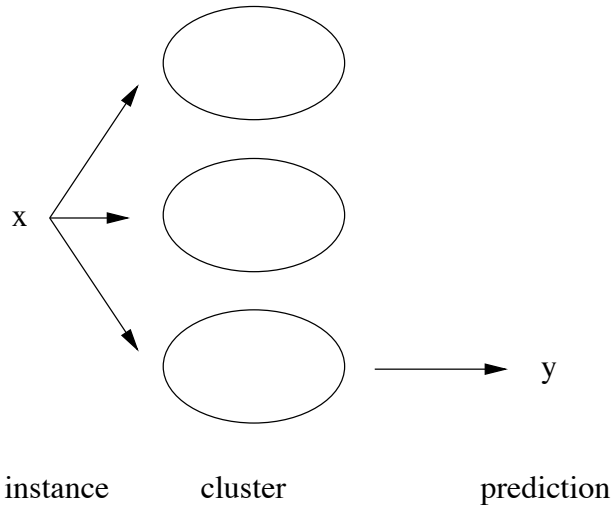

Figure 8.10: Clustering-based prediction. During the learning phase, clusters are formed. In the prediction phase, there are two steps: (1) a new instance is assigned to a cluster; (2) based on the information in the cluster, a prediction is made for the instance.

the depth of the tree, as opposed to $O(N^3)$, which is the complexity of bottom-up methods for extensional hierarchical clustering. In a balanced tree that is fully developed (i.e., the leaves contain only one example), d is proportional to $\log N$, thus decision trees scale much better than bottom-up EHC as far as the number of examples is concerned.

8.4.3 Clustering-based flexible prediction

While clustering is seen as unsupervised learning and as such not directly related to predictive modelling, we can easily think of methods for prediction that are based on clustering.

Generally, we can see clustering-based prediction as a two-step process. Given a set of clusters in the data, the first step consists of assigning the new instance to one of the clusters. The second step consists of predicting the missing information for that instance based on a local model, one that is correct for this cluster. Figure 8.10 illustrates this two-step process. To apply this process successfully, two things are important : (1) given limited information about a new instance, it should be easy to assign it to the right cluster, and (2) given the right cluster, it should be easy to predict the missing information.

Exercise 8.7 Express the conditions (1) and (2) just listed in terms of predictiveness and predictability of attributes that are given and attributes that are to be predicted.

Decision trees are an obvious example of clustering-based prediction. A prediction is made for a new instance by sorting it down the tree (assigning it to one leaf, i.e., one cluster) and then making a prediction based on the information in that leaf. The leaves are constructed in such a way that they are coherent with respect to the information that needs to be predicted (e.g., for classification trees, we want each leaf to consist of examples from one class). The tests in the internal nodes determine what leaf an instance belongs to. Thus, requirements (1) and (2) as listed above are fulfilled.

Also rule-based methods can be seen as an instance of clustering-based prediction. A difference is that the "clusters" defined by the rules may now overlap: an instance can generally belong to more than one cluster (be covered by more than one rule).

Clustering-based prediction can easily be generalized towards so-called **flexible prediction**. In flexible prediction, it is not known in advance what the target attribute is. Models should be built that allow us to accurately predict the value of a missing attribute, whatever that attribute is, from the values of given attributes, whatever those may be. Clusters with a high overall predictiveness and predictability (averaged over all attributes) are likely to perform best in this context. The COBWEB system, with its category utility heuristic, was explicitly designed to find clusterings that are optimal in this respect.

8.5 Bibliographic notes

Hartigan (1975) presents an early overview of clustering algorithms; our description of the LEADER algorithm was inspired by it. The COBWEB system is due to Douglas Fisher (1987) . The EM algorithm was originally proposed by Dempster, Laird and Rubin (1977). Blockeel, De Raedt and Ramon (1998) describe the use of decision trees for clustering. Variants of the k-means clustering approach has been described by many authors; the same holds for the agglomerative clustering methods described here.

Chapter 9

Computational Learning Theory

Computational learning theory, sometimes abbreviated as COLT, studies the complexity of the computations involved in learning. Questions are considered such as:

- how much time is needed to correctly learn a particular concept or model from data;
- how much data is needed for this;
- how do these numbers depend on the complexity of the target model, the complexity of the data instances, or the complexity of the class of models (or the hypothesis space) that is considered
- how much time / data is needed to *approximate* the correct target theory within certain bounds (rather than learn a completely correct theory)

Computational learning theory aims at answering fundamental questions about learning, and gives us a good theoretical grasp of it. It is, however, also a rather technical area, and many of the results obtained in this field are beyond the scope of this text. In this chapter we familiarize the reader with a number of important concepts and intuitions. Much of this chapter is based on Mitchell's discussion of computation learning theory as given in his machine learning textbook [27].

9.1 Settings for learning

In this text, we will focus on the task of learning concepts (functions that map instances to a boolean value); we will disregard other learning tasks such as regression. In the context of concept learning, data elements are usually called examples. An example is an instance $\mathbf{x} \in \mathcal{X}$ together with a boolean value y that indicates whether $\mathbf{x}$ belongs to the target concept or not. An important question in this context is: "How many examples are needed for learning a particular target concept accurately"?

Before we can answer that question, we need to be more precise about a number of points. An obvious one is what we mean by "accurately"? But an even more fundamental question is: in what way are these examples chosen? (We might need fewer examples if we choose them cleverly.)

9.1.1 Who provides the examples?

We distinguish four settings with respect to how examples are provided to the learner.

teacher provides (x)
teacher provides c(x)

1. A benevolent teacher determines which examples are provided to the learner. Such a teacher knows the target concept and also understands how the learner works, and chooses the examples in such a way that the learner can learn as fast as possible the correct target concept. The teacher *guides* the learner towards the right target concept. One could call this learning setting **active teaching**. It has not received very much attention in machine learning, although one could argue that among the settings we discuss here, this one is most similar to how human teachers would teach human students.

learner proposes (x)
teacher provides c(x)

2. The learner can choose the examples to learn from. More specifically, it can propose an instance $\mathbf{x}$ and ask for the class of that instance. In principle, the learner can be aware of its current knowledge and understand what information is missing to extend that knowledge, and use this understanding to select new examples. However, it does not know the target concept; this is the main difference with the previous situation. This setting is usually called **active learning**. The main challenge in active learning, beyond the learning procedure itself, is how to determine which examples to ask for.

3. A (possibly infinite) sequence of examples is available, generated according to some procedure that enumerates the whole instance space, that is: it guarantees that each instance from the instance space occurs somewhere in the sequence. This is only possible if the instance space is *enumerable* (not all spaces are enumerable, see later). No other guarantees are given about the sequence.

   If it can be proven that a learner will learn the target concept correctly after seeing enough examples in such a sequence, we say that the learner *identifies the concept in the limit*. Depending on the complexity of the space of possible target concepts among which the correct concept must be identified, it may be possible to identify it in the limit, or not. Identification in the limit is an important paradigm for characterizing certain types of learners and learning problems, and we will discuss it in some more detail in Section 9.3.

PAC
teacher provides random (x)
teacher provides c(x)

4. Examples are generated randomly. The learner has access to an infinite sequence of examples, which it can read in the order of the sequence. However, each individual element in the sequence is generated randomly and independently from a particular distribution over the instance space. As such, it is possible that the same example reoccurs several times, or that certain examples never occur. In this setting it is typically not possible to guarantee that the learner will ever learn the correct concept, but it may be possible to characterize how likely it is that it finds a "reasonably accurate" approximation. An important notion in this setting is *PAC-learnability*, which we discuss in Section 9.4.

The settings are ordered from easy to difficult. In the first setting, the teacher understands how the learner works and knows the target concept; he can optimize the sequence of examples using this knowledge, to make the learner learn as fast as possible. In the second setting, we assume that the learner knows its current state and understands its own learning behavior, but does not know the target concept. It has strictly less information than the teacher, and therefore cannot optimize its example sequence as well as the teacher could. In the third setting, the example

sequence cannot be optimized at all, but there is a guarantee that if we wait long enough, any example that the learner wants to see will sooner or later show up. Finally, in the fourth setting, even this guarantee is not given, although there is some kind of probabilistic guarantee: the longer the sequence, the more likely it becomes that the needed example occurs somewhere in it (but it will never become certain).

The complexity of learning has been studied in all these different frameworks. In this text we will focus mostly on the third and fourth framework.

9.1.2 Learning "accurately"

The second point that we need to clarify is what it means to learn a target concept "accurately". One possible meaning is that the hypothesis output by the learner equals the target concept. We can relax this requirement somewhat as follows. The *error* of a hypothesis h is defined as

$$Pr_{\mathbf{x}\sim\mathcal{D}}(h(\mathbf{x}) \neq c(\mathbf{x}))$$

i.e., the probability that $h(\mathbf{x})$ is not equal to $c(\mathbf{x})$ when $\mathbf{x}$ is drawn according to a distribution $\mathcal{D}$ over $\mathcal{X}$. The *accuracy* of a hypothesis is then 1 minus this error.

9.2 Some terminology

In the following we will use the following assumptions and terminology. The task we consider is "concept learning", and it can be regarded as a binary classification task. Given a space of instances $\mathcal{X}$, a concept C is a subset of this space. Equivalently, we can say that C is a function from $\mathcal{X}$ to $\{true,false\}$; $C(\mathbf{x}) = true$ is then equivalent to $\mathbf{x} \in C$ and $C(\mathbf{x}) = false$ is equivalent to $\mathbf{x} \notin C$. Depending on the context we will use one notation or the other (whichever is most natural). We say that an element $\mathbf{x} \in \mathcal{X}$ belongs to the concept if $\mathbf{x} \in C$ (or $C(\mathbf{x}) = true$), and does not belong to the concept otherwise.

To quantify how difficult it is to learn a concept, we first need to agree on what the *possible* concepts are, among which we have to choose. After all, learning the right concept out of only two possibilities is much easier than if we start with millions of possibilities. Thus, in all the ensuing problem definitions, we assume that there is a set of concepts (usually called a *class* of concepts) $\mathcal{C}$, one among which is the target concept that we want to learn.

The notion of a class of concepts, from which to identify the correct one, is very similar to that of a hypothesis space, among which to select the correct (or at least most accurate) hypothesis. There is a somewhat subtle difference however: the hypothesis space contains the concept descriptions that the learner uses, and among which it must choose; thus it is a property of a particular learner. The class of concepts, on the other hand, is part of the problem description. We cannot say whether an individual concept is learnable, we can only say whether a class of concepts is learnable. Similarly, how difficult it is to learn a particular concept is related not so much to how complex this concept is, but to how complex the class of concepts is among which it has to be identified.

Example 9.1 Consider a target function $y = 2x_1 - \sin x_2 + 5$. How difficult is it to identify this function from data? If we know in advance that the target function has the form $y = ax_1 + b\sin x_2 + c$, this learning problem is not very difficult. If we know that the true function is one of $\{y = 2x_1 - \sin x_2 + 5, y = 3x_1 - 5, y = 7x_2\}$ (that is, the "concept class" has only three elements, instead of being infinite), finding out which is the correct one is much easier: it may even be possible from one example.

Finally, if we only know that the true function is a continuous function, it will be very difficult to identify the correct function.

The concept learning problem can now formally defined as follows.

Definition 9.1 (Task: Concept learning) *Given a class of possible concepts $\mathcal{C}$, and a set T of examples $(\mathbf{x}, C(\mathbf{x}))$ for some $C \in \mathcal{C}$, identify C.*

In the following sections we will study the difficulty of this task, from different points of view.

9.3 Identification in the limit

Identification in the limit is a notion that was originally proposed in the context of learning languages or grammars from sentences (sequences of symbols), which is a special kind of concept learning. We here reformulate it in the more general concept learning context.

Take a learner L and a (possibly infinite) set of examples $T = \{\mathbf{u}_1, \mathbf{u}_2, \ldots\}$ where $\mathbf{u}_i = (\mathbf{x}_i, C(\mathbf{x}_i))$, with C the target concept. We define $T_{[i,j]} = \{\mathbf{u}_i, \mathbf{u}_{i+1}, \ldots \mathbf{u}_j\}$ as the subset of T containing its i'th to j'th element. For any set S, let $L(S)$ denote the result of applying L to S.

Definition 9.2 (Identification in the limit) *L identifies C in the limit from a set T if and only if there exists a number $n \in \mathbb{N}$ such that for any set $S \supseteq T_{[1,n]}$, it holds that $L(S) = C$.*

In plain English: as soon as the learner L has seen the first n elements of T, it has correctly learned C, and seeing more elements will not make it change its mind again.

Definition 9.3 (Enumeration) *An* **enumeration** *of a (finite or infinite) set S is a sequence $s_1, s_2, \ldots$ such that each element of S occurs somewhere in the sequence. S is* **enumerable** *if an enumeration for it exists. (A synonym for enumerable is "countable".)*

Example 9.2 The sequence [1,2,3,4,5,6] is an enumeration of the set of possible outcomes of a die. Similarly, [heads, tails] is an enumeration of the possible outcomes of a coin toss. The infinite sequence [1,2,3,4,...] enumerates all natural numbers, but the infinite sequence [2,4,6,8,...] does not (it only enumerates the even numbers; 3, for instance, will never occur in this sequence).

All finite sets are enumerable, but not all infinite sets are. It is well-known that the set of all natural numbers is enumerable, and so are the sets of integers and of rational numbers, but the set of real numbers is not. Generally, given an alphabet Σ, the set of all finite strings over this alphabet (denoted Σ^*) is enumerable, while the set of infinite sequences over Σ is not.

Definition 9.4 (Identifiability in the limit) *A class of concepts $\mathcal{C}$ is identifiable in the limit if and only if there exists a learner L such that for each $C \in \mathcal{C}$, and for each enumeration $\mathbf{x}_1, \mathbf{x}_2, \ldots$ of the instance space $\mathcal{X}$, there exists a finite number n such that L is guaranteed to identify C from $\{(\mathbf{x}_1, C(\mathbf{x}_1), (\mathbf{x}_2, C(\mathbf{x}_2), \ldots, (\mathbf{x}_n, C(\mathbf{x}_n))\}$.*

Example 9.3 A string is a finite sequence of elements from an alphabet Σ. Let the instance space $\mathcal{X}$ be the set of all such strings. A language is a set of strings; in our case, it is a subset of $\mathcal{X}$. Thus, in concept learning terminology, a concept is equivalent to a language, and a concept class is equivalent to a class of languages.

$\mathcal{X}$ is enumerable, since the set of all finite strings is enumerable. Consider then a sequence $T = [(\mathbf{x}_1, C(\mathbf{x}_1)), (\mathbf{x}_2, C(\mathbf{x}_2)), \ldots]$ such that $[\mathbf{x}_1, \mathbf{x}_2, \ldots]$ enumerates $\mathcal{X}$. What classes of concepts, or classes of languages, are identifiable in the limit?

One of the theoretical results on language learning is that the class of all regular languages is identifiable in the limit. This means that, for instance, if we know that the target language we want to learn is regular, then it is possible to make an algorithm that, from a set of examples such as $\{(ab, +), (abab, +), (aa, -), (aba, -), (bbab, -), \ldots\}$ (with $+$ and $-$ denoting *true* and *false*), and assuming the set is large enough, can learn the regular language given by the expression $(ab)^*$ (which denotes the language containing all strings that just repeat the substring ab any number of times).

There are many theoretical results related to identification in the limit, and the concept is still very relevant for learning from sequences. In many practical machine learning settings, however, it cannot be used because the instance space $\mathcal{X}$ may not be enumerable; even if it is, there may not be a practical process available that enumerates $\mathcal{X}$; the number of instances to be inspected by the algorithm may be unrealistically large in practice; and we may be interested in relaxing the requirement that the hypothesis is provably correct towards a requirement that it is "sufficiently close" to the target concept. The PAC learning setting will address these concerns.

9.4 PAC Learning: learning models that are Probably Approximately Correct

The setting of identification in the limit assumes that sooner or later every example that is relevant for the learning task will occur. In practice, we often have no control over the way in which instances are presented to the learner. They may be presented in a random order, some examples may occur multiple times, others may never occur at all, etc. There is no guarantee, then, that the learner will in the limit converge to the right target concept. Further, with identification in the limit there is no indication of when the learner will finally have learned the target concept; even if we know it will happen at some point in time, there may never be a moment when we can be reasonably certain that the target concept has been found.

Consider now the case where examples are not listed according to some enumeration of the instance space $\mathcal{X}$, but drawn randomly from a certain distribution $\mathcal{D}$ over $\mathcal{X}$. It is often possible to show, in this setting, that after seeing some number of examples N, it is very likely (though not certain) that the target hypothesis has been learned correctly, or that at least a good approximation of the target hypothesis has been learned. This situation is addressed in the PAC learnability framework.

"PAC" stands for "Probably Approximately Correct". When we say that in a given learning problem the target hypothesis is PAC-learnable, we mean that there exists an algorithm that with high probability, but not certainly (hence *Probably*), will learn a hypothesis that is *Approximately Correct*, which means that there may still be a small difference between the learned hypothesis and the target hypothesis, but it should be small.

This explanation is of course still a bit vague. We say that the algorithm should have "high" probability to learn a hypothesis that is "close" to the target hypothesis; but how high should this probability be, and how close should the hypothesis be?

It is not possible to give a generally valid answer to this. The concept of "close enough" will depend on the application. Let us consider classification tasks only, where a predictive model is to be learned that makes discrete predictions, which

may be correct or incorrect. In some cases one might be happy with a predictive model that has an accuracy of 99%; in other cases one may require 99.99%, or even 99.9999999999% (which implies only one in a trillion predictions would be wrong). Similarly, when we say that we want to find with "sufficiently high probability" a hypothesis that fulfills this condition, this may mean we want to be 99% certain, or perhaps 99.9999%, that the hypothesis indeed has the required accuracy.

Generally, we can indicate the first number, the probability that the learned hypothesis makes a correct prediction, as $1 - \epsilon$ with ϵ a very small number (we use here a scale from 0 to 1, not from 0 to 100). Similarly, we can write the second number as $1 - \delta$, with δ again being a very small number.

For ease of writing, we will say that a hypothesis is **ϵ-correct** if it approximates the target hypothesis well enough, according to ϵ, that is: if it makes the same prediction as the target hypothesis with a probability of at least $1 - \epsilon$ (or, in other words: if its error is at most ϵ).

Now, generally, we will say that the target hypothesis in a given problem is PAC-learnable if there exists an algoritm that for *any* δ and for *any* ϵ, no matter how small we choose them, is able to learn an ϵ-correct hypothesis with probability $1 - \delta$.

The above can of course only be stated under the assumption that there are enough data available to learn from. In fact, the difficulty of a learning problem can be expressed as the number of examples that one needs to look at before one can be almost certain (with probability $1 - \delta$) that an ϵ-correct hypothesis has been learned.

This leads us to the following definitions.

Definition 9.5 (PAC-learnability) *Given a class $\mathcal{C}$ of concepts over an instance space $\mathcal{X}$ over which a probability distribution $\mathcal{D}$ is defined, we say that $\mathcal{C}$ is PAC-learnable if there exists a learner L, such that for all $\delta \in (0, 1]$ and for all $\epsilon \in (0, 1]$, there exists an N such that, when given a random dataset T with at least N elements (sampled i.i.d. from $\mathcal{X}$), there is a probability of at least $1 - \delta$ that L returns a hypothesis h such that $Pr_{\mathbf{x} \sim \mathcal{D}}(h(\mathbf{x}) \neq C(\mathbf{x})) < \epsilon$, that is, the error of h w.r.t. C is at most ϵ.*

It is clear that, the smaller ϵ and δ are, the larger N will typically be. We can therefore consider N a function of these parameters. This function is called the sample complexity of the learning task.

Definition 9.6 (Sample complexity under PAC-learnability.) *The minimal number of examples N that a PAC-learning algorithm needs to see to guarantee that the learned hypothesis is ϵ-correct with probability at least $1 - \delta$, expressed as a function of $1/\epsilon$ and $1/\delta$, is called the sample complexity of a concept class.*

9.5 Expressiveness of hypothesis spaces: The Vapnik-Chervonenkis dimension

The Vapnik-Chervonenkis dimension, usually briefly referred to as VC-dimension, is a property of a hypothesis space or concept class; more specifically, it is a measure of how expressive the hypothesis space is, or how difficult to learn the concept class is.

To get some intuition about what the VC-dimension expresses, let us first consider finite hypothesis spaces. Generally, we can assume that the more hypotheses a hypothesis space contains, the more expressive it is, i.e., the more semantically different hypotheses it can represent.

It is useful here to distinguish representations of hypotheses, and the actual hypotheses (functions or concepts) they represent. For instance, the rules

if A and B **then** positive
if B and A **then** positive

both have the same meaning, even they are written differently. They are *different representations* of the *same function*. Also in the context of decision trees, it is easy to write down two different trees for the concept "A **xor** B", one with A in the root and two B nodes lower down, and one with B in the root and two A nodes below.

Thus, we could say that the expressiveness of a hypothesis space is not really related to the size of the hypothesis space itself (in terms of how many different hypothesis representations it contains), but rather to the number of different concepts or functions that can be expressed by these representations.

If we have an instance space $\mathcal{X}$, and consider as possible concepts any subset of $\mathcal{X}$, then there are $2^{|\mathcal{X}|}$ different concepts. Depending on the learning method we use (conjunctive concepts, rules, trees, ...), our hypothesis space may be able to represent all of them, or only some of them. The more of these concepts can be represented, the more expressive the hypothesis space is.

Just counting the number of concepts may not work well, however, especially when we have an infinite hypothesis space. If we have two hypothesis spaces, both of which can represent an infinite number of concepts, are they automatically equally expressive? Intuitively, this seems unlikely. For instance, it is reasonable to say that the class of thresholded quadratic functions:

$$f(x) = \textit{true} \text{ if } ax^2 + bx + c > t, \text{and } \textit{false} \text{ otherwise}$$

(call this class Q) is more expressive than the class of thresholded linear functions L:

$$f(x) = \textit{true} \text{ if } ax + b > t, \text{and } \textit{false} \text{ otherwise.}$$

Indeed, there exist parameters a, b, c, t for which no a', b', t' exists such that $ax^2 + bx + c > t \Leftrightarrow a'x + b' > t'$. Thus, the class Q of thresholded quadratic functions can represent some concepts that the class L of thresholded linear functions cannot represent. The opposite is not the case: all functions representable in L can be represented in Q.

While Q can represent "more" functions than L, it is difficult to quantify this, since both have "equally many" functions, namely infinitely many. One could argue that Q has four parameters while L has only three, therefore the cardinality of Q is $|\mathbb{R}|^4$ while that of L is only $|\mathbb{R}|^3$, therefore Q is more expressive. But this "calculating with infinities" does not work well: it is difficult to say when an infinity is really greater than another infinity. Moreover, one can easily find classes of functions with an equal number of parameters (hence equal "cardinality") that are yet more or less expressive. For instance, the function class LL:

$$f(x) = \text{true if } |a|x + |b| \log(x) + c > t, \text{and false otherwise}$$

has four parameters, like Q, but is in a sense less expressive than L. Note that $\log(x)$ and x are both monotonically increasing, so the function f as a whole is monotonically increasing, whereas the linear functions in L are monotonically increasing or decreasing, and those in Q are non-monotonic.

Clearly, we need a more practical way to decribe the expressiveness of a hypothesis space. The Vapnik-Chervonenkis dimension will give us that. But before we define it, we need to introduce the concept of shattering.

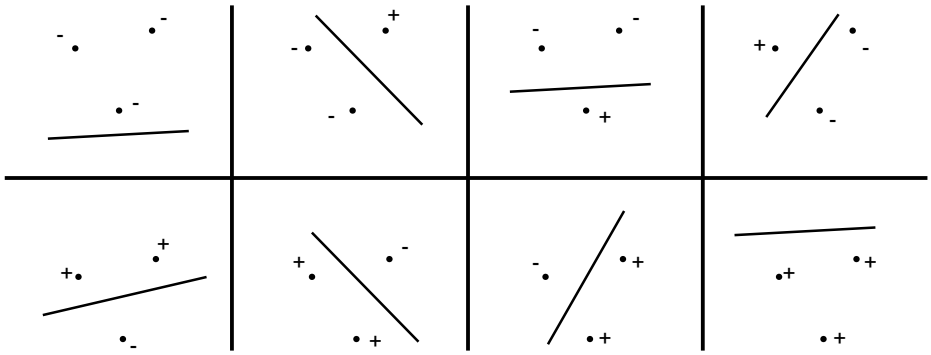

Figure 9.1: The hypothesis space consisting of all half-planes in $\mathbb{R}^2$ shatters the set of three points shown. There are eight different ways of labeling those three points. For each labeling of the points a straight line can be drawn such that all the positive points are on one side and all the negative points are on the other side of it.

9.5.1 Shattering

Shattering is a property of a set of concepts (or boolean functions, or hypotheses). We can say that a hypothesis space shatters a set of data points, or that a concept class shatters it. In the following part we will define shattering for a hypothesis space H, but it can just as well be used for a concept class $\mathcal{C}$.

Definition 9.7 (Shattering) *Given an instance space $\mathcal{X}$, a hypothesis space H and a finite set of points $T \subseteq \mathcal{X}$, we say that H shatters T if and only if for each subset S of T there is a $h \in H$ such that $h(\mathbf{x}) = \text{true} \Leftrightarrow \mathbf{x} \in S$.*

In words, a hypothesis space H shatters a set of instances T if for each subset of T, H contains a hypothesis that represents a concept that on T coincides perfectly with this subset. Or: whatever the labeling (positive/negative) of the elements of T is, H always contains a hypothesis that is consistent with T.

Example 9.4 Figure 9.1 illustrates the definition for a 2-dimensional $\mathcal{X}$. We consider the three points shown, and a hypothesis space consisting of all rules of the form $f(\mathbf{x}) =$ **if** $ax_1 + bx_2 > t$ **then** *true* **else** *false*. Each such rule separates the concept instances from the non-concept instances by a straight line. For each possible labeling of the points as true or false, there exists a hypothesis that is consistent with that labeling.

Example 9.5 Consider a 2-dimensional boolean input space and a hypothesis space consisting of all conjunctive concepts over the two input variables. Consider the three points shown in the left part of Figure 9.2. These points are not shattered by the hypothesis space; there is a particular labeling of the points that does not correspond to a conjunctive concept.

Exercise 9.1 Show that the set of all boolean formulas over the two input variables does shatter the three points in Figure 9.2.

Example 9.6 Applying the notion of shattering to the thresholded linear and quadratic functions, it is easy to see that class L shatters any pair of two points

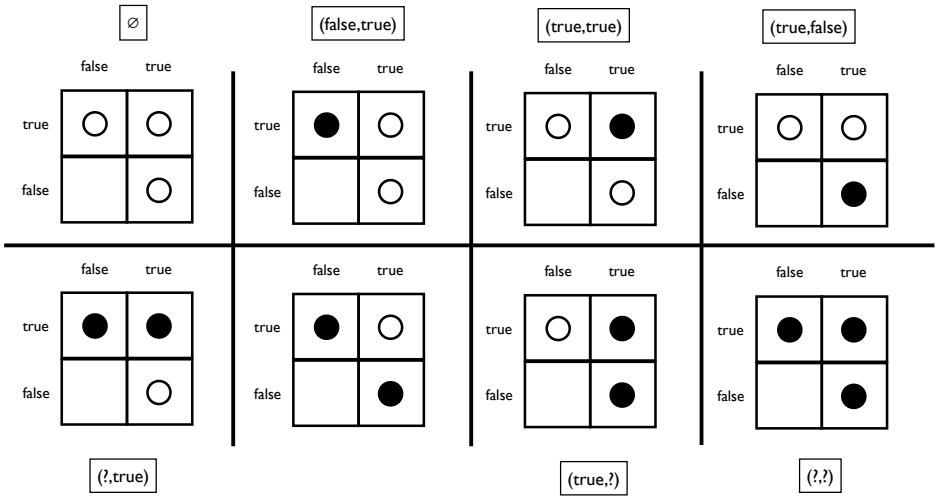

Figure 9.2: A set of three points in $\mathbb{B}^2$ that is not shattered by the class of all conjunctive concepts over $\mathbb{B}^2$. We show eight possible labelings of the points (black means the point belongs to the concept, white means it does not). For each labeling we show a conjunctive concept equivalent to it, if there exists one. There is one labeling of the points for which no equivalent conjunctive concept exists.

(as long as they are different), but does not shatter any set of three points. Indeed, if $x_1 < x_2 < x_3$, then for the labeling $C(x_1) = \mathit{true}, C(x_2) = \mathit{false}, C(x_3) = \mathit{true}$ (or equivalently, for the subset $\{x_1, x_3\}$), there is no thresholded linear function consistent with it. The class Q shatters any triplet of points.

Exercise 9.2 Show that the class LL shatters any set with one point, but never shatters a pair of two points.

It is important to remember that a hypothesis space shatters a set of points if it contains for *each* subset a hypothesis consistent with that subset. It is not sufficient to find just one subset for which a consistent hypothesis exists.

A direct consequence of this is the following property.

Proposition 9.1 *If H shatters a set of n points, then H must contain at least 2^n hypotheses.*

Proof: This is trivial: there are 2^n different subsets of T, and for each of these a different hypothesis is needed, so we must have at least 2^n hypotheses. □

The converse does not hold: if a hypothesis space has 2^n or more hypotheses, that does not imply that it can shatter n points. Many of the different hypotheses may just yield exactly the same subset. This is clear in Example 9.6: the class of thresholded linear functions cannot shatter three points, even though it has an infinite number of hypotheses. The point is that many of these hypothesis look exactly the same on T: they yield an identical labeling on it. These hypotheses cannot be distinguished by looking only at T.

9.5.2 VC-dimension

We now finally come to the definition of VC-dimension.

Definition 9.8 (VC-dimension) *The Vapnik-Chervonenkis dimension, or VC-dimension, of a hypothesis space H is the cardinality of the largest set of points that is shattered by H. If sets of arbitrary cardinality can be shattered, the VC-dimension is infinite.*

Note that as soon as one set of n points is shattered, the VC-dimension is at least n. It is not necessary that all sets of n points can be shattered.

Example 9.7 The VC-dimension of the thresholded linear functions (class L) in Example 9.6 is 2, while that of the thresholded quadratic functions (class Q) is 3. Indeed, L shatters at least one set of two points, but does not shatter any set of three points. The class Q shatters at least one set of three points, but does not shatter any set of four points. (The first part of these statements is easily checked in the examples; the second part is more difficult to prove.)

It is often easier to show that the VC-dimension of a hypothesis space is at least n (we just need to find an example of a set of n points that is shattered) than to show it is at most n (here we need to prove that no set of $n+1$ points exists that is shattered).

For finite hypothesis spaces, one upper bound for the VC-dimension can easily be derived:

Proposition 9.2 $VC(H) \leq \log_2(|H|)$.

Proof: This follows immediately from the fact that to shatter n examples, we need at least 2^n hypotheses. Since a hypothesis space H by definition shatters $VC(H)$ examples, we have $|H| \geq 2^{VC(H)}$, from which the above bound follows. □

The higher the VC-dimension of a hypothesis space, the larger the sets of examples that it can fit perfectly. Thus, VC-dimension is a measure of expressiveness that does not just measure the number of hypotheses, but the number of "significantly different" hypotheses, where "significantly different" implies that the hypotheses result in a different way of separating a concrete set of points.

Example 9.8 Linear separators in an n-dimensional space have a VC-dimension of $n+1$.

Example 9.9 When learning from boolean data, where an instance x is described by boolean attributes A_1 to A_n, consider as possible hypotheses H any conjunctions over conditions of the form A_i or $\neg A_i$. There are 3^n such hypotheses (for each of the n attributes there are three possibilities: it occurs positively, negatively, or not at all in the conjunction). Hence, the VC-dimension of this hypothesis space is at most $\log_2 3^n = n \log_2 3$, rounded down.

A sharper bound can be obtained by observing that the boolean "and" function is linearly separable. Linear separators in an n-dimensional space have a VC-dimension of $n+1$; there does not exist a linear separator that shatters $n+2$ points, and hence conjunctions of n attributes cannot do so either. So we have $VC(H) \leq n+1$, which together with the previous observation becomes $VC(H) \leq min(n+1, \lfloor n \log_2(3) \rfloor)$.

9.6 Sample complexity of concept learning tasks

Using the notions of cardinality of a hypothesis set (for finite hypothesis sizes) and VC-dimension, it is possible to derive certain number of bounds on the number of examples needed to PAC-learn a hypothesis.

9.6.1 Finite hypothesis space

Consider the case where the hypothesis space used by the learner is equal to the class of concepts: $H = \mathcal{C}$. Assume there is no noise in the data. The version space $VS(H,T)$ contains all hypotheses in H that are consistent with the data T seen up till now. When T is empty, it contains all of H: $VS(H,\emptyset) = H$. As data become available, each data element is an additional constraint on the version space: all hypotheses that predict a different outcome for the data element than the observed outcome are removed from the version space. Any hypothesis that is still in the version space at a given moment could be the correct one. If the learner picks a random hypothesis in the version space and suggests that one as the target hypothesis, what is the probability that it has an error of more than ϵ? This probability, which will be a function of the data set size, should be less than δ.

We can characterize that probability as follows. Take a hypothesis h with error ϵ. For a single random instance, the probability that h predicts it correctly is $1-\epsilon$. For n random instances, the probability that h predicts all of them correctly is $(1-\epsilon)^n$. Hence, if we have a dataset T with n instances and a hypothesis h with error ϵ, the probability that $h \in VS(H,T)$ is $p_\epsilon = (1-\epsilon)^n$.

If there are x hypotheses with error ϵ, each of them has a probability p_ϵ of being in $VS(H,T)$. The probability that none of them is in $VS(H,T)$ is then $1-(1-p_\epsilon)^x$. If there are x hypothesis with error *at least* ϵ, then this probability must be *at least* $1-(1-p_\epsilon)^x$ (since hypotheses with a greater error are more likely not to be in $VS(H,T)$).

We say that the version space is **ϵ-exhausted** if it does not contain any hypotheses with an error of ϵ or higher. The above reasoning shows that if the hypothesis space contains x hypotheses with error at least ϵ, the probability that the version space is ϵ-exhausted, which we will denote P_ϵ, is at least $1-(1-p_\epsilon)^x$ with $p_\epsilon = (1-\epsilon)^n$.

In general we do not know x, but it is obvious that it cannot be more than the total number of hypotheses, $|H|$. Since $x \leq |H|$, we have

$$P_\epsilon \geq 1-(1-p_\epsilon)^x \geq 1-(1-p_\epsilon)^{|H|} = 1-(1-(1-\epsilon)^n)^{|H|}$$

Because this is a rather complicated formula, it is usually approximated. First, we can relax the bound by noting that generally $1-(1-p)^n \leq np$. (The first is the probability of a disjunction of n independent events with probability p, the second is an overestimate of that probability.) Applying this to the above formula, we have $1-(1-p_\epsilon)^{|H|} \leq |H|p_\epsilon$, which after substitution in the formula gives

$$P_\epsilon \geq 1-|H|p_\epsilon = 1-|H|(1-\epsilon)^n$$

To ensure that P_ϵ is at least $1-\delta$, it is sufficient to demand

$$1-|H|(1-\epsilon)^n \geq 1-\delta$$

and hence

$$|H|(1-\epsilon)^n \leq \delta$$

which gives

$$(1-\epsilon)^n \leq \delta/|H|$$

$$n \log(1-\epsilon) \leq \log \delta - \log |H|$$

$$n \geq (\log \delta - \log |H|)/\log(1-\epsilon)$$

(The inequality in the last line changes because $\log(1-\epsilon)$ is negative). Changing the sign of enumerator and denominator, we get

$$n \geq (\log |H| - \log \delta)/(-\log(1-\epsilon))$$

and, using the property that $-\log x = \log(1/x)$,

$$n \geq (\log |H| + \log(1/\delta))/\log(1/(1-\epsilon))$$

We can make some further approximations: when ϵ is close to 0, $1/(1-\epsilon) \simeq 1+\epsilon$, and $\log(1+\epsilon) \simeq \epsilon$. Applying both in sequence yields

$$n \geq \frac{\log |H| + \log \frac{1}{\delta}}{\epsilon}$$

Thus, we finally arrive at the conclusion that the number of instances n needed to have a probability of at least $1-\delta$ that the version space is ϵ-exhausted, is logarithmic in the size of the hypothesis space $|H|$, logarithmic in $1/\delta$, and linear in $1/\epsilon$.

9.6.2 Infinite hypothesis space

When the hypothesis space is infinite, the above bound is not very useful. The VC-dimension of H can provide alternative bounds in this case.

Multiple bounds have been derived using the VC-dimension, and deriving them is significantly more challenging than the bound in terms of $|H|$ that we derived above. We limit ourselves here to listing a few examples of bounds that have been derived.

It has been shown that the version space is ϵ-exhausted with a probability of at least $1-\delta$ if the number of examples satisfies

$$n \geq 1/\epsilon(4 \log_2(2/\delta) + 8VC(H) \log_2(13/\epsilon)).$$

Besides describing the sample complexity, one can also use VC-dimension to bound the error of a hypothesis. If we learn a hypothesis h from a hypothesis space with VC-dimension $VC(H)$, where $VC(H) < n$, and h turns out to have an error on the training set of err_T, then with probability $1-\eta$ the true error err is bounded by

$$err \leq err_T + \sqrt{\frac{VC(H)(\log(2n/VC(H)) + 1) - \log(\eta/4)}{n}}.$$

9.7 Other characterizations

The difficulty of a learning task is naturally described in terms of sample complexity, the total number of examples needed to learn the target concept sufficiently accurate. However, this is not the only possible way in which it can be characterized.

An example of an alternative setting is the following. Consider the so-called online learning setting, where the learning phases and the prediction phase overlap. This might be the case when we have a robot that is performing a certain task,

but keeps on learning while performing that task. In the concept learning setting, consider a learner that is making predictions and while doing this receives feedback: if a prediction is wrong, it is corrected by a teacher. How long it will take before the correct concept is learned, is less relevant than how many mistakes the learner will make before it has learned the correct concept. If we know that the learner will not make more than 10 mistakes before it has learned the correct concept, we may not care whether those 10 mistakes occur sooner or later in the learning process. What counts is the total number of mistakes that may be made.

Example 9.10 Consider a learner that keeps track of the version space $VS(H, T)$ while seeing new examples. In the online learning setting, when seeing a new example, this learner will first make a prediction for that example. If the prediction is wrong, the learner is corrected. A reasonable way of making predictions with a non-singleton version space is the following: predict whatever the majority of the hypotheses in the version space predict. In this way, each time a prediction turns out wrong, this means that at least half of the hypotheses in the version space were wrong. All of these are now removed from the version space. Thus, with each mistake made by the learner, the version space is at least halved (i.e., at most half of the hypotheses remain). Because of this property, this learning algorithm is called the Halving algorithm.

Since after m mistakes the size of the version space has been divided by at least 2^m, the total number of mistakes cannot be larger than $\log_2(|H|)$: after this number of mistakes have been made, at most a single hypothesis remains in the version space.

In this setting, the difficulty of learning (or rather, the cost involved in learning) can be expressed as the **optimal mistake bound**. Given a concept class $\mathcal{C}$, the optimal mistake bound $Opt(\mathcal{C})$ is the minimal number of mistakes that will be made by any learning algorithm in this setting, before it has learned the correct target concept.

Since the halving algorithm makes at most $\log_2(|H|)$ mistakes, also $Opt(H)$ must be at most $\log_2(|H|)$. In fact, it is known that in general, for a concept class $\mathcal{C}$, the following inequality holds:

$$VC(\mathcal{C}) \leq Opt(\mathcal{C}) \leq \log_2(|\mathcal{C}|)$$

9.8 Exercises

Exercise 9.3 Consider a hypothesis space with VC-dimension v, and a data set with N examples. Indicate for each of the following statements whether they are true or false:

- If $N \leq v$, the version space is guaranteed to be non-empty.
- If $N > v$, the version space is guaranteed to be empty.
- There exists a dataset with N examples with $N \leq v$ for which the version space is non-empty.

Exercise 9.4 Consider a concept learning task where the hypothesis space H has a VC-dimension of v. Show that there exists at least one training set T of size $v-1$ for which at least two hypotheses are in the version space $VS(H,T)$ that make a different prediction on at least one unseen instance.

Chapter 10

Artificial Neural Networks

Neural networks, as a modelling tool, have had an important influence on machine learning, as well as other areas, for decades. In this chapter we briefly review a number of basic elements that define neural networks. A complete treatment of neural networks is beyond the scope of this text; whole textbooks have been written on just this subject. Instead, we provide a high-level overview of the basics of neural networks and put them in context with respect to the other machine learning techniques discussed in this text.

10.1 Neural networks

While in this text the term "neural network" usually refers to artificial neural networks, the term can in fact also be used for biological networks of neurons, as they are found in many organisms. The human neural system, including the brain, is an example of such a neural network.

A neural network is composed of a number of neurons. A biological neuron is a cell with a very characteristic form. It has a compact nucleus with a "spiked" surface; these spikes are called dendrites, and have a somewhat tree-like shape, branching into sub-branches which again have sub-branches, and so on. A neuron has a single long "tail", called the axon, which at the end branches into "subtails". The non-branching part of the axon is covered by a substance that, among other things, isolates it from the outside world. Figure 10.1 shows a sketch of a neuron.

While the behavior of neurons is complicated (as in many biological processes, practically all general "rules" that can be stated have exceptions to them), the following is a good approximation of how they work. The dendrites are a receptor for certain chemicals in their environment. If enough of these chemicals touch the dendrites, the nucleus sends an electrical pulse over the axon, causing it to release chemicals at its end points. Thus, the axon can be seen as conveying information (the presence of chemicals) from one place (the dendrites, which are near the nucleus) to another place (the endpoints of the axon). However, the nucleus builds in a kind of filter: the output produced by the neuron (the chemicals released at the end of the axon) is not a linear function of the sum of the inputs. When the sum of the inputs is too low, no output is produced. Only when this sum exceeds a certain threshold, the neuron will "fire", its axon releasing chemicals.

Neurons do not occur in isolation; the dendrites of a neuron are typically physically close to the axon of other neurons. As such, when a neuron fires, the chemicals it releases may be picked up by other neurons. In this way, a network of neurons is formed, where the different neurons pass information to one another.

A single neuron is a relatively simple structure from a computational point of

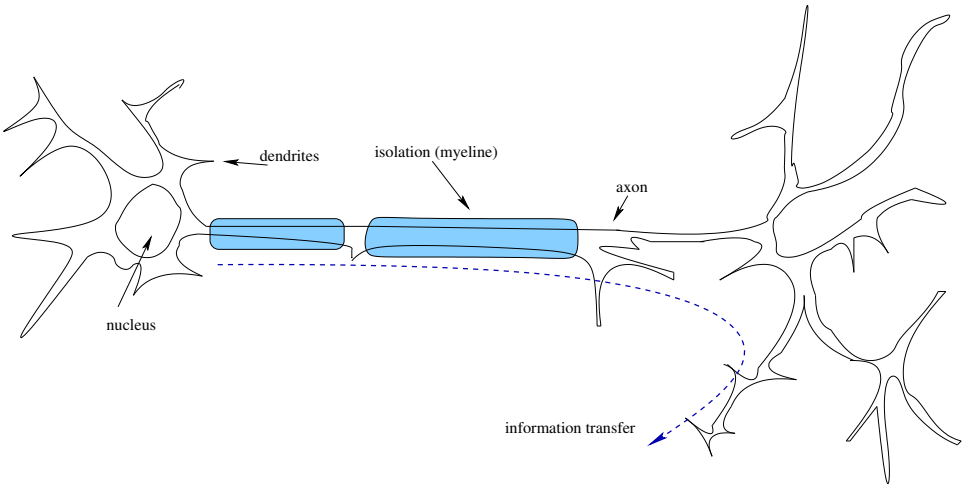

Figure 10.1: A biological neuron.

view. However, it turns out that a network of neurons can perform quite complex computations. **Artificial neural networks** are inspired by this: they consist of a network of artificial "neurons", each of which is relatively simple in itself, while the network as a whole is a quite powerful modelling tool.

We start our discussion with a particular but very important kind of neural network, the "perceptron", which in its simplest form consists of a single neuron. We will discuss the representational power of a single-neuron perceptron, and methods for training it so that it represents a function that fits the data on which it is trained. Next, we will have a look at networks of perceptrons, more specifically, so-called multi-layer perceptrons. Finally, a brief discussion of a rather different type of neural network, the Kohonen network, is added.

10.2 Perceptrons

10.2.1 The single-neuron perceptron

The simplest perceptron consists of a single artificial neuron. An artificial neuron is a simple tree structure that has a number of input nodes and a single output node, which is connected to each input node. In Figure 10.2, this structure is shown; the output node is represented here with some internal structure rather than as a single point.

With each input node n_i, $i = 1, \ldots, D$, is associated a numerical value x_i, which in principle can be any real number. With each connection is associated a weight w_i, again this can be any real number. With the output is associated a number y that is a function of a weighted sum of the inputs, as follows:

$$y = f(\sum_{i=1}^{D} w_i \cdot x_i)$$

with f a so-called "transfer function", typically chosen among a small set of suitable functions.

A trivial case is when f is the identity function, $f(x) = x$. In that case y is a linear combination of the x_i. But, as said, a biological neuron has a certain treshold built into it; it fires only when the sum of its inputs exceeds that threshold. An

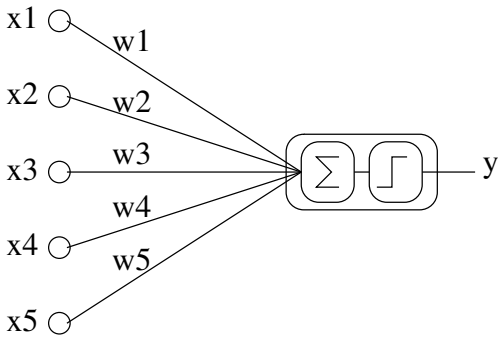

Figure 10.2: An artificial neuron.

artificial neuron can mimic this by using a **step function** for f:

$$\begin{aligned} f(x) &= 0 \quad \text{if} \quad x < 0 \\ f(x) &= 0.5 \quad \text{if} \quad x = 0 \\ f(x) &= 1 \quad \text{if} \quad x > 0 \end{aligned}$$

(Usually the step function is defined with $f(0) = 1$ instead of 0.5. We define it as 0.5 because this slightly simplifies some reasoning later, but this is not an important point in practice.)

Such a step function causes the output to be a discontinuous function of the inputs, which can cause problems with mathematical processing of these structures. For that reason, a continuous variant called a "sigmoid" function is often used. One example of such a sigmoid function is the so-called **logistic function**,

$$f(x) = \frac{1}{1 + e^{-x}}$$

which approximates 0 if $x \to -\infty$ and 1 if $x \to \infty$, and rises relatively quickly from nearly-zero to nearly-one in an area around $x = 0$.

An alternative function that has roughly the same shape is

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

which approximates -1 for $x \to -\infty$ and 1 for $x \to -\infty$, and again rises relatively steeply in the area near $x = 0$.

The step, logistic and tanh functions are plotted in Figure 10.3. As the tanh function behaves quite similar to the logistic function, we will mostly disregard it in the remainder of this chapter, focusing on the logistic function as a representative example of smooth approximations of the step function.

It is instructive to see how the output of an artificial neuron changes as a function of its multiple inputs. This is easy to visualize for two-dimensional inputs. Figure 10.4 shows how the output changes with the inputs for a particular configuration of the weights.

Using perceptrons for classification

Like other machine learning methods that produce numerical outputs, a perceptron can discriminate two classes by agreeing that any output above a certain threshold, for instance 0 or 0.5, indicates that the input instance belongs to one class (the

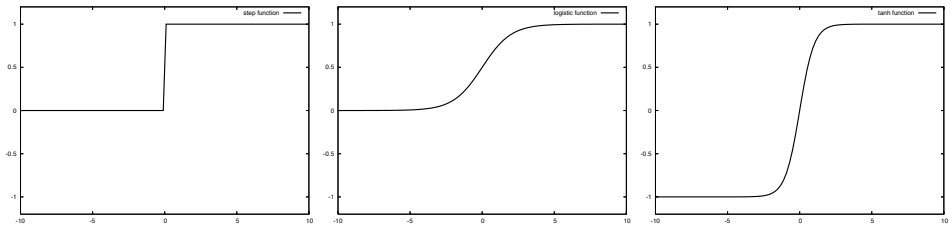

Figure 10.3: Transfer functions for artificial neurons. Left: step function. Middle: logistic function. Right: tanh function.

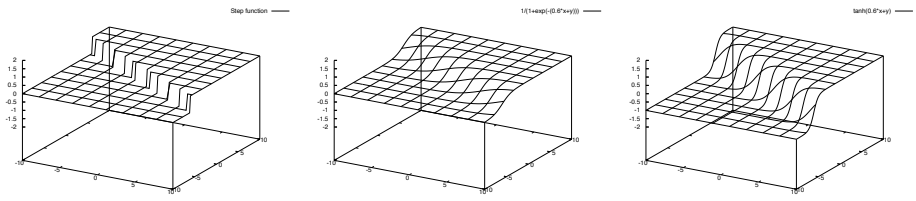

Figure 10.4: Output of the neuron as a function of its inputs, for a two-dimensional input space, assuming weights $w_1 = 0.6$ and $w_2 = 1$.

positive class), while an output below the threshold indicates that the corresponding input is a member of the negative class. The line where the output equals the threshold then indicates the boundary between the classes. This line is shown in Figure 10.5 for a threshold of $f(0)$, which is 0 for the tanh function, and 0.5 for the sigmoid function and the step function.

Note that in all cases, the boundary between classes is a straight line. Indeed, due to the fact that the argument of f is a linear combination of the inputs, for any threshold c the set of points for which the output equals c is given by:

$$f(\sum_i w_i x_i) = c$$

which is equivalent to (assuming f is invertible in c)

$$\sum_i w_i x_i = f^{-1}(c)$$

which defines a straight line in terms of the x_i.

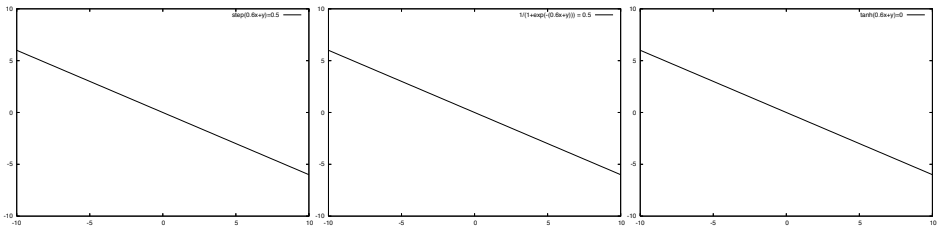

Figure 10.5: The line indicating the boundary between two classes, if we use $f(0)$ as the boundary constant and weights $w_1 = 0.6$ and $w_2 = 1$. The line is given by $f(0.6x + y) = f(0)$ or, equivalently, $0.6x + y = 0$.

A constant input x_0

In the above description, the straight line separating the two classes always runs through the origin (0,0). For continuous transfer functions, this restriction can be removed if we allow another threshold than $f(0)$ to decide which class an example belongs to. But there is another simple method that removes the restriction, which also works for the (discontinuous) step function, and which is equivalent to changing the threshold in the case of continuous functions.

In addition to the inputs $x_1, \ldots, x_D$, we can add an input x_0 that is always 1, whatever the example is. We then also get an extra weight w_0. By including $w_0 x_0 = w_0$ into the summation, we get

$$\sum_{i=0}^{D} w_i \cdot x_i = w_0 + \sum_{i=1}^{D} w_i \cdot x_i$$

which means that for $x_i = 0$, $i = 1 \ldots D$, the result is $f(w_0)$ and not $f(0)$. So, the effect of changing the threshold from $f(0)$ to some other number c is the same as that of keeping the threshold $f(0)$ but including a weight w_0. More precisely, for a continuous and invertible function f, we have

$$f(\sum_{i=1}^{D} x_i w_i) = c \quad \Leftrightarrow \quad \sum_{i=1}^{D} x_i w_i = f^{-1}(c) \quad \Leftrightarrow \quad \sum_{i=0}^{D} x_i w_i = f^{-1}(c) + w_0$$

which for $w_0 = -f^{-1}(c)$ becomes

$$\sum_{i=0}^{D} x_i w_i = 0$$

and hence

$$f(\sum_{i=0}^{D} x_i w_i) = f(0).$$

In other words, any separation that can be expressed using a threshold c other than $f(0)$, can be expressed using a threshold $f(0)$ by including an extra weight $w_0 = -f^{-1}(c)$. This weight is automatically learned if we include an extra input $x_0 = 1$ in the perceptron. Thus, by including x_0 and w_0 into the perceptron, the perceptron can in a sense learn the optimal threshold c together with all the other parameters.

For this reason, the issue of choosing the threshold is usually not discussed separately. Conceptually, the input space is D-dimensional (i.e., x_0 is not part of the input space) and the threshold is not necessarily 0, and we will continue taking this view; but in practice this can be implemented by adding a constant input x_0.

Vector notation

It is clear that the inputs x_i, $i = 1 \ldots D$ can be seen as components of a D-dimensional vector $\mathbf{x}$. An example is then a couple $(\mathbf{x}, y)$, and the perceptron can be seen as mapping an input vector $\mathbf{x}$ onto a scalar output y.

Interestingly, since there is one connection weight w_i for each input variable x_i, the weights together also form a D-dimensional vector $\mathbf{w}$. While in principle this vector just represents the parameters of the perceptron, and does not "live" in the input space, we can interpret it as a vector in that input space, just like the input vector $\mathbf{x}$. This leads to an interesting intuitive view.

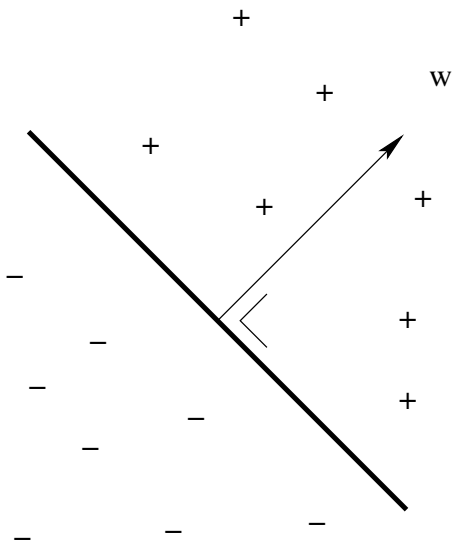

Figure 10.6: Interpretation of the weight vector $\mathbf{w}$ as a vector indicating the direction in which the positives lie, relative to the negatives. The line separating positives and negatives, defined by $\mathbf{w} \cdot \mathbf{x}$, is orthogonal to $\mathbf{w}$.

Geometrically, using the vectors $\mathbf{x}$ and $\mathbf{w}$, the calculation performed by the perceptron is simply

$$y = f(\mathbf{w} \cdot \mathbf{x}),$$

that is, the perceptron calculates the dot product of the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}$ and then applies f to it. (If we include a constant input $x_0 = 1$, then a constant term w_0 is simply added to this dot product.)

The dot product $\mathbf{w} \cdot \mathbf{x}$ is related to the cosine of the angle between $\mathbf{x}$ and $\mathbf{w}$: it is positive if the angle between $\mathbf{x}$ and $\mathbf{w}$ is less than $\pi/2$, zero if $\mathbf{x}$ and $\mathbf{w}$ are orthogonal, and negative if the angle is between $\pi/2$ and π.

Intuitively, this means that the weight vector $\mathbf{w}$ can be seen as indicating the direction of the positive examples, relative to the negative examples, in the input vector space. Imagine a plane going through the origin and orthogonal to $\mathbf{w}$. Any vector on the same side of the plane as $\mathbf{w}$ will yield a positive dot product, any vector on the opposite side as $\mathbf{w}$ a negative one. Thus, the plane through the origin and orthogonal to $\mathbf{w}$ forms the boundary between the two classes.

If we plot the examples and boundary in the original space, excluding w_0 and the constant input x_0 from $\mathbf{w}$ and $\mathbf{x}$, and introducing a threshold c, i.e., we have a positive prediction for $\mathbf{x}$ if $f(\mathbf{w} \cdot \mathbf{x}) \geq c$, then it still holds that the decision boundary is a plane orthogonal to $\mathbf{w}$, though it does not necessarily go through the origin. (It does when $c = f(0)$.)

Figure 10.6 illustrates the relationship between $\mathbf{w}$ and the decision boundary.

Perceptrons take numerical inputs

Note that, given the nature of the computations performed by a perceptron, perceptrons only work on numerical data.

This does not mean that they cannot be applied in any domain where symbolic (nominal) attributes are present. It justs implies that these nominal data have to be converted into a numerical format.

A boolean attribute can easily be transformed into a numerical attribute by using the values 1 and 0 (or 1 and -1) for true and false. Conversion of a nominal attribute with k values is a bit more tricky. Using the numerical values 1, 2, ..., k for them is often a bad idea because that implicitly imposes an ordering on the nominal values: it suggests that the first nominal value is "smaller" than the second, etc. In practice, nominal values may not have such an ordering.

A better solution is to encode a nominal attribute A with k values $a_1, \ldots, a_k$ as k boolean (actually 0/1) attributes, where the i'th attribute takes the value 1 if $A = a_i$, and 0 otherwise.

10.2.2 Training a perceptron

Having looked at the structure of a perceptron, and the kind of models it can represent, we now turn to the question of how to learn the parameters of a perceptron, in other words: how to compute the weight vector $\mathbf{w}$ such that the function represented by the perceptron maximally fits the input data T.

A commonly used method is the following. We start with initializing the weights randomly. Then, we perform for each single example in T the following procedure: given the example $\mathbf{x}$ and the current weight vector $\mathbf{w}$, we compute $\hat{y} = f(\mathbf{w} \cdot \mathbf{x})$, the value for y that is predicted by the perceptron, and compare this to the observed value y. If $\hat{y} > y$, the prediction is too high. We should then adapt the weights $\mathbf{w}$ such that $f(\mathbf{w} \cdot \mathbf{x})$ becomes smaller. If $\hat{y} < y$, we should adapt $\mathbf{w}$ such that $f(\mathbf{w} \cdot \mathbf{x})$ increases.

There are many different ways to make $f(\mathbf{w} \cdot \mathbf{x})$ increase or decrease. Note that *in the end* we want to obtain that $\mathbf{w} \cdot \mathbf{x} = f^{-1}(y)$, or:

$$\sum_i w_i x_i = f^{-1}(y).$$

This can easily be done by adapting a single w_i. Note that with the current weights,

$$\sum_i w_i x_i = f^{-1}(\hat{y}).$$

Let $d = f^{-1}(\hat{y}) - f^{-1}(y)$. We want $\sum_i w_i x_i$ to decrease with d. We can achieve that by making $w_1 x_1$ decrease with d, which means decreasing w_1 with d/x_1. Obviously, we could do something similar with any w_i, not just w_1. We could also distribute the reduction d over several of the w_i, or equally over all of them.

Which of all these is best, then? Are they all equally good?

They are not. In fact, none of them is really good. First of all, note that the proposed adaptation will ensure that for this one training example, the prediction equals the output, but for the other training examples the prediction has also changed, and that prediction may be incorrect. We can turn to a next example and adapt the weights to fit that example, but then, obviously, the prediction for the first example may be incorrect again. By adapting the weights in this way, the perceptron may never converge to a weight vector that works reasonably well for all the training examples together.

Second, note that for several of the transfer functions f we have considered, it may be impossible to obtain $f(\mathbf{w} \cdot \mathbf{x}) = y$. In many training settings, the class variable y will be given values 1 or 0. For the sigmoid function we have considered, there is no value z for the argument for which $f(z) = 0$ or $f(z) = 1$; for any z, $0 < f(z) < 1$. So reducing the error to zero is impossible: it would require $\mathbf{w} \cdot \mathbf{x}$ to be infinite.

For these reasons, it is better to not immediately try to reduce the error to zero, but make only a small change in $\mathbf{w}$, in such a way that $|f(\mathbf{w} \cdot \mathbf{x}) - y|$ is reduced

by a small amount. By doing this repeatedly, once for each training example, and repeating the process from the beginning when we have run out of training examples, we can hope that $\mathbf{w}$ will gradually move towards a value that works reasonably well for the whole training set.

Generally, when looking at a specific example $\mathbf{x}$, we want to change $\mathbf{w}$ as little as possible, in order to disturb the predictions for other examples as little as possible, while at the same time trying to reduce $\hat{y} - y$ as much as possible. In other words, we look for a minimal change to $\mathbf{w}$ with a maximal effect on $\hat{y}$.

Let $g(\mathbf{w}) = f(\mathbf{w} \cdot \mathbf{x})$. In mathematics, the direction in which $g(\mathbf{w})$ increases most for a given change in $\mathbf{w}$ is called the *gradient* of g. This gradient is equal to the vector of partial derivatives of g to the components of its argument vector:

$$\nabla g(w_1, w_2, \ldots, w_D) = (\frac{\partial g}{\partial w_1}, \frac{\partial g}{\partial w_2}, \ldots, \frac{\partial g}{\partial w_D})$$

If we want to change $\mathbf{w}$ by adding to it a vector of a given length, $g(\mathbf{w})$ will increase maximally if the vector is in the direction of the gradient, and it will decrease maximally if the vector is in exactly the opposite direction.

We will now look at how all this applies to the transfer functions f.

Gradients of transfer functions

In the special case where the transfer function f is the identity function, i.e.,

$$g(w_1, \ldots, w_D) = x_1 w_1 + \cdots + x_D w_D,$$

this gradient vector simply becomes

$$\nabla g(w_1, \ldots, w_D) = (x_1, x_2, \ldots, x_D) = \mathbf{x}$$

Thus, in this case, we need to change each w_i proportionally to the corresponding x_i.

If f is the step function, there is a problem. This function is constant everywhere and discontinuous in 0. That means its gradient is 0 everywhere except on the line where $f(\mathbf{x}) = f(0)$, where it is undefined. Intuitively, if we have a vector $\mathbf{x}$ for which the prediction $f(\mathbf{w} \cdot \mathbf{x})$ is wrong, we have no idea in which direction we should change it in order to reduce the error, because a small step in any direction leads to exactly the same error as before.

The logistic function, $f(x) = 1/(1 + \exp(-x))$, has the property that $\frac{df(x)}{dx} = f(x)(1 - f(x))$, and since $g(\mathbf{w}) = f(\mathbf{w} \cdot \mathbf{x})$, we get

$$\nabla g(w_1, \ldots, w_n) = g(\mathbf{w})(1 - g(\mathbf{w}))\mathbf{x}.$$

Exercise 10.1 (Some background in calculus required.) Show first that when $f(x) = 1/(1 + \exp(-x))$, $\frac{df(x)}{dx} = f(x)(1 - f(x))$. Next, show that with $g(\mathbf{w}) = f(\mathbf{w} \cdot \mathbf{x})$, $\nabla g(\mathbf{w}) = g(\mathbf{w})(1 - g(\mathbf{w}))\mathbf{x}$.

Note that just like in the previous case, the optimal change of $\mathbf{w}$ is in the direction of $\mathbf{x}$, but this time the change of $g(\mathbf{w})$ for a given change step size becomes smaller if $g(\mathbf{w})$ is close to 0 or 1, i.e., when the absolute value of $\mathbf{w} \cdot \mathbf{x}$ is large.

Making the change proportional to the error

We have now determined the direction in which $\mathbf{w}$ should be updated, namely: it should be updated in the direction of $\mathbf{x}$; but how large should the change in that direction be?

function TrainPerceptron(training set T, step size σ) **returns** weight vector:
    let $T = \{(\mathbf{x}_i, y_i) | 1 \leq i \leq N\}$
    $\mathbf{w}$:= random weight vector
    $i := 0$
    repeat
        $i := i + 1$
        if $i > N$ **then** $i := 1$
        $\hat{y}_i := f(\mathbf{w} \cdot \mathbf{x}_i)$
        $\mathbf{w} := \mathbf{w} + \sigma(y_i - \hat{y}_i)\mathbf{x}_i$
    until stopping criterion fulfilled
    return w

Figure 10.7: Perceptron training algorithm.

Since we want $g(\mathbf{w}) = f(\mathbf{w} \cdot \mathbf{x})$ to increase if $y > f(\mathbf{w} \cdot \mathbf{x})$ and decrease otherwise, and assuming that we want to make the change proportional to the error (i.e., take bigger steps when we're still far away from the goal), it is reasonable to make the step size dependent on $y - \hat{y} = y - g(\mathbf{w})$. Hence, we update $\mathbf{w}$ as follows:

$$\mathbf{w} = \mathbf{w} + \sigma(y - g(\mathbf{w}))\nabla g(\mathbf{w}),$$

where σ influences the step size: a large value for σ causes us to make larger steps, relative to $(y - \hat{y})$, whereas a smaller σ causes us to make smaller steps.

In the special case where $\nabla g(\mathbf{w}) = \mathbf{x}$, and writing the current prediction $g(\mathbf{w})$ as $\hat{y}$ for simplicity,

$$\mathbf{w} = \mathbf{w} + \sigma(y - \hat{y})\mathbf{x}$$

Returning to scalar notation, this means that the weights w_i are updated as follows:

$$w_i = w_i + \sigma(y - \hat{y})x_i$$

This update rule is the so-called **perceptron learning rule**. It leads to the perceptron learning algorithm shown in Figure 10.7. The algorithm works as follows: first, a random value for $\mathbf{w}$ is chosen; next, the algorithm simply considers the examples $\mathbf{x}_i$ one by one and for each $\mathbf{x}_i$ updates $\mathbf{w}$ by moving it a bit in the direction of $\mathbf{x}_i$ when $y_i - \hat{y}_i$ is positive, and in the opposite direction when $y_i - \hat{y}_i$ is negative, using precisely the formula we just derived. When all examples have been handled, the algorithm starts again at the beginning of the dataset. It continues updating $\mathbf{w}$ until all its predictions are correct or until a maximum number of iterations have been performed.

When $\nabla g(\mathbf{w}) = g(\mathbf{w})(1 - g(\mathbf{w}))\mathbf{x}$ (i.e., when using the logistic function), essentially the same update rule can be used but with the factor $g(\mathbf{w})(1 - g(\mathbf{w}))$ added in the update rule. This factor causes $\mathbf{w}$ to be updated more strongly when $g(\mathbf{w}) = f(\mathbf{w} \cdot \mathbf{x})$ is far from 0 or 1, i.e., when $\mathbf{w} \cdot \mathbf{x}$ is near zero.

Gradient descent

The reasoning used above is similar to a general method for finding minima or maxima of functions in mathematics, and it is known as **gradient descent** (in the case of minimizing a function) or **gradient ascent** (when maximizing). Intuitively, it can be compared with finding the top of a hill, or lowest point of a valley, by stepping always in the direction of the steepest slope, up or down.

Formally, the gradient descent principle can be used to derive a training algorihtm very similar to the perceptron training algorithm. Given a data set $T =$

$\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ and a function $h(\mathbf{x})$ (in our case represented by a perceptron) that we wish to have the property that $h(\mathbf{x}_i) = y_i$, we can define the squared error of h as

$$e(h) = \sum_{i=1}^{N} (h(\mathbf{x}_i) - y_i)^2.$$

The function $e(h)$ can be seen as a quality measure for h: the smaller $e(h)$ is, the closer h approximates our true target function, and when $e(h) = 0$ the function h perfectly models the training set.

The function h that a perceptron represents, depends entirely on the weight vector $\mathbf{w}$. Thus, we can define the error e directly as a function of $\mathbf{w}$ instead of h. That gives

$$e(\mathbf{w}) = \sum_{i=1}^{N} (f(\mathbf{w} \cdot \mathbf{x}_i) - y_i)^2$$

where we have rewritten $h(\mathbf{x}_i)$ as $f(\mathbf{w} \cdot \mathbf{x}_i)$.

We have now turned the problem of learning the weights of a perceptron into a minimization problem (look for the vector $\mathbf{w}$ that minimizes $e(\mathbf{w})$), which can itself be solved using the general technique of gradient descent. This technique consists of adapting $\mathbf{w}$ in the direction of the negative gradient of $e(\mathbf{w})$. Thus we get the update rule

$$\mathbf{w} \leftarrow \mathbf{w} - \sigma \nabla e(\mathbf{w})$$

where σ is, as above, a parameter controlling the size of the steps we take in the direction of the negative gradient.

Using the definition of e, we have

$$\begin{aligned}
\nabla e(\mathbf{w}) &= \nabla \sum_{i=1}^{N} (f(\mathbf{w} \cdot \mathbf{x}_i) - y_i)^2 \\
&= \sum_{i=1}^{N} 2(f(\mathbf{w} \cdot \mathbf{x}_i) - y_i) \nabla (f(\mathbf{w} \cdot \mathbf{x}_i) - y_i) \\
&= \sum_{i=1}^{N} 2(\hat{y}_i - y_i) \nabla f(\mathbf{w} \cdot \mathbf{x}_i)
\end{aligned}$$

Filling in for $\nabla f(\mathbf{w} \cdot \mathbf{x}_i)$ what we saw before for a number of particular f's, we get:

- when f is the identity function: $\nabla f(\mathbf{w} \cdot \mathbf{x}_i) = \mathbf{x}_i$
- when f is the logistic function: $\nabla f(\mathbf{w} \cdot \mathbf{x}_i) = f(\mathbf{w} \cdot \mathbf{x}_i)(1 - f(\mathbf{w} \cdot \mathbf{x}_i))\mathbf{x}_i$

Filling in the first of these two into the update rule, moving the σ factor into the summation, and rewriting $\hat{y}_i - y_i$ as $-(y_i - \hat{y}_i)$, we get

$$\mathbf{w} \leftarrow \mathbf{w} + \sum_{i=1}^{N} 2\sigma(y_i - \hat{y}_i)\mathbf{x}_i \tag{10.1}$$

Comparing this with the perceptron training rule, we see that the update is of exactly the same form, except for using 2σ instead of σ (which is not important, it simply changes the meaning of the parameter σ a bit). There is one subtle difference, however. In the gradient descent approach, a single update of the $\mathbf{w}$ vector uses information about the whole dataset, whereas in the perceptron training algorithm, examples are processed one by one and $\mathbf{w}$ is updated immediately after seeing an

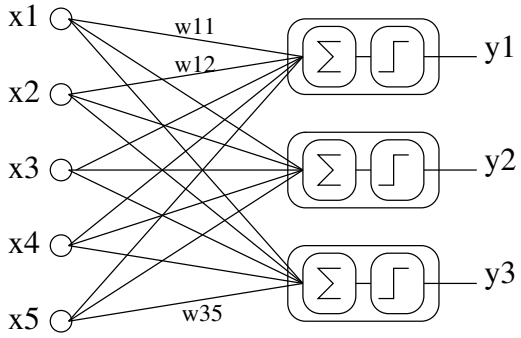

Figure 10.8: A multi-output perceptron.

example. So, the gradient descent approach could be seen as doing the same as the perceptron training rule, with this difference that while going over the dataset it just accumulates the errors of the examples, deferring the updates of $\mathbf{w}$ until the whole dataset has been seen, and then making a single update based on all the errors.

The two algorithms do not yield exactly the same results. If $\mathbf{w}$ is changed after seeing example $(\mathbf{x}_1, y_1)$, then $\hat{y}_2$ will be computed based on the new version of $\mathbf{w}$, whereas in the gradient descent approach $\hat{y}_2$ is computed using the same $\mathbf{w}$ as $\hat{y}_1$.

While they can give different results, both approaches turn out to work well in practice. One could argue that the "deferred update" approach is mathematically more elegant, since it can be interpreted as an instance of the more general "gradient descent" approach to minimization, which has been studied extensively.

Note that filling in the formula for the logistic function, and rewriting $f(\mathbf{w} \cdot \mathbf{x}_i)$ as $\hat{y}_i$, we get the following alternative update rule:

$$\mathbf{w} \leftarrow \mathbf{w} + \sum_{i=1}^{N} 2\sigma(y_i - \hat{y}_i)\hat{y}_i(1 - \hat{y}_i)\mathbf{x}_i \tag{10.2}$$

which is the same except for the factor $\hat{y}_i(1-\hat{y}_i)$. This is the correct gradient descent update rule for perceptrons that use the logistic function as transfer function.

10.2.3 Multi-output perceptrons

We now turn to perceptrons that have more than one output. We assume that each output is connected to each input. Thus we have:

- associated with each input node n_i, a numerical variable x_i;
- associated with each output node o_j, a numerical variable y_j;
- associated with each connection from n_i to o_j, a weight w_{ji} (note that the first index indicates the output node and the second index the input node).

This structure is drawn in Figure 10.8.

A multi-output perceptron is essentially just the union of a number of single-output perceptrons, each of which can be looked at separately: they have their own weights, represent their own decision boundary, and in every way behave like a normal single-neuron perceptron.

While the single-output perceptron was useful to distinguish two classes, the multi-output perceptron can be used to distinguish multiple classes. In this case,

we associate one output node o_j with each class c_j. Each input data point has a vector of class values y_j where $y_j = 1$ if the point belongs to class c_j and $y_j = 0$ otherwise. In other words, for each example, exactly one output variable y_j has the value 1.

The perceptron can then be used to make predictions as follows: given a new instance $\mathbf{x}$, we calculate the associated value y_j of each output node o_j. We assign to $\mathbf{x}$ that class c_M for which the corresponding output y_M is highest, that is, $y_M \geq y_j$ for all y_j.

In the previous section, we pointed out that the x_i can be seen as components of a vector $\mathbf{x}$, the weights form a vector $\mathbf{w}$, and the output is a scalar $y = f(\mathbf{x} \cdot \mathbf{w})$. In the multi-output perceptron, we have multiple outputs y_j, which together form a vector $\mathbf{y}$. For each output y_j we now have a vector $\mathbf{w}_j$ consisting of components w_{ji}. Thus, the weights now form a matrix $\mathbf{W}$, and we have

$$\mathbf{y} = f^*(\mathbf{W}\mathbf{x})$$

where the function f^*, operating on vectors, is defined as the componentwise application of f to all the components of the vector:

$$f^*([t_1, t_2, \ldots, t_n]) = [f(t_1), f(t_2), \ldots, f(t_k)]$$

In the following we will usually not distinguish f and f^* but will use the symbol f in both cases, since it is always clear from the type of the argument exactly what is meant.

10.2.4 Properties of perceptrons

An important property of the perceptron is that it can only learn accurate models in cases where the classes are linearly separable. In two dimensions, this means that the classes can be separated with a straight line. This is clearly a strong restriction. Perhaps the most convincing argument for this is the following: if we are trying to learn boolean functions such as **and**, **or**, **not** and **xor**, using the values 0 and 1 to express false and true, then it turns out that the **xor** function, one of the simplest imaginable boolean functions, cannot be represented by a perceptron.

A property of the perceptron *training algorithm*, as seen here, is that once the perceptron classifies the input data correctly, the learning task is considered solved and the learning process stops. However, there might be multiple possible solutions, all of which correctly distinguish the different classes in the input data. Consider for instance Figure 10.9. In a 2-dimensional input space, several linear separators are shown, all of which are correct in the sense that they separate the two classes correctly. Yet for some of these separators it seems unlikely that they will never make a mistake on unseen data, as the "positive area" they define comes very close to a negative example, or the other way around.

Note that if we use the logistic or tanh function, this is not so much a problem, as $\hat{y}$ will never become equal to y; e.g., for the logistic function, y takes values of 1 or 0 only, while $\hat{y}$ is always strictly in between these. $\hat{y}$ will be closer to y if $\mathbf{x}$ is further away from the decision surface, hence, this decision boundary has a natural tendency to stay away from the training instances. (Note, however, that since the condition that $\hat{y} = y$ is never reached for any instances, some stopping criterion is needed to decide when to stop training.)

Generally, one would expect that a linear separator that stays at maximal distance from the actually observed instances, one that is "in the middle", is a safer guess than a linear separator that gets very close to one of the training instances. In fact, there is a unique linear separator for which the distance between the separator and the closest training examples is maximal. We might prefer to find that unique

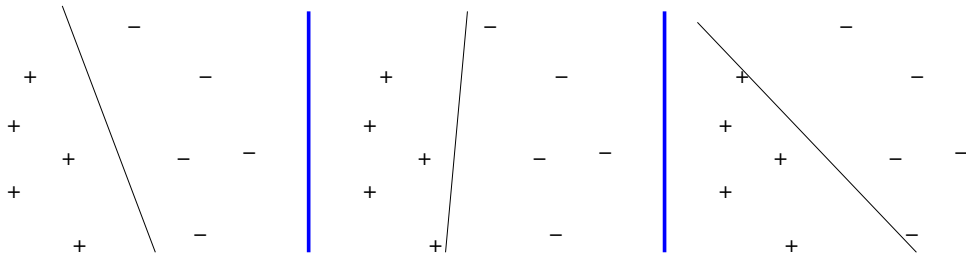

Figure 10.9: A perceptron may converge on multiple solutions, all of which have zero training set error; yet some of these solutions are more convincing than others.

solution. Support vector machines, which are discussed in Chapter 11, explicitly look for that solution.

10.3 Multi-layer perceptrons

Multi-layer perceptrons are also known as "feed-forward neural networks". They are more complex networks than the ones we discussed up till now. Multi-layer perceptrons consist of multiple neurons that are organized in layers: the outputs of the neurons in one layer serve as inputs for the neurons in the next layer. In many cases, multi-layer perceptrons have only two or three layers, although there is no theoretical bound on the number of layers they may have.

Note that when we are given training data for a multi-layer perceptron, these data are in the form of a set of points $(\mathbf{x}, \mathbf{y})$ with $\mathbf{x}$, the description of the example, forming the input of the network, and $\mathbf{y}$, the target vector, forming the desired output of the network for the example. Thus, when training a network, we have information about the input and output nodes, but not about the nodes in between. The layers between inputs and outputs are therefore called *hidden layers*. See Figure 10.10 for a graphical illustration of a two-layer perceptron (consisting of inputs, one hidden layer, and one output layer).

Typically, multi-layer perceptrons are fully connected, in the sense that there is a connection from each perceptron in layer n to each perceptron in layer $n+1$. This is also the case in Figure 10.10. Such full connectivity is not required in multi-layer perceptrons, but it is common: deviations from this structure typically occur only when there are good arguments for that.

10.3.1 Representation power of multi-layer perceptrons

We have seen that a single perceptron can only represent linear separations between classes. The situation is quite different, however, for a multi-layer perceptron.

To see this, first consider the case where the transfer function is a step function, which outputs 1 if its sum of weighted inputs exceeds some threshold, and 0 otherwise. Each of the nodes in the first (hidden) layer of the network then outputs the value 1 if the input lies at one side of a particular hyperplane in the input space, and 0 otherwise. Figure 10.11 shows for a 2-dimensional input space and three neurons a, b and c the boundaries corresponding to them.

Now take into consideration that a perceptron can learn boolean functions such as **and** and **or**. As a consequence, the neuron in the output layer, which takes as inputs 3 values that each express on which side of a particular hyperplane an example lies, can express whether an input is at a given side of one hyperplane **and** at a given side of another hyperplane. As a result, the output can express whether an instance is inside or outside a polygon. Figure 10.11 illustrates this.

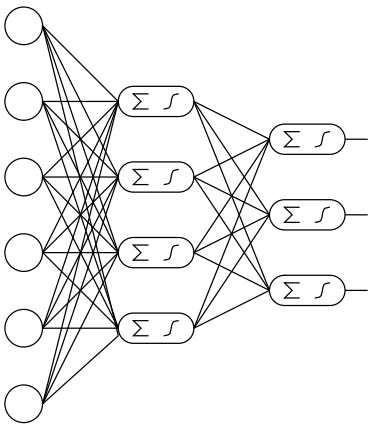

Figure 10.10: A fully connected two-layer perceptron with 6 inputs, a hidden layer with 4 nodes, and an output layer with 3 nodes.

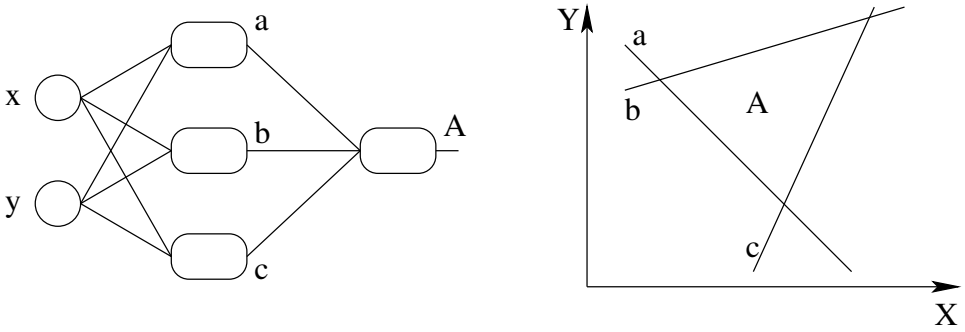

Figure 10.11: Three neurons in the first layer can express three different linear separations (a, b, c). A neuron taking the outputs of these first layer neurons as inputs can express the conjunction of the conditions expressed by the neurons in the first layer, and thus can express, for instance, a polygon (A).

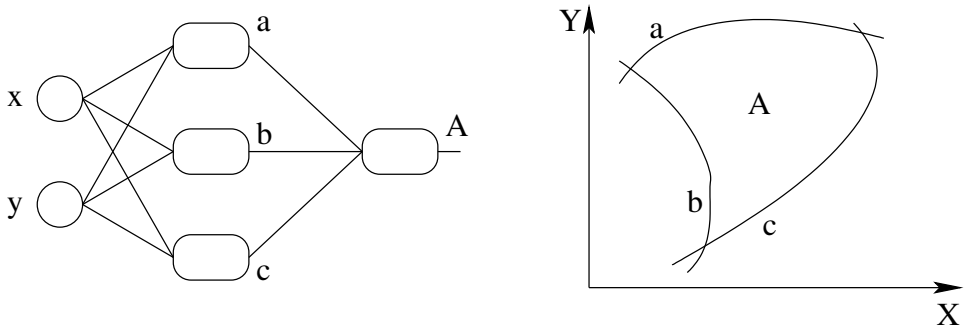

Figure 10.12: Areas identified by a multi-layer perceptron with a smooth non-linear transfer function. The straight lines bounding the polygon from Figure 10.11 have now become curves.

Not every polygon can be expressed in this way, though. The area for which the final neuron fires is generally a convex area surrounded by straight lines (with at most as many lines as there are neurons in the hidden layer), or the complement of such an area. If we add an additional layer, these areas can again be combined into more complex shapes. At that point, every polygon can be represented, under the condition that there are enough neurons in the hidden layers.

The situation is a bit more complicated if we do not use a step function as transfer function in the neurons. Note that if the identity function is used, the expressive power of the network is exactly the same as that of a single-layer perceptron. Indeed, each hidden node is then a linear combination of the input nodes, and the outputs are a linear combination of these hidden nodes — but a linear combination of a linear combination of $\mathbf{x}$ can always be written directly as a linear combination of $\mathbf{x}$. So, the expressive power of a multi-layer perceptron is only higher than that of a single-layer perceptron if a non-linear transfer function is used.

If we use, instead of a step function, a smooth function such as the logistic or tanh function mentioned earlier, then roughly the same reasoning can be used as made above for the step function. But because of the non-linearity of the transfer functions, the sides of the "polygons" formed may not be linear anymore. While each hidden node in itself "fires" (i.e., its output is greater than $f(0)$) in an area that is bounded by a hyperplane, the interaction of the different nodes causes nonlinearities in the boundary lines between areas. See Figure 10.12 for an illustration.

Exercise 10.2 Show that, while a single-layer perceptron cannot learn the boolean **xor** function, a multilayer perceptron can. To this aim, show the values of $f(x, y) = x$ **xor** y on a two-dimensional diagram where both x and y can be 0 or 1. Show that a single straight line cannot separate the points where the function is true from the points where it is false. Show that they can be separated if multiple straight lines can be used. Finally, draw a two-layer perceptron (with weights included) that expresses the **xor** function.

10.3.2 Training a multi-layer perceptron

To train a multi-layer perceptron, we can use basically the same reasoning as for a single-layer perceptron. However, such a multi-layer perceptron has many more parameters, or degrees of freedom, than a single-layer perceptron.

As before, we will train the network using couples of input vectors $\mathbf{x}$ and output vectors $\mathbf{y}$. Assume we have a two-layer network with D inputs, E hidden nodes, and F outputs. The E hidden nodes are all connected with all D input nodes,

which gives us a weight matrix $\mathbf{W}$ that has E rows and D columns. The F output nodes are all connected with all E hidden nodes, giving us a second weight matrix $\mathbf{W}'$ with F rows and E columns. The relationship between the input vector $\mathbf{x}$, the vector of hidden nodes $\hat{\mathbf{h}}$ and the predicted output vector $\hat{\mathbf{y}}$ is:

$$\hat{\mathbf{h}} = f(\mathbf{W}\mathbf{x})$$
$$\hat{\mathbf{y}} = f(\mathbf{W}'\hat{\mathbf{h}}) = f(\mathbf{W}'f(\mathbf{W}\mathbf{x}))$$

If for each instance we knew the vector $\mathbf{h}$, then the weights in $\mathbf{W}$ could be learned using gradient descent, reducing the difference between $\hat{\mathbf{h}}$ and $\mathbf{h}$, as described before. $\mathbf{W}'$ could be learned similarly, trying to reduce $\hat{\mathbf{y}} - \mathbf{y}$.

But generally we don't know $\mathbf{h}$; $\mathbf{h}$ contains the values in the hidden layer, which is called thus exactly because of the fact that we don't observe these values.

So we have an input $\mathbf{x}$, an output $\mathbf{y}$, and two weight matrices such that

$$\hat{\mathbf{y}} = f(\mathbf{W}'f(\mathbf{W}\mathbf{x}))$$

and we need to minimize $\hat{\mathbf{y}} - \mathbf{y}$.

The reasoning we will use is as follows. The difference $\hat{\mathbf{y}} - \mathbf{y}$ can be attributed to both $\mathbf{W}$ and $\mathbf{W}'$, and it seems reasonable to distribute the weight updates over both matrices.

From the point of view of the second layer, the situation is as follows. The input for this second layer is $\hat{\mathbf{h}}$. The layer computes $\hat{\mathbf{y}} = f(\mathbf{W_2}\hat{\mathbf{h}})$. Any difference between $\hat{\mathbf{y}}$ and $\mathbf{y}$ can be attributed to the fact that the weight matrix $\mathbf{W}'$ is incorrect, or to the fact that the input vector $\hat{\mathbf{h}}$ is incorrect. After all, we did not observe $\mathbf{h}$ and hence we cannot be sure how much $\hat{\mathbf{h}}$ deviates from the real $\mathbf{h}$.

Therefore, when updating the parameters, we will reduce $\hat{\mathbf{y}} - \mathbf{y}$ by changing both $\mathbf{W}'$ and $\hat{\mathbf{h}}$. Both of these are moved a bit in the direction of decreasing output error. We can update $\mathbf{W}'$ using the gradient descent method, acting as if $\hat{\mathbf{h}}$ were the correct input of the second layer. To do this we need exactly the same weight update rule as before, but now with $\hat{\mathbf{h}}$ taking the place of $\mathbf{x}$. Assuming the use of the logistic transfer function, this gives:

$$w'_{ji} = w'_{ji} + \sigma(y_j - \hat{y}_j)\hat{y}_j(1 - \hat{y}_j)\hat{h}_i \tag{10.3}$$

Similarly, we adapt $\hat{\mathbf{h}}$ acting as if $\mathbf{W}'$ were the correct weight matrix. This can again be done using the gradient descent principle: we want to make changes to $\hat{\mathbf{h}}$ in the direction that reduces the output error the most. This gives us a "requested change" vector $\mathbf{\Delta h}$.

$$\Delta h_i = \sum_j (y_j - \hat{y}_j)\hat{y}_j(1 - \hat{y}_j) \cdot w'_{ji}$$

We need to add $\mathbf{\Delta h}$ to $\hat{\mathbf{h}}$, but note that we cannot change $\hat{\mathbf{h}}$ directly. To change $\hat{\mathbf{h}}$, we need to update the weight matrix $\mathbf{W}$, since after all $\hat{\mathbf{h}} = f(\mathbf{W}\mathbf{x})$. Knowing $\mathbf{\Delta h}$, i.e., the change that is requested in each node of the hidden layer, we can again apply the gradient descent principle to update the weights in $\mathbf{W}$ (note that the $\hat{h}_j$ and Δh_j now play the role of the $\hat{y}_j$ and $(y_j - \hat{y}_j)$ previously):

$$w_{ji} = w_{ji} + \sigma \Delta h_j \hat{h}_j(1 - \hat{h}_j)x_i \tag{10.4}$$

The whole procedure just described is performed by an algorithm that is often referred to as "back-propagation". The name refers to the fact that errors are back-propagated through the network. The output error $\hat{\mathbf{y}} - \mathbf{y}$ is partially reduced by

function BackProp(training set T) **returns** weight matrices:
    $\mathbf{W}$:= random weight matrix (E rows, D columns)
    $\mathbf{W}'$:= random weight matrix (F rows, E columns)
    $k := 0$
    repeat
        $k := k + 1$
        if $k > N$ **then** $k := 1$
        $\hat{\mathbf{h}} := f(\mathbf{W} \cdot \mathbf{x}_k)$
        $\hat{\mathbf{y}} := f(\mathbf{W}' \cdot \mathbf{h})$
        for $j := 1$ to F:
            for $i := 1$ to E:
                $\Delta h_i = \sum_j (y_j - \hat{y}_j)\hat{y}_j(1 - \hat{y}_j) \cdot w'_{ji}$
                $w'_{ji} = w'_{ji} + \sigma(y_j - \hat{y}_j)\hat{y}_j(1 - \hat{y}_j)\hat{h}_i$
        for $j := 1$ to E:
            for $i := 1$ to D:
                $w_{ji} = w_{ji} + \sigma \Delta h_j \hat{h}_j(1 - \hat{h}_j)x_i$
    until max number of iterations reached
    return $(\mathbf{W}, \mathbf{W}')$

Figure 10.13: The Backpropagation algorithm.

changing the second layer's weight matrix, but part of this error is also transferred to the hidden layer, leading to an "error" $\hat{\mathbf{h}} - \mathbf{h}$ in this hidden layer, which is next reduced by updating $\mathbf{W}$.

Of course, this whole procedure is not executed for a single point but for each data point consecutively, returning to the first point in the data set when the end is reached and the error is not yet sufficiently low.

This gives the backpropagation algorithm shown in Figure 10.13.

10.3.3 Multiple layers = multiple representations

Each layer can be seen as a transformation from one space to another space. These transformations are not defined in advance, but are learned during the learning process. Recall that a single layer perceptron has limited discriminative power, being able to construct only linear separations between classes. A multilayer perceptron can be seen as transforming its input data in such a way that they *become* linearly separable. This transformation and the linear separator are learned in parallel. If the transformation produces a result that is linearly separable but not with the current linear separator, the latter is adapted a bit to be more compatible with the outcome of the transformation. If the linear separator is well-tuned to the transformation but the transformation is currently yielding a result that is not yet entirely linearly separable, the transformation is adapted a bit.

Thus, a multilayer perceptron can be seen as learning a transformation from a D-dimensional space to an E-dimensional space such that the classes are linearly separable in the latter space.

10.4 Kohonen maps

Kohonen networks, also known as self-organizing maps (SOMs), are quite different from the kind of neural networks we have seen before. First of all, the goal here is not classification, but is more related to clustering. The way in which a Kohonen network performs clustering is a bit similar to the k-means clustering algorithm.

However, Kohonen networks have an additional advantage: they map the clusters to, for instance, a two-dimensional plane, thus providing a visualisation, or "map", of the clustering.

10.4.1 A simplified Kohonen network

Take a simple perceptron. It was pointed out earlier that the weight vector $\mathbf{w}$, while it is normally not seen as a vector in the input space, could in principle considered as living in that space, since it has the same dimensionality; $\mathbf{w} \cdot \mathbf{x}$ is then the dot product of the input vector and the weight vector, and the output of the perceptron, $f(\mathbf{w} \cdot \mathbf{x})$, is then a function of this dot product.

A Kohonen network has the same structure as a single-layer perceptron with multiple outputs. Its inputs are vectors $\mathbf{x}$ without any classification y attached to it (unsupervised learning). Consequently, the weights of the network cannot be trained so that the outputs of the network become similar to the observed y values.

Instead, we will train the weights such that for each output node, the corresponding weight vector becomes similar to a number of input vectors, in other words, so that it becomes representative for those input vectors. The effect is somewhat similar to adapting the cluster seeds in k-means: starting with random seeds, the seeds move after each iteration in the direction of a cluster center. In the end, each seed is in the center of some cluster, and as such is maximally representative for the points in that cluster. The k-means clustering process can therefore also be interpreted as "find k points that together represent the set of all points as accurately as possible".

In its simplest form, a Kohonen network will basically do something similar to k-means. Each output node corresponds to a seed in k-means: its weight vector is exactly the seed. Instead of recomputing the seeds after each iteration, as in k-means, in a Kohonen network we will look at the data points one by one, and for each data point, we will look up the seed that is closest to that point, and move that seed a little bit in the direction of the data point. This means that given an input $\mathbf{x}$, the weight vector $\mathbf{w}_j$ for that output node o_j that is already closest to $\mathbf{x}$ is moved so that it becomes even closer to $\mathbf{x}$.

If we do this for each input data point $\mathbf{x}$, and start again from the first point once we've reached the last one, until the procedure converges (i.e., the weight vectors do not change anymore), then the weight vectors of the different output nodes will each be representative for a subset of the input vectors, in the same way that the seeds in k-means are.

10.4.2 Kohonen maps

As said earlier, however, Kohonen networks do a bit more than just clustering points in a way similar to k-means. The outputs of a Kohonen network are often considered to have a certain configuration with respect to each other. They might for instance be positioned on a two-dimensional grid. Output nodes that are next to each other in this configuration, horizontally or vertically, are called neighbors.

Note that "next to each other" refers to the grid configuration, not to the actual position of the corresponding weight vectors in space! Output node o_1 might be next to o_2 in the grid, but that does not necessarily mean that the corresponding weight vector $\mathbf{w}_1$ is also close to $\mathbf{w}_2$ in the input space. The idea of a Kohonen map is, however, that we try to make the weight vectors of neighboring output nodes similar. After convergence, some will indeed be similar, while others may still be far apart.

To achieve this, the update rule for weights in a Kohonen network is adapted a bit. When an instance $\mathbf{x}$ is closest to the weight vector of an output node o_j, not only $\mathbf{w}_j$ is adapted, but also the weight vectors of the neighbors of o_j, though

function LearnKohonenMap(training set T) **returns** weight matrix:
    W := random weight matrix
    $k := 0$
    repeat
        $k := k + 1$
        if $k > N$ **then** $k := 1$
        let i be such that $||\mathbf{x}_k - \mathbf{w}_i||$ is minimal
        $\mathbf{w}_i := \mathbf{w}_i + \sigma(\mathbf{x}_k - \mathbf{w}_i)$
        for each weight vector $\mathbf{w}_j$ in the neighborhood of $\mathbf{w}_i$:
            $\mathbf{w}_j := \mathbf{w}_j + \sigma_{ij}(\mathbf{x}_k - \mathbf{w}_j)$
    until stopping criterion fulfilled
    return W

Figure 10.14: Training a Kohonen network

these will be adapted a bit less than $\mathbf{w}_j$ itself. The amount of adaptation can be expressed by an update factor σ for the closest output node o_j, and a matrix of factors σ_{ij} that tells us how much o_j is updated when the closest vector is o_i. This gives us the algorithm for learning Kohonen maps shown in Figure 10.14.

By choosing this update rule, the weight vectors of neighboring nodes on the map will tend to converge to the same points. As a result, after convergence of the procedure, the grid in which the output nodes are positioned forms a kind of map of the data points, or a "sheet" that covers the actual input data.

While the principle of Kohonen maps is easily visualized with two-dimensional grids, one can also take a one- or three-dimensional grid, or define "neighbors" as being close in some other directions than the principle axes. For instance, one might have a hexagonal two-dimensional grid.

10.5 Bibliographical notes

In this chapter we have only scratched the surface of the massive amount of knowledge about neural networks that has been produced over several decades. Many important details, such as suitable stopping criteria (which help to avoid overfitting), how to choose a good step size σ, etc. have not been discussed here. Many books offer a good introduction or a more advanced treatment of neural networks. A very popular book that treats neural networks in detail is Bishop's *Neural networks for Pattern Recognition*; a good working knowledge of calculus and linear algebra is required however. Many texts are also available online; the Wikipedia entry on Artificial Neural Networks is a good starting point for finding online resources.

Chapter 11

Support vector machines

Support vector machines have become very popular in the late 1990's as a broadly applicable machine learning technique that in many cases yields very good predictive models. They rely on a lot of mathematical technicalities, which we will not discuss in detail in this text. However, there are also many interesting intuitions underlying SVMs that provide insight in how they work and why they work well. We will focus on these intuitions.

11.1 Maximum margin classifiers

Consider a set of examples $(\mathbf{x}_i, c(\mathbf{x}_i))$ where $\mathbf{x}_i \in \mathbb{R}^D$ is a vector in a D-dimensional vector space and $c(\mathbf{x}_i)$ is its class ("positive" or "negative"). In the context of support vector machines, $c(\mathbf{x}_i)$ is usually represented as 1 (for positive) or -1 (for negative).

We have seen that the perceptron attempts to learn a hyperplane in the D-dimensional input space that separates the positive examples from the negative examples. Assuming this is indeed possible, i.e., the example sets are linearly separable, then there may be multiple separating hyperplanes. The situation is sketched in Figure 11.1, where $D = 2$ and the hyperplane is simply a straight line. Several such lines separate the positive and negative examples perfectly.

While all of the lines perfectly separate the examples in the training set, we should be aware that this training set is a sample from the instance space X, and the line may not perfectly separate unseen examples.

Under the assumption that examples that are close together are likely to have the same classification, it makes sense to prefer the hyperplane that stays as far away from the training examples as possible. Indeed, when the line passes very near a training example $\mathbf{x}$, then some instances on the other side of the line will be very close to $\mathbf{x}$ and yet be predicted to have the opposite class.

So let us assume it is a good idea to try to find the hyperplane that separates the two sets of examples *and keeps a maximal distance from any of these examples.* The closest positive examples will be at the same distance from that hyperplane as the closest negative examples.[1] If we construct through the closest positive examples, as well as through the closest negative examples, a hyperplane that is parallel to the separating hyperplane, we obtain an area in between these hyperplanes that is

[1] Indeed, if the closest positive example is at distance d and the closest negative example at distance $d + \epsilon$, then we can shift the hyperplane in the direction of the negative example over a distance $\epsilon/2$; this new hyperplane will be at a larger distance from any example than the original, which contradicts the assumption that that hyperplane was at maximal distance from all examples.

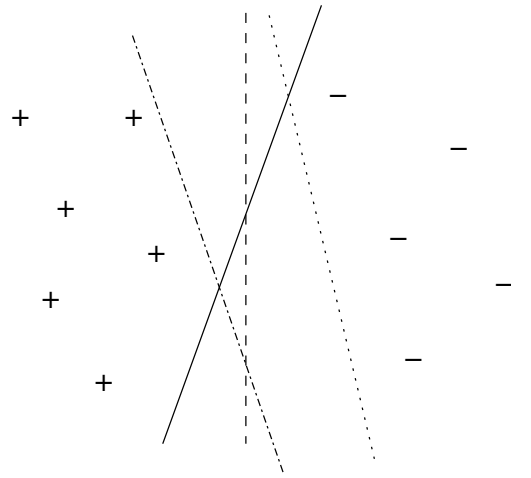

Figure 11.1: Several linear separators between positive and negative examples.

called the **margin** (see Figure 11.2), and which has the separating hyperplane right in the middle.

The margin contains no training examples, and has all the positive examples on one side and the negatives on the other side. We can consider it a "buffer" between the positives and the negatives. The margin has a width of $2d$ with d the distance between the separating hyperplane and the closest training examples. Thus, the separating hyperplane that is at maximal distance from the training examples creates a **maximum margin** between them. Figure 11.2 shows such a maximum margin.

To summarize the above reasoning: we are interested in finding the separating hyperplane that is at maximal distance from any training examples, or: that creates a maximum margin. We will call this optimal hyperplane a **maximum margin classifier** because of this property.

The next question is then: how can we find this hyperplane?

11.1.1 Computing the optimal hyperplane

There is a reasonable amount of technical detail and advanced mathematical methods involved in finding the maximum margin hyperplane. In this section we just discuss the intuitions. The main point is that the task of finding the hyperplane with maximal distance from the nearest training examples can be reformulated mathematically as a minimization problem.

To explain how this works, let us consider first the case of one-dimensional input examples. That is, the positives and negatives can be positioned on a single axis. A hyperplane in such a one-dimensional space is a single point. Linear separability means that all positives are to one side of this point, and all negative to the other side. The maximum margin separator is in this case the point right in the middle between the positive and negative example that are the closest to each other.

In one dimension, we can of course easily find these two points and compute the middle of these two. But in more dimensions this may be more difficult. Therefore, we will use a method that does not "directly" compute this midpoint, but finds it as the solution of a minimization problem. We here explain this minimization problem by first illustrating it for the one-dimensional case.

Assume we are given a set of N examples $(x_i, c(x_i))$, with $x_i \in \mathbb{R}$ and $c(x_i)$ equal

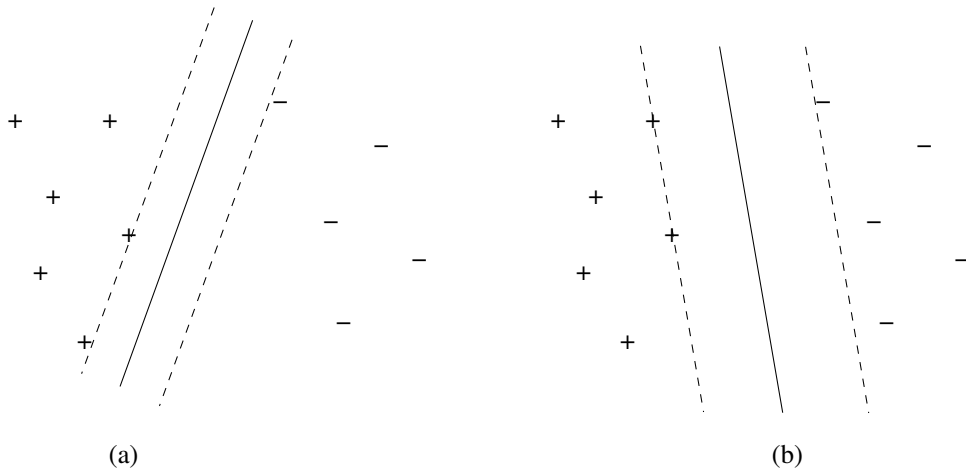

Figure 11.2: The "margins" induced by two linear separators. The right margin is wider than the left one, and is in fact the maximum margin: any other hyperplane would yield a narrower margin.

to $+1$ or -1. We can plot these examples in a two-dimensional X, Y-space with the value x indicated on the X axis and $c(x)$ on the Y axis, as is done in Figure 11.3.

Now consider a function $f(x) = ax + b$, with a and b some parameters. In the two-dimensional X, Y space, the line $y = f(x)$ is a straight line. It crosses the X-axis where $ax + b = 0$. The point where the line crosses the X-axis will be our separating point.

To find it, we first impose the following constraints on $f(x)$: for any x_i in the training set, we require that

$$\begin{aligned} &\text{if } c(x_i) = 1 \text{ then } f(x_i) \geq 1 \\ &\text{if } c(x_i) = -1 \text{ then } f(x_i) \leq -1. \end{aligned}$$

Mathematically, these two constraints are equivalent to the single constraint

$$f(x_i)c(x_i) \geq 1$$

and this is the form we will use in the remainder of the text.

The above constraints impose that the function $f(x) = ax+b$ should give a result of at least 1 for all positive examples, and at most -1 for all negatives. Choosing $f(x) = 0$ as the separating point means that we will predict positive whenever $f(x) > 0$ and negative otherwise, which together with our constraints implies that each positive training example will get a positive prediction, and every negative example a negative prediction.

Under these constraints, multiple functions $f(x)$ are still possible. All of them will correctly separate the positives and the negatives, crossing the X-axis at some point between the leftmost positive and the rightmost negative example. But not all of them cross the X-axis exactly in the middle of the closest positive and negative example. Figure 11.3(a) shows a few $f(x)$ functions.

We can obtain a unique solution by choosing that $f(x) = ax + b$ for which a is minimal. This is the line with the least steep slope, the line that is as horizontal as possible without violating the constraint that $f(x_i)c(x_i) \geq 1$ for each x_i in the training set.

It is intuitively clear that this line will be such that $f(x) = 1$ for exactly one positive example, and $f(x) = -1$ for exactly one negative example. More importantly,

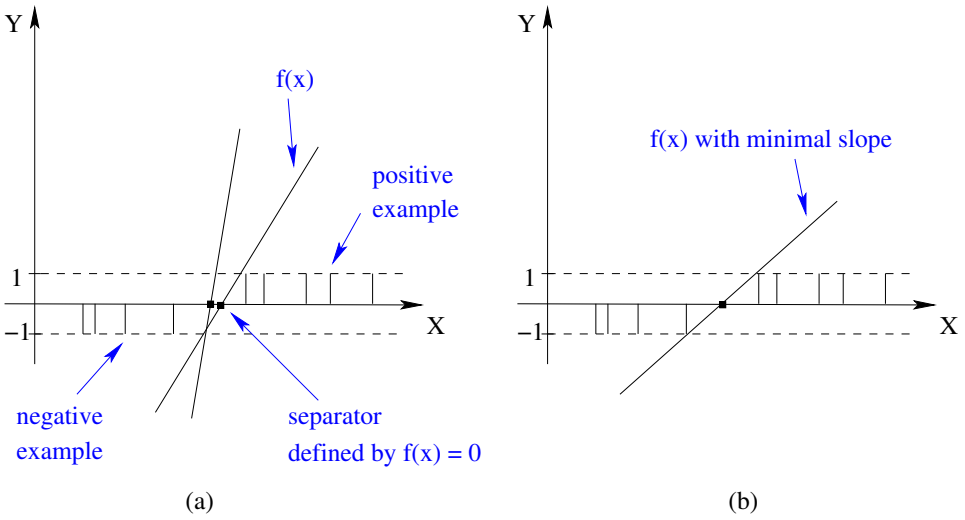

Figure 11.3: A point separating the positives and the negatives can be defined by $f(x) = ax + b = 0$, where $f(x)$ is such that $f(x)c(x) \geq 1$. Figure (a) shows two such points with the corresponding $f(x)$. By choosing that $f(x) = ax + b$ for which a is minimal, we obtain a separating point right in the middle of the two examples that are closest to the border between the positives and the negatives (b). These examples are called support vectors.

it is also clear that $f(x) = 0$ right in the middle of that positive and negative example, i.e., that $f(x) = 0$ defines a maximum margin separator. Finally, it is clear that there is exactly one minimal a: the minimization problem has a unique solution.

The examples closest to the separating point are called **support vectors**, expressing the physical intuition that the line $f(x)$ defining the separating point "rests" on them, or is supported by them.

Thus, the method for finding a maximum margin hyperplane in a one-dimensional input space can be formulated as follows:

Given: a set of data $(x_i, c(x_i))$,
Find: a function $f(x) = ax + b$ such that $\forall i : f(x_i)c(x_i) \geq 1$ and a is minimal.

Exactly the same reasoning can be used for higher-dimensional inputs. The input examples are then vectors $\mathbf{x}$ with $c(\mathbf{x}) = 1$ or $c(\mathbf{x}) = -1$, and we look for a linear function $f(\mathbf{x}) = \mathbf{a}\mathbf{x} + b$ such that $f(\mathbf{x})c(\mathbf{x}) \geq 1$ and $||\mathbf{a}||$, the norm of the vector $\mathbf{a}$, is minimal. The area in the input space where $f(\mathbf{x}) = 0$ is then a separating hyperplane.

Figure 11.4 illustrates this for a two-dimensional input space. Note that there are now three support vectors. Indeed, while the straight line $y = ax + b$ in the one-dimensional input space example is determined by two points, the plane $y = a_1x_1 + a_2x_2 + b$ is determined by three points. Intuitively, we can visualize the minimization problem as follows: imagine the positive cases as lying 1 cm above the page, and the negative lying 1 cm below the page; position a plane in the 3-D world such that it is above all the positives and below all the negatives; then turn the plane so that it becomes as parallel to the page as possible, without moving past any positives or negatives. The straight line where the plane crosses the page is the 2-dimensional linear separator.

The problem statement, for general D-dimensional spaces, is thus:

Given: a set of data $(\mathbf{x}_i, c(\mathbf{x}_i))$,

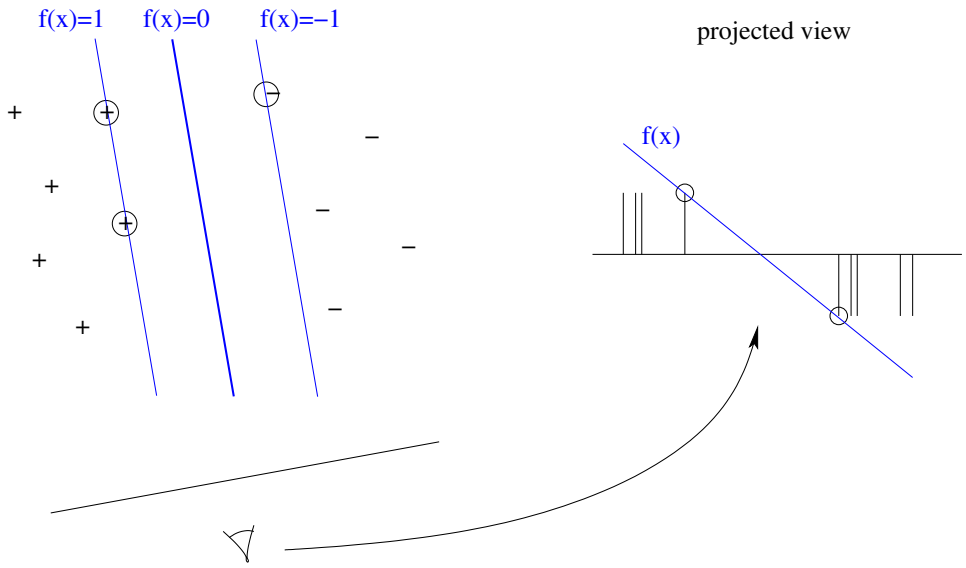

Figure 11.4: Solving the constrained minimization problem $f(x) = a_1x_1 + a_2x_2 + b$ with $f(x)c(x) \geq 1$ and $||\mathbf{a}|| = \sqrt{a_1^2 + a_2^2}$ minimal yields a linear separator with maximum margin in the two-dimensional input space.

Find: a function $f(\mathbf{x}) = \mathbf{a}\mathbf{x} + b$ such that $\forall i : f(\mathbf{x}_i)c(\mathbf{x}_i) \geq 1$ and $||\mathbf{a}||$ is minimal.

This is a minimization problem with constraints, for which methods are known for solving them. We briefly discuss them in the next section. This is the most mathematical part of this chapter.

11.1.2 Solving the minimization problem

The problem we are facing is a minimization problem with inequality constraints. It is a special case of a class of problems that can be described as follows:

Maximize $f(\mathbf{x})$ under a number of equality constraints $g_i(\mathbf{x}) = 0$ and inequality constraints $h_i(\mathbf{x}) \geq 0$.

(Of course, the same problem setting covers just as well minimization problems (just maximize $f'(\mathbf{x}) = -f(\mathbf{x})$ and problems with constraints $h_i(\mathbf{x}) \leq 0$ (just express those constraints using $h_i'(\mathbf{x}) = -h_i(\mathbf{x})$.)

When only equality constraints are given, this problem requires relatively simple calculus techniques; the method of Lagrange multipliers is used to solve such problems. With the introduction of inequality constraints, the problem becomes a bit more difficult, but the solution strategy is still quite similar. We will first discuss the Langrange method for equality constraints, then show how it is extended to handle inequalities.

Equality constraints

To understand how a function $f(\mathbf{x})$ is optimized under equality constraints, consider Figure 11.5, which shows a contour plot of a 2-D function f, and a 1-D line that corresponds to a constraint $g(\mathbf{x}) = 0$. Imagine $f(\mathbf{x})$ as a hill (seen from above in Figure 11.5), and the line $g(\mathbf{x}) = 0$ as a path on the hill; we are looking for the highest point along the path.

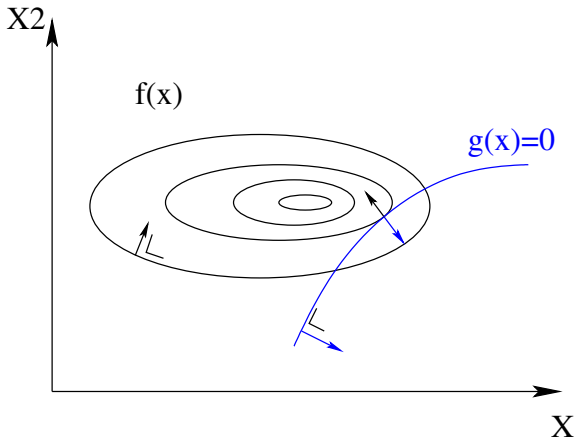

Figure 11.5: The figure shows a contour plot of a function f and a constraint $g(\mathbf{x}) = 0$. The maximum of $f(\mathbf{x})$ under the constraint $g(\mathbf{x}) = 0$ is obtained in a point where the line $g(\mathbf{x}) = 0$ runs along a contour line of f; in this point, the gradients ∇f and ∇g (indicated by arrows) are parallel.

On its highest point the path becomes horizontal. That implies that, at that point, the path coincides with a contour line (i.e., is tangential to the contour line going through that point).

Note that the contour lines of a function, in any point, are always perpendicular to the gradient of that function in that point. This means that the gradient of f, ∇f, is perpendicular to the line $g(\mathbf{x}) = 0$ in the optimum we are looking for.

Up till now we have used the function g simply as a means of stating a constraint; but let us now look at the function g itself. The line $g(\mathbf{x}) = 0$ is a contour line of this g, and the gradient of g, ∇g, is therefore always perpendicular to it.

When the line runs along a contour plot of f, this therefore means that the contour plots of f and g, and therefore also their gradients (which are perpendicular to these contour plots) are parallel. We can write this as $\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x})$: the gradient vector of f is equal to that of g up to a constant (which may be negative, in which case ∇f and ∇g are still parallel, but point in opposite directions).

Note that this λ can be interpreted as follows: if we would change the constraint g such that the line $g(\mathbf{x}) = 0$ moves a little bit in the direction of its gradient, then the optimal value of f changes with λ times that amount. λ can be seen as the "marginal effect" of g on the value of f in its optimum. (If, say, f is the profit obtained from a production process and g indicates constraints on the input resources, with $g = 0$ meaning that all resources must be consumed, then λ tells us how much more (or less) profit would be made if we had a bit more (or less) resources.)

The method of the Lagrange multipliers consists of finding the constrained maximum by defining a new function, the so-called Lagrangian L, as follows:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda g(\mathbf{x})$$

and optimizing L (without any further constraints) by setting all its partial derivatives to zero. That gives the following two equations:

$$\begin{aligned} \nabla f(\mathbf{x}) - \lambda \nabla g(\mathbf{x}) &= 0 \qquad \text{(after deriving to } \mathbf{x}\text{)} \\ g(\mathbf{x}) &= 0 \qquad \text{(after deriving to } \lambda\text{)} \end{aligned}$$

where the second equation expresses precisely our constraint, and the first expresses that $\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x})$. Thus, by defining this L, we have turned the constraint

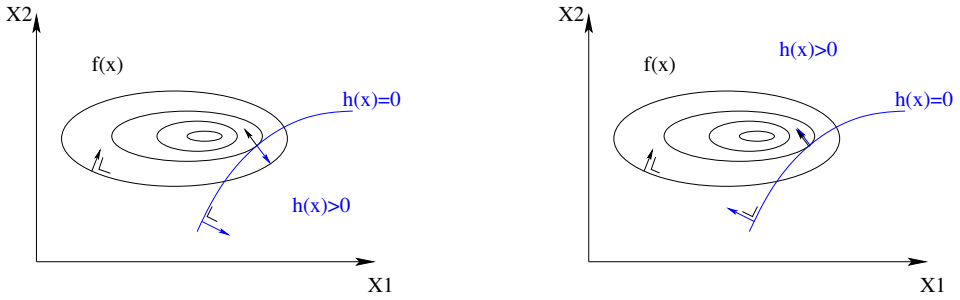

Figure 11.6: The figures shows a contour plot of f and a constraint $h(\mathbf{x}) \geq 0$. In the left figure, $h > 0$ to the lower right of the line $h(\mathbf{x}) = 0$; in the right figure, $h > 0$ to the upper left of that line. In the left figure, the maximum of $f(\mathbf{x})$ under the constraint $h(\mathbf{x}) \geq 0$ is obtained in the same point as in Figure 11.5, i.e., on the line where $h(\mathbf{x}) = 0$ (the constraint is active). In the right figure, we need to be in the area to the upper left of the $h = 0$ line. The optimum is then the same as what we would have if there were no constraints at all, namely at the top of the f-"hill".

optimization problem into an unconstrained one that gives us exactly the same solution for $\mathbf{x}$ and λ.

Inequality constraints

When we have an inequality constraint $h(\mathbf{x}) \geq 0$, there are two possibilities. The region $h(\mathbf{x}) \geq 0$ is bordered by the line $h(\mathbf{x}) = 0$, and the gradient ∇h is perpendicular to this line and points in the direction of increasing h, i.e., it points to the area where $h(\mathbf{x}) > 0$. In the hill-climbing example, imagine the line $h(\mathbf{x}) = 0$ as a fence: we can walk along the fence or walk away from it, but we cannot cross it.

Now, walking along the line $h(\mathbf{x}) = 0$, we can again walk until we arrive in a point where $\nabla f = \mu \nabla h$, i.e., the contour lines of f and h are parallel (the path along the fence becomes horizontal; we can only go up by moving away from the fence). Now there are two possibilities (see Figure 11.6):

- The gradient of f is in the same direction as the gradient of h ($\mu > 0$): this means that moving away from the border line into the acceptable region increases f. In that case, the line is not constraining the maximization problem in this point. We say that the constraint h is *inactive*. (The fence blocks our way down, but not our way up; so we can go up by simply moving away from the fence.)

- The gradient of f is in the opposite direction as the gradient of h ($\mu < 0$): this means that to increase f we would have to cross the border, which is not allowed. Moving away from the border only decreases f. The highest reachable point for f is therefore located on the line $h(\mathbf{x}) = 0$. We say that the constraint h is *active* in this point: it constrains the optimization problem. (The fence blocks our way up, and the path along the fence has reached its highest point.)

Note that here, again, the μ factor can be seen as how much higher or lower we get by making one step away from the fence. In terms of the "maximizing profit" example, the $h(\mathbf{x}) \geq 0$ corresponds to resource constraints, imposing for instance a maximum on the number of resources that can be used (but allowing for fewer

resources to be used).[2] The constant μ for an active constraint refers to the extra profit that relaxing that constraint yields.

Concerning the terminology of active and inactive constraints: whether a constraint is active or inactive depends of course on the point that we are considering. But once we have found a maximum for f under the constraints, either this maximum is on a line $h(\mathbf{x}) = 0$, in which case we simply call h active, or it is somewhere in the region $h(\mathbf{x}) > 0$, in which case we call h inactive.

As with equality constraints, the constrained optimization problem can be solved by defining the Lagrangian:

$$L(\mathbf{x}, \mu) = f(\mathbf{x}) - \mu h(\mathbf{x})$$

and optimizing L. Here, however, we need to distinguish two cases, depending on whether the constraint is active or not. Deriving the Lagrangian to $\mathbf{x}$ gives us, as before,

$$\nabla f(\mathbf{x}) - \mu \nabla h(\mathbf{x}) = 0$$

which implies

$$\nabla f(\mathbf{x}) = \mu \nabla h(\mathbf{x}).$$

For an active constraint, we will have $h(\mathbf{x}) = 0$ and $\mu < 0$. The requirement $\mu < 0$ is there because ∇h needs to point in the opposite direction from ∇f, otherwise the constraint cannot be active. (In our "fence terminology", with $\mu > 0$, the fence would be blocking our way down, not our way up.)

For an inactive constraint, the optimum will be characterized by $\nabla f(\mathbf{x}) = 0$, like any unconstrained optimum. ∇h can be anything in this point. The relationship between ∇f and ∇h then still holds if and only if $\mu = 0$. Note that this is consistent with the interpretation for μ we just mentioned: if a constraint is inactive, then modifying it a bit will not affect the optimal value for f, so also in this case, μ is the marginal effect on the optimum for f of any change in h.

We can summarize the two cases (active/inactive) as follows: either $h(\mathbf{x}) = 0$ and $\mu < 0$ in our optimum (the constraint is then active), or $h(\mathbf{x}) > 0$ and $\mu = 0$. This can be summarized as: $h(\mathbf{x}) \geq 0$, $\mu \leq 0$, and $h(\mathbf{x})\mu = 0$.

Multiple constraints

Maximizing $f(\mathbf{x})$ under equality constraints $g_i(\mathbf{x}) = 0$ and inequality constraints $h_i(\mathbf{x}) \geq 0$ can be done as follows. We define a function

$$L(\mathbf{x}, \mathbf{\Lambda}, \mathbf{M}) = f(\mathbf{x}) - \sum_i \lambda_i g_i(\mathbf{x}) - \sum_i \mu_i h_i(\mathbf{x})$$

where $\mathbf{\Lambda}$ and $\mathbf{M}$ represent vectors that have as components the λ_i and μ_i, respectively.

The solution to the constrained maximization problem is reached in a point where the partial derivatives of L to all components of $\mathbf{x}$ and to all λ_i is zero. For μ_i it is a bit more complicated: we have the conditions that, first, $\mu_i \leq 0$ and second, either $\mu_i = 0$ or $h_i(\mathbf{x}) = 0$.

Putting all these conditions together gives:

$$\begin{aligned} \nabla f(\mathbf{x}) &= \sum_i \lambda_i \nabla g_i(\mathbf{x}) + \sum_i \mu_i \nabla h_i(\mathbf{x}) \\ \forall i : g_i(\mathbf{x}) &= 0 \end{aligned}$$

[2] This constraint of course would use $\leq$ instead of $\geq$, but remember that any inequality constraint can be written in $\geq$ form by just multiplying h with -1 if necessary.

$$\begin{aligned} \forall i : h_i(\mathbf{x}) &\geq 0 \\ \forall i : \mu_i &\leq 0 \\ \sum_i \mu_i h_i(\mathbf{x}) &= 0 \end{aligned}$$

If we are minimizing f, instead of maximizing it, then exactly the same reasoning can be followed. The conditions then look as follows:

$$\nabla f(\mathbf{x}) = \sum_i \lambda_i \nabla g_i(\mathbf{x}) + \sum_i \mu_i \nabla h_i(\mathbf{x}) \tag{11.1}$$

$$\forall i : g_i(\mathbf{x}) = 0 \tag{11.2}$$

$$\forall i : h_i(\mathbf{x}) \geq 0 \tag{11.3}$$

$$\forall i : \mu_i \geq 0 \tag{11.4}$$

$$\sum_i \mu_i h_i(\mathbf{x}) = 0 \tag{11.5}$$

Note that the conditions on μ_i are now that $\mu_i \geq 0$. Indeed, if we are minimizing f (going down the hill), then obviously the constraint is active (the fence blocks our way) if it blocks our way down, rather than our way up.

The above conditions are known as the **Karush-Kuhn-Tucker conditions**. Leaving aside certain degenerate cases, these are necessary conditions that an optimum of f under the given constraints must satisfy. They are not sufficient conditions: as is generally the case in optimization, having all partial derivatives equal to zero does not imply having an optimum; second order derivates are relevant too. Moreover, even real optima may be local, rather than global, optima. But for the particular kind of problems that we consider here, there will always be a unique point that fulfills these conditions and that is optimal (this is because the functions that we will consider are convex).

Applying this to maximum margin classifiers

How does all the above apply to the constrained minimization problem that we encountered in the context of maximum margin classifiers? In this context, we want to minimize $||\mathbf{a}||$ (note that it is not a function of $\mathbf{x}$ but a function of $\mathbf{a}$ that we are minimizing). In practice, we will minimize $F(\mathbf{a}, b) = ||\mathbf{a}||^2/2$; clearly the optimal solution is the same in both cases. We call the function to be optimized F instead of f, to avoid confusion with the function $f(\mathbf{x}) = \mathbf{a}\mathbf{x} + b$; and we include b as a parameter, even though F is constant in b, for a reason that will be clear in a moment.

We have no equality constraints. We do have inequality constraints, which are of the form $f(\mathbf{x}_i)c(\mathbf{x}_i) \geq 1$. This can be rewritten as $(\mathbf{a}\mathbf{x}_i + b)c(\mathbf{x}_i) - 1 \geq 0$, which gives us N constraint functions $h_i(\mathbf{a}, b)$ defined as follows:

$$h_i(\mathbf{a}, b) = (\mathbf{a}\mathbf{x}_i + b)c(\mathbf{x}_i) - 1. \tag{11.6}$$

Since we have a minimization problem, μ_i is constrained by $\mu_i \geq 0$.

Note that although b does not occur in the function to be minimized, it does occur in the constraints, and therefore the Lagrangian will be a function of b as well. We will need to set the derivatives to both $\mathbf{a}$ and b to zero. That is why we included b as a parameter of F, even though it did not seem necessary there.

The relationship between the gradients of F and the h_i then becomes

$$\nabla F(\mathbf{a}, b) = \sum_i \mu_i \nabla h_i(\mathbf{a}, b)$$

which, with $F(\mathbf{a}, b) = ||\mathbf{a}||^2/2$ and h_i as above (Equation 11.6), gives (deriving consecutively to $\mathbf{a}$ and to b)

$$\mathbf{a} = \sum_i \mu_i \mathbf{x}_i c(\mathbf{x}_i) \tag{11.7}$$

$$0 = \sum_i \mu_i c(\mathbf{x}_i) \tag{11.8}$$

Eliminating $\mathbf{a}$ from the Lagrangian using Equation 11.7 and observing that b only occurs there in a term $(\sum_i \mu_i c(\mathbf{x}_i))b$, which is zero according to Equation 11.8, we obtain:

$$\begin{aligned}
L(M) &= 1/2||\mathbf{a}||^2 - \sum_i \mu_i((\mathbf{a}\mathbf{x}_i + b)c(\mathbf{x}_i) - 1) \\
&= 1/2||\mathbf{a}||^2 - \sum_i \mu_i \mathbf{a}\mathbf{x}_i c(\mathbf{x}_i) - \sum_i \mu_i b c(\mathbf{x}_i) + \sum_i \mu_i \\
&= 1/2||\mathbf{a}||^2 - \sum_i \mu_i \mathbf{a}\mathbf{x}_i c(\mathbf{x}_i) + \sum_i \mu_i \\
&= 1/2||(\sum_i \mu_i \mathbf{x}_i c(\mathbf{x}_i))||^2 - \sum_i \mu_i (\sum_j \mu_j \mathbf{x}_j c(\mathbf{x}_j))\mathbf{x}_i c(\mathbf{x}_i) + \sum_i \mu_i \\
&= 1/2 \sum_i \mu_i c(\mathbf{x}_i)(\sum_j \mu_j c(\mathbf{x}_j)\mathbf{x}_j)\mathbf{x}_i - \sum_i \mu_i (\sum_j \mu_j \mathbf{x}_j c(\mathbf{x}_j))\mathbf{x}_i c(\mathbf{x}_i) + \sum_i \mu_i \\
&= 1/2 \sum_i \sum_j \mu_i \mu_j c(\mathbf{x}_i) c(\mathbf{x}_j) \mathbf{x}_i \mathbf{x}_j - \sum_i \sum_j \mu_i \mu_j c(\mathbf{x}_i) c(\mathbf{x}_j) \mathbf{x}_i \mathbf{x}_j + \sum_i \mu_i \\
&= -1/2 \sum_i \sum_j \mu_i \mu_j c(\mathbf{x}_i) c(\mathbf{x}_j) \mathbf{x}_i \mathbf{x}_j + \sum_i \mu_i
\end{aligned}$$

This function is to be optimized under the constraints given by Equations 11.8 and 11.4. This can be done using efficient numerical methods. The details of this are beyond the scope of this text, but the form of the above equation is interesting, as will become clear later.

Solving this optimization problem gives us values for the μ_i. The support vectors are those training instances $\mathbf{x}_i$ for which $\mu_i > 0$. Indeed, the notion of active and inactive constraints corresponds to the vectors being support vectors or not (recall that for a support vector $\mathbf{x}_i$, $f(\mathbf{x}_i)c(\mathbf{x}_i) = 1$, which means $h_i = 0$ for this vector).

Note that the above system does not give us the value of b, since b has disappeared from the equations. We will see in a moment how b can be computed.

11.2 Linear support vector machines

The previous section introduces the basic principles underlying **linear support vector machines**. Under the assumption that the positives and negatives are linearly separable, i.e., there exists a hyperplane that separates them, a linear support vector machine finds the maximal margin hyperplane by solving the optimization problem just described.

Besides the fact that support vector machines are maximum margin classifiers, they have another crucial property: the solution to the minimization problem, i.e., the separating hyperplane, is not represented using the parameters that define the hyperplane, but simply by listing the support vectors together with one parameter for each support vector. For linear SVMs, this may not seem so very important, but for non-linear SVMs (which we will see later) this is an important property.

In the following, we will discuss what this support vector representation looks like, and how predictions can be made using this representation.

11.2.1 Representation of the solution

The solution of the minimization problem, the hyperplane $f(x) = \mathbf{a}\mathbf{x} + b$, can of course be represented simply using that equation. We just need to store $\mathbf{a}$ and b. But there is another representation possible, which will be very useful when we consider non-linear support vector machines.

Remember that the hyperplane is uniquely determined by the support vectors. In fact, the mathematical formulation of the minimization problem directly yields parameters μ_i, one μ_i for each training example. The μ_i are zero for each example that is not a support vector, and non-zero for each support vector.

Observing that $\mathbf{a} = \sum_i \mu_i \mathbf{x}_i c(\mathbf{x}_i)$ (see Equation 11.7), we can fill this in in the equation $f(\mathbf{x}) = \mathbf{a}\mathbf{x} + b$ to obtain

$$f(\mathbf{x}) = (\sum_i \mu_i \mathbf{x}_i c(\mathbf{x}_i))\mathbf{x} + b = \sum_i \mu_i c(\mathbf{x}_i)\mathbf{x}_i\mathbf{x} + b.$$

Since the μ_i are zero except for the support vectors, we can have the summation range just over these support vectors:

$$f(\mathbf{x}) = \sum_{\mathbf{s}_i \in S} \mu_i c(\mathbf{s}_i)\mathbf{s}_i\mathbf{x} + b$$

with S the set of support vectors.

Finally, the constant b can be determined by observing that for any support vector $\mathbf{s}_i$ we have $f(\mathbf{s}_i)c(\mathbf{s}_i) = 1$. Thus, taking a random support vector $\mathbf{s}$, we have

$$f(\mathbf{s})c(\mathbf{s}) = (\sum_i \mu_i c(\mathbf{s}_i)\mathbf{s}_i\mathbf{s} + b)c(\mathbf{s}) = 1$$

which implies

$$\sum_i \mu_i c(\mathbf{s}_i)\mathbf{s}_i\mathbf{s} + b = 1/c(\mathbf{s}) = c(\mathbf{s})$$

and hence

$$b = c(\mathbf{s}) - \sum_i \mu_i c(\mathbf{s}_i)\mathbf{s}_i\mathbf{s}.$$

Since this holds for any support vector $\mathbf{s}$, for reasons of numerical stability the right hand side can also be computed for each separate support vector and then averaged:

$$b = \frac{1}{|S|}\sum_{\mathbf{s}\in S}(c(\mathbf{s}) - \sum_i \mu_i c(\mathbf{s}_i)\mathbf{s}_i\mathbf{s}).$$

Thus, for any instance $\mathbf{x}$, the prediction $f(\mathbf{x})$ can be written in terms of the scalar products of $\mathbf{x}$ with the support vectors. The vector $\mathbf{a}$ itself need not be explicitly computed.

So we essentially have two ways of making a prediction $f(\mathbf{x})$ for a new vector $\mathbf{x}$:

1. computing $\mathbf{a}\mathbf{x} + b$
2. computing $\sum_{\mathbf{s}_i \in S} \mu_i c(\mathbf{s}_i)\mathbf{s}_i\mathbf{x} + b$

The first makes use of an explicit representation of the function $f(\mathbf{x})$, the second makes use of the fact that the support vectors implicitly represent it. While there may not seem to be any advantage to using the second representation instead of the first, this will change when we look at non-linear support vector machines.

11.3 Linear support vector machines for non-linearly-separable data

What if we have data that are not linearly separable? In that case, there does not exist a separating hyperplane; in other words, a function f such that $f(\mathbf{x}_i)c(\mathbf{x}_i) \geq 1$ for all training instances $\mathbf{x}_i$ does not exist.

We can then still find a solution if we soften the constraints, demanding

$$f(\mathbf{x}_i)c(\mathbf{x}_i) \geq 1 - \xi_i$$

where the $\xi_i \geq 0$ are not necessarily all zero, but preferably still as small as possible. Figure 11.7 shows how any instance within the margin or on the wrong side of the margin has a non-zero ξ_i.

Since we want to minimize not only the $\mathbf{a}$ vector but also the ξ_i, the minimization problem now becomes

$$\text{minimize} \quad ||\mathbf{a}|| + C\sum_i \xi_i \quad \text{where } f(\mathbf{x}) = \mathbf{a}\mathbf{x} + b \text{ and } f(\mathbf{x}_i)c(\mathbf{x}_i) \geq 1 - \xi_i$$

where the user-defined parameter C trades off the importance of minimizing $||\mathbf{a}||$ with the importance of minimizing the so-called "slack variables" ξ_i. With a large C we obtain a hyperplane that tries hard to separate all the positives from all the negatives; with a smaller C we get a hyperplane that accepts that some examples may be "on the wrong side", and tries to maximize the distances of "most" examples from the separating hyperplane. Note that this may result in exceptional cases lying on the wrong side of the separating line even if the data are linearly separable.

11.4 Non-linear support vector machines

Support vector machines construct a maximum margin hyperplane, where the maximum margin constraint can be relaxed by introducing slack variables. While this relaxation allows us to handle non-linearly-separable datasets, the separator is still linear. We now turn to the construction of non-linear SVMs, which can construct non-linear boundary surfaces.

11.4.1 Transformation to another space

The basic idea behind non-linear support vector machines is the following. Given an input space with vectors $\mathbf{x}$, we can transform these vectors to another space, sometimes called the **feature space**, using some transformation function Φ. That is, we map each vector $\mathbf{x}$ onto its image $\Phi(\mathbf{x})$ before proceeding. For instance, we could define Φ as follows: $\Phi([x_1, x_2]^T) = [x_1, x_1x_2, x_2]^T$.

A dataset that was not linearly separable in the original space might be linearly separable in the new, typically higher-dimensional, space. Whether this is indeed the case, or not, will in general depend on the function Φ as well as on the original dataset.

Example 11.1 Consider a 1-D input space with the following training examples $(x, c(x))$: $\{(-2, -1), (-0.5, +1), (0.3, +1), (1.2, -1), (1.5, -1)\}$. (See Figure 11.8.) This dataset is not linearly separable: there is no single point separating the examples with class $+1$ from those with class -1. Now take $\Phi(x) = x^2$. After transforming the data using this Φ, the dataset becomes $\{(4, -1), (0.25, +1), (0.09, +1), (1.44, -1), (2.25, -1)\}$. The transformed data are linearly separable and the maximum margin separator in this new space is $(0.25 + 1.44)/2 = 0.845$.

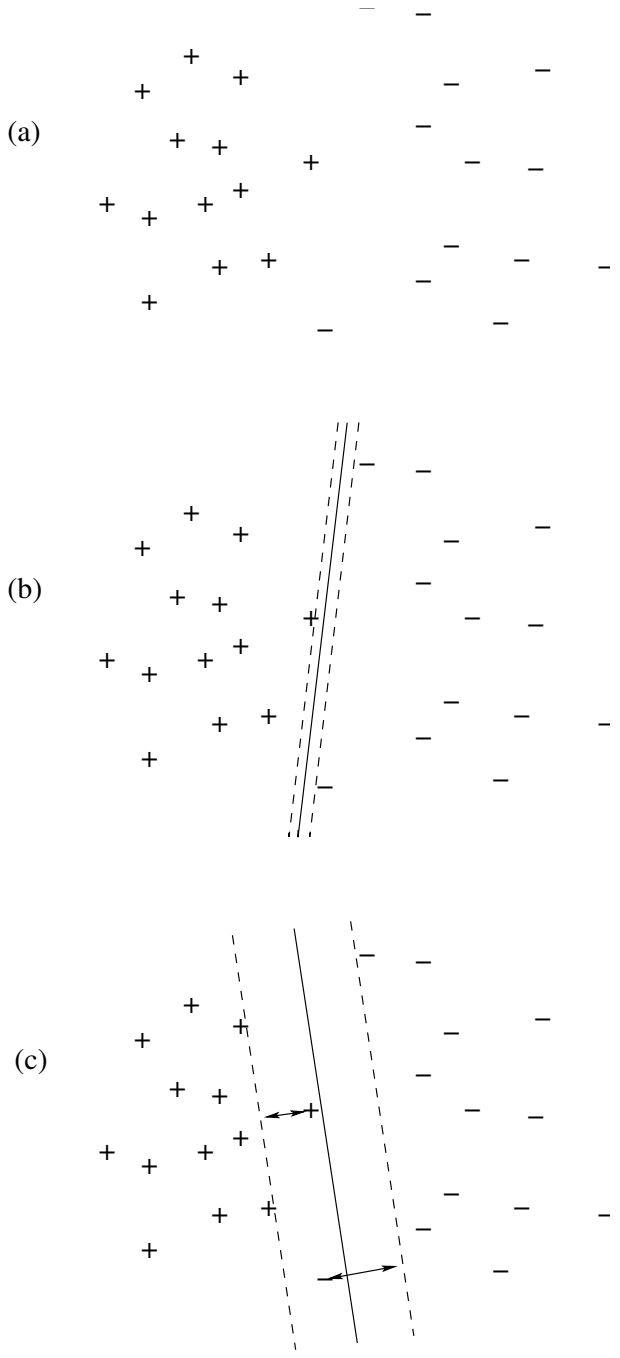

Figure 11.7: (a) A dataset that is linearly separable. (b) The optimal separator under strict conditions (allowing no exceptions) has a very narrow margin. (c) A "separating" hyperplane where two training instances are inside the margin, or on the wrong side of the separator. These instances violate the constraint that $f(\mathbf{x}_i)c(\mathbf{x}_i) \geq 1$. The distance to the area where they should be, in order not to violate the constraints, is indicated by an arrow. This distance is proportional to the slack variable ξ_i. While a few exceptional cases are now in a forbidden area, the margin of the separator when we disregard these exceptions is much wider.

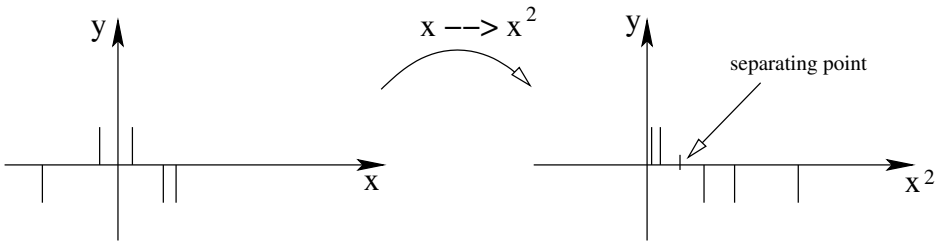

Figure 11.8: (a) A one-dimensional dataset that is not linearly separable. (b) After applying a transformation that maps each x onto its square, the transformed dataset is linearly separable.

Assuming that the data are linearly separable in the new space, we can construct a linear maximum margin separator there and transform it back to the original space (assuming Φ is invertible). Note that the separator is linear in the feature space but may not be linear in the original space. Also, its maximum margin property may not hold in the original space: the separator constructed in the original space may not be at maximal distance from all the training examples, even though its image in the transformed space is.

Example 11.2 The previous transformation $\Phi(x) = x^2$ was not invertible: given a result 4, we do not know whether it comes from $x = 2$ or $x = -2$. A similar transformation, $\Phi(x) = [x, x^2]$, is invertible. $\Phi^{-1}([x_1, x_2]) = x_1$ if $x_2 = x_1^2$ and undefined otherwise.

For the data from Example 11.1, the transformed data set is now $\{([-2, 4], -1)$, $([-0.5, 0.25], +1)$, $([0.3, 0.09], +1)$, $([1.2, 1.44], -1)$, $([1.5, 2.25], -1)\}$. We get as maximum margin separator $x_2 = -2/3x_1 + 1.265$. There are two support vectors: $[0.3, 0.09]$ and $[1.2, 1.44]$.

The line $x_2 = -2/3x_1 + 1.265$ can be transformed back to the original space, but only those points on the line where $x_2 = x_1^2$ have a defined image (for the other points, Φ^{-1} is undefined). Filling in $x_2 = x_1^2$ and rearranging, we get $3x_1^2 + 2x_1 - 3.795 = 0$. Solving this equation we get solutions $x_1 = 0.80$ and $x_1 = -1.47$.

The consecutive steps are illustrated in Figure 11.9.

Note that the two separating points in the original space indeed separate the positives from the negatives, but are not at a maximal distance from the training examples. The point $x = -1.47$, for instance, is closer to -2 than to -0.5.

For the purpose of illustration we have used very simple transformations and low-dimensional input spaces up till now. But the same reasoning can of course be used for more complex spaces.

Example 11.3 Consider a 2-D input space (x_1, x_2). Consider the following transformation, which transforms this 2-D space (x_1, x_2) into a 5-D space $(x'_1, x'_2, x'_3, x'_4, x'_5)$ as follows.

$$\Phi([x_1, x_2]^T) = [x_1, x_2, x_1x_2, x_1^2, x_2^2]^T$$

The linear separator in this new five-dimensional space looks as follows:

$$ax'_1 + bx'_2 + cx'_3 + dx'_4 + ex'_5 + f = 0$$

which according to the definition of Φ can be written in the original coordinates as

$$ax_1 + bx_2 + cx_1x_2 + dx_1^2 + ex_2^2 + f = 0.$$

Note that what is a linear equation (a hyperplane) in the transformed coordinates, is a second-degree equation (a quadratic curve) in the original coordinates. Thus, the

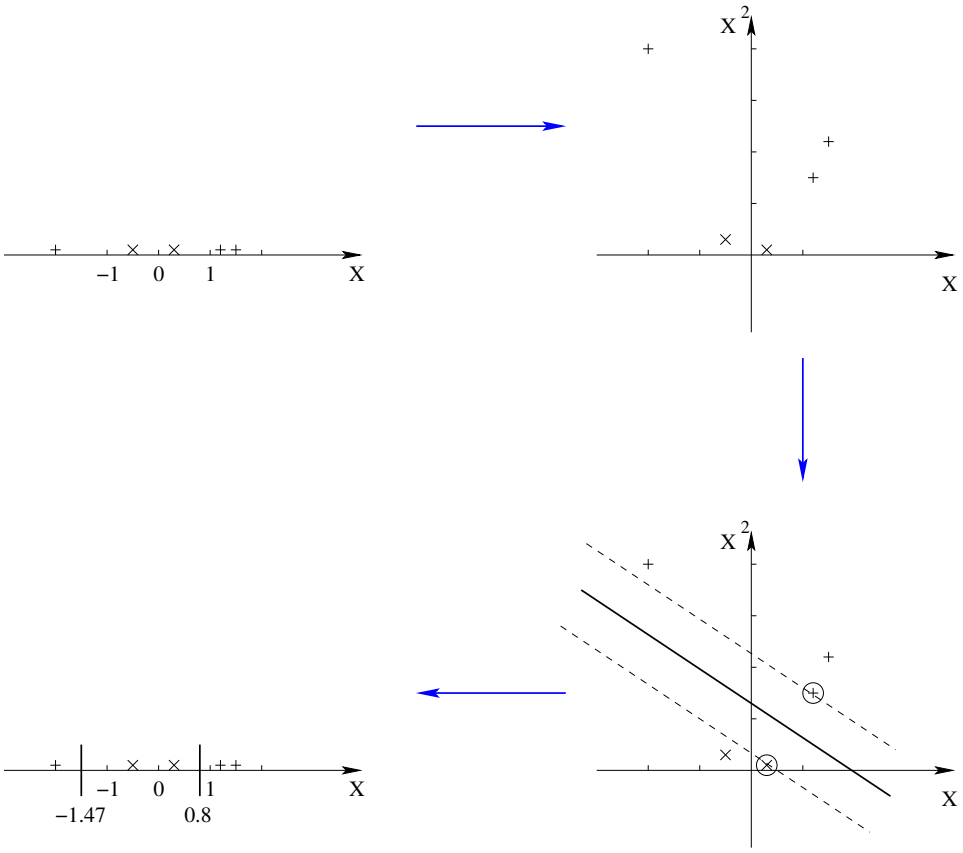

Figure 11.9: Example of mapping x onto $(x_1, x_2) = (x, x^2)$, constructing the maximum margin hyperplane in that space, and mapping that hyperplane back to the original space. The positive and negative examples are here represented by +'s and ×'s.

separating hyperplane in the transformed space will map to an elliptic, hyperbolic or parabolic separator in the original space.

Generally, for any n-D vector, we can transform that vector to a new vector containing as components all second-degree combinations of the original coordinates, or even all higher-degree combinations of them. Thus, separators defined by any polynomial equation in the original coordinates can be obtained.

Obviously, the resulting space will have many dimensions. For instance, with a 10-D space, the transformed space when using only second-degree combinations (in addition to the original coordinates) has the 10 original coordinates x_i, 10 squared coordinates x_i^2, and $10 * 9/2 = 45$ variables of the form $x_i x_j$ with $i \neq j$. This yields a 65-D transformed space. If we include third-degree polynomials, we have to add 10 (from the x_i^3) + 90 ($x_i^2 x_j$) + 120 ($x_i x_j x_k$) = 220 dimensions, yielding a 285-D space.

The dimensionality of the new space can clearly become very high. On the one hand, this is good: data are much more likely to be linearly separable in such a high-dimensional space. On the other hand, it raises worries about the curse of dimensionality: in such a high-dimensional space, isn't there a high chance of overfitting? A second concern is efficiency: how complex will the computations become?

Interestingly, maximum margin approaches tend not to suffer much from the curse of dimensionality. The reasons for this have been studied in statistical learning, but they are beyond the scope of this text.

As for the complexity of the approach: the nice thing about support vector machines is that they avoid the complex computations needed for computing the transformation, finding the maximum margin hyperplane, and transforming that back to the original space. They do all this *implicitly*: they perform other, simpler, computations, for which it can be proven that the result is the same as the result that one would get when performing the above computations. The key to this is twofold: first, the use of *kernel functions*, and second, the property of support vector machines that we earlier saw, namely that the *separating hyperplane is represented implicitly by the support vectors*, and predictions can be made by just computing the scalar product of the vector for which the prediction is being made with these support vectors.

11.4.2 Kernels

Let us first consider again the use of the support vectors, and their scalar product with a new vector, to make predictions for that vector:

$$f(\mathbf{x}) = \sum \mu_i c(\mathbf{s}_i) \mathbf{s}_i \mathbf{x} + b.$$

Since in the new setting of non-linear SVMs, we construct an SVM in the transformed space, the natural method to predict the class of a vector $\mathbf{x}$ is to transform it to the higher-dimensional space and then apply the SVM in that space to make a prediction for $\mathbf{x}$:

$$f(\mathbf{x}) = \sum \mu_i c(\mathbf{s}_i) \Phi(\mathbf{s}_i) \Phi(\mathbf{x}) + b$$

where i ranges over all $\mathbf{s}_i$ for which $\Phi(\mathbf{s}_i)$ is a support vector in the transformed space. We now call $\mathbf{s}_i$ a support vector if the corresponding $\Phi(s_i)$ is a support vector in the transformed space.

Now consider a function $K(\mathbf{x}, \mathbf{y})$ defined as $\Phi(\mathbf{x})\Phi(\mathbf{y})$. K is called a **kernel function**. Generally, any function K for which a transformation Φ exists such that $K(\mathbf{x}, \mathbf{y}) = \Phi(\mathbf{x})\Phi(\mathbf{y})$ (i.e., there exists a transformation such that K expresses the

scalar product of two vectors after applying that transformation to them) can be called a kernel function.

Using this kernel function K, we can rewrite the previous equation as

$$f(\mathbf{x}) = \sum \mu_i c(\mathbf{s}_i) K(\mathbf{x}, \mathbf{s}_i) + b.$$

Thus, it is enough to have the support vectors $\mathbf{s}_i$ with their corresponding μ_i, and the kernel function K, to make a prediction.

Now the interesting thing is, also the minimization problem itself can be formulated in terms of K instead of Φ. Recall the Lagrangian that we needed to optimize in the linear case:

$$L(M) = -1/2 \sum_i \sum_j \mu_i \mu_j c(\mathbf{x}_i) c(\mathbf{x}_j) \mathbf{x}_i \mathbf{x}_j + \sum_i \mu_i$$

Here, too, the $\mathbf{x}_i$ and $\mathbf{x}_j$ only occur as scalar products. If we optimize in the feature space, the vectors are replaced by $\Phi(\mathbf{x}_i)$ and $\Phi(\mathbf{x}_j)$, which gives for their scalar product $K(\mathbf{x}_i, \mathbf{x}_j)$. Thus, learning a maximum margin classifier in the feature space reduces to optimizing

$$L(M) = -1/2 \sum_i \sum_j \mu_i \mu_j c(\mathbf{x}_i) c(\mathbf{x}_j) K(\mathbf{x}_i, \mathbf{x}_j) + \sum_i \mu_i$$

under the constraints

$$\begin{aligned} \forall i : \mu_i &\geq 0 \\ \sum_i \mu_i c(\mathbf{x}_i) &= 0 \end{aligned}$$

We see that finding a maximum margin separator in the feature space is not more difficult than finding a maximum margin separator in the original space: we simply replace each scalar product by an evaluation of the kernel function.

11.4.3 Useful kernel functions

Many kernel functions can in principle be used by a support vector machine. There are a number of specific kernel functions that are often used. These include:

- polynomial kernel functions: these are of the form $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}\mathbf{y} + 1)^p$ with p a parameter. For $p = 2$, we get the quadratic decision surfaces indicated earlier.
- Gaussian kernel functions: $K(\mathbf{x}, \mathbf{y}) = \exp(-||\mathbf{x} - \mathbf{y}||^2 / 2\sigma^2)$.

Note that the scalar product $\mathbf{x}\mathbf{s}_i$ can be seen as a similarity measure; it measures to what extent $\mathbf{x}$ and $\mathbf{s}_i$ point in the same direction, and hence, are the same vector up to a scalar factor. Thus, also kernel functions can be considered a similarity measure: $K(\mathbf{x}, \mathbf{y})$ should have a large value when $\mathbf{x}$ and $\mathbf{y}$ are similar. The notion of similarity is somewhat different now: similarity is measured as the scalar product in the feature space, rather than in the original space.

The question remains how to choose the most appropriate kernel function. This is to some extent a matter of experimentation. Some kernel functions, such as the Gaussian, generally give quite good results.

Whether a kernel performs well also depends on whether a good balance is found between the expressiveness of a kernel and its risk of overfitting (in other words, roughly, between the bias and the variance of the method using the kernel). To

find a good balance between these, the principle of **structural risk minimization** (SRM) has been proposed.

When learning a hypothesis using a certain hypothesis space, a probabilistic upper bound can be constructed on the difference between the error on the training set and the actual error. This upper bound relies on the VC-dimension of the hypothesis space.

Let $e(h)$ be the expected error of h on the instance space, as obtained by adding this upper bound to the error of h on the training set.

The idea behind structural risk minimization is the following. Assume given a sequence of hypothesis spaces $\mathcal{H}_i$ with increasing VC-dimension, and correspondingly an increasing upper bound $u(\mathcal{H}_i)$. In each of these hypothesis spaces, learn a hypothesis $h_i \in \mathcal{H}_i$. Let $e(h_i, T)$ be the error of h_i on the training set T. That h_i will be chosen for which $e(h_i, T) + u(\mathcal{H}_i)$ is minimal.

In the context of support vector machines, the different hypothesis spaces can be made to correspond to learning using different kernel functions. A kernel with the right balance between expressiveness and risk of overfitting can then be chosen using the SRM principle.

11.5 Bibliographic notes

Vladimir Vapnik is generally credited with laying the basis for what is now known as support vector machines. An interesting introductory text on the matter is Anthony Burges' tutorial on support vector machines (Data Mining and Knowledge Discovery 2(2):121−167, 1998), but this tutorial still requires a reasonable amount of familiarity with mathematics. Thorsten Joachims contributed much to the popularity of the method by implementing the SVM-light system and by applying SVMs to text classification, showing that they exhibited better performance than other methods at that time [22].

Many variants of support vector machines have been developed. There are variants that learn from examples of a single class, variants that perform regression, etc. As many of these methods have in common that they use kernels (in a way similar to SVMs), they are also called *kernel methods*. Several recent books offer good introductions to kernel methods (e.g. [10]).

11.6 Exercises

Exercise 11.1 Assume a 2-D input space and a linearly separable data set. Under what conditions will there be two support vectors, under what conditions will there be three?

Chapter 12

Probabilistic approaches

Generally, the aim of inductive learning is to build models that describe accurately the population from which the data set is a sample. For instance, when we learn a predictive model that allows us to predict an attribute y from attributes x_i, we are trying to learn a model that captures a certain relationship between y and x_i, and more specifically a relationship that does not only hold for the data set itself but also for unseen data, i.e., for all elements of the population from which our data set is drawn, whether they were in the data set or not.

Many learning methods assume there is a deterministic relationship between the variables, or an almost deterministic one (allowing some noise). But it is possible that such a relationship simply does not exist; for instance, for multiple objects with exactly the same description $\mathbf{x}$, it might be that the class is positive in 30% of the cases and negative in the remaining 70%. In that case, the best we can hope for is a model that describes which classes the object might belong to, and the probability with which it belongs to them. We call such a model a probabilistic model.

The ultimate model, from this perspective, is the joint probability distribution over all variables together. This joint probability distribution describes for each combination of values of the variables the probability that when we observe an individual, exactly this combination of values is observed for the variable. From this joint probability distribution we can infer any probability, e.g., $Pr(y|\mathbf{x})$, the probability of having some class y given a description $\mathbf{x}$.

12.1 Background

This chapter assumes familiarity with basic probability theory. In particular, the following concepts are considered known:

- stochastic experiments, outcomes of experiments, events;
- the probability of an event A, denoted $Pr(A)$;
- the conditional probability of an event A given an event B, denoted $Pr(A|B)$;
- mutually exclusive events
  (A and B are mutually exclusive if $A \cap B = \emptyset$)
- independent events, conditionally independent events
  (A and B are independent if $Pr(A \cap B) = Pr(A)Pr(B)$; A and B are conditionally independent given C if $Pr(A \cap B|C) = Pr(A|C)Pr(B|C)$
- the following rules of probability:

- $Pr(\bar{A}) = 1 - Pr(A)$, with $\bar{A}$ the complement of A ("not A")
- $Pr(A \cap B) = Pr(A)Pr(B|A) = Pr(B)Pr(A|B)$
- $Pr(A \cap B) = Pr(A)Pr(B)$ if A and B are independent
- $Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B)$
- $Pr(A) = \sum_i Pr(A|B_i)Pr(B_i)$
  if $\forall B_i, B_j : B_i \cap B_j = \emptyset$ and $A \subseteq \bigcup_i B_i$ (rule of total probability)
- $Pr(A|B) = Pr(B|A)Pr(A)/Pr(B)$ (Bayes' rule)

- discrete and continuous probability distributions;
- joint probability distributions
- marginal distributions

A brief introduction to these concepts and formulas is given in Appendix A. We advise the reader to read this appendix if any of the above concepts and formulas seem unfamiliar.

12.2 The use of probabilities in machine learning

Probabilities can be used in machine learning in several ways. We can use them to indicate how likely it is that a given *hypothesis* is correct, or how likely it is that a certain *prediction* is correct. Thus we can define, for instance, which hypothesis from a given hypothesis set has the highest probability of being the correct one. This way of using probabilities is described in the current section.

In addition, we can build learning systems that build so-called probabilistic models. Here, the hypotheses themselves are expressed by means of probabilities. This will be the subject of section 12.3 and the sections following it.

12.2.1 Maximum a posteriori hypothesis

In deterministic learning, the concept learning problem can be defined as follows: **given** a description space X and target space Y, a hypothesis space H (containing hypotheses which are functions from X to Y), an unknown target concept $c : X \to Y$, and a training set T which is a sample from $X \times Y$, **find** $h \in H$ such that $\forall x \in X : h(x) = c(x)$ (this h is called "the correct hypothesis").

Consider now the following, slightly modified, setting: we are given not just a hypothesis space H, but a distribution over H, which associates each $h \in H$ with the probability $Pr(h)$ that h is the correct hypothesis. We call this probability distribution the **prior distribution** over H, and the probability it assigns to a hypothesis h is called h's **prior probability**. (Sometimes the Latin expression *a priori* is used instead of "prior"; it means the same.)

The prior probabilities express our preconceptions about which hypotheses are likely and which are less likely to be true. After seeing the training set T, we get new probabilities for each h, denoted $Pr(h|T)$; these are called **posterior**, or **a posteriori**, probabilities, as opposed to the prior probabilities $Pr(h)$. $Pr(h|T)$ may differ from $Pr(h)$ because, for instance, a hypothesis h that was considered possible before seeing T ($Pr(h) > 0$) might be considered impossible after seeing the training set ($Pr(h|T) = 0$) because it logically contradicts the training data.

According to Bayes' rule, we have

$$Pr(h|T) = \frac{Pr(T|h)Pr(h)}{Pr(T)}. \tag{12.1}$$

The training set is a set of labelled examples $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ is the description part of the example and y_i the label. Let us use $\mathbf{x}'_i$ to denote the whole example (description plus label).

When the dataset T consists of stochastically independent $\mathbf{x}'_i$ (that is, each example is drawn randomly from the population, and the probability of any example $\mathbf{x}'_i$ being drawn is not influenced by which examples have already been drawn; we also say that the examples are i.i.d., independent and identically distributed), we have

$$Pr(T) = \prod_i Pr(\mathbf{x}_i)$$

and

$$Pr(T|h) = \prod_i Pr(\mathbf{x}'_i|h),$$

which we can fill in in Equation 12.1 to obtain

$$Pr(h|T) = \frac{\prod_i Pr(\mathbf{x}'_i|h)Pr(h)}{\prod_i Pr(\mathbf{x}'_i)}. \tag{12.2}$$

Here $Pr(\mathbf{x}'_i)$ is the probability that when we draw a random instance from the population X and observe its class, we get $\mathbf{x}'_i$, and $Pr(\mathbf{x}'_i|h)$ is the probability that if we draw a random instance from X in a world where h is true, and we observe its class, we obtain $\mathbf{x}'_i$.

Consider the special case where the target function is deterministic, i.e., for any $\mathbf{x} \in X$, the class of $\mathbf{x}$, denoted $c(\mathbf{x})$, is uniquely determined by $\mathbf{x}$ (no two copies of x can have a different class). Assume further that there is no noise in the data: $y_i = c(\mathbf{x}_i)$.

Note that $Pr(\mathbf{x}'_i|h)$ is the probability that we observe $\mathbf{x}'_i$ given that h is correct. Now, if h is correct, then whenever we observe $\mathbf{x}_i$, we must observe $h(\mathbf{x}_i)$ for its class, and it is impossible that we observe another class. So

$$\begin{array}{ll} Pr(\mathbf{x}'_i|h) = Pr(\mathbf{x}_i) & \text{whenever } c(\mathbf{x}_i) = h(\mathbf{x}_i), \text{ and} \\ Pr(\mathbf{x}'_i|h) = 0 & \text{whenever } c(\mathbf{x}_i) \neq h(\mathbf{x}_i). \end{array}$$

Since $Pr(T|h)$ is the product of all the individual $Pr(\mathbf{x}'_i|h)$, we have $Pr(T|h) = 0$ as soon as T contains a single element that contradicts h, and $Pr(T|h) = \prod_i Pr(\mathbf{x}_i)$ when each element of T is consistent with h. Hence, filling in in Equation 12.2, we get

$$Pr(h|T) = \frac{\prod_i Pr(\mathbf{x}_i)}{\prod_i Pr(\mathbf{x}'_i)} Pr(h)$$

if all $\mathbf{x}'_i \in T$ are consistent with h, and

$$Pr(h|T) = 0$$

as soon as one $\mathbf{x}'_i \in T$ is inconsistent with h.

If we want to find a correct hypothesis, our best bet is by definition on the hypothesis that has the highest probability to be true. This hypothesis is called the **maximum a posteriori probability hypothesis**, or **MAP-hypothesis**:

$$h_{MAP} = \arg\max_{h \in H} Pr(h|T) = \arg\max_{h \in H} \frac{Pr(T|h)Pr(h)}{Pr(T)} \tag{12.3}$$

This can be simplified further by noting that the arg max function is invariant under multiplication with a constant; that is, $\arg\max_x f(x) = \arg\max_x cf(x)$ if c is constant in x. (Indeed, $x_0 = \arg\max_x f(x) \Leftrightarrow \forall x' : f(x_0) \geq f(x') \Leftrightarrow \forall x' : cf(x_0) \geq cf(x') \Leftarrow x_0 = \arg\max_x cf(x)$.) In the last term, $Pr(T)$ is constant with respect to h and hence can be dropped. That yields

$$h_{MAP} = \arg\max_{h \in H} Pr(T|h)Pr(h) \tag{12.4}$$

12.2.2 Maximum likelihood hypothesis

The MAP-hypothesis is by definition the hypothesis that has the highest probability of being correct, given the data. Consistent with Bayes' rule, it is the hypothesis that maximizes the product of $Pr(T|h)$ and $Pr(h)$. Now $Pr(T|h)$ can often be computed relatively easily; but what about $Pr(h)$? This a priori probability distribution over H reflects our preconceptions about which hypotheses are probable and which are less probable.

In most cases, we do not have sufficiently precise knowledge about $Pr(h)$, and we cannot measure it. If we have no a priori preference for certain hypotheses over other hypotheses, this is reflected by choosing $Pr(h)$ to be uniform: $\forall h \in H : Pr(h) = 1/|H|$ (each $h \in H$ is a priori considered equally probable).

Note the difference between choosing a uniform $Pr(h)$, i.e., expressing no preference for any particular h, and not knowing what $Pr(h)$ is, i.e., expressing no preference for any particular distribution over H. If we choose a uniform $Pr(h)$, we *are* making an assumption about which hypotheses are more likely and which are less likely (namely, we assume that they are all equally likely).

Under the assumption of a uniform $Pr(h)$, i.e., $\forall h \in H : Pr(h) = 1/|H|$, the formula for the MAP-hypothesis becomes

$$h_{MAP} = \arg\max_{h \in H} Pr(T|h)Pr(h) = \arg\max_{h \in H} Pr(T|h)\frac{1}{|H|} = \arg\max_{h \in H} Pr(T|h). \tag{12.5}$$

where again the constant factor $1/|H|$ is dropped because it doesn't influence the result of arg max.

Thus, under the assumption of a uniform prior distribution over H, the MAP-hypothesis is simply the hypothesis h^* for which the conditional probability of the training set, given the hypothesis, is highest (among all possible hypotheses). The probability of the data given a hypothesis is also called the *likelihood* of the data, and hence the hypothesis that maximizes this likelihood is called the **maximum likelihood** hypothesis, denoted h_{ML}. So we have

$$h_{ML} = \arg\max_{h \in H} Pr(T|h). \tag{12.6}$$

Note that we cannot claim that h_{ML} is the most probable hypothesis, given the data. This claim can only be made for h_{MAP}, and $h_{MAP} = h_{ML}$ **only if** we know that $Pr(h)$ is uniform. The hypothesis that is really most probable, unfortunately, simply cannot be computed unless we know $Pr(h)$. Therefore, returning the maximum likelihood hypothesis h_{ML} is in many cases considered the best we can do.

12.2.3 Bayes optimal prediction

In the previous sections we saw how probability theory can be used to find the most probable hypothesis h in a hypothesis space H. A second possible use of probability theory is predicting the most probable target value.

Consider first the case where a hypothesis space H was given together with a distribution over it (the prior distribution $Pr(h)$). Given a data set T, we can find the posterior probabilities $Pr(h|T)$ of the hypotheses, and select the most probable hypothesis given the data, h_{MAP}.

Given a new instance $\mathbf{x}$, it makes sense to use h_{MAP} to make a prediction for $\mathbf{x}$, i.e., to predict the class of $\mathbf{x}$ as $\hat{y} = h_{MAP}(\mathbf{x})$.

However, this does **not** give us the target value that is most probable! Indeed, according to the rule of total probability, the probability that the class of an instance

$\mathbf{x}$ is y, denoted $Pr(y|\mathbf{x})$, is

$$Pr(y|\mathbf{x}) = \sum_{h \in H} Pr(y|\mathbf{x}, h) Pr(h)$$

and after seeing a training set T, this becomes

$$Pr(y|\mathbf{x}, T) = \sum_{h \in H} Pr(y|\mathbf{x}, h) Pr(h|T).$$

In the case of deterministic hypotheses, each hypothesis predicts one particular value of y, which corresponds to giving a probability of 1 to that value and 0 to other values. That is, $Pr(y|\mathbf{x}, h) = 1$ if $h(\mathbf{x}) = y$ and $Pr(y|\mathbf{x}, h) = 0$ otherwise.

This translates to

$$Pr(y|\mathbf{x}, T) = \sum_{h \in H : h(\mathbf{x}) = y} Pr(h|T).$$

Thus, the most probable prediction (also called the MAP-prediction) is not the prediction made by the most probable hypothesis; the probability of a prediction is the sum of the probabilities of *all* the hypotheses making that prediction, and the most probable prediction is the one where this sum is highest.

Example 12.1 Assume that our hypothesis space consists of three hypotheses, $H = \{h_1, h_2, h_3\}$, and assume that after seeing the training set T we have the following probabilities: $Pr(h_1|T) = 0.4, Pr(h_2|T) = 0.3, Pr(h_3|T) = 0.3$. Note that $h_{MAP} = h_1$. Now assume that for a new instance $\mathbf{x}$, h_1 predicts positive and h_2 and h_3 predict negative. While the most probable hypothesis predicts positive, the probabilities of $\mathbf{x}$ being positive or negative, respectively, are $Pr(pos|\mathbf{x}, T) = Pr(h_1|T) = 0.4$ and $Pr(neg|\mathbf{x}, T) = Pr(h_2|T) + Pr(h_3|T) = 0.6$. The instance $\mathbf{x}$ is most probably negative.

When target values are predicted according to the formula

$$Pr(y|\mathbf{x}, T) = \sum_{h \in H} Pr(y|\mathbf{x}, h) Pr(h|T),$$

this is called **Bayes optimal prediction**. In the context of classification, classifiers that yield the most probable class according to this formula are called **Bayes optimal classifiers**. Bayes optimal prediction gives us the best possible prediction, in the sense that no other prediction method can exist that has a higher probability of making a correct prediction for a randomly chosen instance. In practice, unfortunately, this method cannot be used for any problems except the simplest ones, for several reasons: we usually don't know the a priori distribution over the hypotheses; even if we do, there may be so many hypotheses that computing the probability of all of them, given the data, is practically infeasible; and even when it is feasible, making a prediction for a single instance is slow since we need to combine the predictions of a huge number of hypotheses to get it.

12.3 Building probabilistic models

In the previous section we assumed that a hypothesis space was given, and we used probability theory to characterize the quality of hypotheses and predictions. The format of the hypotheses was not relevant: they could be decision trees, neural networks, or whatever.

In the remainder of this chapter we will look at hypotheses that are themselves represented in a probabilistic format; we call them probabilistic models. The most general type of a probabilistic model is the joint probability distribution. Once this joint probability distribution is known, Bayes optimal prediction of any set of variables from any other set of variables is possible. The joint probability distribution is, in this sense, the most accurate model of the population that is possible.

12.3.1 Representing a joint probability distribution

Let us start with considering only discrete variables. Given a set of variables $X_1, \ldots, X_D$, the joint probability distribution of this set of variables is a function $p_{X_1,\ldots,X_D}$ mapping any vector $(x_1, x_2, \ldots, x_D)$ to the probability that the variables take exactly these values, i.e., $p_{X_1,\ldots,X_D}(x_1, \ldots, x_D) = Pr(X_1 = x_1 \wedge \ldots \wedge X_D = x_D)$.

In practice we usually drop p's subscript for ease of notation, so we write $p(x_1, \ldots, x_D)$ instead of $p_{X_1,\ldots,X_D}(x_1, \ldots, x_D)$. The idea is that the arguments are written in such a way that the subscript is implied. That is, when we write $p(x)$, we mean $p_X(x)$; when we write $p(y)$, we mean $p_Y(y)$ (note that p_Y may be another function than p_X); when we write $p(x, y)$, we mean $p_{X,Y}(x, y)$, the joint distribution over X and Y; when we write $p(x|y)$, we mean $p_{X|Y}(x, y)$, the function mapping x and y onto the conditional probability that $X = x$ given $Y = y$; etc.

Assuming that each X_i can take only a finite number of values, a straightforward way of representing p is simply listing the value of $p(x_1, \ldots, X_D)$ for each possible combination of $x_1, \ldots, x_D$. If each X_i can take V_i values, then there are $\prod_i V_i$ entries in this table. The size of this table is thus exponential in the number of variables.

This approach clearly does not scale well when we have many variables. Moreover, it does not work at all when we have variables with an infinite number of possible values, for instance continuous variables. In these cases, instead of the tabular format, a format is needed where the JPD is represented in symbolic form as a function with a finite number of parameters.

For instance, the JPD of three variables X, Y, Z might be given by

$$p(x, y, z) = \frac{1}{2^x} \frac{x!}{y!(x-y)!} z^y (1-z)^{x-y}$$

(which would be a correct JPD in the case where the data consist of (x, y, z) tuples generated as follows: toss an unbiased coin until it shows heads, and call x the number of tosses needed; choose a random z uniformly between 0 and 1; generate x random numbers between 0 and 1 and let y be the number of such numbers that are less than z).

The JPD of D independent Gaussian variables X_i, each with mean μ_i and variance σ_i^2, is

$$p(x_1, \ldots, x_D) = \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} \exp(-\frac{(x_i - \mu_i)^2}{2\sigma_i^2})$$

Depending on the situation, different formats for the JPD may be appropriate. *The choice of a particular format imposes a bias on the learning system*: once the format is chosen, there will be JPDs that can be represented within this format and there will be JPDs that cannot. This is in contrast with the tabular format (which could be used for discrete variables only): the tabular format does not impose any bias at all, any JPD over the considered variables can be represented exactly using the table.

Representing JPDs for continuous variables in tabular format is impossible because the number of entries would be infinite. The symbolic format is finite as long

as it uses a finite number of parameters, but this implies that a bias is imposed: with a finite number of parameters not every imaginable JPD can be approximated with arbitrary precision.

In the remainder of this text, we will not focus on the problem of exactly how the JPD is represented; in most cases it can simply be assumed that the JPD is given in tabular form.

12.3.2 Using the JPD for prediction

Let us now look at how we can predict the value of a target variable Y from input variables $X_1, \ldots, X_D$. More precisely, we are interested in predicting the probability that Y takes certain values, given $X_1, \ldots, X_D$.

From the joint distribution $p(x_1, \ldots, x_D, y)$, one can easily derive the conditional probability function

$$p(y|x_1, \ldots, x_D) = p(x_1, \ldots, x_D, y)/p(x_1, \ldots, x_D)$$

according to the definition of a conditional probability.

This formula can be used in two ways. First, obviously, given an example $\mathbf{x}$ with values $x_1, \ldots, x_D$, the formula tells us how probable each value y for the target (class) variable is, or in other words: it gives us a probability distribution over Y given $\mathbf{x}$. This probability distribution may be useful knowledge in itself. Second, the formula allows us to predict a specific value for Y: typically, if we need to predict a single value, we would just predict the value that has the highest probability. This prediction is optimal in the sense that it is most likely to be correct, among all possible predictions. The probability that it is correct is $p(y|x_1, \ldots, x_D)$.

These last two sentences are very important, and give us a third way of applying the above formulae. A model that always predicts

$$y^* = \arg\max_y p(y|x_1, \ldots, x_D)$$

has the highest accuracy (probability of predicting correctly) of all possible models, and this accuracy is

$$p^* = p(y^*|x_1, \ldots, x_D).$$

This implies that p^* is a theoretical upper bound on the accuracy that any model predicting Y from $X_1, \ldots, X_D$ could have. No models can exist that perform better than this.

Thus, the probabilistic approach towards prediction gives us not only a way to make predictions, but also a "gold standard", a reference point that indicates what the best possible predictions are and what the highest accuracy is that can be achieved with any model, whatever its format is. Moreover, as the JPD contains full information on the whole set of variables, it provides this information for whichever variable we want to predict from whatever other set of variables.

Be careful not to overinterpret this result. The conclusion is not that probabilistic models are guaranteed to work better than other models. The formulas above are written in terms of the true JPD, which we usually don't know. We can try to approximate it, given a set of data. Whether the resulting model is good or not depends on how good that approximation is.

So the conclusion is: if we can approximate the JPD really well, then we can obtain optimal predictions from it. The JPD is the best possible model of the data, in that respect. But the learning problem (including the problem of bias, for instance) is simply moved from learning a particular prediction function towards learning the JPD from a training set.

Much of the remainder of this chapter will be about estimating the JPD from data.

12.3.3 Estimating a JPD from data

With limited data, it may be impossible to find a really good approximation of the JPD. With an unlimited amount of data, we can make the approximation as good as we want, in theory, but we may need very many examples for that.

Consider the case of discrete variables where the JPD is represented in tabular format. As mentioned, the table may be huge: it has $D+1$ columns (containing the values x_1, x_2, ..., x_D, and $p(x_1, \ldots, x_D)$), which is not too bad, but $\prod_i V_i$ rows, where V_i is the number of values X_i may take. The size of the table is thus roughly exponential in the number of variables. Besides the fact that an exponential amount of space is necessary to store the probabilities, there is also the problem that for each row we need a reasonable number of instances having $X_1 = x_1, \ldots, X_D = x_D$ to get a good estimate of $p(x_1, \ldots, x_D)$. Thus the number of instances needed is also exponential in the number of variables.

Example 12.2 Consider an experiment where a computer simulates the rolling of a fair die (having a 1/6 probability of rolling any number from 1 to 6). In principle each die roll should be independent from the others, but due to the use of a poor random number generator, there might be dependencies between the rolls. For instance, given that the first die roll was 5, the outcome of the second die roll might be distributed as $(0.1, 0.2, 0.1, 0.2, 0.2, 0.2)$ instead of $(1/6, 1/6, 1/6, 1/6, 1/6, 1/6)$.

Assume we have a sequence of 10 die rolls. If the random numbers had been independent, we would have $p(x_1, \ldots, x_{10}) = p(x_1)p(x_2)\cdots p(x_{10})$. But since they are not independent, we cannot use that formula; instead, we need to estimate $p(x_1, \ldots, x_{10})$ for each possible combination $(x_1, \ldots, x_{10})$ separately. There are $6^{10} = 60,466,176$ (approximately 60 million) such combinations. To have a reasonably good estimate of how often one particular combination occurs, we must have seen it at least some number of times. For instance, if we have $100 \cdot 6^{10}$ (about six billion) die rolls, then a combination that has a true probability of p will occur approximately $100 \cdot 6^{10} p$ times in this sequence. If all combinations are equally likely ($p = 1/6^{10}$), they would all occur approximately 100 times.

Due to the randomness of the process, some will occur more and others less. The actual number of occurrences of a particular combination follows roughly a Gaussian distribution with standard deviation of 10, hence about 99% of the combinations will occur between 74 and 126 times. For about 1% of the combinations we would get an estimate that's relatively far off the actual probability, i.e., even if their true probability is indeed $1/6^{10}$ we might estimate it as less then $0.74/6^{10}$ or more than $1.26/6^{10}$. Thus, even with 6 billion data points, and a table with 60 million cells to fill in, we get relatively inaccurate probability estimates.

The number of die rolls in the examples corresponds to the number of attributes in a data table. That number can easily be much larger than 10, and the number of attribute values can easily be larger than 6. This shows that the approach of estimating the JPD as a table of all possible combinations is bound to fail in many practical application domains.

In the following sections we will look at methods that are more efficient and can learn from smaller numbers of examples, this at the cost of introducing an inductive bias.

12.4 Naive Bayes

To briefly summarize what we saw before: Given the conditional probability distribution $p(y|x_1, \ldots, x_D)$, the optimal prediction is given by

$$y^* = \arg\max_y p(y|x_1, \ldots, x_D)$$

and we can derive the conditional probility function from the joint probability distribution:

$$p(y|x_1,\ldots,x_D) = p(x_1,\ldots,x_D,y)/p(x_1,\ldots,x_D)$$

where

$$p(x_1,\ldots,x_D) = \sum_y p(x_1,\ldots,x_D,y)$$

As explained before, both $p(y|x_1,\ldots,x_D)$ and $p(x_1,\ldots,x_D,y)$ are difficult to estimate directly from the data: too many data points would be needed to have an accurate estimation.

The Naive Bayes method estimates $p(x_1,\ldots,x_D,y)$ by making an assumption of independence of the X_i. More precisely, it assumes that all the attributes X_i are *conditionally independent from each other given the class*. This means that

$$p(x_1,\ldots,x_D|y) = p(x_1|y)p(x_2|y)\cdots p(x_D|y).$$

Under this assumption, $p(y|x_1,\ldots,x_D)$ can be rewritten as follows:

$$\begin{aligned} p(y|x_1,\ldots,x_D) &= p(x_1,\ldots,x_D,y)/p(x_1,\ldots x_D) && (12.7)\\ &= p(x_1,\ldots,x_D|y)p(y)/p(x_1,\ldots,x_D) && (12.8)\\ &= p(x_1|y)p(x_2|y)\cdots p(x_D|y)p(y)/p(x_1,\ldots,x_D) && (12.9)\end{aligned}$$

and hence

$$\begin{aligned} y^* &= \arg\max_y p(y|x_1,\ldots,x_D) && (12.10)\\ &= \arg\max_y p(x_1|y)p(x_2|y)\cdots p(x_D|y)p(y)/p(x_1,\ldots,x_D) && (12.11)\\ &= \arg\max_y p(x_1|y)p(x_2|y)\cdots p(x_D|y)p(y) && (12.12)\end{aligned}$$

where the denominator $p(x_1,\ldots,x_D)$ is again dropped because it does not depend on y.

The last equation gives us a very simple method to compute the most probable value for Y, y^*, from attribute values $x_1,\ldots,x_D$. It tells us that for each value y of Y, we should simply estimate $p(y)$ and $p(x_i|y)$ for all the x_i, and multiply all this. *This simple procedure gives us with certainty the most probable value for Y given $x_1,\ldots,x_D$, if the assumption of conditional independence is correct.*

This leaves us of course with the question whether it is realistic for many problems to assume that the attributes are conditionally independent given the class, and what happens if the assumption is violated.

To answer the first question: in many cases the assumption is *not* realistic. As an example, take the so-called Iris dataset, a small dataset that is often used as a benchmark problem in machine learning (and publically available in the UCI repository of machine learning datasets [29]). In this dataset, iris flowers are described by petal length (PL), petal width (PW), sepal length (SL), and sepal width (SW). The general shape of the petals is relatively constant, so when a petal has a greater length, it probably also has a greater width. This means PL and PW are not independent. Also within one class, it seems likely that this dependency exists: if a flower of class A has a larger petal length, it is likely that it also has a larger petal width. Figure 12.1 plots sepal lengths and widths, and confirms this dependency.

So what happens if the assumption is violated? Does the method break down in that case?

Extensive experiments over the years have shown that Naive Bayes often performs surprisingly well, even if the assumptions are violated. This effect has to some extent been explained theoretically by Domingos and Pazzani in 1997.

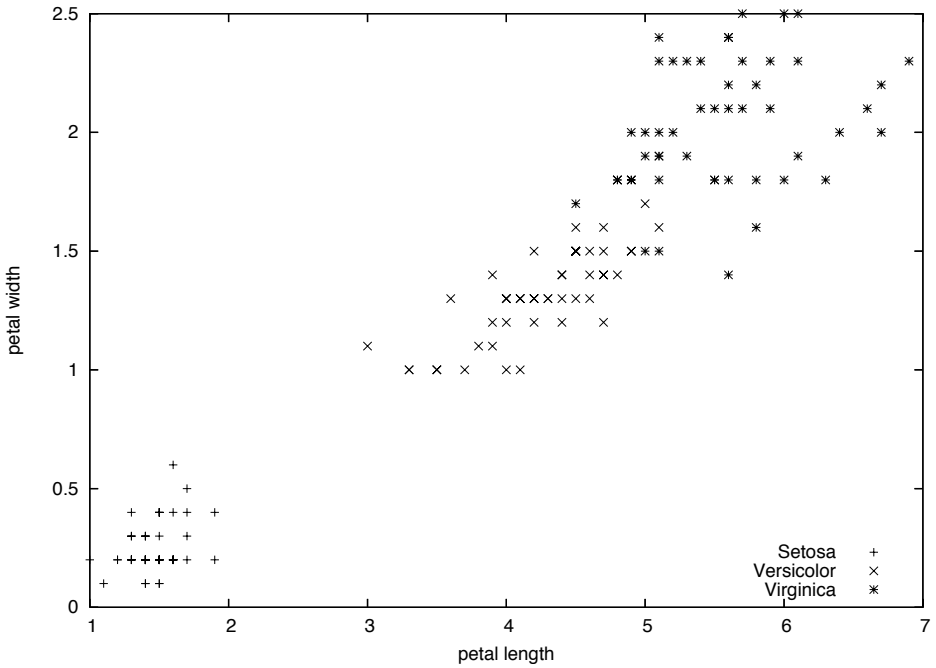

Figure 12.1: Petal length and width for iris flowers.

The main reason why Naive Bayes works well is the following: if the assumptions are violated, then the actual probabilities $p(y|x_1, \ldots, x_D)$ computed by Naive Bayes are indeed quite wrong, but this turns out not to have that much influence on which value for y maximizes $p(y|x_1, \ldots, x_D)$. Note that, even when

$$p(y|x_1, \ldots, x_D) \neq p(x_1|y)p(x_2|y) \cdots p(x_D|y)p(y)/c$$

(with c the denominator), it may still be the case that

$$\arg\max_y p(y|x_1, \ldots, x_D) = \arg\max_y p(x_1|y)p(x_2|y) \cdots p(x_D|y)p(y)/c,$$

i.e., two functions of y may be quite different and yet reach their maximum for the same value for y.

It turns out that in practice this happens indeed quite often. Domingos and Pazzani performed an analysis showing under what conditions the different functions reach their maximum for *different* values of y, and this analysis showed that this only happens under relatively restricted conditions. For instance, it was shown that when the input attributes $X_1, \ldots, X_D$ are boolean, and the target concept is a conjunctive concept, the Naive Bayes learner yields Bayes optimal predictions.

Note that we are here analyzing the error introduced by the Naive Bayes assumption alone, that is, the error introduced by replacing $p(y|x_1, \ldots, x_D)$ with $p(x_1|y)p(x_2|y) \cdots p(x_D|y)p(y)$. In practice we need to estimate the $p(x_i|y)$ and $p(y)$ from a data set, and this may introduce additional errors.

The conclusion of this is that Naive Bayes can be a Bayes optimal classifier even when the assumption of conditional independence is violated. Note, however, that while Naive Bayes is often a good *classifier*, the *probabilities* it computes are not necessarily accurate. That is, if the Naive Bayes procedure predicts that an example is positive, it is reasonable to believe that; but if Naive Bayes tells you that the probability that the example is positive is 90% (in other words, of all examples with description $\mathbf{x}$, it estimates that 90% are positive), this is likely to be incorrect (unless the conditional independence assumption indeed holds).

12.4.1 Estimating the probabilities

How can we estimate a (conditional) probability from a dataset? Generally, the probability that a random instance fulfills a condition A is approximated by the proportion of cases in a finite sample that fulfill A. E.g., if we want to know what percentage of people smoke, we can ask 200 people, and if out of these 200, 30 smoke, then we estimate the probability that a randomly chosen person smokes as $30/200 = 0.15$.

Similarly, a conditional probability $Pr(A|B)$ can be estimated by counting the number of instances that fulfill B and how many of those also fulfill A, and dividing the latter by the former:

$$\hat{Pr}(A|B) = |\{\mathbf{x} \in T|A(\mathbf{x}) \wedge B(\mathbf{x})\}|/|\{\mathbf{x} \in T|B(\mathbf{x})\}|$$

where $\hat{Pr}(A|B)$ is an estimate of $Pr(A|B)$.

While this is a reasonable way of estimating probabilities (the probability estimates can be proven to be statistically unbiased and relatively accurate), it unfortunately does not work very well when used inside the Naive Bayes method. The reason for this is the possible occurrence of zero-counts. Whenever a certain condition $A \wedge B$ has never been observed in the dataset, we estimate its probability as zero.

This probability estimate will be interpreted by the Naive Bayes procedure as "$A \wedge B$ is impossible" rather than "$A \wedge B$ did not occur in the data" (indeed, that is what a probability of zero implies). But this is obviously a very strong conclusion, for which the arguments are relatively weak.

How does this translate to the Naive Bayes procedure? Essentially, since we are estimating the conditional probabilities of attribute values x_i given the class y, an estimate of 0 is to be interpreted as "this attribute value can never occur for this class". So if a certain attribute value never occurred for instances of a particular class in the training set, we consider it completely impossible that it would ever occur for instances of that class, also in unseen data.

This is clearly a much too strong conclusion, and it can easily lead to the result that all classes have a probability of 0. Indeed, all these probability estimates will be multiplied. As soon as one estimate is zero, the result of the multiplication will be zero.

The problem here is that a probability estimate of 0 is OK as long as one is aware that it is only a probability *estimate*, and the actual probability might be slightly different. But Naive Bayes treats the estimates as the actual probabilities, and because of that it may draw too strong conclusions.

To alleviate this problem, probabilities are in practice usually estimated in such a way that zero estimates do not occur. A simple procedure to achieve that is to make an a priori estimate of the probabilities, stating essentially that we believe that the probabilities are non-zero, and adapting these probabilities based on the data available. One method to do that is the so-called **Laplace estimate**.

Generally, if we perform an experiment n times and A occurred n_A times, then instead of estimating $Pr(A)$ as n_A/n, we estimate it as

$$Pr(A) = \frac{n_A + 1}{n + 2}.$$

That is, we act as if two additional experiments were performed, where A was true for one experiment and false for the other.

This estimate corresponds to assuming a priori that $P(A|B) = 0.5$ (roughly interpretable as "we have no idea what the probability of A is given B, so we have no bias towards probable or improbable, and hence we choose 0.5"), and then

adapting this estimate to the ratio observed in the data. The more data is available, the closer the ratio comes to n_A/n. If there are few data, the ratio remains closer to 1/2.

If A in fact expresses that some variable takes a particular value, out of V possible values, then it may make sense to a priori assume that each of the V values is equally likely (i.e., has a probability $1/V$), which gives:

$$Pr(A) = \frac{n_A + 1}{n + V}$$

These estimates always give an interpolated value between an a priori estimate $1/V$ imposed by us, and the observed proportion of cases where A was observed in the data. The weight of the a priori estimate is comparable to having previously seen a set of V elements where each value occurred exactly once.

We can in fact give this a priori estimate a higher weight by acting as if we have previously seen a set of mV elements where each of the V values occurred m times. The probability estimate then becomes:

$$Pr(A) = \frac{n_A + m}{n + mV}$$

This estimate is called the m-estimate, and was introduced by Čestnik (1990). By choosing a value for m, one may decide to give more weight to the a priori estimate, or more weight to the data. For $m = 1$ and $V = 2$, we obtain the original Laplace estimate.

Exercise 12.1 Show that the m-estimate is indeed a linear interpolation between the a priori estimate $1/V$ and the ratio n_A/n. Do this by writing the m-estimate as $\alpha 1/V + (1 - \alpha)n_A/n$ and showing how α depends on m.

12.4.2 The Naive Bayes Learning Algorithm

The learning algorithm corresponding to the Naive Bayes method is very simple: it consists of simply estimating for each value x_{ij} of each attribute X_i, and for each value y_k of the class attribute Y, the probability that $X_i = x_{ij}$ given $Y = y_k$, i.e., for each i, j, k, determining the conditional probability $Pr(X_i = x_{ij}|Y = y_k)$. In addition, the unconditional probability $Pr(Y = y_k)$ for each class value y_k needs to be determined. All of these are stored in a matrix, and when a new prediction is to be made, the relevant entries in the matrix are retrieved.

Figures 12.2 and 12.3 show pseudo-code for the algorithms used for learning and predicting using Naive Bayes.

12.4.3 Naive Bayes with continuous variables

Up till now, we have discussed a version of Naive Bayes that works with discrete variables. What happens if we have continuous variables? The main change is that the probabilities turn into probability densities, and that we need to estimate probability density functions.

Estimating probabilities is easy, as we have seen before; essentially it is based on counting how often certain values occur, although a small modification is necessary to avoid too extreme estimates, hence we use the m-estimate. The m-estimate can be seen as trading off the proportions we observe in the data with some prior estimate of the probabilities that we consider reasonable.

For estimating probability density functions, the problem of non-observed values is even more severe: for a real-valued attribute, we will necessarily observe only a finite number of values from an infinite domain, and it would be foolish to think

function NaivesBayesLearn(T: dataset) **returns** 3-D hash cp and 1-D hash up:
    for each $y \in Dom(Y)$:
        for each $i = 1 \ldots D$:
            $V = |Dom(X_i)|$
            for each value $x_i \in Dom(X_i)$:
                $cp[i, y, x_i] :=$ estimate(T, $Pr(X_i = x_i|Y = y_k)$, 1, V)
        $up[y] :=$ estimate(T, $Pr(Y = y_k)$, 1, V)
    return (cp, up)

function estimate(T: dataset, $Pr(A|B)$, m, V) **returns** real:
    $c := |\{\mathbf{x} \in T|A(\mathbf{x}) \wedge B(\mathbf{x})\}|$
    $d := |\{\mathbf{x} \in T|B(\mathbf{x})\}|$
    return $(c + m)/(d + mV)$

Figure 12.2: Naive Bayes: Learning Algorithm

function NaivesBayesPredict($\mathbf{x}$, cp, up) **returns** class;
    for each $y \in Dom(Y)$:
        $c[y] := up[y]$
        for each $i = 1 \ldots D$:
            $c[y] := c[y] * cp[i, y, x_i]$
    return $\arg\max_y c[y]$

Figure 12.3: Naive Bayes: Prediction Algorithm

that for any non-observed value the probability density must be zero. The question is: how do we estimate a probability density from a finite sample of values?

The most straightforward method, in the sense of being closest to the discrete case, is the following. Divide the continuous domain of an attribute X into a number of intervals. For each interval, count how many observed values lie in it, and divide this by the total number of observations. This is an estimate of the probability that a random value for X lies in this interval. Dividing this by the width of the interval gives us an estimate of the probability density. See Figure 12.4 for a graphical illustration.

Histograms are rather clumsy to work with, though, when used as a probability density function. They are typically represented in a tabular format rather than as a mathematical function. The probability density they define is not continuous, and the steps it makes occur at relatively random points (the end points of the intervals). The histogram is just an approximation of some real density function, and the approximation becomes better if the intervals become smaller, assuming there are still enough observations in each interval to make an accurate estimate of the probability.

Once we have a histogram, we can try to fit a continuous curve through the points (m_j, f_j) where m_j is the middle of the j'th interval and f_j is the probability density estimate for that interval. Fitting such a curve implies choosing a parametrized family of functions and determining the parameters such that the curve fits the histogram as well as possible.

In many cases, the density function is assumed a priori to belong to some family of distributions, for instance, in practice it is very often (and perhaps too often) assumed that a variable has a Gaussian distribution. One would then determine the parameters μ and σ^2 of a Gaussian distribution in such a way that the density function maximally fits the histogram. One question is then: what fitting criterion

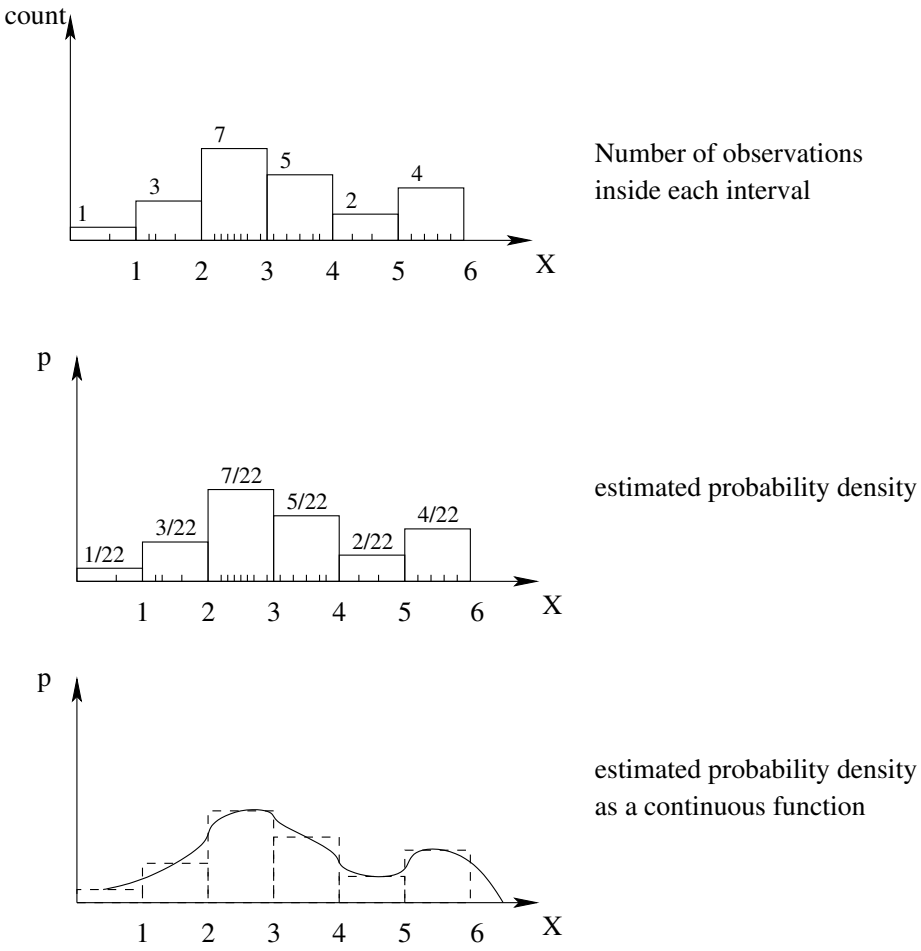

Figure 12.4: A histogram indicates how many instances were observed with an X value within some interval. Dividing the number of observations in an interval by the total number and dividing that by the interval width gives an approximation of the probability density of X in that interval.

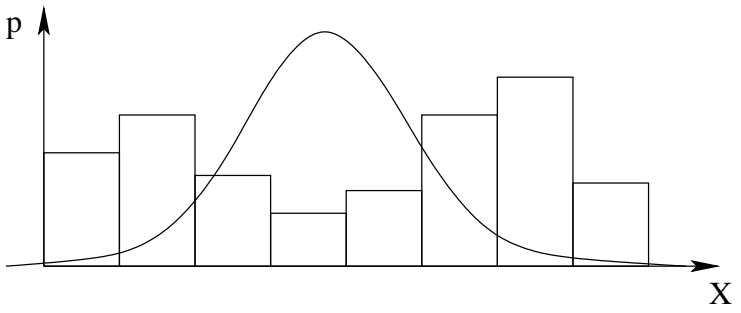

Figure 12.5: Bimodal histogram with Gaussian fitted. The Gaussian is clearly a bad approximation of the true density: it is high where the true density is low, and vice versa.

should we use?

Minimizing the sum of squared errors $\sum_j (g(m_j) - f_j)^2$ is one option. But the f_j of an interval with many observations is probably more accurate than that of an f_j with few observations, and the f_j of a narrow interval is a more accurate estimation of the probability density at its middle m_j than the f_j of a wide interval. So we may want to weight the sum of squared errors, giving higher weights to narrow intervals with many observations.

Finally, we might just compute a number of properties of the histogram, such as its mean, variance, etc., just as many properties as there are parameters to determine, and fit a curve such that these properties remain the same. An advantage of this last approach is that we may not even have to compute the entire histogram: we only need to compute a few of its properties.

In fact, in the case of a Gaussian distribution, the latter is typically the approach that is taken. There are two parameters to be determined: the mean μ and variance σ^2 of the Gaussian. These can be estimated directly from the sample, using the standard estimators from statistics: $\hat{\mu} = \sum_i X_i/N$ and $\hat{\sigma}^2 = \sum_i (x_i - \hat{\mu})^2/(N-1)$. [1]

The approach of estimating the parameters directly from the data, without going through a histogram, has an important disadvantage. Any sample has a mean and variance, hence for any sample we can compute the best fitting Gaussian, but that does not necessarilly mean it is a good fit. Figure 12.5 illustrates this. While the Gaussian and the actual density may have the same mean and variance, their actual densities in given points may be totally different, and when this happens, the variable may severely disturb the Naive Bayes model.

Fitting a single Gaussian is a bad idea unless one has reason to assume the variable distribution is indeed approximately Gaussian. But there are more general-purpose methods for estimating probability functions, methods that can fit almost any shape of the probability density function. One possibility is to assume that the probability density is a mixture of Gaussians. This allows us to model density functions with multiple peaks. More generally, one can use a combination of locally accurate functions, similar to what happens in radial basis function networks. (See the chapter on "instance based learning".)

[1] The reason for dividing by $N-1$ instead of N for the variance is the use of $\hat{\mu}$ instead of μ in its computation, which causes the variance to be underestimated if we divide by N, while dividing by $N-1$ makes it unbiased. We refer to statistics textbooks for more explanations.

12.4.4 Linearity of Naive Bayes

Naive Bayes is in some sense a linear classifier. Linear classifiers are characterized by the fact that the effect of one variable on a target variable does not depend on the values of other variables. That is, if, on average, having a certain property increases the likelihood of belonging to some class, then having that property must increase this likelihood regardless of what else we know about the instance.

For instance, assume that you tend to like cold drinks with a lemon taste, but for hot drinks (e.g., soup) you tend to dislike a lemon taste. Any linear classifier can either consider a "lemon taste" to be good or bad with respect to whether you will like a drink, but it cannot express that a lemon taste is good in some situations and bad in other situations. In other words, it cannot express that the effect of a lemon taste on liking the drink depends on the temperature of the drink. This is a typical property of linear classifiers.

The linearity of Naive Bayes can easily be shown by performing a logarithmic transformation. Assume that we have two classes 0 and 1. We assign any instance to that class for which $p(x_1|y)p(x_2|y)\cdots p(x_D|y)p(y)$ is largest, that is, we assign 1 if

$$p(x_1|1)p(x_2|1)\cdots p(x_D|1)p(1) > p(x_1|0)p(x_2|0)\cdots p(x_D|0)p(0)$$

and 0 otherwise. This condition is equivalent to assigning 1 if

$$\frac{p(x_1|1)p(x_2|1)\cdots p(x_D|1)p(1)}{p(x_1|0)p(x_2|0)\cdots p(x_D|0)p(0)} > 1$$

or

$$\frac{p(x_1|1)}{p(x_1|0)}\frac{p(x_2|1)}{p(x_2|0)}\cdots\frac{p(x_D|1)}{p(x_D|0)}\frac{p(1)}{p(0)} > 1$$

which after taking the logarithm becomes

$$\log(\frac{p(x_1|1)}{p(x_1|0)}) + \log(\frac{p(x_2|1)}{p(x_2|0)}) + \cdots + \log(\frac{p(x_D|1)}{p(x_D|0)}) + \log(\frac{p(1)}{p(0)}) > 0.$$

The conclusion of this is: if we transform each attribute X_i of an instance into X_i' where $X_i' = \log(p(X_i|1)/p(X_i|0))$, and define $t = -\log(\frac{p(1)}{p(0)})$, then in this transformed space, we are really learning a linear separator of the form

$$x_1' + x_2' + \cdots x_D' > t.$$

Example 12.3 Assume you have a database of cars, where cars are described by their type (sedan, break, or SUV) and color (blue, red or green), and their class is whether you like them or not. Assume that among the cars you like, 60% are blue, 30% are red, and 10% are green; whereas among the cars you don't like, 50% are green, 20% red, 30% blue. The attribute *Color* of the car can be transformed into a numeric attribute *Color'* which takes the value $\log(0.6/0.3) = \log(2)$ for Color=blue; $\log(0.3/0.2) = \log(1.5)$ for red; and $\log(0.1/0.5) = \log(0.2)$ for green. A similar transformation is possible for *Type*. A linear classifier in the transformed space (with coordinates *Color', Type'*) corresponds to the Naive Bayes classifier in the original space.

Note: we have seen before that the XOR function defines classes that are not linearly separable, and that for instance the perceptron cannot learn it because of that. Similarly, Naive Bayes is essentially a linear classifier that cannot learn XOR.

12.5 Bayesian networks

12.5.1 Factorizing a JPD

We have seen earlier that the JPD is the completest possible model of a dataset, in the sense that the exact probability distribution of any subset of variables given any other subset of variables can be derived from it (and as such it allows Bayes optimal prediction of any variables given any other variables), but that estimating the JPD without any bias is difficult because the number of instances needed is roughly exponential in the number of variables. We have also seen that the Naive Bayes method avoids that problem by assuming that the X_i are conditionally independent of each other given Y, which means

$$\begin{aligned} p(x_1, \ldots, x_D, y) &= p(x_1, x_2, \ldots, x_D|y)p(y) \\ &= p(x_1|y)p(x_2|y)\cdots p(x_D|y)p(y) \end{aligned}$$

where the first equality is generally valid and the second equality expresses the assumption of conditional independence.

Rewriting the JPD as a product of simpler distributions is called *factorizing* the JPD. The Naive Bayes method uses one particular way of factorizing the JPD, but other ways are possible, as we will soon see.

What are the advantages of factorizing a joint probability distribution into simpler ones? Perhaps the main advantage is the following. To estimate a joint or conditional distribution $p(\mathbf{x}|\mathbf{x}')$, we need a number of examples that is roughly exponential in the total number of variables occurring in $\mathbf{x}$ and $\mathbf{x}'$. When Naive Bayes, the maximal number of variables in any distribution is 2 (the $p(x_i|y)$ distributions). As soon as each combination of x_i and y occurs relatively frequently in the training set, we can estimate $p(x_i|y)$ relatively accurately. This is often a massive reduction of required data, compared to estimating the full JPD directly, where each combination of $(x_1, x_2, \ldots, x_D, y)$ needs to occur relatively frequently in the training set.

Another advantage is that the factorization expresses certain structure in the data: it gives information about which independencies exist. From another point of view, this is a disadvantage: we need to know which dependencies exist in order to find a good factorization. Any non-trivial factorization is only correct if certain assumptions are made; if these assumptions are violated, the factorized form may give us an approximation of the JPD but will not be equal to it. The question is then to what extent being able to estimate the factor distributions more accurately compensates for having this approximation.

12.5.2 An example factorization

Example 12.4 Let X_1 and X_2 be die rolls, let $X_3 = X_1 + X_2$, let X_4 and X_5 be coin tosses and X_6 be a boolean variable which is true if $X_4 = X_5$ and false otherwise. There is then a group X_1, X_2, X_3 of variables that is entirely independent from X_4, X_5, X_6. Therefore, $p(x_1, \ldots, x_6) = p(x_1, x_2, x_3)p(x_4, x_5, x_6)$.

If all variables within a group were independent, we could factorize the probability distribution further. The situation is somewhat complicated, however, because while X_1 and X_2 are independent, they are not independent of X_3. In such cases, it is still possible to factorize the probability function further, as we will see in a moment.

Note that while the factorization into marginal probability functions is only possible in the case of independence, it is always possible to factorize the JPD into conditional dependencies, e.g.,

$$p(x_1, x_2, x_3) = p(x_1|x_2, x_3)p(x_2|x_3)p(x_3).$$

12.5.3 From trivial to non-trivial factorizations

Generally, without making any assumptions, and without needing to treat the class variable separately (just assume Y is one of the X_i), we have

$$\begin{aligned} p(x_1, \ldots, x_D) &= p(x_1|x_2, \ldots, x_D)p(x_2, \ldots, x_D) \\ &= p(x_1|x_2, \ldots, x_D)p(x_2|x_3, \ldots, X_D)p(x_3, \ldots, x_D) \\ &= \ldots \\ &= p(x_1|x_2, \ldots, x_D)p(x_2|x_3, \ldots, X_D) \cdots p(x_{D-1}|x_D)p(x_D) \end{aligned}$$

That is: we can separate out one variable and multiply its conditional probability given the rest with the joint probability of the rest; and we can recursively apply the same rule on that second joint probability, repeating this until the last unconditional probability contains only one variable.

Now what does this formula tell us? It gives us an alternative factorization of the JPD into multiple simpler conditional density functions. In itself, this does not help us much, because the most complex of these simpler functions, $p(x_1|x_2, \ldots, x_D)$, still requires as much data as the JPD itself: we need counts for every combination of $x_1, \ldots, x_D$. The good news is that this factorization is correct; the bad news is that it doesn't buy us anything.

However, consider the possibility that X_1 is independent of some variable X_i, given the other variables X_j, $j \neq i$. Then that X_i of which X_1 is independent can be removed from the condition part (this is correct by definition of conditional independence), and then the "most complex" factor becomes a bit less complex. Of course we can remove from the condition part any X_i of which X_1 is independent in this way.

The same principle can be applied to any of the factors. For instance, if X_{D-1} is independent of X_D, then the last two factors become $p(X_{D-1}|X_D)p(X_D) = p(X_{D-1})p(X_D)$. Similarly, if X_{D-2} is independent of X_{D-1} given X_D, then we have $p(X_{D-2}|X_{D-1}, X_D) = p(X_{D-2}|X_D)$.

Generally, the amount of data needed to estimate the distributions will be exponential in the highest number of variables occurring in any single factor. In other words, the most complex factor is the bottleneck: the estimation of the JPD is just as difficult as the estimation of its most complex factor. Hence, the more the factors can be simplified, the better.

Exercise 12.2 We use the term "complexity" informally in the above description. As mentioned before, the amount of data needed to accurately estimate a joint probability distribution depends on the number of variables, but also on the number of values for each variable. Try to formulate an expression indicating the complexity of a factor in the above factorization that takes into account the number of values of each variable as well as the number of variables.

Note that the factorization of the JPD into multiple conditional probability distributions can be done in many different ways. In the above factorization we first separated X_1, then X_2, etc., but the X_i could be separated in any order. There are $n!$ possible orderings. For each of these orderings, the factors can be simplified in a different way, and some orderings may lead to simpler factorizations than others.

Exercise 12.3 In Naive Bayes, we assume that each X_i is independent from X_j, $j \neq i$, given the class Y (and this is the only independence assumption made). Under this assumption, factorize $p(X_1, X_2, X_3, X_4, X_5, Y)$ as above, using three different orderings:
1) X_1, X_2, X_3, X_4, X_5, Y;
2) X_1, X_5, X_3, X_2, X_4, Y;
3) X_3, X_2, Y, X_1, X_4, X_5.
Which condition should the ordering fulfill in order to get the optimal factorization, i.e., the one used by Naive Bayes? Which property of the ordering influences the complexity of the most complex factor?

12.5.4 Indicating the conditional dependencies graphically

Given an ordering of the X_i, we can graphically indicate which variables are conditionally independent of which other variables, given yet other variables. This can be done using a graph structure where each X_i is a node and an arrow from X_i to X_j indicates that X_i occurs in the condition part of $p(X_j|\ldots)$. For instance, if $p(X_1|X_2, X_3, X_4, X_5)$ can be reduced to $p(X_1|X_3, X_4)$, which means that X_1 is independent of X_5 and X_2 given X_3 and X_4, we indicate this in the graph by drawing arrows from X_3 and X_4 to X_1. The *absence* of arrows from X_2 and X_5 indicates that X_2 and X_5 do not influence X_1 directly: the values of X_2 and X_5 do not need to be known to estimate X_1 (assuming we know X_3 and X_4). Note the qualification: they do not influence X_1 *directly*. This does not mean that X_2 and X_1 are independent: X_2 may influence X_1, but this influence, if it exists, has to go through X_3 or X_4. If we know the values of X_3 and X_4, then the value of X_2 cannot influence the probabilities of X_1, its influence is already taken into account by X_3 and X_4. We say that the variables X_3 and X_4 *block* the influence of X_2 on X_1.

Example 12.5 Consider the following variables: for any student, X is the average score of a student for mathematics in high school, B is the score for Calculus in the first year at University, and C is the ranking of the student for Calculus (i.e., whether the student is the best of his class, second best, etc.). We can expect that there is some stochastic dependency between B and C: students with a higher score will have a better ranking. We also expect a correlation between A, the score for mathematics in high school, and B, the score for Calculus at University, since people who are good at mathematics will tend to score high on both. Finally, we can expect a correlation between A and C for exactly the same reason: people who are good at mathematics will tend to have better ranks.

However, while there is a correlation between A and C, it goes entirely through B. That is, the rank of a student depends solely on his score for Calculus. If we do not know a student's score for Calculus, it makes sense to use his score for mathematics in high school to estimate the ranking of the student for Calculus. But if we know his score for Calculus, the ranking follows entirely from that, and the student's score for mathematics at high school is totally irrelevant.

This is an example of a case where C is independent of A given B. Note that we cannot just say that C is independent of A, the qualification "given B" is essential.

Exercise 12.4 Draw the graph corresponding to the previous example. I.e., for the ordering C, B, A, indicate that B depends on A, C depends (globally) on both A and B, but C is independent of A given B.

What happens if we use the ordering A, B, C?

Note that the "ordering" in the factorization we described earlier is the opposite of the topological ordering in which the variables occur in the graph (where x comes

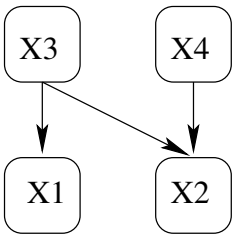

Figure 12.6: Graph corresponding to the factorization $p(x_1, x_2, x_3, x_4) = p(x_1|x_3)p(x_2|x_3, x_4)p(x_3)p(x_4)$.

before y if there is a path from x to y in the graph). If we first separate X_1, this means X_1 can depend on all the other variables, which means that there may be arrows from any of those variables to X_1 but not in the opposite direction; hence X_1 is an "end" node in the graph (it can have only incoming arrows, no outgoing ones). Usually, when people talk about the order of the variables, they mean the topological order in the graph. *From here on, when talking about an order, we refer to the topological order in the graph.*

Example 12.6 Assume that a distribution $p(x_1, x_2, x_3, x_4)$ can be factorized as follows: $p(x_1, x_2, x_3, x_4) = p(x_1|x_3)p(x_2|x_3, x_4)p(x_3)p(x_4)$. The graph in Figure 12.6 shows the dependencies and independencies that this factorization implies.

12.5.5 Bayesian networks

Given a factorization, there is one particular directed graph that corresponds with it, and this graph is always acyclic. Conversely, given an acyclic directed graph, one particular factorization corresponds to it. While the graph indicates how the JPD is factorized, it does not indicate the factor distributions themselves. That is, having an edge from X_3 to X_1 implies we will have a factor $p(x_1|x_3)$, but it does not define the p function itself. This information needs to be added to the graph. Thus, for each node X with incoming arrows from a set of nodes S, the node is annotated with the distribution $p(x|S)$.

Example 12.7 Figure 12.7 shows the graph corresponding to the previous example, together with tables showing the factor distributions. It is assumed that X_1 and X_2 both take values *red* and *green*; X_3 takes values a, b, c; and X_4 takes values T and F. The tables show the conditional probability of a node given any combination of values for its parents; for nodes without parents this becomes an unconditional probability.

The probability $p(red, red, a, T)$ can then be computed as $p(red, red, a, T) = p_{X_1|X_3}(red|a)p_{X_2|X_3,X_4}(red|a, T)p_{X_3}(a)p_{X_4}(T) = 0.3 \cdot 0.3 \cdot 0.2 \cdot 0.5 = 0.009$.

The graph indicating the dependencies among variables, together with the tables that define the conditional probability distributions, is called a **Bayesian network**. If there is an arrow from X to Y, we call X a parent of Y, and Y a child of X. If there is a path from X to a node Y, we call X an ancestor of Y, and Y a descendant of X. The independencies expressed by the Bayesian network can then be formulated as follows: *each variable is conditionally independent of all its non-descendants, given its parents.*[2]

[2]To see how this follows from the factorization, note that (1) by definition, each variable is conditionally independent of all the variables preceding it in the ordering, given its parents (indeed

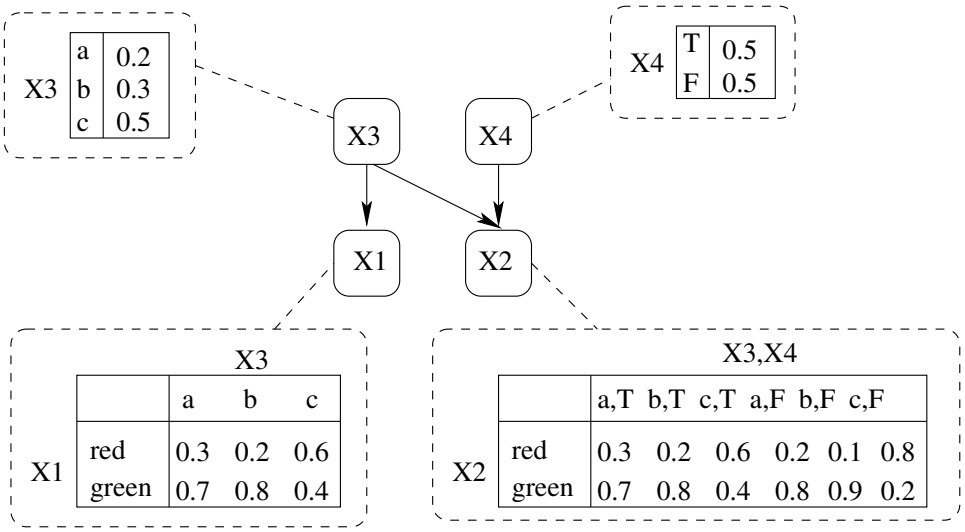

Figure 12.7: The graph of Figure 12.6, annotated with the (conditional) probability distribution for each node.

A classic example of a Bayesian network is shown in Figure 12.8. There are five boolean variables, indicating for some house in, say, California, whether: there is a burglary in the house (B); there is an earthquake (E); the alarm goes off (A); John, one of the neighbors, calls the house owners on the phone (J); Mary, another neighbor, calls them (M).

It is instructive to reason about this example and see what the dependencies indicated in the graph mean in practice. In the following it is important to clearly distinguish conditional independence ($p(x, y|z) = p(x|z)p(y|z)$) and unconditional independence ($p(x, y) = p(x)p(y)$), which we will also refer to as *global independence*.

The intuition behind this network is that a burglary or an earthquake increases the probability that the alarm goes off, and that if the alarm goes off, neighbors are likely to call to check if everything is OK. These are causal dependencies, with naturally give rise to stochastic dependencies (we will return to the difference between these two later). Thus we get the graph shown in Figure 12.8. Note that the graph expresses the assumption that a burglary has no direct effect on whether the neighbors call. Roughly, this means that we assume that the neighbors have no other information about whether there might be a burglary or not besides the alarm going off. Similarly, whether there is an earthquake or not does not influence the neighbors calling, except through the activation of the alarm. If the alarm does not go off, then an earthquake does not make them more likely to call. If the alarm goes off, there is a higher probability that they call, but how high it is does not depend on whether there is an earthquake or not.

Note, further, that the network expresses that John calling has no influence on Mary calling, given that the alarm does or does not go off. This does not mean that J and M are independent: if John calls, it is more likely that Mary calls as well. This is because it is more likely that the alarm is going off at that time (since that's a possible reason for John calling), and that increases the probability that Mary calls too. So J and M are not independent, but they are conditionally independent given A.

---

that was how we simplified the factorization); (2) for any variable y that is not a descendant of x, i.e., for which there is no path from x to y, we can find a total ordering consistent with the graph in which y precedes x.

Exercise 12.5 Write down the factorization that the network in Figure 12.8 represents. Compute the probability that there is a burglary but no earthquake, the alarm goes off, John calls, and Mary does not call. Next, compute the same probability but without the condition that there is no earthquake (i.e., there might or might not be an earthquake). Finally, compute the probability that the alarm goes off (without knowing anything else). Hint: you will need several of the probability rules we have seen.

In Figure 12.9 two alternative graphs are shown, corresponding to different factorizations of the JPD. The left hand side of Figure 12.9 shows the network we just used; on the right hand we see the network that would be obtained for a different ordering: J, M, A, B and E. Starting with J means J has no incoming edges. We then introduce M, and need to decide which incoming edges M gets. At this point the question is: does M depend on J? The answer is: yes. As just explained, J and M are conditionally independent given A, but not globally independent. Since A has not yet been introduced in the network, we cannot express the conditional independence between J and M at this point, and since J and M are not globally independent, we have to draw an edge from J to M. (Not drawing that edge would indicate independence between the two.)

Next we introduce A. Does A depend on J, M, or both? The probability of the alarm going off is higher if John calls or Mary calls, and probably highest if both call. Is A independent of M given J? That would mean that given that John calls, the probability that the alarm goes off is the same whether Mary calls or not. As just mentioned, this is not the case: if both are calling, this makes it more likely that the alarm is going off than if one of them calls.

Example 12.8 That A is not independent of M given J can also be shown using the Bayesian network in Figure 12.8.

We need to show that $p(a, m|j) \neq p(a|j)p(m|j)$. First compute $p(a, m|j)$:

$$p(a, m|j) = p(a, m, j)/p(j) = (p(j|a)p(m|a)p(a))/p(j)$$

where the first step is correct by definition and the second step follows from the factorization of $p(a, m, j)$ that the Bayesian network expresses.

For $p(a|j)$ we apply similar steps to get

$$p(a|j) = p(a, j)/p(j) = p(j|a)p(a)/p(j)$$

and for $p(m|j)$ we get

$$p(m|j) = p(m, j)/p(j) = (\sum_{a'} p(m, j, a'))/p(j) = (\sum_{a'} p(m|a')p(j|a')p(a'))/p(j).$$

In these formula, all the conditional probabilities are immediately found in the tables; only $p(a)$ and $p(j)$ are to be computed.

$$p(a) = \sum_{b,e} p(a, b, e) = \sum_{b,e} p(a|b, e)p(b)p(e)$$

$$p(j) = \sum_{a} p(a, j) = \sum_{a} p(j|a)p(a)$$

This gives (we add the subscripts for A and J when filling in constants T and F to avoid confusion between p_A with p_J):

$p_A(T) = 0.9 \cdot 0.05 \cdot 0.01 + 0.8 \cdot 0.05 \cdot 0.99 + 0.4 \cdot 0.95 \cdot 0.01 + 0.01 \cdot 0.95 \cdot 0.99 = 0.053255$
$p_A(F) = 0.1 \cdot 0.05 \cdot 0.01 + 0.2 \cdot 0.05 \cdot 0.99 + 0.6 \cdot 0.95 \cdot 0.01 + 0.99 \cdot 0.95 \cdot 0.99 =$
$0.946745 \qquad (= 1 - p_A(T))$
$p_J(T) = 0.8 \cdot p_A(T) + 0.1 \cdot p_A(F) = 0.1372785$
$p_J(F) = 0.2 \cdot p_A(T) + 0.9 \cdot p_A(F) = 0.8627215$

Thus we find, by filling in for instance the values T, T, F for a, m, j, respectively:

$$\begin{aligned}
p_{A|J}(T|F) &= p_{J|A}(F|T)p_A(T)/p_J(F) = 0.2 \cdot 0.053255/0.8627215 \\
&= 0.0123458149588251 \\
p_{M|J}(T|F) &= (\sum_a p_{M|A}(T|a)p_{J|A}(F|a)p_A(a))/p_J(F) \\
&= (0.9 \cdot 0.2 \cdot 0.053255 + 0.2 \cdot 0.9 \cdot 0.946745)/0.8627215 \\
&= 0.208642070471178 \\
p_{A,M|J}(T,T|F) &= p_{J|A}(F|T)p_{M|A}(T|T)p_A(T)/p_J(F) = 0.2 \cdot 0.9 \cdot 0.053255/0.8627215 \\
&= 0.0111112334629426
\end{aligned}$$

and we see that $p_{A|J}(T|F) \cdot p_{M|J}(T|F) \neq p_{A,M|J}(T,T|F)$, which contradicts the hypothesis that for all a, m, j: $p_{A|J}(a|j)p_{M|J}(m|j) = p_{A,M|J}(a, m|j)$, i.e., that A and M are independent given J. In particular, the probability that Mary calls and there is an alarm, given that John calls, is higher than what would be expected if Mary calling and the alarm going off were independent given that John calls.

Next, we introduce B. It is more likely that there is a burglary if the alarm is going off than if it's not. Given that the alarm goes off, is it more likely that there is a burglary if John is calling? No, because as said before the alarm "blocks" the path between B and J: given that there is an alarm (or not), J and B are independent. The same holds for B and M. So, B depends only on A.

Finally, we introduce E. Like B, E depends on A but is independent from J and M given A. Is E also independent of B given A? No. The intuitive reasoning goes as follows. The alarm tends to go off only when there is a burglary or an earthquake. So if the alarm goes off, it is likely that there is either a burglary or an earthquake. If we now find out that there is no burglary, that clearly increases the probability that there is an earthquake. So information on B is relevant for E, when we know the value of A.

Note that, interestingly, if we have *no* information about A, then B and E are independent. So B and E are globally independent, but not conditionally independent given A. This is the opposite situation of J and M, which were conditionally independent but not globally independent.

Clearly, global independence should not be interpreted as independence "under all circumstances", but as independence in those circumstances where the values of other variables are unknown. I.e., such independence is not really unconditional, it is conditioned on *not knowing* the values of certain variables.

Exercise 12.6 Write down the factorization of $p(A, B, E, J, M)$ according to the two graphs in Figure 12.9.

From the point of view of complexity of the most complex factor in the factorization (i.e., the number of data needed to estimate the factor distributions), which of these factorizations is the simplest one?

12.5.6 Independencies implied by Bayesian networks

The above intuitive reasoning gave us already some idea of in what manners variables can influence other variables, or, put differently, how information can flow

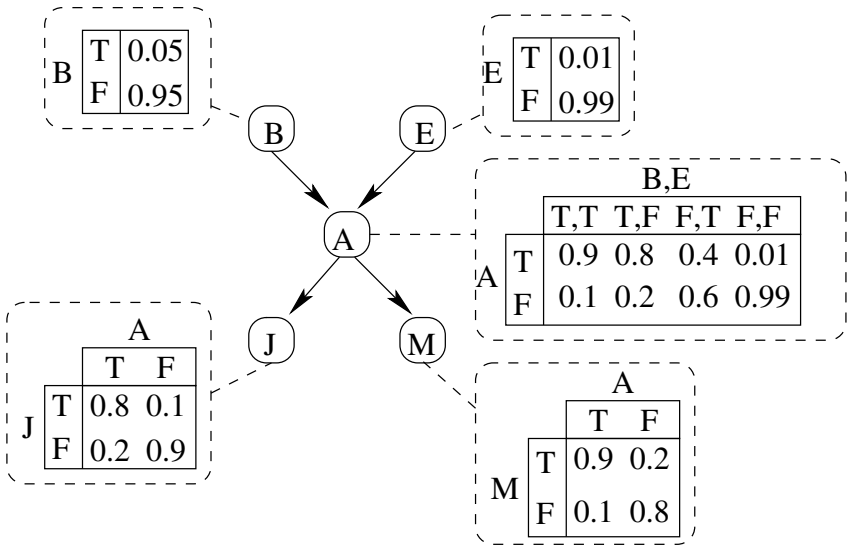

Figure 12.8: A Bayesian network for the Alarm example.

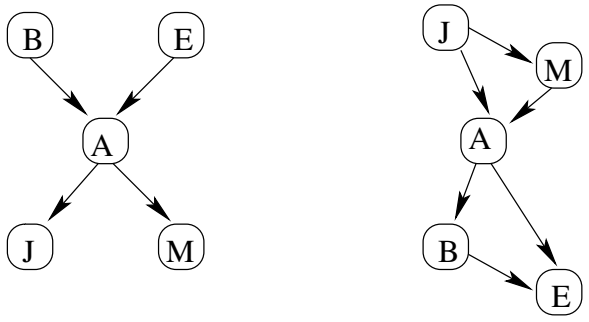

Figure 12.9: Two Bayesian networks (graph structure only) indicating dependencies between the five variables of the Alarm example. Both correspond to factorizations obtained with different initial orderings of the variables. Note that the left network has fewer edges than the right network.

between two variables (how does knowledge of one variable carry over to knowledge of another variable).

Consider two variables A and B. Under what circumstances does knowledge about A influence our estimate of B? Clearly, if A and B are directly connected, information about one of them is relevant for the other, in both directions. (If a burglary makes it more likely that the alarm goes off, then conversely, the alarm going off carries information about whether there is a burglary). So information can flow from A to B or from B to A, whatever the direction of the edge is.

What if A and B are connected through one intermediate variable X? We distinguish three possibilities (see also Figure 12.10):

1. $A \rightarrow X \rightarrow B$: A influences B only through X. That means that if we know the value for X, the value of A does not influence our estimate for B (assuming there are no other paths between A to B). If we don't know X, A does influence B. This is the "Calculus" case mentioned in an earlier example.

   We could say that X *blocks* the flow of information from A to B (or from B to A) if X has a *known* value. If the value of X is *not known*, the flow is *not blocked*.

2. $A \rightarrow X \leftarrow B$: A and B both influence X, and therefore X carries information about A and B. If we know X, then A tells us something about B, but if we don't know X, then A does not give us information about B. This is the case "earthquake, burglary and alarm" case just mentioned: if the alarm goes off, then knowledge about whether there is a burglary influences our estimate of whether there is an earthquake. If we don't know anything about the alarm going off, then knowledge about a possible burglary does not influence our estimate of whether there is an earthquake.

   Thus, in this situation, X *blocks* the flow of information between A and B when it is *unknown*, but it *enables* this flow when it is *known*.

   The term "unknown" is slightly ambiguous here; it should be interpreted as "there is no information whatsoever about X". This is a stronger condition than just stating that X is not part of the evidence (the variables of which the values are known). Even if X has not been observed, partial information about X may be available, for instance because children of X are known, and such partial information enables information flow between A and B. To be precise, we should say that X blocks a path $A \rightarrow X \leftarrow B$ if none of the descendents of X have a known value.

3. $A \leftarrow X \rightarrow B$: this by definition means that A and B are conditionally independent given X, which means that if X is known, A can give no information about B. A and B are not necessarily globally independent, so when X is unknown (not given), A can carry information about B. As in the first case, *knowing* X *blocks* the information flow between A and B along this path; *not knowing* X *enables* this flow.

Generally, information can flow from A to B if there exists at least one undirected path between A and B that is not blocked by an intermediate variable. To see if A is relevant for B, we consider all paths from A to B, where edges can be followed in any direction. If at least one such path is not blocked, A is relevant for B. If no such path exists, i.e., all paths between A and B are blocked in at least one point, we say that A and B are **d-separated**. When two variables are d-separated, given a certain context of known and unknown variables, it means they are independent in this context: knowing the value of one variable does not influence the probability distribution of the other variable.

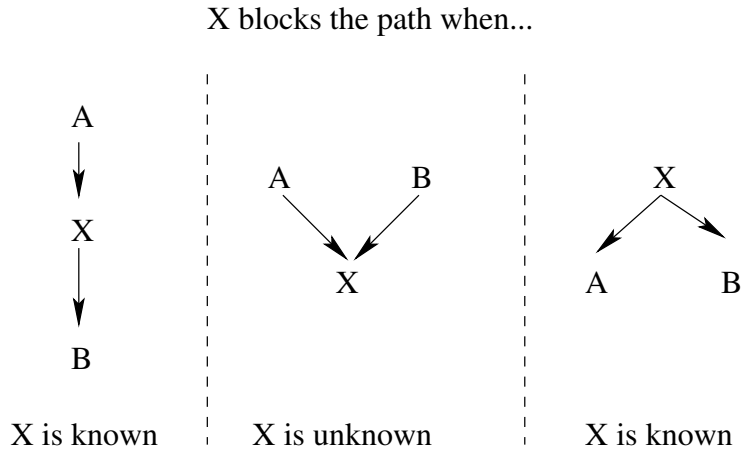

Figure 12.10: When X is on an undirected path between A and B, there are three possible configurations for X relative to this path: X may have two incoming edges, two outgoing edges, or one incoming and one outgoing edge. In the case of two incoming edges, knowing the value of X enables information flow between A and B (i.e., from A to B or from B to A). In the other two cases, knowing X blocks this information flow.

A set of variables S is said to d-separate two variables in a Bayesian network if knowing the values of all variables in S and not knowing any other variables causes the variables to be d-separated.

Example 12.9 Assume that there is a path between variables X and Y that looks as follows: $X \rightarrow A \leftarrow B \rightarrow C \rightarrow Y$. A blocks this path if it is unknown, B and C block the path if they are known. If this is the only path between X and Y, then it is sufficient to know B or C, or to know nothing about A, to have d-separation between X and Y. We can say that the set $\{B\}$ d-separates X and Y. The set $\{B, C\}$ also d-separates X and Y. The set $\{A\}$ does not d-separate X and Y.

12.5.7 The Markov blanket of a variable

Assume that we want to predict the value of a variable A, or more generally, that we want to estimate the probability distribution for A. Further assume that all the other variables have known values. Which of these other variables do we actually need to know to predict A's distribution? Which variables are relevant for predicting A?

The smallest set of variables that carries complete information about A (in the sense that adding other variables to it cannot change the probability distribution for A) is called the **Markov blanket** of A.

Basically, if U represents all variables, $U' = U - \{A\}$ represents all variables besides A, and B is the Markov blanket of A, then it holds that

$$p(A|U') = p(A|B).$$

Note the difference between this density and the factors in the factorization we saw before. Those factors were the conditional probability function for a single variable given a limited set of other variables, namely those that had not yet been separated out. The form of the factor was typically $p(X_i|X_{i+1}, \ldots, X_D)$, whereas here we are interested in $p(X_i|X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_D)$. The former function is

explicitly represented in the Bayesian net as the CPD associated with X_i, but the latter is not explicit in the network.

What variables are in the Markov blanket? Since we assume all variables known (except A), some of these variables will block possible influences of other variables on A.

The parents of A obviously influence A, and block all influences through them. Indeed, if X is a parent of A, then we have an $X \to A$ edge, and we have seen that in any path containing an edge $X \to A$ (this combines the two cases $B \to X \to A$ and $B \leftarrow X \to A$), X blocks the information flow to or from A on that path.

There was a third case: $A \to X \leftarrow B$. Here X, being a child of A, does not block information flow to A when it is known, but on the contrary *enables* it. If X were not known, more precisely, if all the children of A are unknown, then the distribution of A would depend entirely on its parents. But if a child X is known, X determines A as well, and what's more, it enables information flow from other parents of X towards A, according to the schema just given. (Note that it does not enable a flow from any of its own children: any child of X is connected to A through a path $A \to X \to B$ and X blocks that path.) The parents of its children, themselves, block any further information from outside, in the same way that A's parents block the information flow towards A; so the information flow stops here.

Thus, A is influenced by its parents, its children, and the parents of its children. These nodes form the Markov blanket of A.

Generally, if we want to determine the probability distribution of a variable A, the more information we have about variables related to A, the more accurately we know this probability distribution. *The Markov blanket contains all the relevant information.* If we know all the variables in the Markov blanket, finding out the value of additional variables will not help. If we do not know the value of some variable in the Markov blanket, information on these other variables may help, but it can only help insofar it provides information about that missing variable in the Markov blanket.

Note that, using the notion of d-separation, the Markov blanket of X can also be defined as the smallest set S that d-separates A from all other nodes in the Bayesian network.

12.5.8 Inference in Bayesian networks

We have discussed Bayesian networks as a compact representation of JPDs, and discussed how information can flow from one variable to another. We now turn to the general question of how to perform inference in Bayesian networks.

The question, stated in the most general way, is: given a partitioning of the variables in a set of known variables K and a set of unknown variables U, compute a probability distribution over one or more variables Q in U. That is, compute $p(Q|K)$ with $Q \subseteq U$. A special case of this is when Q contains only one variable, the target variable Y. Note that $p(Y|K) = p(Y|B_Y)$, with B_Y the Markov blanket of Y, if $B_Y \subset K$.

Inference in Bayesian networks can be quite complicated, and is in the general case NP-complete. To give some indication of how complicated it is, consider the simple case of a Bayesian net with 2 nodes and one edge: $A \to B$. There are four combinations of known/unknown variables.

1. A is known, and we want to predict B. I.e., we look for $p(B|A)$. This is a trivial case: $p(B|A)$ is explicitly represented in the Bayesian net, it is the conditional probability table associated with B.

2. A is unknown, we want to predict B. This means we want to compute $p(B)$. $p(B)$ is obtained from $p(A)$ and $p(B|A)$, which are both present in the Bayesian

net, using the rule of total probability: $p(B) = \sum_a p(B|a)p(a)$ where we sum over all possible values a for A.

3. B is known, we want to predict A; i.e., we need to compute $p(A|B)$. This can be obtained using Bayes' rule: $p(A|b) = p(b|A)p(A)/p(b)$. $p(b|A)$ and $p(A)$ are stored, and $p(b)$ can be computed with the rule of total probability as before.

4. B is unknown, we want to predict A. We need $p(A)$, which is again trivial: it's stored in the Bayesian network.

Thus, four different kinds of inference in a network with two nodes and one edge already require four quite different solution strategies. The situation becomes much more complex with more nodes. We have already seen an example of that in Example 12.8.

A general technique to compute any conditional probability in the Bayesian net is to write it in terms of the JPD. Remember that the JPD of a Bayesian net is simply the product of all the (conditional or unconditional) probability distributions associated with the nodes. Any marginal distribution can be computed from that using the rule of total probability, and any conditional distribution as the ratio of two distributions. That gives:

$$p(\mathbf{q}|\mathbf{k}) = p(\mathbf{q},\mathbf{k})/p(\mathbf{k}) = \frac{\sum_{\mathbf{u}} p(\mathbf{q},\mathbf{u},\mathbf{k})}{\sum_{\mathbf{q}}\sum_{\mathbf{u}} p(\mathbf{q},\mathbf{u},\mathbf{k})} \tag{12.13}$$

where

$$p(\mathbf{q},\mathbf{u},\mathbf{k}) = \prod_X p(x|\mathbf{p}_X))$$

with the variable X ranging over all variables in the Bayesian network and $\mathbf{p}_X$ indicating the parents of X.

Note that the sums go over all possible vectors $\mathbf{u}$ and $\mathbf{q}$, that is, over all combinations of values for the variables in U and Q. The number of such combinations is roughly exponential in the number of variables. Thus, while the formula may seem relatively simple, it is very expensive to compute.

In practice, it may be possible to simplify the formula. For instance, all those nodes in U for which all the descendants are also in U can be left out of the summation, which gives:

$$p(\mathbf{q}|\mathbf{k}) = p(\mathbf{q},\mathbf{k})/p(\mathbf{k}) = \frac{\sum_{\mathbf{u}'} p(\mathbf{q},\mathbf{u}',\mathbf{k})}{\sum_{\mathbf{q}}\sum_{\mathbf{u}'} p(\mathbf{q},\mathbf{u}',\mathbf{k})} \tag{12.14}$$

where $U' \subseteq U$ is the set of nodes for which it holds that at least one descendent is in Q or K.

Exercise 12.7 Look at the four reasoning mechanisms for the two-node network that we discussed earlier, and indicate where the above property has been used implicitly. Try to prove that the rule holds in general.

A detailed treatment of inference and simplification techniques is beyond the scope of this text. We do give an alternative generally applicable method (the Monte Carlo inference method described below) and a simplified method for predicting the conditional probability distribution of a variable given its Markov blanket.

Monte Carlo inference

A very simple empirical method is the following: draw a large sample from the distribution defined by the Bayesian network, retaining only those instances where all the variables in K have the same value as the observed one. Among these, count how many times each value for Q occurs. Note that this process is empirical: only if the sample is large enough will we have a good estimate of the probability distribution, and it is still only an estimate.

Drawing a sample from the distribution defined by the Bayesian network is easy. For each variable X that has no parents, we draw a random value for X according to $p(X)$. Next, for each variable X for which all the parents have been instantiated (received a value), we use $p(X|parents(X))$ to generate a random value for X. We continue doing this until each variable has a value. That gives us one observation for our sample.

This is a very simple method, but it is also very inefficient. A large majority of the examples generated will not have the correct values for some variable in K and will have to be rejected. We may need to generate billions of instances to keep a few thousand. Nevertheless, in many cases it is more efficient than the NP-complete inference method mentioned above.

Predicting a variable given its Markov blanket

Let Y be the variable for which we want to compute the conditional probability distribution given its Markov blanket; let B denote the Markov blanket of Y, and let P denote the parents of Y, C its children, and CP those children's parents (the "co-parents"). $B = P \cup C \cup CP$, so we have

$$p(y|\mathbf{b}) = p(y|\mathbf{p}, \mathbf{c}, \mathbf{cp}) = p(y, \mathbf{p}, \mathbf{c}, \mathbf{cp})/p(\mathbf{p}, \mathbf{c}, \mathbf{cp}). \tag{12.15}$$

We can factorize $p(y, \mathbf{p}, \mathbf{c}, \mathbf{cp})$ as follows:

$$p(y, \mathbf{p}, \mathbf{c}, \mathbf{cp}) = p(\mathbf{c}|y, \mathbf{cp}, \mathbf{p})p(y|\mathbf{cp}, \mathbf{p})p(\mathbf{cp}, \mathbf{p}) = p(\mathbf{c}|y, \mathbf{cp})p(y|\mathbf{p})p(\mathbf{cp}, \mathbf{p}). \tag{12.16}$$

The first equality generally holds. The second equality introduces two simplifications. $p(\mathbf{c}|y, \mathbf{cp}, \mathbf{p}) = p(\mathbf{c}|y, \mathbf{cp})$ because Y and CP, being all the parents of C, block P (unless P would be a descendent of C, but that would mean that there is a cycle in the Bayesian net, which is not possible). $p(y|\mathbf{cp}, \mathbf{p}) = p(y|\mathbf{p})$ because CP can only influence Y through P or C; the variables in P are known and block the influence, while C does not occur in the condition part (hence is not given, or "unknown" from the point of view of this distribution).

We can further write $p(\mathbf{p}, \mathbf{c}, \mathbf{cp})$, the denominator of Equation 12.15, as

$$p(\mathbf{p}, \mathbf{c}, \mathbf{cp}) = \sum_Y p(y, \mathbf{p}, \mathbf{c}, \mathbf{cp}) = \sum_y p(\mathbf{c}|y, \mathbf{cp})p(y|\mathbf{p})p(\mathbf{cp}, \mathbf{p}) \tag{12.17}$$

Now combining the righthand sides from Equations 12.16 and 12.17 into Equation 12.15, and dropping $p(\mathbf{cp}, \mathbf{p})$ from both enumerator and denominator, we get

$$p(y|\mathbf{p}, \mathbf{c}, \mathbf{cp}) = p(\mathbf{c}|y, \mathbf{cp})p(y|\mathbf{p})/\sum_y p(\mathbf{c}|y, \mathbf{cp})p(y|\mathbf{p}) \tag{12.18}$$

which gives us the conditional probability of Y, given its Markov blanket, in terms of the conditional probability tables of Y, given its parents, and of the children of Y given their parents.

12.5.9 Learning Bayesian networks

After all the complex mathematics involved in using Bayesian networks, we now turn to learning Bayesian networks. Fortunately, learning such networks is relatively simple!

We consider two settings for learning. In the first setting, we assume the graph structure given, and we try to estimate the parameters, i.e., the conditional probability function associated with each node. In the second setting, we consider learning the graph structure itself.

Learning the parameters for a given graph

For a given node X, we need to estimate $p(x|\mathbf{p})$ with P the set of parents of X. If X has no parents, this is simply $p(x)$, which can be estimated simply by counting how often each value for X occurs. If X has parents, then for each combination of values for $\mathbf{p}$ we count the relative frequency of each value for X given that combination. As with Naive Bayes, this counting can be made a bit more sophisticated by using the m-estimate for these probabilities instead of the unmodified relative frequencies.

Exercise 12.8 *(Advanced.)* Show that by estimating the conditional probability distributions in a Bayesian network using count ratios, we get a maximum likelihood JPD; that is, of all JPDs that can be expressed by the Bayesian network, the JPD that uses these distributions gives the training set the highest likelihood.

And that's it! Estimating the parameters of a Bayesian network is no more complicated than estimating the parameters of a Naive Bayes model.

Well, there is one "but". We have assumed here that all the variables are observable. In some cases, Bayesian networks may contain unobservable variables, which means that no values are available for these variables in the data set. This is a bit similar to neural networks having hidden nodes.

The question may arise why one would ever want to have such a network. Remember that a Bayesian network reflects a certain structure in the joint distribution. Sometimes this structure can be described more compactly by introducing new, unobserved variables.

Example 12.10 Assume we have four boolean variables $A - D$ that are independent, and three variables E, F, G each of which depends on $A - D$ in a very specific way, namely: they are true with a certain probability if and only if $A \wedge B \wedge C \wedge D$ holds. We could express this boolean AND-function in terms of A, B, C, D for each of E, F, G separately, but it is more compact to introduce variable a X that depends on A, B, C, D once and represents the boolean AND-function, and to have E, F, G depend on X. The structure of the network becomes much simpler this way, and there are fewer parameters to be estimated. Figure 12.11 shows the network structure in both cases.

How can we learn the conditional probability distribution for a node for which we have not observed any values? This can be done using an EM process.

We first fill in the values for the unobserved nodes at random. Next, we estimate all the probability distributions in the usual way. Now, given the distributions, we can compute the probability distribution of an unobserved variable given its Markov blanket. This should tell us which values for the hidden variable are likely for each possible combination of values of its Markov blanket. We can regenerate the values for the unobserved variable randomly, now using this distribution. Next, we recompute the probability distributions with these values. We continue the process until convergence. Figure 12.12 shows the algorithm.

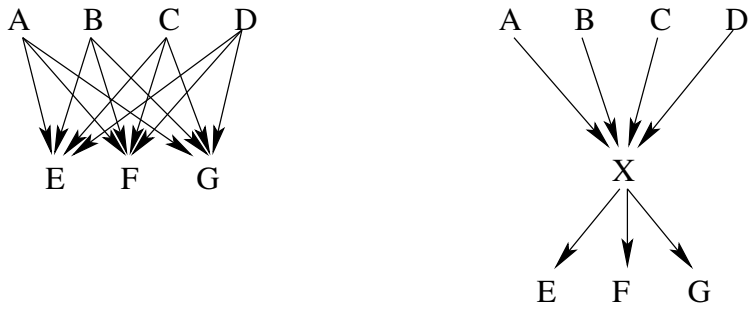

Figure 12.11: Two Bayesian nets describing how E, F and G depend on variables A, B, C, D; in the left case, without an additional (unobserved) variable X, and in the right case with X added to the network.

procedure EstimateParametersUsingEM(training set T, bayesian net BN)
returns set of CPD's:
// assume the data elements in T are vectors
// $(x_1, \ldots, x_m, u_1, \ldots, u_n)$;
// where u_i are unobserved variables;
for each unobserved variable U:
give a random CPD to U
repeat
for each unobserved variable U:
for each instance $\mathbf{x}$ in the training set T:
give U a random value according to $p(u|\mathbf{b})$,
with $\mathbf{b}$ the values of U's Markov blanket in the instance
for each variable V in the network:
estimate $p(v|\mathbf{p}_V)$, with $\mathbf{p}_V$ the parents of V
until convergence

Figure 12.12: An EM algorithm for estimating the conditional probability distributions in a Bayesian network with unobserved variables.

procedure StructureSearch **returns** Bayesian Network;
    B := random-network;
    b := evaluate(B);
    b^* := $-\infty$;
    while $b > b^*$:
        b^* := b
        B^* := B
        B := $\arg\max_{B' \in \rho(B)}$ evaluate(B');
        b := evaluate(B)
    return (B^*)

Figure 12.13: Learning Bayesian networks: Structure search.

Learning the structure

We can learn the structure of a Bayesian network using an iterative process where we start with a simple structure and repeatedly make small modifications to it (refine it, using a fixed refinement operator), modifying it in the direction of a better fitting network, and repeating this process until no modifications further improve the network. This is the standard greedy search procedure.

The refinement operator in this "structure search" is typically an operator that can add an edge between any two nodes, delete an edge, or reverse the direction of an edge.

When can we say that a network is "better" than another network? Generally one could say that it is better if it has a higher likelihood. But more complex networks will always have a higher likelihood on the training data: they have more parameters and can therefore be made to fit the training data more easily. As with other models, we need to either estimate the likelihood of the network on a set of validation data (not used for training), or build in a kind of penalty for the complexity of the network. The latter kind of approach often relies on some version of the MDL principle: a higher likelihood of the data set given the model implies a smaller coding length for the data, which is traded off with the increased coding length of a more complicated model.

The structure search algorithm is summarized in Figure 12.13.

A recent improvement to structure learning was proposed in 2005, when Teyssier and Koller introduced **ordering search**. Instead of searching over the space of all structures, they search greedily over the space of all orderings of variables. For each possible ordering, they construct a Bayesian network by choosing for each variable as parents those variables that precede it and that influence it most strongly. Each variable can have only a limited number of parents. Teyssier and Koller showed that this technique is more efficient than structure search while yielding networks with equally high likelihood.

12.5.10 Bayesian networks and causality

An arrow $A \to B$ in a Bayesian net can be interpreted as indicating that A has a "direct influence" on B. Indeed, A occurs explicitly in the conditions for the conditional probability distribution of B.

It is tempting to interpret this direct influence as a causal influence, to say that a change in A somehow causes B to change. But this is not correct. The "alarm" network shown in Figure 12.9 already illustrates this. Based on our general knowledge about the world, we can guess that a burglary or an earthquake causes the alarm to go off, and that the alarm going off causes the neighbors to call. So it

seems reasonable to say that the left network in Figure 12.9 indeed indicates causal influences among the variables. But the network to the right is an equally correct network, and here, clearly, the connections are not causal; the neighbors do not cause the alarm to go off by calling, and the alarm does not cause a burglary or earthquake.

Consider a very small network with only two variables. Either the two variables are independent, in which case there is no edge in the network, or they are dependent. If they are dependent, we know that

$$p(a, b) = p(a|b)p(b) = p(b|a)p(a).$$

Both factorizations are equally valid, there is no reason to prefer one over the other. But one factorization is expressed graphically as $A \rightarrow B$, whereas the other is expressed as $B \rightarrow A$.

With only two variables, there is no reason to prefer one factorization over the other. But with more variables, it is possible that some factorizations yield simpler networks (e.g., networks with fewer edges) than others. Figure 12.9 illustrates this. In such situations, it is natural to prefer the simplest network; it is easier to understand, and the parameters of such a network tend to be easier to estimate. On the other hand, a causal network (where the arrows do indeed indicate causal influences) is sometimes more intuitive to understand. But the simplest network does not necessarily coincide with the causal network (it does in the alarm case, but in general this is not guaranteed).

The most important conclusion from this is that it is not always obvious how a Bayesian network should be interpreted. Humans tend to have a strong tendency to interpret influences as causal, even when they know they shouldn't. It is wise to always remember that the network is just a representation of certain independencies in the data, that it represents one particular possible factorization of the joint distribution.

12.6 Markov networks

Markov networks are another kind of graphical models. They are somewhat similar to Bayesian networks, but consist of undirected graphs, instead of directed ones. The joint distribution defined by a Markov network is the product of a number of so-called potential functions, which can be seen as distributions over a subset of variables. There is one such potential function for each maximal clique in the network.[3]

For instance, in a network $A - B - C$, there is a potential function over A and B, denoted $\phi_{A,B}$, and one over B and C, $\phi_{B,C}$. The joint probability distribution over A, B, C is then defined as

$$p(a, b, c) = \frac{1}{Z}\phi_{A,B}(a, b)\phi_{B,C}(b, c)$$

where Z is a normalization factor that ensures that the probabilities sum to one.

We can interpret the edges in a Markov network as follows. *There is an edge between two variables A and B if A and B directly influence each other.* We say that A and B directly influence each other if no set of variables exists such that knowing the values of the variables in that set blocks the information flow between A and B. In other words: A and B directly influence each other if they cannot be d-separated.

[3] A set of nodes in a graph forms a clique if each node in the set is directly linked to each other node in the set. It is a maximal clique if there exists no strict superset of it that also forms a clique.

Sometimes the joint distribution of a set of variables can be factorized in such a way that all the factor distributions are functions of a strict subset of the variables in the original distribution. In other cases, this is not possible. It turns out that it is possible to do this if and only if in a Markov net the variables do not form a clique, i.e., if not all variables directly influence each other. In other words, the JPD over a set of variables can always be written as a product of potentials over the cliques in the Markov network.

Proposition 12.1 *The JPD of a set of variables V can be written as a product of two functions of strict subsets of variables in V if and only if the variables do not form a clique in the Markov network.*

Proof: For ease of writing, we prove the implication in only one direction; but all the steps in the proof can be reversed, showing that the implication holds in both directions.

Assume that the JPD over V can be written as a product of two functions of variables in a set X and a set Y: $p_{X,Y}(\mathbf{x}, \mathbf{y}) = \phi_X(\mathbf{x})\phi_Y(\mathbf{y})$, with $V = X \cup Y$, $X \subset V$ and $Y \subset V$.

From $V = X \cup Y$ and $X \subset V$ follows that Y must contain at least one variable not in X; call this variable A. Similarly, since $Y \subset V$, X must contain at least one variable not in Y. Call this variable B, and let $X' = X - \{B\}$ and $V' = V - \{B\}$.

For B it holds that

$$p(b|\mathbf{v}') = \frac{p(\mathbf{v})}{p(\mathbf{v}')} = \frac{1/Z\phi(\mathbf{x})\phi(\mathbf{y})}{\sum_B 1/Z\phi(\mathbf{x})\phi(\mathbf{y})} = \frac{1/Z\phi(\mathbf{x})\phi(\mathbf{y})}{1/Z\phi(\mathbf{y})\sum_B \phi(\mathbf{x})} = \frac{\phi(\mathbf{x})}{\sum_B \phi(\mathbf{x})} = \frac{\phi(\mathbf{x})}{\phi(\mathbf{x}')} = p(b|\mathbf{x}')$$

where the third equality holds because $B \notin Y$, which implies that $\phi(\mathbf{y})$ is constant with respect to B and can be moved out of the summation.

We see that $p(b|\mathbf{v}')$ is only a function of the variables in X, not of those in Y, which implies that B is independent of all variables in $Y - X$ given X'. In particular, X' d-separates B from A, which implies there should be no edge between A and B in the Markov network.

□

Markov networks are somewhat less popular in the machine learning world than Bayesian networks. Compared to the latter they have both advantages and disadvantages. A disadvantage is, for instance, that the potential functions are less easy to interpret than the conditional probability distributions in a Bayesian net.

An advantage of Markov networks, however, is that they are undirected models, and thus there is no risk of incorrectly interpreting arrows as indicating causal influences. While Bayesian networks may seem easier to interpret than Markov networks, there is also a higher risk of interpreting them incorrectly.

Further, the structure of the Markov blanket is easier to specify in a Markov network. The Markov blanket of a variable X simply consists of all the variables that are directly connected to X. Recall that in a Bayesian network, the co-parents of the children of a node are also in the node's Markov blanket.

Markov nets have their own, sometimes quite sophisticated, inference methods (such as Loopy Belief Propagation, Gibbs sampling, ...) We refer to the specialized literature for more details on this.

12.7 Bibliographic notes

Graphical models were introduced and have been treated extensively by Judea Pearl in several books [30, 31]. Neapolitan [28] provides an excellent and up-to-date

textbook on Bayesian networks.

Chapter 13

Ensemble methods

To motivate this chapter, consider the following. Different learning systems, when confronted with some data set, will probably learn different models. It is unlikely that these models are perfect: they will sometimes make mistakes, and different models may disagree on certain predictions. In such cases, how do we know which model is probably right? Is there a way to combine the predictions of the models into one "most likely" prediction?

The situation of having multiple models is not necessarily undesirable. It seems reasonable to assume that when all these models agree on a prediction, it is more likely that that prediction is correct than when about half of them predict one class and about half of them predict the other. Moreover, we might expect that the combined prediction is on average better than the individual predictions, because it makes a mistake only when the majority of the individual models makes a mistake. Thus, having multiple models, learned using different learners, is good because it can give us both *better predictions*, and a *better estimate of how certain* these predictions are.

A set of models all trying to approximate the same function is called an ensemble, and methods learning such ensembles are ensemble learners.

Definition 13.1 *An* **ensemble** *is a set of predictive models f_i, where each f_i predicts the same target variable from the same input variables.*

Definition 13.2 *An* **ensemble learner** *is a learning system that, given a data set, learns a predictive model that is in fact composed of a number of different predictive models (i.e., it is an ensemble), together with a rule for combining the predictions of the component models into one final prediction.*

The component models are learned using standard learning systems, which we call in this context the **base learners**.

In the following we first discuss how the predictions of a given set of models can be combined to obtain more accurate predictions, using some kind of voting process. Next, we will have a look at several ensemble learning methods. These differ not only with respect to how they combine the predictions of given models, but also with respect to how they construct these models.

13.1 Voting

13.1.1 Basic voting

The principle of voting is simple. When given n models f_i, we combine their predictions into one prediction which is a kind of "average". If the models are classifiers,

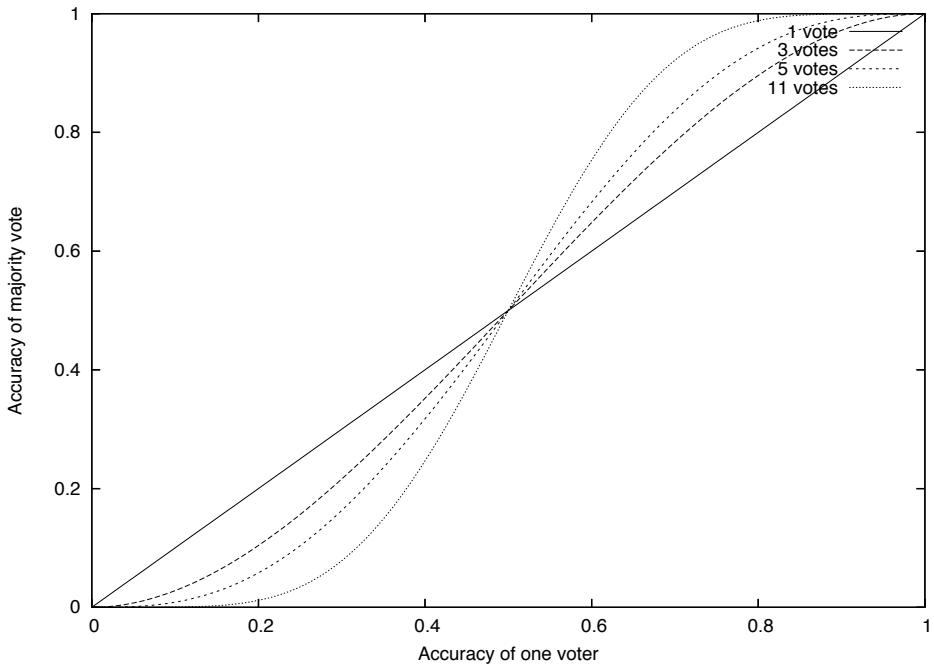

Figure 13.1: The effect of voting on accuracy. The curves show the accuracy of the majority prediction among n predictions each of which has an accuracy of a, as a function of a, for $n = 1, 3, 5, 11$. The curves are constructed under the assumption that the different predictors make mistakes independently from each other.

we take the mode (the class that is predicted most often) as the final prediction of the ensemble. If the models make numerical predictions, we predict the arithmetic mean of the predictions.

To understand the effect of voting, consider the special case where we have n binary classifiers (the two classes are *pos* and *neg*), all with the same error rate e, and assume that their errors are statistically independent; that is, if f_i makes an incorrect prediction, this does not influence the probability that f_j make an incorrect prediction, for $i \neq j$.

Consider now a random example x. Each f_i has a probability e of making an incorrect prediction for x. The ensemble's prediction is the most frequent prediction among the f_i, so the ensemble makes an incorrect prediction if at least half of the f_i make an incorrect prediction. So what is the probability that this happens? This is given by the binomial distribution:

$$Pr(E > k) = \sum_{i=k+1}^{n} Pr(E = i) = \sum_{i=k+1}^{n} C_n^i e^i (1-e)^{n-1}$$

where the C_n^i are binomial coefficients: $C_n^i = n!/(i!(n-i)!)$.

Figure 13.1 shows how this probability varies as a function of e for different n. It is clear that for $e < 0.5$, the probability that the ensemble makes an incorrect prediction is smaller than e. Moreover, this probability will approximate 0 when n goes to infinity.

There is a similar effect if the different classifiers have different errors e_i, as long as all $e_i < 0.5$. The error of the ensemble will then decrease with increasing n. However, it is not necessarily smaller than the smallest e_i. Selecting the best classifier from the ensemble and using just that classifier may be better than taking

a vote from all the classifiers in the ensemble.

For numerical predictors, similar statements can be made. For instance, if each f_i is an unbiased estimator with a squared error of e_i^2, then the ensemble is an unbiased estimator with a squared error of $\frac{\sum_{i=1}^{n} e_i^2}{n^2}$, which may be smaller than the smallest e_i, though this is not guaranteed.

Note that all the above reasoning is under the assumption that the different models make mistakes *independently from each other*. In general, this assumption is almost always violated: generally, some instances will be more difficult than others to predict because of inherent properties of these instances. The actual improvement obtained using voting will therefore in general be less spectacular than what Figure 13.1 suggests.

It is important to see the difference between saying that the predictions are independent, and saying that the errors are independent. The latter is equivalent to conditional independence of the predictions given the target variable.

For instance, in the case of classification, independent *predictions* would imply that, for all classes c and for all models f_i, f_j where $j \neq i$:

$$Pr(f_i(x) = c | f_j(x) = c) = Pr(f_i(x) = c)$$

whereas having independent *errors* implies that

$$Pr(f_i(x) = c | c(x) = c \wedge f_j(x) = c) = Pr(f_i(x) = c | c(x) = c)$$

where $c(x)$ refers to the true class of x.

The predictions themselves are almost certainly not independent. Assume we have two models f_i and f_j, both with an accuracy of 0.9; then when $f_j(x) = c$ it is very likely that the true class is c and therefore that $f_i(x) = c$ as well, so f_j and f_i make strongly correlated predictions. Nevertheless, the errors they make may be independent (though there is still no guarantee that they are).

13.1.2 Weighted voting

The above discussion has already shown that, while voting can improve the accuracy of predictions, it may not be optimal, especially when there are large differences in accuracy between the different models. When one model is much more accurate than the others, averaging its predictions with those of other, much less good, models may actually yield a lower accuracy.

One way to counter this effect is the use of weights. If model A is better than model B, it should get a higher weight in the vote. In the limit, if A is infinitely much better than B (i.e., A has zero error), only A should influence the result.

For numerical predictions, and assuming again independence of errors, the theory of statistics provides us with a way to optimally combine different models: the weights should be inversely proportional to the squared errors. This result carries over to classifiers if we count an incorrect prediction as an error of 1, and a correct prediction as an error of 0.

Unfortunately, errors are not necessarily independent, so the statistical methods may not be optimal in practice, and it is useful to look for other methods.

In the following, we will look at several ensemble learning methods. Many of them use weighted or unweighted voting to combine the predictions of the models in the ensemble.

function bagging(training set T, learner L, number of bags B) **returns** classifier:
    for $i = 1$ **to** B:
        create T_i by drawing N elements randomly with replacement from T
        $h_i := \mathrm{L}(T_i)$
    return $\lambda x : mode(\{h_i(x)\})$.

Figure 13.2: The bagging algorithm.

13.2 Bagging

13.2.1 The algorithm

Bagging is short for "bootstrap aggregating". The basic idea behind it is that different models f_i are constructed by *creating several variants of the original training set*. Each variant, being slightly different from the others, may give rise to a different model f_i. The predictions of all these models will be combined using unweighted voting.

How are the variants of the original training set created? Each variant is the result of drawing N elements from the training set, randomly and with replacement (i.e., the same element can be drawn several times). The result is a new data set of size N, in which some examples from the original training set do not occur (they are called out-of-bag examples), some occur once, and some occur multiple times.

Thus, the whole bagging procedure can be described as in Figure 13.2, where the notation $\lambda x : exp$ is used to represent the function that maps x onto exp.

Bagging was originally defined for classification, but can easily be extended towards regression by changing the *mode* function into *mean*.

13.2.2 When does it work well?

We have already seen that the voting procedure is guaranteed to yield improved performance when the different classifiers have comparable errors that are below 0.5 and make mistakes independently of each other. What happens when the mistakes are not independent? As an extreme case of this, consider the situation where all classifiers are exactly the same. In that case, obviously, the ensemble yields exactly the same predictions as any single model in it, so there is no improvement in performance at all.

In the specific case of bagging, each model is learned by the same learning system L, from similar training sets T_i (resamplings of one training set T). One can expect that in this situation, the models h_i will be similar too. They may even be exactly the same, if the differences in the training sets T_i are too small for the learner L to produce different models.

A learner is said to be *stable* if small changes in the training set do not influence the learned model much. It is said to be *unstable* if small changes in the training set may give rise to large differences in the model learned. Using this terminology, one can say that **bagging works well if the learner L that is being used is an unstable learner**.

Decision tree learners are generally considered unstable, and are therefore good candidates to use as the base learner in a bagging procedure.

13.2.3 Random Forests

A variation on bagging with decision trees is the **Random Forests** procedure proposed by Breiman [4]. Here, the variation among the decision trees in the ensemble is increased by not only learning the trees from resamplings of the data, but by also introducing randomness in the decision tree building procedure. This is done as follows. When an attribute is to be filled in in the node of a decision tree, instead of evaluating all possible attributes and choosing the one with the highest heuristic value, only a randomly chosen subset of all attributes is considered, and the best attribute from this subset is selected. The subset may be different for each node, and it may be as small as 50%, 10%, or even less of the original set of attributes. Breiman suggests to use a subset of size $\sqrt{a}$ with a the number of attributes.

Assume that in the top node of the tree, attribute A yields a clearly higher information gain than any other attribute, so much higher that resampling the data set still leaves A the best attribute. In the standard bagging approach, all trees in the ensemble will then have A as the attribute in the top node. In the Random Forests procedure, however, there is a (high) probability that A is simply not included in the subset that is being considered, and therefore an other attribute is selected instead. This obviously increases the variability among the induced decision trees.

Several other techniques to increase the variability of the learned models have been proposed in the literature, but we will not discuss them in detail here.

13.3 Boosting

Boosting generally refers to the approach of learning multiple weak models (simple models with relatively high error) and trying to combine all these models into one model that works much better than its component models.

In the machine learning context, the term boosting very often refers to one specific kind of approach that is very similar to bagging, and of which the AdaBoost algorithm is probably the best known method. The boosting approach is similar to bagging, in the sense that here, too, different models are constructed by creating variants of the training set and learning a model from those variants using a fixed learner. But it differs from bagging in the following ways:

- The variants of the training set are not resamples of the original set where each elements occurs 0, 1, or multiple times. Instead, each element is assigned a weight, which may be any positive real number. If the weight is an integer k, the effect is the same as having that element k times in the dataset. It is assumed that the learner used in the procedure L can handle training sets where examples have non-integer weights.

- The variants are not constructed randomly, instead the weight of each example is computed according to a specific function. Whether an example gets a high or low weight depends on how accurately it was predicted in previous iterations. "Difficult" examples (examples that are predicted incorrectly in the current iteration) will get a higher weight in the next iteration, "easy" examples get a lower weight.

- The models in the ensemble will in the end be combined using weighted voting. The weights are again computed according to a specific function.

In the first iteration of boosting, all examples have a weight $1/N$. A learner L is used to construct a model h_1. Then, for each example x in the training set, the predictions $h_1(x)$ is compared to its actual class $c(x)$. If $h_1(x) = c(x)$, the weight

function ADAboost(training set T, learner L, number of iterations B) **returns** classifier:
    let $T = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ with $y_i = 1$ or -1
    $w_i := 1/N$, $i = 1..N$
    for $t := 1$ **to** B
        learn h_t from T with weights w_i, using L
        $\epsilon_t := \sum_{h_t(x_i) \neq y_i} w_i$
        if $\epsilon_t > 0.5$ **then** stop
        $\beta_t := \epsilon_t/(1 - \epsilon_t)$
        for $i = 1$ **to** N: **if** $h_t(x_i) = y_i$ **then** $w_i := w_i * \beta_t$
        // the following lines just normalize w_i *such that* $\sum_i w_i = 1$
        $s := \sum_i w_i$
        for all w_i: $w_i := w_i/s$
    return $\lambda x : argmax_c(\sum_{t:h_t=c} log(1/\beta_t)$

Figure 13.3: The ADAboost algorithm.

of x is reduced; otherwise it is increased. Thus we get a new version of T, call this version T_2. In the second iteration, we run L on T_2; because of the changed weights, L is now forced to look for a hypothesis h_2 that compensates for the errors of h_1, i.e., one that predicts correctly those examples that were predicted incorrectly by h_1. After learning h_2, weights are again updated based on the errors made by h_2, given a training set T_3, on which a model h_3 is built, and so on.

In the end, the B hypotheses obtained in this way form together an ensemble. The prediction of this ensemble is a weighted vote among the predictions of the different models.

The exact functions that are used may vary from one boosting implementation to another. A very well-known learning algorithm that performs boosting is ADABOOST, proposed in 1996 by Freund and Schapire. This algorithm is described in Figure 13.3. We do not go into detail about the reasons for computing ϵ and β as indicated in Figure 13.3.

13.4 Stacking

Weighted and unweighted voting are relatively simple ways of combining the predictions of different models. We might ask ourselves: if we are confronted with predictions from multiple models, and we are not sure how to combine them, why don't we *learn* how to combine them optimally, using some machine learning approach?

This is the basic idea behind stacking. Given n models, we create a table with N rows and $n+1$ columns, where each row contains for one example the predictions of the n models and the example's true class. We then use a learner (sometimes called the *top-level learner*, as opposed to the *base learners* that produced the models h_i) to learn a function that maps the n predictions onto the true class. In a sense, the top level learner is stacked on top of the base learners, which explains the name of this method.

Note that stacking is just a way of combining the predictions of a set of models. How those models are obtained is not specified. They could be generated from variants of the training set, as in bagging or boosting; or they could be the outcomes of different learning methods.

A stacking algorithm is shown in Figure 13.4. In this description the "base

function stacking(training set T, n base learners L_i, top-level learner L) **returns** classifier:
    let $T = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ with $y_i = 1$ or -1
    for each base learner L_i, $i = 1 \ldots n$:
        $h_i := L_i(T)$
    let $T = \{(x'_1, y_1), \ldots, (x'_N, y_N)\}$ with $x'_{ij} = h_j(x_i)$, $j = 1, \ldots, n$
    $h := L(T')$
    return h

Figure 13.4: The Stacking procedure.

learners" L_i are assumed to internally create a variant of the training set T if needed.

Some flexibility is possible with respect to the "predictions" that are used by the top level learner. Consider the case where the base learners are binary classifiers that, besides the prediction itself (-1 or 1), indicate the certainty with which they make this prediction (for instance, they output a number from -1 to 1; a number close to -1 or 1 indicates more certainty than a number near 0). This certainty indicator holds extra information that may be useful when combining the predictions. In the same way that an accurate model should get a higher weight than a less accurate one, a prediction with high certainty should be considered more thrustworthy, and therefore given a higher weight, than an uncertain prediction. In 1999 Ting and Witten [40] have shown that learning from such rank predictions, instead of from binary predictions, may yield a better classifier.

Instead of creating a new table that contains only the outputs of the n base learners as attributes, one could also extend the original dataset with these n additional attributes. The top-level learner can then, for instance, use the original attributes to impose conditions under which the predictions of certain base learners should or should not be used. If one base learner performs well in a certain area of the instance space, whereas another one performs well in another area, then using this approach the top-level learner can construct a model that uses the prediction of one learner or the other, depending on the area in which the example lies. The original procedure could not take that kind of information into account.

13.5 Bibliographical notes

The principle of bagging was first proposed by Breiman in 1996. In 2001, the same author presented Random Forests as an improvement of bagging. The ADAboost algorithm is described by Freund and Shapire (1996). The principle of stacking is due to Wolpert (1992); in 1999 Ting and Witten showed that stacking rank classifiers instead of regular classifiers improves the performance of the ensemble.

Chapter 14

Reinforcement learning

Reinforcement learning is a learning task that is quite different from any we have discussed up till now. The general context is that an environment is given in which an agent can perform certain actions, and performing the right action in the right situation may provide some reward. Besides possibly leading to an immediate reward, actions also influence the agent's situation, and thus an action may be good for several reasons: because it leads to an immediate reward, or because it brings the agent closer to a situation in which it can get a reward.

The goal of the agent is to learn a policy that maximizes its rewards in the long run. The setting is semi-supervised in the sense that the rewards provide some information about which actions are good or bad, but there is no complete information. A high immediate reward for some action does not imply that the action was optimal: another action might not give an immediate reward, and yet be better because it leads to a higher reward a bit later. Also, when a series of actions finishes with a high reward, this does not guarantee that all the actions in the series were optimal: some actions may have been unnecessary or even detrimental.

14.1 The Reinforcement Learning Problem

The system that is trying to solve the reinforcement learning problem will be referred to as the *agent*. Such an agent will interact with its *environment*, sometimes also referred to as its *world*. This interaction consists of actions and perceptions as depicted in Figure 14.1. The agent is (through its perceptions) supplied with an indication of the current *state* of the environment and chooses an *action* to execute, to which the environment reacts by presenting an updated state indication. Also available in the perceptions presented to the agent is a *reward*, a numerical value that is given for each taken action (or for each reached world-state). The **policy** of the agent is a function that translates any state into the action the agent will perform in that state. To solve the reinforcement learning problem, the agent will try to find a policy that maximizes the received rewards over time.

The reinforcement learning problem can be formally defined as follows:

Definition 14.1 *A reinforcement learning task is defined as follows:*

Given

- *a set of states* $\mathcal{S}$*,*
- *a set of actions* $\mathcal{A}$*,*
- *a* (possibly unknown) *transition function* $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$*,*

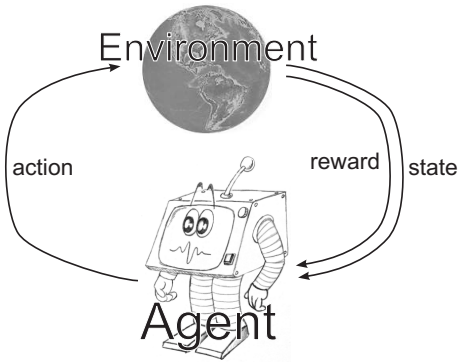

Figure 14.1: The interaction between an agent and its environment in a reinforcement learning problem.

- *a* (possibly unknown) *real-valued reward function* $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$.

Find *a policy* $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ *that maximizes a value function* $V^\pi(s_t)$ *for all states* $s_t \in \mathcal{S}$. *The* utility value $V^\pi(s)$ *is based on the rewards received starting from state* s *and following policy* π.

We consider environments in which the agent interacts with the world in a *turn-based* manner. This means that at each point in time t, the reinforcement learning agent is in state s_t, one of the states of $\mathcal{S}$, and executes an action $a_t = \pi(s_t) \in \mathcal{A}$. It is possible that not all actions can be executed in all world states. In environments where the set of available actions depends on the state of the environment the possible actions in state s will be indicated as $\mathcal{A}(s)$. Executing action a_t in a state s_t will put the agent in a new state $s_{t+1} = \delta(s_t, a_t)$. The agent also receives a reward $r_t = r(s_t, a_t)$.

The target of a reinforcement learning problem is a policy function π. A policy function allows the agent to select an action in each state s, represented by $\pi(s)$. The value $V^\pi(s)$ indicates the *value* or *utility* of state s, often related to the cumulative reward an agent can expect starting in state s and following policy π. The task of reinforcement learning is to find an optimal policy, i.e., a policy that will maximize the chosen value function. The optimal policy is denoted by π^* and the corresponding value function by V^*, i.e.

$$\forall s \in S, \forall \pi : V^{\pi^*}(s) \equiv V^*(s) \geq V^\pi(s).$$

Example 14.1 For an agent learning to play chess, the set of states $\mathcal{S}$ are all the legal chess states that can be reached during play, the set of actions $\mathcal{A}(s)$ are all the legal moves in state s according to the rules of the chess game. The transition function δ includes both the result of the action chosen by the agent and the result of the counter move of its opponent. Unless the agent was playing a very predictable opponent, δ is unknown to the agent in this case.[1] The reward function could be defined easily to present the agent with a reward of 1 when it wins the game, -1 when it looses and 0 for all other actions. The task of the learning agent is then to find a policy which maximizes the received reward and therefore maximizes the number of won games.

[1] In case most common case, where the opponent does not always play the same action in the same game-state, δ is also non-deterministic. More on this in section 14.1.2.

14.1.1 Value Functions

The most commonly used definition of the state utility V^π, and the one that will be used in the rest of this text, is the discounted cumulative future reward:

Definition 14.2 (Discounted Cumulative Reward)

$$V^\pi(s_t) \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{14.1}$$

with $0 \leq \gamma < 1$.

This expression computes the discounted sum of all future rewards that the agent will receive starting from state s_t and executing actions according to policy π. The discount factor γ keeps the cumulative reward finite, but it is also used as a measure that indicates the relative importance of future rewards. Setting $\gamma = 0$ will make the agent try to optimize only immediate rewards, while setting γ close to 1 means that the agent will regard future rewards almost as important as immediate ones. Requiring that $\gamma < 1$ ensures that the state utilities remain finite. As different values of γ will lead to different optimal policies, the value of γ needs to be given as part of the problem specification of a reinforcement learning task.

Example 14.2 When dealing with money as a reward signal, there is an automatic discount factor that should be taken into account under the form of inflation, i.e., the devaluation of currency caused by increasing product prices. Of course, human nature can also add its own preference to this discount factor. Many people will prefer receiving 100 euro today over receiving 110 euro in a year's time, even if the inflation is less than 10%.

The definition of the utility function as given in Equations 14.1 is not the only possibility. A number of other possible definitions are:

Finite Horizon : $V^\pi(s_t) \equiv \sum_{i=t}^{h} r_i$

The finite horizon value function only considers rewards up to a fixed time-step (h). This makes sense in tasks where the agent has a limited life-span or time to act, for example if it wants to maximize its rewards during a single day or in a game with a fixed number of turns.

Receding Horizon : $V^\pi(s_t) \equiv \sum_{i=t}^{t+h} r_i$

In the receding horizon value function, rewards are only considered up to a fixed number of steps (h) *starting from the current time-step*. Although this definition avoids the use of the (often arbitrary) discount factor, it is of limited use because of the danger of the horizon effect, where some actions might postpone negative events beyond the horizon, but at the same time make them inevitable afterwards. This is closely related to the horizon effect in minimax search algorithms.

Average Reward : $V^\pi(s_t) \equiv \rho^\pi(s_t) \equiv \lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$

The average reward value functions consider the long-run average reward, also known as the *gain* ρ. This definition is quite popular because it is better suited for cyclical tasks than the discounted definition above. A key observation is that for environments where any state can be reached from the starting state, the average reward of any policy is state independent, i.e.

$$\forall x, y \in S : \rho^\pi(x) = \rho^\pi(y) = \rho^\pi.$$

In this case, the *bias* or *relative* value of a state $h(s)$ is usually defined as

$$h(s_t) \equiv \lim_{N\to\infty} \sum_{i=0}^{N} (r_i - \rho^\pi)$$

which represents the relative difference in total reward gained by starting in state s_t as opposed to some other state.

14.1.2 Nondeterministic Environments

The environment the agent interacts with is not required to be deterministic. When the execution of an action in a given state does not always result in the same state transition, the transition function δ can be replaced by a transition-probability function.

Definition 14.3 *A* transition-probability *function* $P_\delta : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ *is a function where the value* $P_\delta(s_t, a_t, s_{t+1})$ *indicates the probability that taking action* a_t *in state* s_t *results in state* s_{t+1}*. This implies that*

$$\forall s \in \mathcal{S}, a \in \mathcal{A}(s) : \sum_{s' \in \mathcal{S}} P_\delta(s, a, s') = 1$$

i.e., the probabilities of each possible resulting state, given a state s *and an action* a*, sum to 1.*

The assumption that P_δ is only dependent on the current state s_t and action a_t is called the **Markov property** and an environment which has the Markov property is often referred to as a **Markov Decision Process** or **MDP**. It allows the agent to make decisions based only on the current state without having to take its history, i.e., how it reached that state, into account.

Not only can the state transitions be stochastic, also the reward function can be nondeterministic. It must be noted however, that while determinism is not required by most reinforcement learning algorithms, most convergence results about those algorithms assume the environment is *static*, i.e., the probabilities of state transitions or rewards do not change over time.

To allow nondeterministic policies, a similar probability function can be defined instead of the deterministic policy π.

Definition 14.4 *A* probabilistic policy *is a function* $P_\pi : \mathcal{S} \times \mathcal{A} \to [0,1]$ *where* $P_\pi(s,a)$ *indicates the probability of choosing action* a *in state* s*. It is required that*

$$\forall s \in \mathcal{S} : \sum_{a \in \mathcal{A}(s)} P_\pi(s,a) = 1$$

When dealing with stochastic environments or policies, the definition of the utility function of Equation 14.1 is changed to represent the *expected* value of the discounted sum of future rewards.

$$V^\pi(s_t) \quad \equiv \quad E_{(\delta,r,\pi)} \left(\sum_{i=0}^{\infty} \gamma^i r_{t+i} \right)$$

Example 14.3 The optimal policy of the well-known game of Rock-Paper-Scissors is probabilistic. When playing a perfectly random strategy that gives equal probability to each of the three actions available in the game, one is impossible to predict and will (in the long run) draw against any opponent.

14.2 Learning a Policy for known δ and r

Notice that the reinforcement learning task definition included both the case where the transition function δ (and even the reward function r) were either known or unknown. Although the case where both these functions are known is a lot simpler than when they are not, solving the reinforcement learning task, i.e., finding an optimal policy can be non-trivial.

To be able to find the optimal policy, i.e., the one that maximizes the appropriate value function, one needs to be able to compute that value function for each state. Looking more closely at the definition of the discounted cumulative reward, one can see that:

$$\begin{aligned} V^{\pi}(s_t) &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \\ &= r_t + \sum_{i=0}^{\infty} \gamma^{i+1} r_{t+1+i} \\ &= r_t + \gamma \cdot \sum_{i=0}^{\infty} \gamma^i r_{t+1+i} \end{aligned}$$

And thus :

$$V^{\pi}(s_t) = r_t + \gamma \cdot V^{\pi}(s_{t+1}) \tag{14.2}$$

where $r_t = r(s_t, \pi(s_t))$ and $s_{t+1} = \delta(s_t, \pi(s_t))$.

Equation 14.2 is known as a **Bellman equation**. It expresses the relation between the value of a state and the value of its successor state under the given policy. This equation gives rise to a number of reinforcement learning algorithms. In the following sections, we will discuss a few algorithms that can be used to find the optimal policy when the transition function and reward function are known.

14.2.1 Policy Evaluation

To be able to find the best policy, we have to be able to evaluate the performance of a policy. The performance of any policy π is indicated by the state-values V^{π} as defined above. Policy evaluation computes these values for all states for any given policy. It uses the Bellman equation shown above as an update-rule in an incremental algorithm.

After an arbitrary instantiation of V_0 for all states[2], the policy evaluation algorithm updates the values of each state using the following formula:

$$V^{\pi}_{k+1}(s) = r(s, \pi(s)) + \gamma \cdot V^{\pi}_{k}(\delta(s, \pi(s))). \tag{14.3}$$

Since each iteration of this algorithm computes a new value for each state of the learning problem, each iteration is called a "*full backup*".

When implementing this algorithm, two important issues arise. The first is whether to use one or two arrays to hold the state-values. If one wants to implement Equation 14.3 literally, one would have to use two arrays, one for the old values, and one for the new. However, using only one array and making updates in place, not only saves half of the used memory, but also quickens convergence as the newly computed values are used as soon as they become available. In this case, the order in which the states are updated can have a significant influence on the convergence speed of the algorithm.

The second issue is when to stop updating. In the worst case, the algorithm will only converge in the limit. Typically, policy evaluation algorithms will stop

[2]Except for any terminal states, which must be given the value of 0.

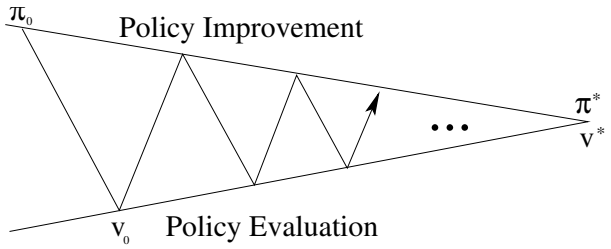

Figure 14.2: Policy iteration interleaves policy evaluation steps with policy improvement steps.

when the maximum difference between two updates during a single full backup is sufficiently small.

14.2.2 Policy Iteration

Once the value of a policy has been computed for each available state, improving that policy (if possible) becomes relatively simple. If a state s can be found for which there exists an action a, different from the one chosen by the current policy, i.e., $a \neq \pi(s)$ for which

$$V^{\pi}(s) < r(s,a) + \gamma \cdot V^{\pi}(\delta(s,a))$$

then the policy π', which is identical to π except for $\pi'(s) = a \neq \pi(s)$, performs better than π.

Thus, by examining all states s and replacing the action chosen by policy π with the action that maximizes $r(s,a) + \gamma \cdot V^{\pi}(\delta(s,a))$, we can make sure that the new policy will be at least as good, and probably better than the old one. This process is called policy improvement.

Initializing the starting policy arbitrarily and then alternating between policy evaluation and policy improvement steps produces a well known reinforcement learning algorithm called **Policy Iteration**. This iterative process is illustrated in Figure 14.2. Policy iteration, although it makes only greedy, one-step lookahead improvements in each iteration, is guaranteed to find the optimal policy and the optimal value function of a finite Markov decision process in a finite number of steps. The iterative process can be halted when no further changes to the policy are made during the policy improvement step. Although each step of policy evaluation is an iterative process in itself, this step is bootstrapped by the value function of the previous policy which often results in a significant reduction of the number of iterations needed to reach convergence. Policy iteration often converges in a surprisingly small number of iterations.

14.2.3 Value Iteration

The policy evaluation step can require multiple iterations through the entire state-space to converge, thereby making each iteration of the policy iteration algorithm unnecessarily slow. As it turns out, it is not necessary to postpone the policy improvement step until the policy evaluation has converged. The two separate steps of the policy iteration algorithm can be combined into a single update step, thereby giving rise to the value iteration algorithm.

Instead of postponing the maximization of the policy improvement step until all state-values have converged, one can alternate single full backups of the policy evaluation step with policy improvement steps, thus alternating between the following

Algorithm 1 The Value-Iteration algorithm.

Input: Transition function δ,
reward function r,
discount factor γ,
approximation measure ϵ
Output: Optimal Value Function V^*

```
Initialize V randomly;
Δ ← ε;
while (Δ ≥ ε) do
    Δ ← 0;
    for all s ∈ S do
        tmp ← V(s);
        V(s) ← max_{a∈A(s)} r(s,a) + γ · V(δ(s,a));
        Δ ← max(Δ, |V(s) − tmp|);
    end for
end while
return V;
```

two sets of updates:

$$\begin{aligned} \forall s \in \mathcal{S} \quad &: \quad V^{\pi}_{k+1}(s) = r(s, \pi(s)) + \gamma \cdot V^{\pi}_{k}(\delta(s, \pi(s))) \\ \forall s \in \mathcal{S} \quad &: \quad \pi(s) = \arg\max_{a \in A(s)} \left[r(s,a) + \gamma \cdot V^{\pi}(\delta(s,a)) \right] \end{aligned}$$

Interleaving these two steps even further and substituting the policy improvement step into the value estimation step produces the following update-rule:

$$V^{\pi}_{k+1}(s) = \max_{a \in \mathcal{A}(s)} \left[r(s,a) + \gamma \cdot V^{\pi}_{k}(\delta(s,a)) \right]. \tag{14.4}$$

Equation 14.4 gives rise to the value iteration algorithm. Value iteration is guaranteed to find the optimal value function V^* in the limit. In practice, once again, one can stop the algorithm when the maximum update during a full backup becomes sufficiently small.

The complete value iteration algorithm is presented as Algorithm 1. The algorithm keeps track of the maximum update during a full backup, denoted Δ, and keeps iterating until Δ is smaller than some threshold ϵ.

Once V^* has been sufficiently approximated, the optimal policy π^* can be derived as follows:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}(s)} \left[r(s,a) + \gamma \cdot V^*(\delta(s,a)) \right]. \tag{14.5}$$

14.3 Learning a Policy for unknown δ and r

If no information about the transition function or reward function is available, it is impossible to find an optimal, or even good, policy. In this case, reinforcement learning algorithms will try to gather information interacting with their environment to discover knowledge about the consequences of actions. During this interaction, the agent will select actions with a non-optimal, exploration policy. When confronted with a learning task where the functions δ and r are unknown, we could try to first learn these functions and then apply one of the algorithms described in the previous section. Interaction with the environment can be seen as sampling examples from δ and r. These examples can be used by supervised learning algorithms to build a

model for δ and r. However, when interested only in learning a good policy, this turns out to be unnecessary.

As seen in Equation 14.5, to translate state-values into a policy, one needs to know both the transition and the reward function. To avoid this, we define state-action values $Q(s, a)$, also known as Q-values, as the part of the equation that needs to be maximized.

Definition 14.5 (Q-value) *The Q-value of a state-action pair is defined as:*

$$Q^\pi(s, a) \equiv r(s, a) + \gamma \cdot V^\pi(\delta(s, a))$$

Comparing this definition with the Bellman equation 14.2, one can see that

$$V^\pi(s) = Q^\pi(s, \pi(s)).$$

Since the optimal policy π^* will choose the action that maximizes state values, we can write the following important relation between the optimal state values V^* and the optimal Q-values Q^*:

$$V^*(s) = \max_{a \in \mathcal{A}(s)} Q^*(s, a).$$

If one can learn these optimal Q-values Q^*, they can be translated (rather trivially) into the optimal policy π^* without requiring knowledge about either δ or r.

$$\pi^*(s) = \arg \max_{a \in \mathcal{A}(s)} Q^*(s, a).$$

The following sections all deal with algorithms that can be used to find these Q-values without having to learn the transition function δ or the reward function r explicitly. We call this **model-free learning**, as no model of the environment in terms of δ or r is constructed.

14.3.1 Monte Carlo Policy Iteration

There exists a model-free version of the policy iteration algorithm discussed before. As before, this algorithm starts with an arbitrary policy. In contrast to before, the algorithm will try to sample the state-action-values or Q-values of this policy through interaction with the environment guided by that policy.

In an episodic environment, i.e., a world where the agent will eventually end up in a terminal state and then be reset for the next trial, we can compute Q-values directly using the sampled transitions and rewards. We will call the sequence of states, actions and rewards that starts in an initial state or starting state s_0 and ends in a terminal state s_N an **episode**. When important, we will indicate the states, actions and rewards encountered at time-step i respectively as s_i, a_i and r_i.

While the ultimate goal of our Monte Carlo policy iteration is Q^*, we will first develop a method to estimate the Q-values of any policy π, i.e., Q^π. We can re-write the definition of Q-values using the definition of state values as follows:

$$Q^\pi(s, a) = r(s, a) + \sum_1^N \gamma^i r_i$$

where $s_1 = \delta(s, a)$, $s_{i+1} = \delta(s_i, \pi(s_i))$ and $r_i = r(s_i, a_i)$. In our unknown environment, $r(s, a)$, r_i and s_i will be the rewards and states the agent encounters during exploration. Thus, $Q^\pi(s, a)$ is equal to the expected (discounted) return starting in state s, executing action a and then following policy π for the remainder of the episode. This value can be easily computed from trial episodes generated by the

agent interacting with its environment and selecting actions based using the policy π.

It is important to note that the first action a does not have to be the action chosen by π in the state s. Thus, to compute Q-values for all possible state-action combinations, the first state and action of an episode should be chosen randomly, guaranteeing that all possible state-action combinations have a non-zero probability of being selected, and then, for the remainder of the episode, the agent should select actions according to its policy π. This is called the approach of "*exploring starts*".

Once the Q-values Q^π are sufficiently approximated, improving policy π can be accomplished by greedily setting

$$\pi(s) = \arg \max_{a \in A(s)} Q^\pi(s, a).$$

Following the same iterative strategy as in policy iteration before, Monte Carlo policy iteration will eventually lead to the optimal policy.

There are two major drawbacks of the described approach. One is the assumption of exploring starts, which may be hard to realize in practice. The second is that during the Monte Carlo policy iteration, computing the correct Q-values for each policy π requires an often unrealistic amount of exploration. In practice, one tends to loosen the approximation requirements for Q^π and alternate between evaluation and improvement on an episode-by-episode basis.

14.3.2 Q-Learning

While the Monte Carlo approach described above tried to estimate the Q-values of each state-action pair individually, it is possible to construct a setup comparable to the value iteration algorithm of section 14.2.3. To accomplish this, we rewrite the Bellman equation 14.2 in terms of Q-values. Using s' as a short notation for $\delta(s, a)$, we can rewrite the Q-value definition as:

$$\begin{aligned} Q^\pi(s, a) &\equiv r(s, a) + \gamma \cdot V^\pi(s') \\ &= r(s, a) + \gamma \cdot Q^\pi(s', \pi(s')) \end{aligned}$$

For the optimal Q-values, this means that:

$$Q^*(s, a) = r(s, a) + \gamma \cdot \max_{a'} Q^*(s', a') \tag{14.6}$$

The Q-learning algorithm in its simplest form, uses the above equation to compute incremental updates of the Q^*-values as it explores the environment. For each transition that occurs by executing action a in state s, Q-learning observes both $r(s, a)$ and $s' = \delta(s, a)$ and updates its estimate of $Q^*(s, a)$ based on these observations and its current estimates of $Q^*(s', a')$ for all actions $a' \in \mathcal{A}(s')$.

The Q-learning algorithm has some very nice properties. It is guaranteed to find the optimal Q-values, and thus the optimal policy, given access to sufficient exploration. In theory, every state-action pair must be visited infinitely often, but in practice the optimal policy is often found after a relatively limited amount of exploration. The convergence speed of the Q-learning algorithm is influenced by the exploration policy that is used, but convergence itself is always guaranteed, whatever the exploration strategy (as long as each state-action pairs is visited often enough). As such, Q-learning is known as an *exploration independent* or **off-policy** learning algorithm. A full description of this simple version of the Q-learning algorithm is given as algorithm 2. Note that this algorithm does not rely on an episodic task as was the case with the Monte Carlo approach above. Episodic tasks can of course also be solved by Q-learning. It requires that the agent is reset by the environment to a randomly chosen starting state after reaching a terminal state and that all Q-values of terminal states are initialized to 0.

Algorithm 2 The Q-learning algorithm for deterministic environments.

Input: discount factor γ
Output: The optimal state-action values Q^*
1: Initialize Q randomly;
2: Initialize s;
3: **while** (**true**) **do**
4: Choose an action $a \in \mathcal{A}(s)$;
5: Take action a, observe r and s';
6: $Q(s,a) \leftarrow r + \gamma \cdot \max_{a'} Q(s',a')$;
7: **end while**
8: **return** Q;

14.3.3 Q-learning in Non-Deterministic Environments

While the above description of the simple Q-learning algorithm works for deterministic environments, stochastic environments force the algorithm to replace the straight update of the Q-values with an update-rule that computes the average of the observed values.

In this case, the update-rule on line 6 of the above algorithm has to be replaced by

$$6: \quad Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s',a') \right].$$

In this equation, the "*step-size*" or "*learning rate*" α identifies the relative importance of the newly observed value compared to the old approximation. α can either be fixed to a sufficiently small value, or be given the value of $\frac{1}{visits(s,a)}$ (with $visits(s,a)$ the number of times state-action-pair (s,a) has been visited), which automatically puts more emphasis on Q-values that have been averaged over a large number of observed samples.

Note that the rule can be rewritten as

$$6: \quad Q(s,a) \leftarrow Q(s,a) + \alpha \left[(r + \gamma \cdot \max_{a'} Q(s',a')) - Q(s,a) \right].$$

which shows that the update term is proportional to the difference between a new estimate for $Q(s,a)$, namely $(r + \gamma \cdot \max_{a'} Q(s',a'))$, and the old estimate $Q(s,a)$. This difference is called a **temporal difference**, and this kind of update-rule is known as a **temporal difference update-rule**.

Given sufficient exploration time, this version of the Q-learning algorithm converges to the optimal Q-values, and thus the optimal policy, for statically nondeterministic (i.e., with fixed, unchanging probabilities) worlds.

14.3.4 SARSA

As will be discussed later, reinforcement learning often happens online, i.e., the agent learns its behavior while performing its task. When dealing with changing environments, this means that we might never want the learning agent to stop exploring its environment. While Q-learning is guaranteed to converge to an optimal policy, because it is an off-policy algorithm, it does not take this exploration factor into account. This ignorance can be harmful in some applications.

Example 14.4 Consider the task shown in Figure 14.3. The task of the agent is to travel from starting state S to the goal state G which will yield the agent a reward. Falling down the cliff will yield a substantial negative reward. An agent following an optimal path between S and G will travel close to the cliff. Consider now the fact that an agent who performs life-long-learning will once in a while

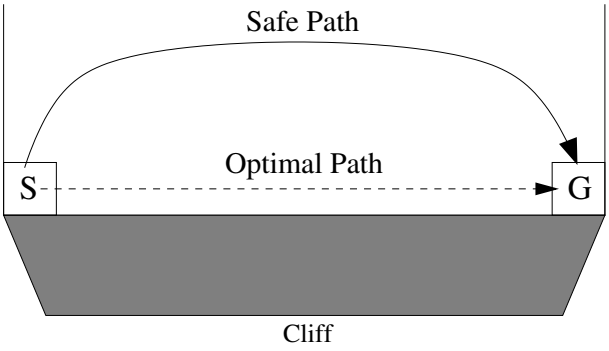

Figure 14.3: Optimal policies with and without exploration.

perform exploration by choosing a random action instead of the optimal one. Even if the probability of choosing a random action is low, one might prefer the agent to take a safer path than the optimal one to avoid the risk of falling down the cliff.

The problem with Q-learning is that it converges to a policy that is optimal *under the assumption that the agent will always be able to follow this optimal policy.* The $\max_{a'} Q(s, a')$ term states that only the best possible next action is relevant. But if there is some possibility that the agent will not be able to choose that best action, then the other actions that might be taken should also be taken into consideration; and this may change our current best action.

Thus, if the policy will be used in a context where a small amount of exploration remains possible, it might be beneficial to use a so-called **on-policy** learning algorithm, which will try to optimize the policy that is used during the learning stage, rather than trying to find a policy that will be optimal once no further learning is necessary.

One such algorithm is Sarsa. The Sarsa algorithm, also a temporal difference algorithm, differs from the Q-learning algorithm only in the update-rule that is used. Instead of the maximization that is employed by Q-learning, Sarsa uses the following update-rule

$$6: \quad Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[r + \gamma \cdot Q(s',a')\right]$$

where a' is the action that was actually chosen by the agent (rather than the best possible action $\arg\max Q(s', a')$) during its interaction with the environment. The name Sarsa stems from the fact that this update-rule uses information from each quintuple $< s, a, r, s', a' >$ it encounters during exploration.

In contrast to Q-learning, Sarsa will not automatically converge to the optimal policy. Its convergence properties depend largely on how the exploration strategy is derived from the computed Q-values. Sarsa converges to the optimal Q-values and policy if the exploration strategy converges in the limit to the greedy policy as indicated by the obtained Q-values.

14.3.5 TD(λ)

Monte Carlo methods and temporal difference algorithms approach the same job, i.e., estimating the correct (and optimal) state-action values, from different ends of a spectrum. Monte Carlo methods estimate Q-values based on complete episodes of experience, i.e.,

$$Q^{\pi}_{MC}(s_t, a_t) = r_t + \gamma r_{t+1} + \cdots + \gamma^{N-t} r_N$$

where N represents the timeframe of the last visited state of the learning episode, $r_t = r(s_t, a_t)$ and the other rewards r_i are received by following policy π starting from state $\delta(s_t, a_t)$. Temporal difference algorithms, such as Sarsa, look only one step into the future and base their Q-value estimates on other Q-value estimates, i.e.,

$$Q^{\pi}_{Sarsa}(s_t, a_t) = Q^{\pi}_{(1)}(s_t, a_t) = r_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1})$$

where $r_t = r(s, a)$ and action $a_{t+1} = \pi(s_{t+1})$.

Between these two extremes, other update scenarios exist such as 2-step lookahead and n-step lookahead which result in the following two update-rules respectively:

$$\begin{aligned} Q^{\pi}_{(2)}(s_t, a_t) &= r_t + \gamma r_{t+1} + \gamma^2 Q^{\pi}(s_{t+2}, a_{t+2}) \\ Q^{\pi}_{(n)}(s_t, a_t) &= r_t + \gamma r_{t+1} + \cdots + \gamma^n Q^{\pi}(s_{t+n}, a_{t+n}) \end{aligned}$$

where the rewards r_i and the action a_{t+n} are generated and chosen by the policy π.

The value used by the n-step lookahead rule is known as the "*corrected n-step truncated return*". This name stems from the fact that the sum of returns in truncated after n steps, but corrected by adding the value of the n^{th} next state. If the episode ends before the agent could take n steps, the returned value becomes the same as the one computed in the Monte Carlo case. This means that the Monte Carlo policy evaluation is just a special case of the temporal difference algorithms.

The estimates of the Q-values as produces by the n-step lookahead rule increases the backward flow of information about already known Q-values, and as such, will often speedup the convergence of the temporal difference algorithm. However, because n-step methods are rather cumbersome to implement, they are seldom used in practice. A more popular algorithm uses an update-rule known as TD(λ), which employs the underlying ideas of the n-step rules in the following way.

Instead of choosing a value n and using that n-step rule to perform backups, the TD(λ) approximation uses a weighted combination of all possible n-step update-rules

$$Q^{\pi}_{\lambda}(s_t, a_t) = (1 - \lambda)\left(Q^{\pi}_{(1)} + \lambda Q^{\pi}_{(2)} + \lambda^2 Q^{\pi}_{(3)} + \cdots\right) \tag{14.7}$$

where the weight-distribution is governed by the parameter λ and $0 \leq \lambda \leq 1$. Because the updates for all n-steps rules where $n > N - t$, with N the total number of steps in the episode, are equal to the Monte Carlo estimate, this can be re-written as:

$$Q^{\pi}_{\lambda}(s_t, a_t) = (1 - \lambda) \sum_{n=1}^{N-t-1} \lambda^{n-1} Q^{\pi}_{(n)} + \lambda^{(N-t-1)} Q^{\pi}_{MC} \tag{14.8}$$

which does no longer include an infinite sum. This formulation also shows what happens when λ becomes 0 or 1. For $\lambda = 1$, the first term disappears and TD(1) turns into Monte Carlo policy evaluation. On the other hand, TD(0) only looks one step into the future and uses the same Q-value estimates as Sarsa.

For other values of λ, the TD(λ) update-rule can be interpreted (and implemented) using the notion of **eligibility traces**. While the update-rule of Equation 14.8 looks at contributions of all future returns to the Q-value of one state-action pair, one could also try to interpret the TD(λ) update-rules looking at the contribution of one return to the Q-values of all encountered states. The weighted updates can be interpreted as a weighted backward propagation of the temporal difference error

$$\Delta_t = r(a_t, s_t) + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t).$$

One way of accomplishing this weighted backward propagation is through the use of eligibility traces. Consider values $e_t(s, a) \in \mathbb{R}$ that represent the contribution for state-action pair (s, a) to any reinforcements received at time t. At each time

Algorithm 3 Sarsa(λ) using eligibility traces.

Input: discount factor γ,
learning rate α,
temporal difference parameter λ

Initialize Q randomly;
Initialize $e = 0$;
Initialize s and a;
while (**true**) **do**
  Take action a, observe r and s';
  Select action a' according to the exploration policy derived from Q ;
  $\Delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$;
  $e(s, a) \leftarrow e(s, a) + 1$;
  for all s,a **do**
    $Q(s, a) \leftarrow Q(s, a) + \alpha \Delta\, e(s, a)$;
    $e(s, a) \leftarrow \gamma \lambda\, e(s, a)$;
  end for
  $s \leftarrow s'$;
  $a \leftarrow a'$;
end while
return Q;

step, the eligibility value for each state-action pair is degraded by $\gamma\theta$ and the value for each *visited* state-action pair is incremented by 1, i.e.:

$$e_t(s, a) = \begin{cases} \gamma\lambda e_{t-1}(s, a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_{t-1}(s, a) & \text{otherwise.} \end{cases} \tag{14.9}$$

If we construct an algorithm that contributes the temporal difference error Δ to each state-action pair proportional to its eligibility value we have constructed a TD(λ)-update version of Sarsa, also known as Sarsa(λ). The explicit derivation that proves the equality of this approach and the above derivation of TD(λ) lies outside the scope of this text. The resulting algorithm is described more formally as Algorithm 3. As before, the convergence of Sarsa(λ) to the optimal policy depends on the convergence of the used exploration policy to the greedy policy as indicated by the Q-values.

Although versions of Q(λ) algorithms have been proposed, they are complicated by the fact that Q-learning is an off-policy algorithm and uses exploratory actions in its policy. The easiest way to deal with these explorations is to reset the eligibility values for all states to 0 whenever an exploratory move is made. The drawback of this approach is that much of the benefits of the eligibility traces are lost.

14.4 Exploration vs. Exploitation

Reinforcement learning is an example of an **on-line learning** setting. On-line learning means that the learning happens in the environment where the model will be applied; collecting the data, learning a model, and applying the model all happen at the same time. This can be contrasted to **off-line learning**, where the procedure is typically that first data are collected (and preprocessed if necessary) into a dataset; then a learning algorithm is run, yielding a model; finally, the model is applied to make predictions for new data. The three phases are clearly distinguished and do not overlap.

In Q-learning, the agent typically starts without any knowledge about which actions are good or bad. It makes sense to perform random actions, collecting data

about the quality of these actions. We can call this the **exploration** phase. At some point, the agent's approximation of the Q-values, represented by $\hat{Q}$, will start giving some indication about which state-action pairs are good and which are not. After a much longer time still, the $\hat{Q}$ function will be close enough to Q to express the correct optimal policy. From that point onward, it makes sense for the agent to choose its actions based on the optimal policy, rather than acting randomly. This is called the **exploitation** phase.

Due to the online learning component of reinforcement learning, one may try to combine the exploration and exploitation phases so that already learned knowledge can be use to guide the agent when learning the rest. An extremely simple approach to the exploration-exploitation tradeoff is ϵ-greedy exploration. ϵ-greedy policies normally choose the action indicated by the current $\hat{Q}$ function, but may, with probability ϵ, pick a random one instead. While this approach guarantees that each state-action pair will be visited eventually, the drawback is that no knowledge about the progress made on the learning task is taken into account. It makes sense to focus on exploration in the beginning of the learning process, and focus more on exploitation near the end.

What is the optimal point for switching from exploration to exploitation? If the switch is made too early, $\hat{Q}$ may not yet have converged to Q and the agent will use a suboptimal policy. If the switch is made too late, learning takes unnecessarily long and a certain amount of opportunities for collecting rewards is not used. On top of that, the agent might be operation in a non-stationary environment and might want to keep exploring occasionally during its entire lifetime, thereby accomplishing life long learning.

In fact, the switch from exploration to exploitation need not be made at one particular point in time; it can be made gradually. The agent can gradually move from randomly selecting actions according to uniform distribution (each action is equally likely to be chosen) towards selecting with a higher probability those actions with a high Q-value. One way to do this is using the following formula that expresses the probability that a certain action is chosen:

$$Pr(a|s) = \frac{k^{\hat{Q}(s,a)}}{\sum_{a'} k^{\hat{Q}(s,a')}}$$

When $k = 1$, each action has an equal probability of being chosen. When k increases, the formula $\sum_{x \in S} k^x$ has the property that it gets closer and closer to $k^{\max_{x \in S} x}$ (i.e., the maximal value in S will dominate the sum). Applied to the above formula, this means that the enumerator $k^{\hat{Q}(s,a)}$ becomes approximately equal to the denominator when a is the action with maximal $\hat{Q}(s, a)$, which gives a $Pr(a|s)$ close to 1. Conversely, any other action gets a probability close to 0. Thus, by starting with $k = 1$ and gradually increasing k, the agent gradually moves towards a setting where mostly the actions with maximal $\hat{Q}$-value are chosen.

More elaborate exploration strategies also exist. One that performs very well in practice is **interval based exploration**, where for each $\hat{Q}$-value the agent also stores a confidence interval for that value. By choosing actions greedily according to the upper bound of the confidence interval, the agent makes a well-founded trade-off between actions with a known high performance and actions with the potential of high performance. Of course, the disadvantage of this technique is the significant computational overhead it introduces.

14.5 Abstraction and Generalization

Up till now, we have assumed that there are finite sets of states $\mathcal{S}$ and actions $\mathcal{A}$, and each state and action is identified by name. That is, when we have a state

called s_1, we know nothing about that state besides the fact that it is state s_1; and similarly for actions. In this context, we have tried to learn the Q-values in the form of a table $Q[s, a]$ that expresses for each state s and action a what the corresponding $Q(s, a)$ is.

This approach has a number of shortcomings. First, note that the number of state-action pairs can be very large in some situations.

Example 14.5 In tic-tac-toe, a very simple game, there are 3^9 possible states (9 squares with a nought, cross, or nothing in them), and 18 actions (put a nought or a cross in any of the 9 squares). (This is an upper bound: not all states can be encountered, and not all actions are possible in any state.) This gives a total of $3^9 \times 18 = 354294$ state-action pairs.

Memory-wise the tic-tac-toe example is OK: a table of this size can be stored in internal memory in today's computers. But considering that each state-action pair has to be encountered a reasonable number of times in the experiments before its $\hat{Q}$ value converges to the real Q-values, it is clear that learning the optimal policy for even such a simple game will require many training episodes.

If we consider chess, defining a state as a configuration of the pieces on the board, and an action as making a valid move, it is clear that Q-learning in its basic form is practically impossible: the number of state-action pairs is much too high.

It is possible to make the Q-learning more efficient, though, by not trivially defining states as board configurations, but defining **abstract states** instead. Abstract states correspond to multiple board states.

Example 14.6 Consider again the tic-tac-toe game. It is clear that there are certain symmetries in the game. If we rotate the board 90, 180 or 270 degrees, then the situation does not essentially change. If for a board with one cross in the upper left corner, and otherwise empty, the optimal move would be to put a cross in the upper middle square, then according to the rotation symmetry, for a board with a cross in the upper right corner the optimal move must be putting a cross in the middle right square.

Besides the rotational symmetry, there is also left-right symmetry, top-bottom symmetry, and mirror symmetry around the two diagonals. Taking all these symmetries into account, the number of possible configurations is greatly reduced.

The key to efficiency in many practical cases is the observation that multiple different states are sufficiently similar to have the same outcome for the same actions, or for corresponding actions. More precisely, we want to identify mappings $(s, a) \to (s', a')$ such that $\delta(s, a) = \delta(s', a')$ and $r(s, a) = r(s', a')$. Two states s and s' are equivalent, denoted $s \sim s'$, if for each action $a \in A$, there is an action a' such that $\delta(s, a) \sim \delta(s', a')$ and $r(s, a) = r(s', a')$, and vice versa. Formally:

$$\begin{aligned} s \sim s' \quad &\Leftrightarrow \quad \forall a \in A, \exists a' \in A : \delta(s, a) \sim \delta(s', a') \wedge r(s, a) = r(s', a') \\ &\wedge \quad \forall a' \in A, \exists a \in A : \delta(s, a) \sim \delta(s', a') \wedge r(s, a) = r(s', a') \end{aligned}$$

Two actions are equivalent in a given state if they lead to equivalent states and yield the same immediate reward:

$$a \sim a' \Leftrightarrow \delta(s, a) \sim \delta(s, a') \wedge r(s, a) = r(s, a')$$

They are fully equivalent if they are equivalent in all states.

For each equivalence class of states, only one abstract state needs to be included in the representation used by the reinforcement learning system, and for multiple actions that are fully equivalent, it is sufficient to keep only one. For multiple

actions that are not fully equivalent but are equivalent in some states, we can in principle adapt the Q-learning algorithm so that when a and a' are equivalent in s, action a' is simply treated as if a were performed.

In some cases, human intuition helps us detect equivalence of states and actions. We can then design the abstract states and actions so that the state-action space is maximally reduced.

When our own intuition is insufficient, or reducing the state-action space is simply too complicated, there is an alternative option: we can use a machine learning system to detect equivalences. This can in principle be done by looking for patterns of the above form.

A simpler way to do it is to use machine learning to learn a $\hat{Q}$ estimator function from examples. In this case, we assume that states are not just identified by name, but have a description.

Example 14.7 In tic-tac-toe, instead of referring to the states as s_1 to s_{354294}, we can represent each state using nine attributes that represent the different squares; e.g., $[x, o, -, -, x, -, -, -, -]$ would mean that the top row contains $[x, o, -]$, the second row $[-, x, -]$, and the third row is empty. Similarly, we could represent actions as couples (position,symbol), e.g., $[3, x]$ means that an x is put on square 3 (upper right corner). A state-action pair can then be represented by concatenating both: $[x, o, -, -, x, -, -, -, -, 3, x]$ means that in the state described above, an x is put in the upper right corner.

Using such a representation, we could do the following. First, run standard Q-learning for a number of times, collecting $\hat{Q}$ approximations for many (but probably not all) states. Once many such approximations have been obtained, output all of them into a data-set listing the description of each state-action pair together with its value $\hat{Q}$. Now run a machine learning system that builds a regression model that tries to predict $\hat{Q}$ from the observed description. From that point onwards, whenever $\hat{Q}(s, a)$ is updated using $\hat{Q}(s', a')$ values, use that regression model to estimate $\hat{Q}(s', a')$. After enough updates have been made, use the new $\hat{Q}$ values to learn a new regression model, and start using that in the $\hat{Q}$ updates. Continue interleaving Q-learning and regression in this way until the Q values stabilize. An episodic algorithm implementing this approach is given by Algorithm 4.

14.6 Direct Policy Learning

The ultimate goal of reinforcement learning is to learn an optimal policy, a function π^* that tells us given a state s what the optimal action a is. While Q-learning as well as the other discussed techniques can achieve this goal, they achieves in fact more than necessary: the Q-values tells us exactly what the total future reward is for each action, instead of simply giving us the action for which it is highest.

Especially when combining Q-learning with generalization, we can easily find situations where Q-learning is unnecessarily difficult.

Example 14.8 Assume a learning problem where states are described using one coordinate x with $1 \leq x \leq 10$; a reward of 1 is given for arriving in the final state 5; and actions are L, moving left (decreasing x with one) or R, moving right (increasing x with one).

An optimal policy is expressed by the following rule: "if $x < 5$ then R else if $x > 5$ then L." However, the actual Q-function is more difficult to learn. The correct Q-function is:

$Q(x, R) = \gamma^{4-x}$ if $x < 5$ and γ^{x-5} if $x > 5$.
$Q(x, L) = \gamma^{x-6}$ if $x < 5$ and γ^{5-x} if $x > 5$.

Algorithm 4 Q-learning with function approximation.

Input: discount factor γ,
an incremental regression algorithm Ψ
Output: A generalized model $\hat{Q}$ of the optimal Q-function

Initialize the Q-function hypothesis $\hat{Q}_0$;
$e \leftarrow 0$;
while (**true**) **do**
  $\mathcal{E} \leftarrow \emptyset$;
  Generate a starting state s_0;
  $i \leftarrow 0$;
  while (s_i is not terminal) **do**
    Choose an action $a_i \in A(s_i)$;
    Take action a_i, observe r_i and s_{i+1};
    $i \leftarrow i + 1$;
  end while
  for ($j = 0$ to $i - 1$) **do**
    Generate example $x = (s_j, a_j, \hat{q}_j)$ where
      $\hat{q}_j \leftarrow (1 - \alpha)\hat{Q}_e(s_j, a_j) + \alpha \left[r_j + \gamma max_a \hat{Q}_e(s_{j+1}, a) \right]$;
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{x\}$;
  end for
  Update $\hat{Q}_e$ using $\mathcal{E}$ and Ψ to produce $\hat{Q}_{e+1}$;
  $e \leftarrow e + 1$;
end while
return $\hat{Q}_e$;

Clearly the π^* function is a lot simpler than the Q function, in this case.

Unfortunately, learning the policy π^* directly is not trivial. The relationship between $Q(s, a)$ and $Q(s', a')$ is relatively clear, and Q-values for successor states carry information about the Q-value for a given state. Simply knowing the optimal action for the successor states, however, does not help with finding the optimal action for the current state. Therefore, it is hard to avoid the use of state- or state-action-values all together. It is possible however, to generate a *locally* optimal policy without the use of an explicit representation of the Q-function, or the need to compute Q-values for all possible state-action pairs.

Policy Gradient

Policy gradient algorithms find a locally optimal policy starting from an arbitrary initial policy and using a gradient ascent search through an explicit policy space. The policy gradients are computed using a function that expresses the value of a policy in an environment, known as the ρ function. It is defined (for probabilistic worlds and policies) as follows:

$$\rho(P_\pi) \equiv \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}(s)} P_\pi(s, a) \cdot Q^\pi(s, a) \tag{14.10}$$

where $d^\pi(s)$ represents the (possibly discounted) probability of being in state s when following policy P_π, also known as the stationary state-distribution.

Assume we only want to investigate policies that can be represented as a function of the state s, the action a and a vector of parameters θ. This creates an explicit policy space with a dimension equal to the size of the parameter vector θ. Also

assume that this policy function is differentiable with respect to its parameter, i.e., that $\frac{\partial P_\pi}{\partial \theta}$ exists.

A property of the ρ function for both average reward reinforcement learning and episodic tasks with a fixed starting state s_0 is that:

$$\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial P_\pi(s,a)}{\partial \theta} \cdot Q^\pi(s,a). \tag{14.11}$$

The proof of this property lies outside the scope of this text. Important to note is that this gradient does not include the term $\frac{\partial d^\pi(s)}{\partial \theta}$. Equation 14.11 allows the computation of an approximate policy gradient through exploration. By sampling states s through exploration of the environment following policy P_π, the value of $d^\pi(s)$ is automatically represented in the generated sample of encountered states. The sum $\sum_a \frac{\partial P_\pi(s,a)}{\partial \theta} Q^\pi(s,a)$ then becomes an unbiased estimate of $\frac{\partial \rho}{\partial \theta}$ and can be used in a gradient ascent algorithm.

Of course, the value $Q^\pi(s,a)$ is unknown, but can be easily estimated for each visited state in the same way as used for Monte Carlo policy evaluation described above. This alleviates the need for an explicit representation (either as table or as a learned function) of the Q-values for all states.

14.7 Bibliographic notes

Q-learning was introduced by Watkins [43, 42], and has, together with its variants, become one of the most popular methods for reinforcement learning. But reinforcement learning is much broader than Q-learning. An excellent overview of the field, on which much of this chapter is based, is offered by Sutton and Barto [38].

Chapter 15

Inductive logic programming

15.1 Introduction

Machine learning and data mining rely on inductive inference: reasoning from the specific to the general. Concretely, this usually means: inducing general laws from specific observations (properties of individual objects, or variable measurements); but it could be broadened to *inducing general knowledge from more specific knowledge*, whatever the form of that knowledge is.

To study inductive reasoning in its broadest form, we first need to consider knowledge representation in its broadest form. Indeed, only when we have a concrete language to express certain kinds of knowledge, and when the meaning of statements in that language is perfectly clear, can we hope to obtain formal methods for inductive reasoning.

Many of the most expressive representation formalisms that are still practically feasible to use, are based on first order predicate logic (FOPL). Concrete knowledge representation languages based on FOPL have been proposed, with well studied semantics and inference methods. For this reason, FOPL is a good starting point for studying inductive inference in its broadest form.

FOPL has roughly the same expressiveness as natural language, due to the fact that it has mechanisms to refer to objects in the world and to properties of (or relationships between) those objects. For instance, a sentence such as "Bart teaches the course 'Introduction to Relational Databases' but he also teaches a course on mathematics in Hasselt" can be written down in first order predicate logic, assuming the right vocabulary is available (i.e., we have words to refer to Bart, to the town of Hasselt, to the concept of teaching, etc.). The structure of the formula that represents this sentence is such that it allows the same kind of reasoning that humans can perform when given this sentence.

15.2 Preliminaries

It is assumed that the student has been familiarized with logic and logic programming in other courses. In this section we briefly recapitulate the main concepts and terminology.

In general, in first order predicate logic, we can make statements about objects in some given universe. It is important to clearly distinguish *the things that we talk about* (represented by terms), and *what we say about them* (predicates and formulas). We discuss each of them in the following subsections.

15.2.1 Terms

In first order predicate logic, we can make statements about objects in some universe. A **term** refers to such an object. We distinguish different kinds of terms: constants, variables, and compound terms. A **constant** always refers to a fixed object in the universe. For instance, the term `5` is normally used to refer to the integer 5; the term `santa` might refer to the person Santa Claus and the term `lapland` to the geographical area of Lapland.

Following conventions from the logic programming language Prolog, we will write non-numerical constants as strings starting with a lowercase letter. Numerical constants are written in the usual way (`5`, `3.14`, ...).

A **variable** refers to some specific object in the universe, but we do not specify which object it is. Again by convention, we will use strings starting with an uppercase letter to write variable names. For instance, the term `X` is a variable, the term `Country` is another variable.

A **compound term** is a term of the form F`(`A_1`, `A_2`, ..., `A_n`)` where F is a so-called **functor** and the A_i, which are called the functor's **arguments**, are terms. For instance, `grade(bob, math101)` is a term. A functor is written in lowercase letters. The number of arguments that a specific functor takes, is called the functor's **arity**. An n-ary functor is a functor with arity n. (The words unary, binary, and ternary are normally used for 1-ary, 2-ary, and 3-ary.) A functor corresponds to a function (in the mathematical sense) in the universe. If functor F refers to the function f and its arguments A_i refer to objects a_i, then F`(`A_1`, `A_2`, ..., `A_n`)` refers to the object $f(a_1, a_2, \ldots, a_n)$.

Example 15.1 If `bob` refers to student Bob Cools, `math101` to the course "Mathematics 101" and `grade` to the grade some student received for some course, then `grade(bob, math101)` refers to Bob Cools' grade on Mathematics 101. Similary, if `kim` refers to tennis player Kim Clijsters, and `father` to the father function, then `father(kim)` refers to Kim Clijster's father, Lei Clijsters. Our vocabulary might already include a constant `lei` referring to this person; in that case we have two different ways to refer to the same person, namely `lei` and `father(kim)`.

A **ground term** is a term that does not contain any variables (and is not a variable itself). Given a vocabulary of constants and functors, and a universe with objects, the function mapping all ground terms onto the objects is called a **pre-interpretation**.

Example 15.2 For the above example, Figure 15.1 shows a pre-interpretation connecting `lei` to Lei Clijsters, `kim` to Kim Clijsters, and `father(kim)` to Lei Clijsters.

15.2.2 Predicates

A **predicate** expresses a property that an object may or may not have, or a relationship between several objects that may or may not hold. (Note that a property is simply a special case of a relationship, namely a unary relationship.) A predicate is mathematically the same as a set or a relation.

Example 15.3 The predicate *Male* can be seen as the set of all males, or as the property of being male. These are equivalent: an individual has the property *Male* if it is a member of the set *Male*. Similarly, the predicate *Lives* can be seen as the set of all couples (x, y) where x is an individual and y is where he/she lives, or as the relationship interpreted as "living somewhere".

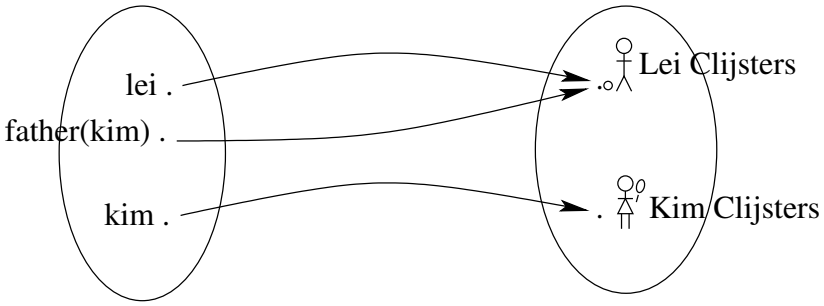

Figure 15.1: A pre-interpretation for the constants `lei` and `kim` and the functor `father`.

A predicate is referred to in our vocabulary using a name, usually called the **predicate symbol** (by convention written as a string starting with a lowercase letter), and an arity, the number of arguments it takes. We write it as P/n with P the predicate symbol and n the arity. We say that P/n is an n-ary predicate. For instance, `passed/1` might refer to the unary predicate *Passed*, which expresses if somebody passed their exams; `lives/2` might refer to the binary predicate *Lives* that states for a person and a geographical area whether the person lives there or not.

An **atom** is of the form P`(`A_1`, ..., `A_n`)` where P/n is an n-ary predicate and the A_i are terms (the atom's **arguments**). For instance, `passed(bob)` is an atom; `lives(santa,lapland)` is another atom.

Note that atoms and compound terms are different things, even though they share some syntax and terminology (arguments, arity). The difference is important at several points in this text. We will return to this in a moment.

Predicates expresses properties or relationships that may or may not hold for certain objects. We thus associate a truth value to atoms: if an atom's predicate holds for its arguments, then the atom is assigned the value *true*, otherwise it is assigned the value *false*. Thus, if Bob passed, then `passed(bob)` is true, and assuming Santa lives in Lapland and not in Belgium, `lives(santa,lapland)` is true while `lives(santa, belgium)` is false.

Given a pre-interpretation and a vocabulary of predicates, we call the pre-interpretation together with a complete truth value assignment (i.e., an assignment of truth values to all the ground atoms that can be constructed using the predicates) an **interpretation**.

Example 15.4 Assume our vocabulary contains constants `belgium`, `santa`, and `lapland`, and predicate symbol `lives`. Then a possible interpretation is the function that maps `belgium` onto Belgium, `lapland` onto Lapland, `santa` onto Santa, and `lives` onto the set $\{(Santa, Lapland)\}$ (which implies that `lives(santa,lapland)` is given the value *true* and all other `lives(`x`,`y`)` atoms are given the value *false*). This interpretation is shown in Figure 15.2.

Note that this is not the only interpretation. Nothing prevents us from interpreting `lives` as the set $\{(Santa, Belgium)\}$, or even interpreting `santa` as *Belgium*. This would obviously not be the most natural interpretation (according to humans) of the terms, but it is a valid interpretation nevertheless.

One particular interpretation that is of some importance is the **Herbrand interpretation**. In the Herbrand interpretation, the "meaning" of a symbol is simply the symbol itself; that is, a Herbrand interpretation maps each term onto itself. The meaning of `kim` is the symbol `kim`, and the meaning of `father(kim)` is the term

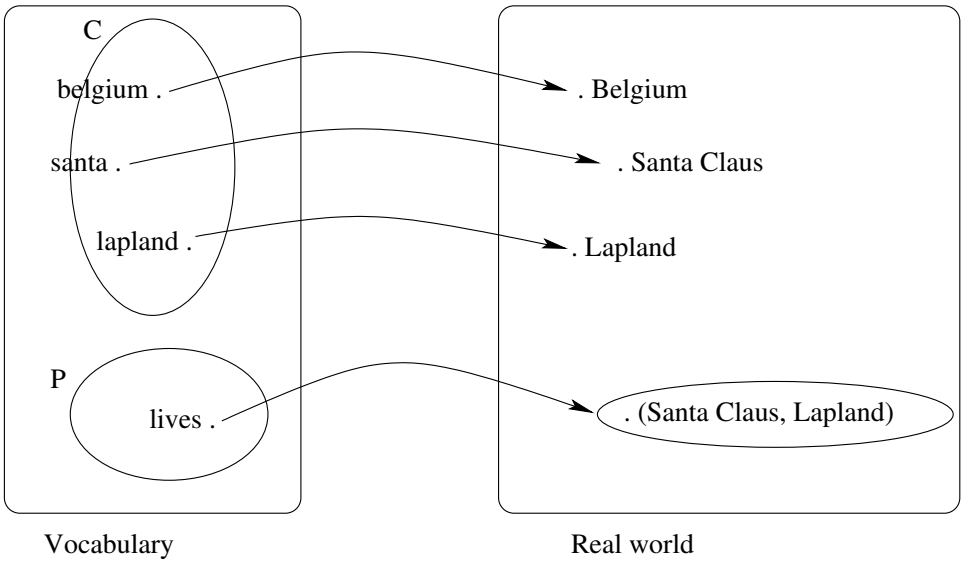

Figure 15.2: An interpretation of the constants `belgium`, `santa`, `lapland` and predicate symbol `lives`. Note that `lives` is mapped to the *set* containing the couple (Santa Claus, Lapland), not to the couple itself.

`father(kim)`. The universe that we make statements about is then the set of all terms that can be constructed using some vocabulary; this set is called the `Herbrand universe`.

Note that there is an important difference between an atom and a compound term, even though their notations are very similar. An atom such as `lives(santa, lapland)` can always be given a truth value. It makes sense to say that it is true (or not true) that Santa lives in Lapland. A compound term such as `father(kim)` refers to an object in the universe. It does not make sense to say that `father(kim)` is true or false. Generally, a (compound) term is related to *what we talk about*; an atom is related to *what we say about it*.

15.2.3 Formulas

A well-formed formula in first order predicate logic is built from atoms, logical operators, and variable quantifiers.

There are three basic operators: the unary operator $\neg$, which expresses negation ("not"); the binary operator $\vee$, which expresses disjunction ("or"); and the binary operator $\wedge$, which expresses conjunction ("and"). In addition, we sometimes use the operator $\rightarrow$ (or $\leftarrow$), which expresses implication; $a \rightarrow b$ is by definition equivalent to $\neg a \vee b$, and $a \leftarrow b$ is by definition equivalent to $b \rightarrow a$. Finally, $a \leftrightarrow b$ is defined as $(a \rightarrow b) \wedge (b \rightarrow a)$.

There are two variable quantifiers: the universal quantifier $\forall$ ("for all ...") and the existential quantifier $\exists$ ("there exists ..."). A variable is *free* in a formula if it is not quantified, and is *bound* otherwise.

A **literal** is of the form A or $\neg A$ with A an atom. A literal that is an atom is called a **positive literal**; a literal that is a negated atom (i.e., is of the form $\neg A$) is called a **negative literal**.

Example 15.5 The formula $lives(santa, lapland) \wedge \neg lives(santa, belgium)$ expresses that Santa lives in Lapland and Santa does not live in Belgium. The formula

$lives(santa, lapland) \vee lives(santa, belgium)$ expresses that Santa lives in Lapland or in Belgium.

The formula $\forall X \exists Y : lives(X, Y)$ expresses that for each X there is a Y such that $lives(X, Y)$ is true. This could be interpreted as "everyone lives somewhere", except for the fact that in this formula X is not restricted in any way; X could be Lapland, for instance, so this formula states that also Lapland lives somewhere. If there is a predicate *person* indicating whether some term refers to a person, then we could write $\forall X : person(X) \rightarrow (\exists Y : lives(X, Y))$, which states that for all X, if X is a person, then that person lives somewhere.

A **clause** is a set of literals. It is interpreted as a disjunction of literals in which all the variables are universally quantified. A **Horn clause** is a clause with at most one positive literal. A **definite Horn clause** is a clause with exactly one positive literal. A **fact** is a clause consisting of a single positive literal.

A **ground** atom (literal, clause, fact) is an atom (literal, clause, fact) that contains no variables.

Clauses will sometimes be written as a set, but at other times we will write the corresponding disjunction, or the implication that is equivalent to that disjunction. As variables in a clause are always universally quantified, the quantification is often not written explicitly.

Note that the disjunction $a \vee b \vee \neg c \vee \neg d$ is equivalent to $(a \vee b) \vee \neg(c \wedge d)$ (due to De Morgan's laws) which is, by definition of implication, equivalent to $a \vee b \leftarrow c \wedge d$. That is: when writing a clause as an implication, the body is to be interpreted as a conjunction of conditions, and the head as a disjunction of conclusions. Since this is always the case in clausal form, we often write the connectives as commas.

Example 15.6 The clause $\{p(X), q(X), \neg r(X, Y)\}$ reflects the well-formed formula $\forall X, Y : p(X) \vee q(X) \vee \neg r(X, Y)$, which can also be written as $p(X) \vee q(X) \leftarrow r(X, Y)$ (by definition of the $\leftarrow$ symbol).

$\{p(X), \neg q(X), \neg r(X, Y)\}$ is a definite Horn clause, as it contains exactly one positive literal. We can write the Horn clause in implication format as $\forall X, Y : p(X) \leftarrow q(X) \wedge r(X, Y)$, or, dropping the variable quantification and writing the conjunction as a comma: $p(X) \leftarrow q(X), r(X, Y)$.

Given an interpretation (an assignment of truth values to the atoms), a formula is assigned a truth value according to the usual meaning of the operators: if ϕ and ψ are well-formed formulas, then $\phi \wedge \psi$ is true if and only if both ϕ and ψ are true; $\phi \vee \psi$ is true if and only if at least one of ϕ and ψ is true; $\neg\phi$ is true if and only if ϕ is false; $\forall X : \phi$, with X a free variable in ϕ, is true if and only if ϕ is always true, whatever object we fill in for X; $\exists X : \phi$, with X a free variable in ϕ, is true if and only if there exists at least one object for which ϕ is true.

A **model** of a formula is an interpretation for which the formula is true. When an interpretation M is a model for a formula ϕ, we write $M \models \phi$. Given two formulas ϕ and ψ, we write $\phi \models \psi$ to denote that each model of ϕ is also a model of ψ. This can be interpreted as "whenever ϕ is true, ψ must be true", or in other words: "ϕ entails ψ".

Example 15.7 For the interpretation shown in Figure 15.2, the formulas

$$lives(santa, lapland) \wedge \neg lives(santa, belgium)$$

and

$$lives(santa, lapland) \vee lives(santa, belgium)$$

are both true, hence the interpretation is a model for those formulas. The formula

$$\forall X \exists Y : lives(X, Y)$$

is false: for instance, for $X = lapland$, there is no Y such that $lives(X, Y)$ is true.

The formula $\forall X : lives(X, lapland)$ entails the formula $lives(santa, lapland)$: there does not exist any model for the first formula that would not be a model for the second. (In other words, whenever the first formula holds, the second formula must hold too.) Thus we have:

$$\forall X : lives(X, lapland) \quad \models \quad lives(santa, lapland).$$

15.2.4 Substitutions

A substitution θ is an assignment of terms to variables. It is written as $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ with the V_i variables and the t_i any kind of terms.

The application of a substition θ to a formula F is denoted $F\theta$. The result of the application is a formula F' obtained by replacing each occurrence of V_i in F with t_i, and this for all V_i/t_i in θ.

Example 15.8 With $\theta = \{X/father(kim), Y/lapland, Z/U\}$, we have:

$lives(X, belgium)\,\theta \quad = \quad lives(father(kim), belgium)$;
$lives(Z, Y)\,\theta \quad = \quad lives(U, lapland)$.

15.3 Inductive Logic Programming

Inductive logic programming refers to a learning paradigm where the input knowledge is represented in the form of first order logic structures, and so is the output knowledge (the hypotheses returned by the learner).

15.3.1 Input knowledge

Attribute value data

Up till now, in this course, we have looked at learning problems where the input data is a set of examples, and each example is described in attribute-value format. That is, each example is described by assigning a value to each attribute from a fixed, predefined, set of attributes. The same attributes are used for each instance. Thus, each element $\mathbf{x}_i$ of a training set T is a tuple $(x_{i1}, \ldots, x_{iD})$ where the x_{ij} are the (nominal or numerical) values given to the D attributes.

In first order logic, the most natural representation of such a dataset T is a single D-ary predicate `t` that assigns the value true to a tuple $(x_{i1}, \ldots, x_{iD})$ if and only if $(x_{i1}, \ldots, x_{iD}) \in T$. (I.e., the predicate `t`, when viewed as a set, is identical to the training set T.)

Example 15.9 The training set $T = \{(a, 1, true), (b, 2, false)\}$ can be represented as a set of logical facts, defining the predicate `t`, as follows:

```
t(a,1,true).
t(b,2,false).
```

Alternatively, we could use D binary predicates a_j, one for each attribute, where $a_j(i, v)$ is true if $x_{ij} = v$. We then need to introduce an explicit identifier i for each example (which could be simply a number).

Example 15.10 The training set $T = \{(a, 1, true), (b, 2, false)\}$ is in this representation represented as

```
a1(ex1,a).
a2(ex1,1).
a3(ex1,true).

a1(ex2,b).
a2(ex2,2).
a3(ex2,false).
```

The first argument of the a_i predicates (`ex1`, `ex2`) is necessary to indicate which individual we are talking about. It is simply an identifier and does not carry any information beyond that. The identifier did not occur in our original description, and we can use any value for it. For instance,

```
a1(x,a).
a2(x,1).
a3(x,true).

a1(y,b).
a2(y,2).
a3(y,false).
```

represents exactly the same training set T as the previous set of facts, we have simply used different identifiers for the instances. In practice we will often use numbers (e.g., `1` for `x`, and `2` for `y`).

The first representation (a single D-ary predicate) is the most natural one, and is used most frequently. The second representation (D binary predicates) is more complex but has the advantage that, for instance, missing values can easily be represented. When the value of a_2 is not known, we can choose to simply not include it in the input data. In the first representation, some special symbol (e.g., `'?'`) with a somewhat special semantics would have to be introduced for this purpose.

While it is very easy to represent attribute-value data in this logical format, it is also possible to represent much more complicated input knowledge, knowledge that cannot be represented in the attribute-value format. In the following we illustrate this.

Structured data

In the attribute value formalism, we assume that each example is described with a fixed set of attributes, and for a given instance each attribute is assigned precisely one value. Sometimes instances may have an internal structure that cannot be converted to this format.

A simple example is that of so-called *multivalued* attributes. These are attributes that take a set of values, rather than a single value. When the elements of the set come from a finite domain, this is not a big problem: one can define one attribute for each element, which takes the value true if the element is in the set, and false if it is not. With infinite domains, however, this is not possible.

Example 15.11 Assume that instances are persons, and we include with each description of a person the job of that person. Some persons might have multiple jobs. An attribute *Job* in the attribute-value framework would allow us to just fill in one job. In the logic framework, just like we could choose not to include a certain fact for an attribute when the value for that attribute was irrelevant or unknown, we can just as well decide to add several facts. So, if we have a person `p1` who is 54 years old and is both a professor at some university and a minister in the government, we can describe him as follows:

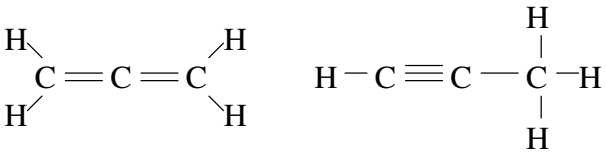

Figure 15.3: Two different molecules that have 3 carbon atoms and 4 hydrogen atoms.

```
age(p1, 54).
job(p1, professor).
job(p1, minister).
```

We could represent this in the attribute-value format, if we have a finite set of possible jobs, as follows:

ID	age	professor?	baker?	minister?	butcher?	...
p1	54	true	false	true	false	...

where we use a separate attribute for each possible job. Note that this trick works only if the set of values is finite; if the multivalued attribute takes, for instance, numerical values, this would not work.

Another example of where an instance cannot easily be represented in the attribute value format, is when an instance is in fact a set of tuples, or when it has a graph structure.

Example 15.12 Assume that an instance is a molecule, and we want to represent the chemical structure of the molecule. One of the instances might be H_2O, some other instances NO_2 and CO_2, etc. How can we represent these structures in the attribute value format? One possibility would be to just have an attribute for each element, indicating how many atoms of that element occur in the molecule. The table containing the instances would look like this:

Name	H	He	Li	Be	B	C	N	O	F	...
H_2O	2	0	0	0	0	0	0	1	0	...
NO_2	0	0	0	0	0	0	1	2	0	...
CO_2	0	0	0	0	0	1	0	2	0	...

But this representation gives only aggregated information, not complete information, on the structure of the molecules. Two molecules with a different structure might have the same counts of elements. For instance, in C_3H_4 the carbon atoms might be connected with double bonds, or with a single and a triple bond (see Figure 15.3). These are different structures with the same atom counts.

To have complete information about the chemical structure of the molecule, the graph structure shown in Figure 15.3 should somehow be represented. But there exists no obvious method for mapping such a graph structure on a single tuple. The only solution is to use another representation, not attribute-value based, in which the graph itself can be represented.

Even if we can represent the graph structure of the molecule, this does not always give us complete information about its spatial configuration. Two molecules may have exactly the same atoms and bonds, but with one molecule being a mirrored version of the other. Therefore it may be useful to add additional information to the graph.

In general, whenever the size of an instance is unbounded (e.g., the instance can be set of graph of any size), it is not possible to represent that instance as a tuple without loss of information.[1]

In the logic framework we are introducing, graphs, sets, and other structures can be represented in a straightforward way. For instance, we can represent any graph by having a fact `node(`x`)` for each node x in the graph, and a fact `edge(`x`,`y`)` for each edge (x, y). If the nodes or edges are labelled, we can add the label as an extra argument to the nodes and edges.

Example 15.13 We continue the C_3H_4 example. The nodes and edges of the graphs are atoms and bonds, so we will use the predicate names `atom` and `bond`. As in earlier examples, we introduce identifiers to indicate what example (molecule) we are talking about, so each atom and bond fact have this identifier as first argument. The second argument of an atom fact identifies a single atom within the molecule, its third argument is a label indicating the chemical element the atom belongs to. For bond facts, the second and third argument are atom identifiers x and y, the fourth argument is a label indicating whether this is a single, double of triple bond (`s`, `d`, and `t`, respectively). That gives the following representation:

```
atom(mol1, 1, c).                atom(mol2, 1, c).
atom(mol1, 2, c).                atom(mol2, 2, c).
atom(mol1, 3, c).                atom(mol2, 3, c).
atom(mol1, 4, h).                atom(mol2, 4, h).
atom(mol1, 5, h).                atom(mol2, 5, h).
atom(mol1, 6, h).                atom(mol2, 6, h).
atom(mol1, 7, h).                atom(mol2, 7, h).
bond(mol1, 1, 2, d).             bond(mol2, 1, 2, t).
bond(mol1, 2, 3, d).             bond(mol2, 2, 3, s).
bond(mol1, 1, 4, s).             bond(mol2, 1, 4, s).
bond(mol1, 1, 5, s).             bond(mol2, 1, 5, s).
bond(mol1, 3, 6, s).             bond(mol2, 3, 6, s).
bond(mol1, 3, 7, s).             bond(mol2, 3, 7, s).
```

Thus, using the logic format, we can easily represent multi-valued attributes (see the job example), graph structures, etc., which would be difficult or impossible in the attribute-value framework.

Looking at the above example, one might find it annoying that we need to introduce identifiers for the instances (in this case, the molecules). These were not necessary in the attribute value framework. In fact, also in the logical framework, we can do without them. Of course, we cannot just drop the identifiers from the facts as shown in the example: then we would not know which instance each fact describes. Some structure in the data needs to indicate which fact is relevant for which instance. We can achieve this by simply representing each instance as a set of facts, or an interpretation. Whereas in the attribute-value format a single instance is described by a vector, in the interpretation format each instance is described by an interpretation.

Example 15.14 The two different C_3H_4 molecules can be represented as follows:

```
{atom(1,c), atom(2,c),atom(3,c), atom(4,h), atom(5,h), atom(6,h),
 atom(7,h), bond(1,2,d), bond(2,3,d), bond(1,4,s), bond(1,5,s),
```

[1] In principle, it is possible to define an ordering for graphs over some domain and then represent each graph as a single number, namely its rank in that ordering. But then the structure of the graph is "encoded" in that number in a very complicated way, and a pattern over the graphs typically will not correspond to an easily definable pattern in the numbers.

```
 bond(3,6,s), bond(3,7,s)}

{atom(1,c), atom(2,c),atom(3,c), atom(4,h), atom(5,h), atom(6,h),
 atom(7,h), bond(1,2,t), bond(2,3,s), bond(1,4,s), bond(1,5,s),
 bond(3,6,s), bond(3,7,s)}
```

For graphs, this representation is clearly more elegant, in the sense that the `atom` and `bond` facts are more similar to the node and edge relations that they represent.

We have now seen two types of logical representations: one where we refer to an instance with an identifier, and each statement about the instance includes that identifier, and one where we drop the identifiers (and consequently we can never explicitly refer to the instance). Both are to some extent interchangeable. However, in the case where our input knowledge includes relations between different instances (we can call this **external structure**), the use of the explicit identifiers is unavoidable.

In this context, it is useful to distinguish **external structure** and **internal structure**. We say that the structure describing an instance is internal if it contains no references to other instances; in the other case it is external. Generally, we can drop all instance identifiers only if all the structure describing an instance is internal to it.

Example 15.15 Web pages are typically coded in HTML, which has a tree structure. We can call this the internal structure of the page. However, pages can also link to each other. This gives us an external structure, a structure that exists not within pages but within a set of different pages. To represent the external structure, page identifiers (e.g., URLs) are required.

Background knowledge

An important advantage of the use of first order logic is that background knowledge about the domain in which we learn can easily be added to the input knowledge. Take for example the chemistry domain. Chemists use many concepts to describe the properties of atoms and molecules. For instance, the elements He, Ne, Ar ... are called noble gases; F, Cl, I, are called halogens, etc. Certain classes of molecules are defined as molecules that contain a certain substructure, called a functional group; for instance, carbon acids are organic molecules that contain a COOH-substructure.

When describing molecules, we could explicitly list all the properties they have. Alternatively, rules could be given that define the different concepts. These rules will ensure that molecules are automatically assigned the correct properties.

Example 15.16 Using the notation for molecules introduced above, the definition of carbon acids can be written as follows:

```
carb_acid(X) :- atom(X,A1,c), atom(X, A2, o), atom(X,A3,o),
                atom(X,A4,h), bond(X,A1,A2,d), bond(X,A1,A3,s),
                bond(X,A3,A4,s).
```

(Since background knowledge is typically written as a Prolog program, we are using the symbol `:-` here, which is the Prolog equivalent of the $\leftarrow$ operator.)

The molecules `mol1` and `mol2` shown are not carbon acids according to this rule, so the literal `carb_acid(mol1)` will automatically be assigned the value false by the rule, and similar for `mol2`.

The use of background knowledge allows us to add a lot of information without burdening the descriptions of individual instances with this information. Especially in knowledge-rich domains, such as chemistry, this proves very valuable.

Learning from entailment, learning from interpretations

Several learning settings have been identified in ILP; we here discuss the two most important ones (for practical purposes).

In the **learning from entailment** setting, an instance is represented by a clause indicating how the target predicate follows from the background knowledge. In this setting, the representation of the instances makes it immediately clear what the target predicate is.

Suppose that for the C_3H_4 example, we want to predict a certain property p. If molecule `mol1` has property p, the following clause then represents a single positive example:

```
p(mol1) :- atom(mol1, 1, c), ..., bond(mol1, 3, 7, s).
```

where the body of the clause contains all the facts that are known about `mol1`. If `mol2` does not have property p, we can write a similar clause, which is now a negative example:

```
p(mol2) :- atom(mol2, 1, c), ..., bond(mol2, 3, 7, s).
```

The idea behind this is that the positive and negatives examples are correct and incorrect "explanations" of the target predicate.

In practice, examples are often written as facts, and their description is made part of the background knowledge. In that case, we would simply add the facts `p(mol1)` and `p(mol2)` to the sets of positive, respectively negative, examples. The positive example `p(mol1)` is then short for the clause `p(mol1) :-` B with B the set of all facts that follow from the background knowledge.

In the **learning from interpretations** setting, an instance is represented by an interpretation, and there is no explicit example identifier; as a consequence, all the information about the instance must be internal. The target predicate is not treated in any different way. For the molecules example, we would write the two molecules `mol1` and `mol2` as

```
{p, atom(1,c), atom(2,c), atom(3,c), atom(4,h), atom(5,h), atom(6,h),
 atom(7,h), bond(1,2,d), bond(2,3,d), bond(1,4,s), bond(1,5,s),
 bond(3,6,s), bond(3,7,s)}

{p, atom(1,c), atom(2,c),atom(3,c), atom(4,h), atom(5,h), atom(6,h),
 atom(7,h), bond(1,2,t), bond(2,3,s), bond(1,4,s), bond(1,5,s),
 bond(3,6,s), bond(3,7,s)}
```

where the first interpretation would be flagged as a positive example (a model for the theory that we want to find) and the second as a negative example (not a model for the theory that we want to find).

We will see later that some ILP approaches are more easily described in the learning from interpretations setting, and others in the learning from entailment setting.

15.3.2 Output knowledge

The language used for representing the output knowledge of the learning process is of course closely related to the learning task.

Many learning tasks can be defined in ILP, but originally, the learning tasks considered in ILP were concept learning tasks, which in the context of ILP corresponds to predicate learning. Given a predicate with arity D, the task is to learn from examples a predicate definition consistent with these examples, i.e., for examples for which the predicate is true, the definition should predict true, and for

the others the definition should predict false. Note that this can also be seen as a binary classification problem, where the predicate definition represents a function that classifies D-ary tuples into a class of tuples for which the predicate is true, and one for which it is false.

Depending on the representation of the instances, the problem statement for predicate learning looks slightly different. We begin with the learning from entailment setting.

Predicate learning from entailment

We can define the task of learning predicate definitions from entailment as follows:

Task description: Predicate learning from entailment

Given: a set of clauses E^+ called the *positive examples*, a set of clauses E^- called the *negative examples*, a logic program B that represents the background knowledge about the domain, and a hypothesis space $\mathcal{H}$ that contains as elements sets of definite Horn clauses (the hypotheses),
Find: a set of clauses $H \in \mathcal{H}$ such that
$\forall e \in E^+ : H \wedge B \models e$ (completeness)
$\forall e \in E^- : H \wedge B \not\models e$ (consistency)

I.e., we want to find a hypothesis, in the form of a set of definite Horn clauses, such that the hypothesis together with the background knowledge B explains (entails) all the positive examples and does not incorrectly predict any of the negative examples as positive.

It is interesting to rephrase this task description in terms of the *covers* relation, which defines when an example is covered, or explained, by a hypothesis.

Definition 15.1 *In the learning from entailment setting, we say that a hypothesis H* **covers** *an example e in the context of background B, denoted $covers(H, B, e)$, if $H \wedge B \models e$.*

We can define the notions of completeness and consistency in terms of the covers relation:

Definition 15.2 *Given a background B, a set of positive examples E^+ and a set of negative examples E^-, a hypothesis H is :*

- **consistent** *if and only if it covers no negative examples, i.e., $\forall e \in E^- : \neg covers(H, B, e)$.*
- **complete** *if and only if it covers all positive examples, i.e., $\forall e \in E^+ : covers(H, B, e)$.*

Using the *covers* relation and the notions of completeness and consistency, we can then rephrase the task description as follows.

Task description: predicate learning

Given: a set of positive examples E^+, a set of negative examples E^-, a logic program B that represents background knowledge about the domain, a hypothesis space $\mathcal{H}$ that contains as elements logic programs (the hypotheses), and a coverage relation *covers*,
Find: a hypothesis $H \in \mathcal{H}$ such that H is complete and consistent.

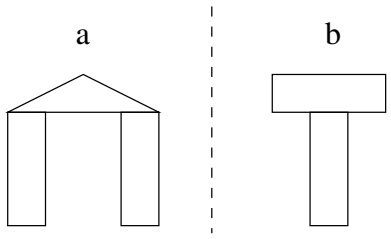

Figure 15.4: Two structures; structure (a) is called an arch, while structure (b) is not.

Note that in this second, rephrased form, we do not refer to the format of the examples (they do not need to be clauses). It is therefore more generally usable, whereas the first formulation of the problem statement was specific to the learning from entailment setting.

Example 15.17 Assume that we want to learn the concept of an arch from examples of arches and other buidings. Consider the following two examples, which represent the structures shown in Figure 15.4:

```
% positive example, element of E+
arch(a) :- contains(a,t1), contains(a,r1), contains(a,r2), triangle(t1),
  points(t1,up), rectangle(r1), rectangle(r2), on(t1,r1), on(t1,r2).

% negative example, element of E-
arch(b) :- contains(b,r3), contains(b,r4), rectangle(r3),
  rectangle(r4), on(r3,r4).
```

The task of predicate learning is to find a set of definite Horn clauses (possibly just one clause) that entails the first clause but not the second. The clause

```
arch(X) :- contains(X,T), contains(X,R1), contains(X,R2),
  triangle(T), points(T,up), rectangle(R1), on(T,R1).
```

is an example of such a clause. Another, simpler clause that is also complete and consistent for this set of two examples, is

```
arch(X) :- contains(X,T), triangle(T).
```

Predicate learning from interpretations

We now move to the task of learning predicate definitions from interpretations.

In this setting, an instance I is an interpretation. We say that a hypothesis H covers an interpretation if H is true in that interpretation, or: $I \models H$. (We could also say that H is consistent with the interpretation – but note that this use of the word "consistent" should be distinguished from the notion of consistency as just defined.)

Definition 15.3 *In the learning from interpretations setting, we say that a hypothesis H* **covers** *an example e in the context of background B, denoted $covers(H, B, e)$, if $e \wedge B \models H$.*

Given this notion of coverage, the definitions of completeness and consistency, as well as the task description, remain the same.

Example 15.18 The arches can be represented as interpretations as follows:

```
% positive example, element of E+
{arch(a), contains(a,t1), contains(a,r1), contains(a,r2), triangle(t1),
 points(t1,up), rectangle(r1), rectangle(r2), on(t1,r1), on(t1,r2)}

% negative example, element of E-
{arch(b), contains(b,r3), contains(b,r4), rectangle(r3),
 rectangle(r4), on(r3,r4)}
```

or, if we drop the example identifiers:

```
{arch, contains(t1), contains(r1), contains(r2), triangle(t1),
 points(t1,up), rectangle(r1), rectangle(r2), on(t1,r1), on(t1,r2)}

{arch, contains(r3), contains(r4), rectangle(r3), rectangle(r4),
 on(r3,r4)}
```

The simplest clause from the previous example now looks as follows:

```
arch :- contains(T), triangle(T).
```

It covers the first interpretation but not the second one.

Theory revision

Note that in both learning settings, it is quite possible for a hypothesis H to cover the positives and not cover any negatives without H containing any reference to the target predicate.

Example 15.19 In the learning from entailment setting, suppose we have
E^+ : $\{p(a,b) \leftarrow, \quad p(c,d) \leftarrow\}$
E^- : $\{p(a,a) \leftarrow, \quad p(d,c) \leftarrow\}$
B: $p(X,Y) \leftarrow q(X), r(Y)$;
then the hypothesis
H: $\{q(a) \leftarrow, \quad q(c) \leftarrow, \quad r(b) \leftarrow, \quad r(d) \leftarrow\}$
covers the positives and no negatives. In fact, this hypothesis predicts that any ground atom $p(x,y)$ is true whenever $x \in \{a,c\}$ and $y \in \{b,d\}$.

In practice, the elements of E^+ and E^- are typically atoms of one target predicate, and the hypothesis space $\mathcal{H}$ is defined in such a way that all the hypotheses are definite Horn clauses with the target predicate in the head. Many ILP algorithms explicitly assume this, and work only in this setting. But strictly speaking it is not a requirement.

When hypotheses of any format can be learned, not just definitions of a given target predicate, the learning task could be called **theory completion**: one could say that the background B is completed with a theory H such that the examples in E^+ are explained by $B \wedge H$. When the theory H can only contain ground facts, this setting is also called **abductive learning**. Abduction refers to the process of inferring likely causes for observations.

We do not go into further detail about these tasks and how they can be solved. It is interesting to note, however, that they are closely related to predicate learning: the conditions imposed on the hypotheses H are simply a bit weaker.

Regression in ILP

Predicate learning can just as well be used for regression. The target predicate is then typically an n-ary predicate with $n > 1$, where one of the arguments represents a target attribute that is to be predicted.

In the case of regression, however, it usually does not make sense to state that we want to find a predicate definition that covers the given positive and negative examples. First, there are typically only positive examples. Second, we want to find a predicate that for training instances usually predicts a value for the target argument that is close to the value occurring in the training set, but which is not necessarily exactly the same.

The best way to view regression is to state it as an **argument prediction** problem. In argument prediction, the goal is not to predict, given a tuple, whether it belongs to a predicate or not, but rather, given a tuple with one missing value (the value for the target attribute), to estimate that value for the target attribute that makes the tuple belong to the predicate.

Example 15.20 The predicate learning task we looked at before can be rephrased as a target prediction task by introducing a binary predicate `shape` that has a second argument the shape of a structure (e.g., `arch` or `t-shape`). The examples can then be represented as follows:

```
shape(a,arch) :- contains(a,t1), contains(a,r1), contains(a,r2),
  triangle(t1), points(t1,up), rectangle(r1), rectangle(r2),
  on(t1,r1), on(t1,r2).

shape(b,t-shape) :- contains(b,r3), contains(b,r4), rectangle(r3),
  rectangle(r4), on(r3,r4).
```

Note that we have no notion of positive or negative examples anymore; each clause is a positive example, and the task is to construct a definition for the shape predicate that fills in the second argument. A possible definition of shape is:

```
shape(X,arch) :- contains(X,T), triangle(t1).
shape(X,t-shape) :- contains(X,R1), rectangle(R1), on(R1, R2).
```

The task of argument prediction can be defined as follows.

Definition 15.4 *The* **observed value** *for the target attribute of an instance e, denoted $obs(e)$, is equal to the value for the target attribute filled in in e.*

Definition 15.5 *The* **predicted value** *of a hypothesis H for the target attribute of an instance e in the context of background B, denoted $pred(H, B, e)$, is equal to v if and only if there exists exactly one atom p such that $H \wedge B \models p$ and p has the same predicate, arity and arguments as e except for the target argument.*

Definition 15.6 *The* **error** *of H on e in the context of B, denoted $err(H, B, e)$, is defined as*
$err(H, B, e) = pred(H, B, e) - obs(e)$ if the target attribute is numerical;
$err(H, B, e) = \delta(pred(H, B, e), obs(e))$ if the target attribute is nominal (with $\delta(x, y) = 0$ if $x = y$ and 1 otherwise)

Task description: argument prediction

Given: a set of examples E, where examples are definite Horn clauses with in the head an n-ary atom a; an integer $t \leq n$ indicating the target attribute; and

background knowledge B;
Find: a hypothesis H such that $\forall e \in E : obs(e) = pred(H, B, e)$

The description is easily adapted to allow for prediction of numerical attributes, where the goal is not to exactly predict all the target values, but to predict values in such a way that, for instance, the sum of least squares of the errors is minimized. This yields:

Task description: least squares regression in ILP

Given: a set of examples E, where examples are definite Horn clauses with in the head an n-ary atom a; an integer $t \leq n$ indicating the target attribute; and background knowledge B;
Find: a hypothesis H such that $\sum_{e \in E} err(H, B, e)^2$ is minimal

---

Clustering in ILP

In extensional clustering, the output knowledge consists of sets of instances; so the hypotheses are typically not logic programs as in the previous cases, but simply enumerations of instances. In intensional clustering, the hypotheses are descriptions of clusters, and as such may be logic programs.

To perform clustering, a distance measure between instances is needed. Such a distance is easiest to define when learning from interpretations: the interpretations are sets of facts, so a distance measure for sets should be used. Such distance measures have been studied in detail in the specialized literature.

Even then, just like in propositional clustering, many definitions of distances are possible, and which definitions are reasonable will depend on the application. We will not treat this topic in more depth here.

15.4 Generality orderings for ILP

We have already seen before that the notion of generality is important in machine learning: it allows us to search a hypothesis space in a systematic way, and to prune parts of the search space that cannot possibly contain a better solution than the best one found up till now.

In the context of first-order logic, we could also take advantage of a generality ordering when looking for patterns, but then we first need to define such an ordering. As it turns out, generality orderings are more difficult to define for first-order theories than for, say, conjunctive concepts (as used in the Versionspaces approach). In addition, simply testing whether one theory is more general than another also becomes more complex.

We first extend the notion of "generality" to patterns, as opposed to concept definitions; next, we will define generality in the context of first order predicate logic. After that, we will have a closer look at a number of particular generality orderings that have been proposed in the context of ILP.

15.4.1 Generality of concepts and patterns

In concept learning, a concept A is said to be more general than another concept B if any object belonging to B must necessarily belong to A as well (while objects

in A are not necessarily in B). For instance, the concept *fruit* is more general than the concept *apple*.

When learning patterns instead of concepts, the notion of generality needs to be rephrased a bit. We do not usually say that an object "belongs to" a pattern or not; rather, we can say that an object *is consistent with* a pattern, or *contradicts* it.

For instance, the pattern "all apples are green" would be contradicted by a red apple. A white mouse is consistent with the pattern: seeing a white mouse does not falsify the claim that all apples are green. Generally, only apples that are not green contradict the pattern.

A possible definition of generality of patterns is then the following.

Definition 15.7 *A pattern A is* **at least as general** *as a pattern B if any instance consistent with B must necessarily be consistent with A. A pattern A is* **strictly more general** *than a pattern B if any instance consistent with B must necessarily be consistent with A, while the opposite does not hold.*

Note that the statement "every instance consistent with B is also consistent with A" implies that when something is not consistent with A, it cannot be consistent with B. So, the statement is equivalent to "every instance that contradicts A also contradicts B". An alternative but equivalent definition of generality is therefore:

Definition 15.8 *A pattern A is* **at least as general** *as a pattern B if any instance that contradicts A must necessarily also contradict B. A pattern A is* **strictly more general** *than a pattern B if any instance that contradicts A must necessarily also contradict B, while the opposite does not hold.*

The above definition of generality may seem natural, but it sometimes gives rise to confusing results. For instance, the pattern A: "all apples are green" is more general than B: "all fruit is green". Indeed, any object that is consistent with the claim that all fruit is green, must be consistent with the claim that all apples are green. (To see why this is, observe that only non-green apples contradict the latter claim, and since all apples are fruit, they are examples of non-green fruit and therefore contradict the statement that all fruit is green.)

Intuitively, we have a tendency to consider a statement as more general if the concepts occurring in it are more general. But according to our above definition, a statement becomes more general if more instances are consistent with it, and in an "if-then" type of pattern this means that *a statement becomes more general if the concepts in the then-part become more general and/or the concepts in the if-part become more specific.*

Example 15.21 The statement "this is an apple" is more specific (less general) than "this is a piece of fruit". This corresponds to the intuition that a specific statement carries more information than a general statement. According to the same intuition, "all fruit is green" is more specific (less general) than "all apples are green"; indeed, the first statement conveys more information than the second. However, the occurence of the more general concept "fruit" in the first statement makes it easy to make the mistake of calling that statement "more general".

Clearly, the concept of "generality" is more intuitive when applied to concepts than when applied to patterns. For this reason one could say that the word "generality" is not very well chosen when we are describing patterns. But the term is quite commonly used in this sense, and it is not obvious what word would better describe this property of patterns (the term "less informative" to denote "more general" might be a candidate). Consistent with the literature, we will continue to use the term "generality" to describe patterns.

The reader should remember that *more general patterns* are patterns that are *less informative* or *less restrictive*, in the sense that more objects, or more states of the world, are consistent with the pattern.

15.4.2 Generality in logic

In the light of the above discussion, we formally define the notion of generality of logic formulas as follows.

Definition 15.9 *A logic formula ϕ is* **at least as general** *as a logic formula ψ, denoted $\phi \preceq_g \psi$, if ϕ is true whenever ψ is true; in other words, if ψ entails ϕ ($\psi \models \phi$). A logic formula ϕ is* **strictly more general than** *a logic formula ψ, denoted $\phi \prec_g \psi$, if $\phi \preceq_g \psi$ and $B \not\preceq_g A$. If ϕ is at least as (more) general than ψ, we can also say that ψ is at least as (more) specific than ϕ.*

Note that we write $x \preceq_g y$ if x is more (or equally) general than y. The $\preceq_g$ is to be interpreted as "precedes in the generality ordering". (Again this notation may be somewhat confusing; but it is consistent with the literature.)

Logical generality applied to concept definitions

When we define a concept C, we can write the definition as an implication $C \leftarrow D$ with D a description of the concept, but this has the awkward consequence that the definition (the implication statement) is more general if the description D is more specific. This is shown by the following proposition.

Proposition 15.1 *With C an atom and D a conjunction of literals, we have $(C \leftarrow D) \models (C \leftarrow D')$ if and only if $D' \models D$.*

Proof: $(C \leftarrow D) \models (C \leftarrow D')$ is by definition equivalent to $(C \vee \neg D) \models (C \vee \neg D')$, which holds if and only if $\neg D \models \neg D'$, which holds if and only if $D' \models D$. □

Example 15.22 Consider the following two definitions of apples:
1) $apple(X) \leftarrow fruit(X), round(X), green(X)$
2) $apple(X) \leftarrow fruit(X), round(X)$
The second definition defines a more general concept "apple" than the first; it allows for red apples, for instance, while the first doesn't. But looking at the implications, we have that the first implication has more models than the second (any world in which round fruit exists that is not an apple, contradicts the second clause, whereas the first clause is only contradicted when green round fruit exists that is not an apple).

To avoid confusion, when we talk about generality of a "concept definition", we consider the body D of the implication as the actual definition, and not the implication as a whole. That way (and only then), the intuitive notion of generality of concepts coincides with generality of concepts in the logical framework.

In the above example with apples, this implies that we will call the second definition, "round fruit", a more general definition than the first, "green round fruit".

Practical generality orderings

The above definition of the generality ordering is theoretically interesting, but not very useful in practice. Testing whether a formula is more general than another formula, according to the above definition, is semi-decidable: no algorithm exists that for any two formulas ϕ and ψ can say within a finite amount of time whether ϕ is more general than ψ or not. Therefore, learning algorithms will necessarily have to use some approximation of the generality ordering as defined above, rather than the theoretically correct ordering.

In the following, we will have a look at two orderings that have been proposed for practical use. They are approximations in two ways: first, they do not necessarily define an ordering on all logic formulas, but only on a subset of them; second, even within this subset, the defined partial order may not be exactly the same as the true generality ordering.

15.4.3 The theta-subsumption ordering

Theta-subsumption is a relationship between clauses that has been studied in much detail in inductive logic programming. It is useful because the partial order it imposes on clauses can be interpreted as a generality ordering.

Definition 15.10 (Theta-subsumption) *A clause c_1 theta-subsumes a clause c_2, denoted $c_1 \preceq_\theta c_2$, if and only if there exists a substitution θ such that $c_1\theta \subseteq c_2$.*

Example 15.23 The following are a few examples of clauses that do or do not theta-subsume each other.

- The clause $\{p(X), q(X)\}$ subsumes the clause $\{p(X), q(X), \neg r(X,Y)\}$, as it is a subset of the latter. Equivalently, using the implication format, we can say that $p(X) \vee q(X) \leftarrow$ subsumes $p(X) \vee q(X) \leftarrow r(X,Y)$.
- The definite Horn clause $\{p(X), \neg r(X,Y)\}$ (or $p(X) \leftarrow r(X,Y)$) subsumes the definite Horn clause $\{p(X), \neg q(X) \neg r(X,Y)\}$ (or $p(X) \leftarrow q(X), r(X,Y)$).
- $p(X) \leftarrow r(X,Y)$ subsumes $p(X) \leftarrow q(X), r(X,X)$, since applying the substitution $\{Y/X\}$ to the first clause yields a subset of the second clause.
- $p(X) \leftarrow r(X,Y)$ subsumes $p(a) \leftarrow q(a), r(a,Y)$ and also subsumes $p(X) \leftarrow r(X,b)$. The two subsumed clauses do not subsume each other.

The last example shows that subsumption is at most a partial order: not each pair of clauses can be ordered using subsumption. It turns out that theta-subsumption is a semi-order. We recall the definitions of semi-order, partial order, and total order:

Definition 15.11 *A relation $\leq$ is a* **semi-order** *if it is reflexive and transitive, i.e., for all x it holds that $x \leq x$ (reflexivity) and for all x, y, z for which $x \leq y$ and $y \leq z$, it holds that $x \leq z$ (transitivity).*

Definition 15.12 *A relation $\leq$ is a* **partial order** *if is a semi-order and it is antisymmetric. In other words, in addition to the conditions for semi-orders, it must hold for all x and y that if $x \neq y$, either $x \leq y$ or $y \leq x$ may hold, but not both.*

Definition 15.13 *A relation $\leq$ is a* **total order** *if it is a partial order and for all x, y it holds that either $x \leq y$ or $y \leq x$.*

Theta-subsumption is a semi-order, but not a partial order. The following example shows that it is not anti-symmetric.

Example 15.24 $p(X) \leftarrow q(X), r(X,X)$ subsumes $p(X) \leftarrow q(X), r(X,X), r(X,Y)$, since it is a subset of it. However, the substitution $\{Y/X\}$ turns the second clause into the first one.[2] Hence, the second clause also subsumes the first.

A semi-order partitions its domain into equivalence classes, where x and y are in the same equivalence class (and are called equivalent), denoted $x \sim y$, if and only if $x \leq y$ and $y \leq x$.

In the context of theta-subsumption, two clauses that theta-subsume each other are called theta-equivalent.

Definition 15.14 *Clauses c_1 and c_2 are theta-equivalent, denoted $c_1 \sim_\theta c_2$, if and only if $c_1 \preceq_\theta c_2$ and $c_2 \preceq_\theta c_1$.*

Theta-subsumption and entailment

Theta-subsumption has the important property that, whenever a clause is true in some interpretation, any clause subsumed by it must also be true in that interpretation. In other words, if c_1 theta-subsumes c_2, then c_1 entails c_2, as the following proposition states.

Proposition 15.2 *If $c_1 \preceq_\theta c_2$, then $c_1 \models c_2$.*

Proof: $c_1 \preceq_\theta c_2$ implies that there exists a θ such that $c_1\theta \subseteq c_2$. Let $c = c_1\theta$. We have $c_1\theta = c$ and $c \subseteq c_2$. It is easy to see that $c_1 \models c$: since all variables in c_1 are universally quantified, any model of c_1 must be a model of $c = c_1\theta$, whatever θ is. Further, $c \models c_2$: this follows from the interpretation of c as a disjunction of literals (adding literals to the distunction can never falsify it). From $c_1 \models c$ and $c \models c_2$ we can conclude $c_1 \models c_2$. □

Example 15.25 The clause $p(X) \vee q(X) \leftarrow$ subsumes the clause $p(X) \vee q(X) \leftarrow r(X,Y)$. The meaning of the first clause is that $p(X) \vee q(X)$ is true for all X. The meaning of the second clause is that $p(X) \vee q(X)$ is true for all X for which $r(X,Y)$ is true. Obviously, if $p(X) \vee q(X)$ is true for all X, it must also be true for those X for which $r(X,Y)$ holds, so the first clause indeed logically entails the second.

It follows immediately from Proposition 15.2 that when two clauses are theta-equivalent (i.e., they subsume each other), they entail each other, which means that they are logically equivalent (i.e., have the same set of models):

Corollary 15.1 *If $c_1 \sim_\theta c_2$, then $c_1 \models c_2$ and $c_2 \models c_1$, i.e., c_1 and c_2 are logically equivalent.*

The fact that $c_1 \preceq_\theta c_2$ implies $c_1 \models c_2$, which is itself equivalent to $c_1 \preceq_g c_2$, means that the theta-subsumption order is sound with respect to the logical generality ordering. When a clause theta-subsumes another clause, it is at least as general as that other clause.

The opposite does not hold, however: when a clause is more general than another clause, this does not imply that it theta-subsumes the other clause. In other words, theta-subsumption is not complete with respect to logical generality.

[2]Note that clauses are sets of literals: when the literal $r(X,Y)$ is turned into $r(X,X)$, and $r(X,X)$ already occurs in the clause, this does not create a second occurrence of $r(X,X)$ in the clause.

Example 15.26 The following clauses c_1 and c_2 have the property that $c_1 \models c_2$ but $c_1 \not\preceq_\theta c_2$:
c_1: $p(f(X)) \leftarrow p(X)$
c_2: $p(f(f(X)) \leftarrow p(X)$
The first clause states that whenever, for instance, $p(a)$ holds, $p(f(a))$ must hold, and therefore (by repeated application of the same clause) $p(f(f(a)))$ must hold as well. Hence it is impossible to find a model for the first clause that is not a model of the second clause. However, there exists no substitution θ such that $c_1\theta \subseteq c_2$.

The difference between theta-subsumption and logical generality is only visible for so-called "self-resolving" clauses, which are a special case of recursive clauses. If we would restrict ourselves to non-recursive clauses, the theta-subsumption ordering would be sound and complete with respect to the logical generality ordering.

Generalization and specialization according to theta-subsumption

From semi-order to partial order Whenever we have a semi-order over a domain, there exists a partial order over the equivalence classes of the semi-order. Indeed, define the order $\preceq'$ over equivalence classes as follows:

$$C_1 \preceq' C_2 \Leftrightarrow \exists c_1 \in C_1, c_2 \in C_2 : c_1 \preceq_\theta c_2.$$

Note that $\exists c_1 \in C_1, c_2 \in C_2 : c_1 \preceq_\theta c_2$ implies

$$\forall c_1 \in C_1, c_2 \in C_2 : c_1 \preceq_\theta c_2$$

(this follows from the transitivity of $\preceq_\theta$ and the fact that for all clauses c and c' in the same equivalence class, $c \preceq_\theta c'$).

The relation $\preceq'$ is a partial order. Its reflexivity and transitivity carry over directly from $\preceq_\theta$. That it is also antisymmetric can be seen as follows. If $C_1 \preceq' C_2$ and $C_2 \preceq' C_1$, this implies that all clauses in C_1 subsume all clauses in C_2 and vice versa, which means that all clauses in C_1 are equivalent to all clauses in C_2, hence C_1 and C_2 must be the same equivalence class.

So the semi-order over clauses induces a partial order over the equivalence classes. Now it turns out that for each equivalence class C_i, there is a unique clause c_i that has, among all the clauses in C_i, the fewest literals. This clause is called the **reduced clause**. Two different reduced clauses must always be in different equivalence sets (since the reduced clause of a single equivalence set is unique). So, if we define C^R as the set of all reduced clauses that are equivalent to a clause in C, then theta-subsumption is antisymmetric in C^R and hence forms a partial order over C^R.

The conclusion of this is that, while theta-subsumption may induce only a semi-order over a set of clauses, it always induces a partial order over any set of reduced clauses.

Computing the reduced clause

Before continuing, we briefly explain how the reduced version of a clause can be computed, as this will be useful later on.

A clause is reduced if and only if there exists no substitution σ such that $c\sigma \subseteq c$. Indeed, if such a substitution exists, then: (1) $c \preceq_\theta c\sigma$ (by definition), (2) $c\sigma \preceq_t$ *hetac* (follows trivially from $c\sigma \subset c$); (1) and (2) imply $c\sigma \sim_\theta c$ and hence c is not reduced (we found a strict subset of it that is equivalent). If such a substitution

does not exist, then there is no clause $c' \subset c$ such that $c \preceq_\theta c'$, again by definition of theta-subsumption.

Thus, a possible procedure for reducing a clause c is the following: try to find a σ such that $c\sigma \subset c$; if no such σ can be found, c is reduced; otherwise, apply the same procedure to $c\sigma$.

Example 15.27 Consider the clause $\{p(X,Y), q(Y,Z), p(X,U), q(U,V)\}$. The substitution $\{U/Y, V/Z\}$ turns this clause into a strict subset of itself: $\{p(X,Y), q(Y,Z)\}$. The new clause is reduced.

Example 15.28 Consider the clause $\{p(X,Y), q(Y,Z), p(X,U), q(U,V), r(Z,V)\}$. This clause is reduced. The substitution mentioned before would yield $\{p(X,Y)$, $q(Y,Z)$, $r(Z,Z)\}$, which is not a subset of the original clause: the literal $r(Z,Z)$ did not occur in the original clause.

Searching a set of clauses Assume that we have a set of clauses C, in which we are searching for a specific clause c with certain properties P (such as: being consistent with the data, being the most general clause that does not cover any negative examples, etc.). We assume that the properties P are semantic, not syntactic, properties, i.e., when two clauses are logically equivalent they must have the same properties.

Since the clauses in an equivalence class are logically equivalent, and there is one reduced clause per equivalence class, it is sufficient to consider only reduced clauses in our search for a clause satisfying a certain semantic property: whenever any clause in C satisfies the property, its reduced clause must also satisfy it. Instead of searching the original set C, it suffices to search the set C^R containing reduced clauses only.

The lattice induced by theta-subsumption Figure 15.5 shows an example where clauses are grouped into equivalence classes. The clause in boldface is always the reduced clause.

We can visualize the partial order over the reduced clauses as a directed acyclic graph where there is an edge from c to d if $c \preceq_\theta d$. In this graph, we call e an ancestor of c if there is a path from e to c. We call e a common ancestor of c and d if it is an ancestor of both. If e is a common ancestor of c and d and there exists no e' such that e' is a common ancestor of c and d and e is an ancestor of e', we call e the nearest common ancestor of c and d. It turns out that the nearest common ancestor of two clauses is always unique, and the same holds for their nearest common descendant; therefore, the directed acyclic graph is actually a lattice.

Since the edges correspond to the "is more general than" relation, as approximated by the theta-subsumes relation, a common ancestor of two clauses c and d is a clause that is more general than both c and d, and the nearest common ancestor is the least general clause that generalizes both c and d. We call this nearest common ancestor the **least general generalization** (abbreviated as lgg) of c and d. The **lgg operator** is a function that maps any two clauses c and d on their least general generalization, denoted $lgg(c,d)$.

Usually, the set of clauses C is not just some random set of clauses, but has certain closure properties, such as: whenever a clause is in C, all subclauses of it are also in C. Under this condition, it holds that all $c, d \in C : lgg(c,d) \in C$, that is, C is closed with respect to the lgg operator.

To summarize all this: theta-subsumption induces only a semi-order over clauses, but it gives rise to a partial order over the set of reduced clauses, and to a lattice over this set. It is sufficient to search the space of reduced clauses because for any

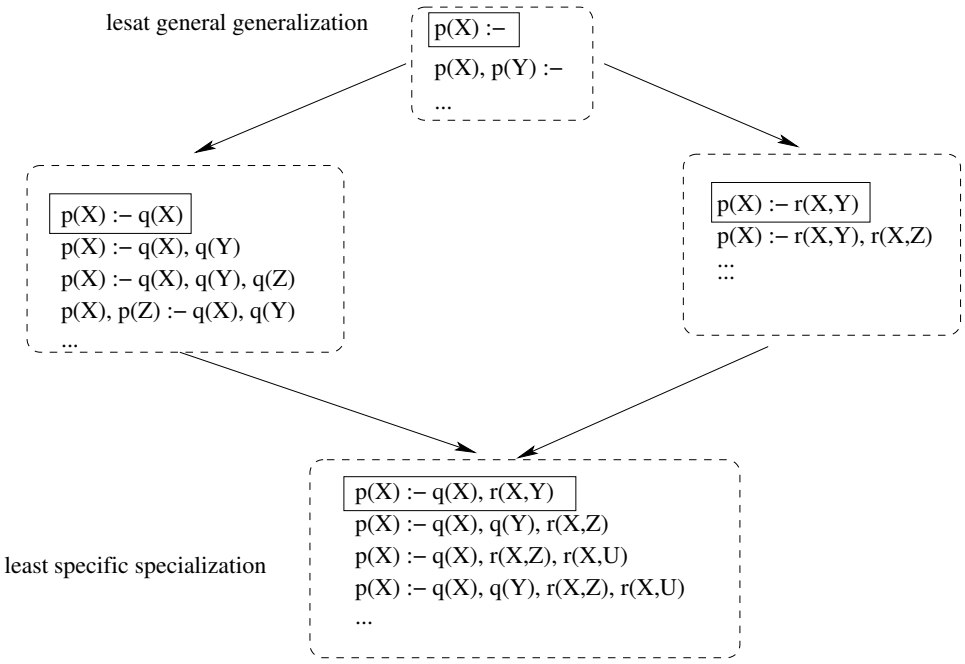

Figure 15.5: Theta-subsumption induces a semi-order over all clauses, and a partial order over reduced clauses. The reduced clauses are indicated by solid rectangles. The arrows indicate the theta-subsumption relationship. The equivalence class at the top contains the least general generalization of the clauses $p(X) \leftarrow q(X)$ and $p(X) \leftarrow r(X, Y)$; the equivalence class at the bottom contains their least specific specialization.

clause fulfilling a certain semantic property there is a reduced clause fulfilling that property.

Even though the theta-subsumption ordering behaves much nicer over the set of reduced clauses than over the set of all clauses, it still has some annoying properties. One of them is that there may exist infinite chains between two nodes. For instance, consider the following two clauses:

$$c\text{:} \leftarrow p(X,Y)$$
$$d\text{:} \leftarrow p(X,X)$$

It is clear that $c \preceq_\theta d$. If C is the set of all clauses of the form $\leftarrow B$ with B a conjunction of p-literals, then C contains an infinite sequence of clauses $c_1, c_2, \ldots,$ for which $c \preceq_\theta c_i$, $c_i \preceq_\theta c_{i+1}$, and $c_i \preceq_\theta d$:

$$c\text{:} \leftarrow p(X,Y)$$
$$c_1\text{:} \leftarrow p(X,Y), p(Y,Z)$$
$$c_2\text{:} \leftarrow p(X,Y), p(Y,Z), p(Z,U)$$
$$\ldots$$
$$d\text{:} \leftarrow p(X,X)$$

Such infinite chains are annoying because of several reasons. If we perform a general-to-specific search through C, starting from c we will never arrive in d. While the clause d is in the lattice, it is "unreachable" if we start at the top of the lattice and make minimal refinements to some clause until a clause satisfying the desired properties is obtained.

We say that c is a minimal generalization of d, and d a minimal specialization of c, if $c \preceq_\theta d$ and $\nexists e : c \preceq_\theta e \wedge e \preceq_\theta d$.

While a clause c always has minimal specializations, it may not always have minimal generalizations. The clause $\leftarrow p(X,X)$, for instance, has generalizations, but does not have minimal generalizations. Indeed, in the infinite chain shown above, for any c_i such that $c_i \preceq_\theta d$, there exists a c_{i+1} strictly between c_i and d, i.e., $c_i \preceq_\theta c_{i+1} \preceq_\theta d$ and $c_{i+1} \not\preceq_\theta c_i$.

Because of these properties of the theta-subsumption lattice, specialization and generalization (i.e., moving down or up in the lattice) are relatively complex operations. Much work has been done in the area of ILP on describing how specialization and generalization should be performed. Refinement operators have been proposed to this end.

Refinement operators for searching the theta-subsumption lattice

Note that, because of the possible non-existence of minimal generalizations, upward refinement is problematic. Most work has therefore been done on downward refinement.

A **downward refinement operator** ρ is any function that maps a clause c onto a set of clauses S such that S contains only specializations of c. Formally:

Definition 15.15 *A* **downward refinement operator** ρ *is a function from* C *to* 2^C *such that*

$$\rho(c) \subseteq \{d \in C | c \preceq_\theta d\}.$$

Definition 15.16 *The transitive closure of an operator* ρ *is a function* ρ^* *defined as follows:*
1. $\rho(c) \subseteq \rho^*(c)$
2. $d \in \rho^*(c) \Rightarrow \rho(d) \subseteq \rho^*(c)$
3. $\rho^*(c)$ *is the smallest set that has the above properties.*

Note that $\rho(c)$ and $\rho^*(c)$ do not necessarily contain all the specializations of c: they are subsets of the set of all specializations of c.

A number of desirable properties of downward refinement operators have been defined.

Definition 15.17 *A downward refinement operator ρ is:*

- **proper** *if and only if* $\nexists d \in \rho(c) : d \preceq_\theta c$.
- **minimal** *if and only if* $\forall d \in \rho(c) : \nexists e \in C : c \prec_\theta e \prec_\theta d$.
- **finite** *if and only if $\rho(c)$ has a finite number of elements.*
- **locally complete** *if and only if $\rho(c)$ contains all the minimal specializations of c.*
- **(globally) complete** *if and only if* $\forall d \in C : c \prec_\theta d \Rightarrow \exists e \in \rho^*(c) : e \sim_\theta d$.
- **weakly complete** *if and only if* $\rho^*(\leftarrow) = C$.
- **non-redundant** *if and only if* $\forall c_1, c_2, d \in C$: $d \in \rho^*(c_1) \cap \rho^*(c_2) \Rightarrow c_1 \in \rho^*(c_2) \vee c_2 \in \rho^*(c_1)$.
- **ideal** *if and only if it is finite, proper and complete.*
- **optimal** *if and only if it is finite, non-redundant and weakly complete.*
- **perfect** *if it is minimal and optimal.*

All of these properties of refinement operators are desirable in one way or another. Properness implies that no clauses are generated that are in fact equivalent to the clause that is being refined; non-redundancy implies that no clause can be reached along different paths through the lattice (which is useful to avoid checking parts of the search space multiple times); minimalness ensures that no clauses are "skipped" during the search; etc.

Unfortunately, there are theoretical results showing that it is impossible to create refinement operators for ILP that have all the desirable properties. As a result, any ILP system searching the theta-subsumption lattice does so in a way that is suboptimal in one way or another.

15.4.4 The resolution ordering

Note that theta-subsumption is an ordering over single clauses. That is, we can use it to (approximately) say whether a clause is more general than another one, but not to say whether any logical theory (possibly consisting of multiple clauses) is more general than another one. In this section, we have a look at the so-called resolution ordering, which defines an ordering over *sets of clauses*, and is more powerful from this perspective.

Resolution is a step in a logical process where from two clausal formulas a new clausal formula is inferred that follows logically from it. A Prolog system that deduces a fact from a set of clauses, does nothing else than apply the resolution step multiple times, until an inference chain from the clauses to the fact has been found.[3] The resolution-based ordering is an ordering between sets of clauses. A set of clauses precedes another set of clauses in the resolution ordering if one can get from the first set to the second set by applying only the resolution operator.

The definition of the resolution operator is as follows.

[3]Technically, an inference chain from the set of clauses extended with the negated fact towards a contradiction is constructed using resolution; but this is equivalent to finding a chain from the set to the fact itself.

Definition 15.18 *Given two clauses c_1 and c_2 that have literals $l_1 \in c_1$ and $l_2 \in c_2$ such that substitutions θ_1, θ_2 exist such that $l_1\theta_1 = \neg l_2\theta_2$,* **resolution** *derives clause $c_3 = (c_1 \cup c_2)\theta_1\theta_2 - \{l_1\theta_1, l_2\theta_2\}$ from c_1 and c_2. We say that c_3 is a* **resolvent** *of c_1 and c_2.*

Example 15.29 Suppose we have the following clauses:

$$p(X) \leftarrow q(X), r(X,Y), s(Y)$$
$$q(X) \leftarrow s(X), t(X,Y)$$

To apply resolution, we first write the clauses in set format, and rename the variables such that all variables from the second clause are different from those in the first clause. (The renaming is necessary because the scope of each variable is just its own clause: the X in the first clause is different from the X in the second clause. When merging both clauses they would mistakenly be considered the same, unless we rename one of them first.) Thus we get:

$$\{p(X), \neg q(X), \neg r(X,Y), \neg s(Y)\}$$
$$\{q(X'), \neg s(X'), \neg t(X',Y')\}$$

Then, we look for opposite literals in both sets, where opposite means that, after applying the right substitution, they contain the same atom but with a different sign ($l_1\theta_1 = \neg l_2\theta_2$). $\neg q(X)$ in the first clause and $q(X')$ in the second clause are in this situation: with a substitution $\theta_2 = \{X'/X\}$ the second clause is changed into

$$\{q(X), \neg s(X), \neg t(X,Y')\}$$

and then we can merge the two sets, leaving out the literals that have been resolved against each other:

$$\{p(X), \neg r(X,Y), \neg s(Y), \neg s(X), \neg t(X,Y')\}$$

In implication form, this clause is

$$p(X) \leftarrow s(X), t(X,Y'), r(X,Y), s(Y).$$

Note that this can be interpreted as the first clause where the literal $q(X)$ has been changed into its "definition" according to the second clause. It is generally the case that if we have a literal l in the body of a clause, and we have another clause $l \leftarrow B$, then l can be replaced by B.

Example 15.30 Suppose we have the following clauses:
$q(X) \leftarrow s(X), t(X,Y)$
$s(a)$
$t(a,b)$
$t(b,c)$
Applying the same technique as in the previous example on the first two clauses, but now choosing a substitution $\{X/a\}$ to apply to the first clause, we get as resolvent

$$q(a) \leftarrow t(a,Y).$$

This will not resolve against the fourth clause, because there is no substitution that can turn a into b, but it does resolve against the third clause, with $\{Y/b\}$, which yields

$$q(a) \leftarrow .$$

Resolution is a generalization of several other (and perhaps better known) rules for logical inference. The *modus ponens* rule, for instance, states that from p and $p \rightarrow q$ we can deduce q. The *modus tollens* rule states that from $p \rightarrow q$ and $\neg q$ follows $\neg p$. Both rules can easily be checked to be special cases of the resolution rule. Indeed, resolution is *complete* in the sense that anything that follows logically from a set of logical formulas can be inferred through repeated application of the resolution rule.

Exercise 15.1 Show that from the set

$$
\begin{array}{l}
parent(X,Y) \leftarrow father(X,Y) \\
parent(X,Y) \leftarrow mother(X,Y) \\
grandparent(X,Y) \leftarrow parent(X,Z), parent(Z,Y) \\
daughter(X,Y), son(X,Y) \leftarrow parent(Y,X) \\
father(X,Y), mother(X,Y) \leftarrow daughter(Y,X) \\
parent(X,Y) \leftarrow son(Y,X)
\end{array}
$$

the following clauses can be inferred through (repeated application of) resolution:

$$
\begin{array}{l}
daughter(X,Y), son(X,Y) \leftarrow father(Y,X) \\
son(X,Y) \leftarrow \neg daughter(X,Y), parent(Y,X) \\
grandparent(X,Y) \leftarrow father(X,Z), daughter(Y,Z) \\
parent(X,Y) \leftarrow daughter(Y,X)
\end{array}
$$

Generalizing and specializing according to resolution

Given a set of clauses, application of the resolution operator yields a set of clauses that follows from it, and therefore this constitutes a specialization step: the new set is less general than the original one. *Resolution can be used to specialize a set of clauses.*

Conversely, a theory can be generalized by applying an **inverse resolution** step: given a set of clauses S, find a set of clauses G such that S can be obtained from G by applying the resolution operator.

Inverting the resolution operator may seem a simple thing to do, but in fact it is not. There are multiple reasons for this. The first is that, roughly speaking, resolution needs to make different things equal, while inverse resolution needs to make equal things different. Roughly speaking, there are many more ways in which equal things can be made different than ways in which different things can be made the same. Second, resolution may result in predicates disappearing from a set of clauses, and consequently, inverse resolution may need to introduce new predicates that were not in the original set. Again, there are many different ways in which a new predicate can be introduced.

Let's look at the first problem. When applying resolution, two opposite literals need to be instantiated in such a way that except for their sign they are exactly the same; we say that their atoms need to be unified. There are many ways to do this, but if we assume that we want to make a "minimal" step, i.e., we want to keep the result as general as possible, there is only one way to do this. In logicians' terms, the *most general unifier* of two atoms is unique (up to variable renamings). The most general unifier of two atoms A and B is a substitution θ such that $A\theta = B\theta$ and there exists no θ' such that: (1) $A\theta' = B\theta'$ and (2) there is a non-empty θ'' such that $\theta'\theta'' = \theta$.

Example 15.31 If we want to resolve $p(X,Y)$ against $\neg p(Z,a)$, we need to find a substitution θ such that $p(X,Y)\theta = p(Z,a)\theta$. The substitution $\theta_1 = \{X/Z, Y/a\}$ is one possible choice. $\theta_2 = \{X/b, Y/a, Z/b\}$ is another choice, and $\theta_3 = \{X/c, Y/a, Z/c\}$

is yet another one. But the latter two substitutions give a result that is more specific than necessary. We have instantiated a variable into a constant without any need for this. θ_1 is the most general unifier of A and B.

Now, let's see what happens if we want to apply resolution in the opposite direction. More precisely: assuming that clauses c_1 and c_2 yield clause c_3 in a single resolution step, how can we reconstruct c_1 from c_2 and c_3?

We need to find a substitution θ and literal $l \in c_2$ such that $c_1\theta \cup c_2\theta - \{l\theta, \neg l\theta\} = c_3$, from which c_1 should then be solved. The literal $l \in c_2$ can be a positive or a negative literal, and should be such that $l\theta \notin c_3$.

Let us first consider the case where the literal in c_2 that has been resolved away in c_3 was negative. Writing the clauses as

$$c_1 = \{a_1\} \cup A \tag{15.1}$$
$$c_2 = \{\neg a_2\} \cup B \tag{15.2}$$
$$c_3 = (A\theta_1 \cup B\theta_2) \tag{15.3}$$

with A and B any set of literals and a_1 and a_2 atoms, and adding as fourth equation

$$a_1\theta_1 = a_2\theta_2, \tag{15.4}$$

we get from Equation 15.4

$$a_1 = (a_2\theta_2)\theta_1^{-1}$$

and from 15.3

$$A = (c_3 - B\theta_2)\theta_1^{-1}$$

where θ_1^{-1} is a so-called inverse substitution. (An inverse substitution consists of changing particular term occurrences into variables.) Filling these in in 15.2 gives us c_1.

Similarly, if c_2 contained a positive literal a_2 instead of the negative literal $\neg a_2$, we get

$$c_1 = \{\neg a_1\} \cup A$$

with a_1 and A exactly as before.

These two different variants of inverse resolution are called the **absorption** operator (if $a_2 \in c_2$) and the **identification** operator (if $\neg a_2 \in c_2$), respectively. They can also be written as follows (now using the implication form of clauses):

Absorption: $\dfrac{p \leftarrow A, B \qquad q \leftarrow A}{p \leftarrow q, B \qquad q \leftarrow A}$

Identification: $\dfrac{p \leftarrow A, B \qquad p \leftarrow A, q}{q \leftarrow B \qquad p \leftarrow A, q}$

The relationship between resolution and these inverse resolution operators can be shown graphically, as in Figure 15.6. Resolution starts with the two upper clauses and derives the lower clause. Inverses resolution starts with the lower clause and one of the upper clauses and derives the other upper clause.

Because of the shape of these schemas, the identification and absorption operators are also called "V-operators".

The two operators mentioned above, unfortunately, are not sufficient to be able to trace backwards on any resolution path from a set S to a set S'. The problem is that a predicate might disappear from the set of predicates through resolution: some predicates in S may not occur in S'. Therefore, when tracing backwards, it may be necessary to introduce new predicates.

Two operators have been proposed to introduce a new predicate in this way. They are called intraconstruction and interconstruction.

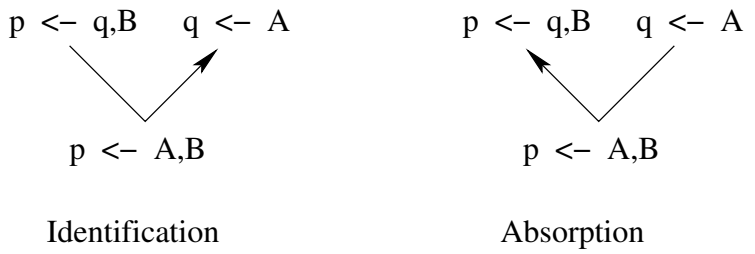

Figure 15.6: The V operators from inverse resolution.

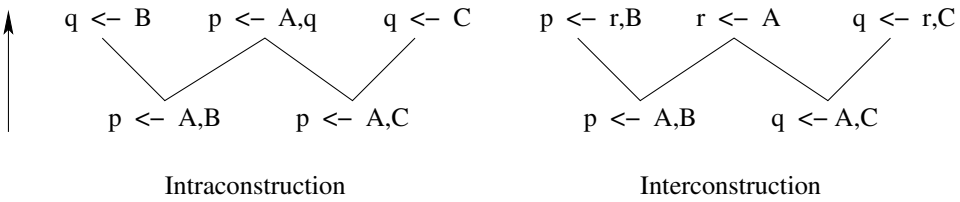

Figure 15.7: The W operators from inverse resolution.

$$\text{Intraconstruction: } \frac{p \leftarrow A, B \qquad\qquad p \leftarrow A, C}{q \leftarrow B \qquad p \leftarrow A, q \qquad q \leftarrow C}$$

$$\text{Interconstruction: } \frac{p \leftarrow A, B \qquad\qquad q \leftarrow A, C}{p \leftarrow r, B \qquad r \leftarrow A \qquad q \leftarrow r, C}$$

Note that these operators each take two clauses as input and return three clauses as output. Their graphical representation is shown in Figure 15.7. Because of this graphical representation, these two operators are called W-operators.

Example 15.32 Consider the following two clauses:
$grandfather(X, Y) \leftarrow father(X, Z), mother(Z, Y)$
$grandfather(X, Y) \leftarrow father(X, Z), father(Z, Y)$
Applying intraconstruction to these clauses leads to the introduction of a new predicate, which we recognize as the *parent* predicate:
$grandfather(X, Y) \leftarrow father(X, Z), parent(Z, Y)$
$parent(X, Y) \leftarrow father(X, Y)$
$parent(X, Y) \leftarrow mother(X, Y)$

It has been shown that the W-operators and the V-operators together are complete, in the sense that by applying only these operators, starting from a set S', any set S can be reconstructed for which it holds that S' can be derived from S using resolution.

While inverse resolution, with these four operations, provides a powerful mechanism for inductive reasoning, practical work on this has shown that induction using these operators is computationally very expensive, due to a heavily branching search space. As a result, the method has mostly been abandoned in recent ILP systems. The older ILP system CIGOL employs inverse resolution; its search is guided by interaction with a human.

15.5 Inductive logic programming: algorithms

15.5.1 Computing the least general generalization of two clauses

The question that we are addressing here is: given two clauses, find a clause that (a) θ-subsumes these two clauses, and (b) is the least general of all clauses fulfilling condition (a).

Recall that $lgg(c_1, c_2) = c$ if and only if $c \preceq_\theta c_1$, $c \preceq_\theta c_2$, and for any c' such that $c' \preceq_\theta c_1$ and $c' \preceq_\theta c_2$ it must hold that $c' \preceq_\theta c$.

How can we compute the least general generalization of two clauses? The logician Gordon Plotkin proposed a procedure for this already in 1970. We describe it here in several steps.

First, we give a procedure for computing the lgg of any two *terms*. Two cases are considered here:

- Case 1: The two terms have the same functor and arity. Then the lgg operator is "pushed inside" the term, i.e., the functor is kept and the lgg operator is applied to the arguments.

$$lgg(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n)) = f(lgg(x_1, y_1), \ldots, lgg(x_n, y_n))$$

- Case 2: The two terms have different functors or arities. Then their lgg is a variable. The same variable should be used whenever the lgg of the same couple of terms is computed. For different couples, different variables are used. We express this here by subscripting the variable name with the terms of which it represents the lgg.

$$lgg(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)) = X_{f(x_1,\ldots,x_n),g(y_1,\ldots,y_m)}$$

Example 15.33 The lgg of $f(a, b, a, c)$ and $f(c, b, c, a)$ is computed as follows. The functors are the same, so we take the lgg's of the arguments. The first argument of the terms is a and c, respectively. These are different, so we introduce the variable X_{ac} to denote their lgg. The second argument is b in both cases, so we can keep it. The third argument is again a in the first term and c in the second term. The variable X_{ac} was already introduced to denote the lgg of these two. The last argument is c in the first clause and a in the second clause. Note that we cannot use X_{ac} here! We need a variable that stands for c in the first clause and a in the second clause. The variable X_{ac} stands for a in the first, and c in the second clause, which is different from what we need here. Hence, a new variable X_{ca} needs to be introduced. This gives as result: $f(X_{ac}, b, X_{ac}, X_{ca})$.

Of course can use any name for the variables; the names X_{ac} and X_{ca} are only used to help us remember what constants these variables generalize. Using the simpler names X for X_{ac} and Y for X_{ca}, we get as final result: $f(X, b, X, Y)$.

We can easily check that the result indeed generalizes both terms. It is supposed to theta-subsume both original terms, i.e., there should exist a θ_1 such that $f(X, b, X, Y)\theta_1 = f(a, b, a, c)$ and a θ_2 such that $f(X, b, X, Y)\theta_2 = f(c, b, c, a)$. With $\theta_1 = \{X/a, Y/c\}$ and $\theta_2 = \{X/c, Y/a\}$, this is indeed the case.[4] Further, since the result is supposed to be the least general generalization, we should not be able to find any term F and substitution θ such that $F\theta$ theta-subsumes both $f(a, b, a, c)$ and $f(c, b, c, a)$, and which is strictly less general (i.e., is subsumed by, and does not subsume) $f(X, b, X, Y)$. We can easily see that this is the case: any term subsuming both $f(a, b, a, c)$ and $f(c, b, c, a)$ must have a variable in the first, third and fourth

[4]Note that, generally, when we use the notation X_{ab} to denote the variable that generalizes over a in the first clause and b in the second clause, the substitution X_{ab}/a will always be in θ_1, and X_{ab}/b in θ_2.

argument, with the variable in the fourth argument differing from the others; such a term must at least theta-subsume $f(X, b, X, Y)$.

Having described the procedure for computing the lgg of two terms, we continue with how the lgg of two literals can be computed. We again consider two cases:

- Case 1: the literals have the same predicate symbol, same arity, and same sign. Then the lgg operator is pushed inside the literal, i.e., applied to the arguments of the literal.

$$lgg(p(x_1, \ldots, x_n), p(y_1, \ldots, y_n)) = p(lgg(x_1, y_1), \ldots, lgg(x_n, y_n))$$

- Case 2: either the name, arity, or sign of the literals differ. In that case, the lgg of the two literals is undefined.

Watch out for the difference between *terms* and *atoms*! A term refers to an object in the universe, and can only occur in a clause as an argument of a predicate. Two different terms can be generalized into a variable. A literal, on the other hand, represents a statement about one or more terms; it is not a term itself. It cannot occur as an argument of another literal, and it cannot be generalized into a variable. This difference is visible in the lgg algorithm: terms and atoms are treated differently.

Example 15.34 Assume f and g are functors and p and q are predicates. Then $f(1)$ is a term, while $p(1)$ is a literal. The lgg of $p(f(1))$ and $p(f(2))$ is $p(f(X))$. The lgg of $p(f(1))$ and $p(g(1))$ is $p(X)$: since $f(1)$ and $g(1)$ have different functor symbols, we need to substitute a variable X for them. The lgg of $p(1)$ and $p(2)$ is $p(X)$. The lgg of $p(1)$ and $q(1)$ is undefined. The lgg of $p(1)$ and $\neg p(1)$ is undefined.

Finally, the lgg of two clauses is defined as containing the lgg's (where defined) of each couple of literals where the first literal comes from the first clause and the second from the second clause.

$$lgg(c_1, c_2) = \{lgg(l_1, l_2) | l_1 \in c_1, l_2 \in c_2, lgg(c_1, c_2) \text{ is defined}\}$$

Example 15.35 The lgg of the clauses
c_1: $p(a), p(b), q(c) \leftarrow p(c)$
c_2: $p(b), p(c) \leftarrow p(d), q(c)$
which in set form are written as
c_1: $\{p(a), p(b), \neg p(c), q(c)\}$
c_2: $\{p(b), p(c), \neg p(d), \neg q(c)\}$
is computed by first computing the lgg of all couples of literals where the first literal comes from the first clause and the second from the second clause:
$lgg(p(a), p(b)) = p(X)$
$lgg(p(a), p(c)) = p(Y)$
$lgg(p(a), \neg p(d))$ is undefined
$lgg(p(a), \neg q(c))$ is undefined
$lgg(p(b), p(b)) = p(b)$
$lgg(p(b), p(c)) = p(Z)$
$lgg(p(b), \neg p(d))$ is undefined
$lgg(p(b), \neg q(c))$ is undefined
$lgg(\neg p(c), p(b))$ is undefined
$lgg(\neg p(c), p(b))$ is undefined
$lgg(\neg p(c), \neg p(d)) = \neg p(U)$
$lgg(\neg p(c), \neg q(c))$ is undefined
$lgg(q(c), p(b))$ is undefined

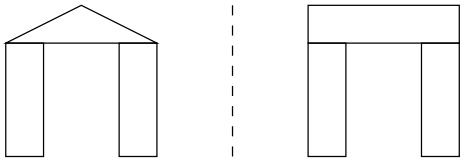

Figure 15.8: Two arches that are described by the two clauses of Example 15.36.

$lgg(q(c), p(c))$ is undefined
$lgg(q(c), \neg p(d))$ is undefined
$lgg(q(c), \neg q(c))$ is undefined
This gives as result the clause

$$\{p(X), p(Y), p(b), p(Z), \neg p(U)\}.$$

This is not a reduced clause: it is equivalent to

$$\{p(b), \neg p(U)\}$$

which in implication form is written as

$$p(b) \leftarrow p(U).$$

Exercise 15.2 Compute the lgg of the following pairs of clauses:

$p(X) \leftarrow q(X), r(Y, X)$	and	$p(X) \leftarrow q(X), r(X, Y)$
$p(a), p(b) \leftarrow p(X)$	and	$p(a), p(c) \leftarrow p(X)$
$gf(X, Y) \leftarrow f(X, Z), m(Z, Y)$	and	$gf(X, Y) \leftarrow f(X, Z), f(Z, Y)$
$p(a) \vee p(b)$	and	$p(b) \vee p(c)$

15.5.2 Generalization based on lgg

We here briefly discuss the role of the lgg in inductive logic programming, and how it can give rise to a practical learning approach.

In the setting of learning from entailment, where examples are clauses, the lgg of two examples represents the least specific clause that covers these two examples. In other words, when we construct a logical theory that covers these examples, either the examples are covered by different clauses, or the single clause covering both of them is a generalization of their lgg. In the latter case, we can just replace the two examples by their lgg: any clausal theory explaining the original example set must also explain the new one. The ILP system Golem learns clausal theories by repeatedly applying this principle (see further).

Example 15.36 To illustrate this, consider the following two clauses:

$$\begin{array}{lll} arch(o1) & \leftarrow & contains(o1, t1), shape(t1, triangle), on(t1, r1), on(t1, r2), \\ & & shape(r1, rectangle), orientation(r1, vertical), \\ & & shape(r2, rectangle), orientation(r2, vertical) \\ arch(o2) & \leftarrow & contains(o2, r1), shape(r1, rectangle), on(r1, r2), on(r1, r3), \\ & & shape(r2, rectangle), orientation(r2, vertical), \\ & & shape(r3, rectangle), orientation(r3, vertical) \end{array}$$

These clauses represent two constructions $o1$ and $o2$ that are considered arches. The properties of these constructions are shown in the bodies of the clauses; both arches contain elements that are triangles or rectangles, and some of these elements rest on other elements. The arches represented by these clauses are drawn in Figure 15.8.

Seeing these two examples of constructions that are arches, we can ask ourselves what their common properties are. Obviously, if there exists a definition of an arch

in the form of a single clause, then this definition must contain only conditions that both examples satisfy; in other words, this clausal definition must θ-subsume both clauses. Of course, there are many definitions that contain only conditions that both examples satisfy; a trivial one is the definition without any conditions, which just states that everything is an arch. This definition is probably too general. It seems more interesting to look for the definition that contains as many conditions as possible, and still contains only conditions that both examples satisfy; in other words, a clause that θ-subsumes both clauses but is as specific as possible. This is exactly the lgg of both clauses.

We leave it as an exercise for the reader to show that the lgg of both clauses above is:

$$\begin{aligned} arch(X) \quad \leftarrow \quad & contains(X,Y), shape(Y,Z), on(Y,U), on(Y,r2), \\ & shape(U,rectangle), orientation(U,vertical), \\ & shape(r2,rectangle), orientation(r2,vertical) \end{aligned}$$

which states that an arch is a construction that contains something (of any shape) that rests on two objects of rectangular shape with vertical orientation. Note that this can intuitively be seen as collecting all the common properties of the two original examples.

Exercise 15.3 Apply Plotkin's lgg algorithm to show that the lgg in the example is indeed correct.

Exercise 15.4 Explain why the `r2` symbol remains in the resulting clause, whereas other constants have disappeared.

15.5.3 Relative least general generalization

The least general generalization of two clauses gives us a single clause that captures as much as possible the similarities between the clauses.

In the practice of ILP, however, input data are often represented as single facts rather than clauses, and the literals that are in the body of the clauses occur in the background knowledge.

Example 15.37 The information in the previous example could also have been written as follows. Note that we are grouping the facts together per predicate, and that we are now avoiding the use of the same name for different objects in different pictures. (I.e., the double use of `r2` in the clausal representation is here avoided.)

```
% examples of the ‘‘arch’’ predicate

arch(o1).
arch(o2).

% background knowledge

contains(o1, t1).
contains(o2, r3).

shape(t1, triangle).
shape(r1, rectangle).
shape(r2, rectangle).
shape(r3, rectangle).
shape(r4, rectangle).
shape(r5, rectangle).
```

```
on(t1, r1).
on(t1, r2).
on(r3, r4).
on(r3, r5).

orientation(r1, vertical).
orientation(r2, vertical).
orientation(r4, vertical).
orientation(r5, vertical).
```

The *contains* predicate links the actual examples, o_1 and o_2, to objects described in the background knowledge.

While in practice objects are often described in this way, it is relatively easy to transform this representation to the clausal representation mentioned above. The simplest way to do this is to simply change the facts about the predicate to be learned into clauses by collecting all the facts implied by the background knowledge and including them in the body of the clause. The lgg of the thus constructed clauses will express the similarity between objects o_1 and o_2 relative to the background knowledge that we have about them. This leads to the definition of the **relative least general generalization** of two facts in the context of some background knowledge.

Definition 15.19 *The relative least general generalization of two clauses c_1 and c_2 given background knowledge expressed as a finite set of facts B, is denoted $rlgg(c_1, c_2, B)$ and defined as*

$$rlgg(c_1, c_2, B) = lgg(c_1 \cup B^\neg, c_2 \cup B^\neg) - B^\neg$$

where $B^\neg = \{\neg a : a \in B\}$.

If the clauses c_1 and c_2 are facts e_1 and e_2, then we can also write this as

$$rlgg(e_1, e_2, B) = lgg(e_1 \leftarrow B, e_2 \leftarrow B) - B^\neg.$$

Example 15.38 The rlgg of `arch(o1)` and `arch(o2)` relative to the background given earlier is:

$$rlgg(arch(o1), arch(o2), B) = lgg(arch(o1) \leftarrow B, arch(o2) \leftarrow B)$$

with $B = \{contains(o1, t1), contains(o2, r3),\ shape(t1, triangle),$
$shape(r1, rectangle),\ shape(r2, rectangle),\ shape(r3, rectangle),$
$shape(r4, rectangle),\ shape(r5, rectangle),\ orientation(r1, vertical),$
$orientation(r2, vertical),\ orientation(r4, vertical),\ orientation(r5, vertical),$
$on(t1, r1),\ on(t1, r2),\ on(r3, r4),\ on(r3, r5)\}$.

Computing the lgg of these long clauses, we will introduce variables to generalize constants in compatible places (i.e., constants that occur as the same argument of the same predicate with the same sign): this will give us variables $X_{o1,o2}$ and $X_{o2,o1}$ (note that both $o1$ and $o2$ occur in each clause, since they both occur in the background B which was included in these two clauses); $X_{t1,r3}$ and $X_{r3,t1}$; $X_{ri,rj}$ for all i, j with $i \neq j$. We simplify the notation a bit by using O_{ij} for X_{o_i,o_j}, R_{ij} for X_{r_i,r_j} and $T_{.i}$ and $T_{i.}$ for $X_{t1,ri}$ and $X_{ri,t1}$. The clause we then get is (we abbreviate *shape* as s, *orientation* as o, *rectangle* as r and *vertical* as v):

$arch(O_{12}) \leftarrow contains(O_{12}, T_{.3}), contains(O_{21}, T_{3.}), s(R_{12}, r), s(R_{13}, r), s(R_{14}, r),$
$s(R_{15}, r),\ s(R_{21}, r),\ s(R_{23}, r),\ s(R_{24}, r),\ s(R_{25}, r),\ s(R_{31}, r),\ s(R_{32}, r),\ s(R_{34}, r),$
$s(R_{35}, r),\ s(R_{41}, r),\ s(R_{42}, r),\ s(R_{43}, r),\ s(R_{45}, r),\ s(R_{51}, r),\ s(R_{52}, r),\ s(R_{53}, r),$

$s(R_{54}, r)$, $s(T_{.1}, S_{.1})$, $s(T_{.2}, S_{.2})$, $s(T_{.3}, S_{.3})$, $s(T_{.4}, S_{.4})$, $s(T_{.5}, S_{.5})$, $s(T_{1.}, S_{1.})$, $s(T_{2.}, S_{2.})$, $s(T_{3.}, S_{3.})$, $s(T_{4.}, S_{4.})$, $s(T_{5.}, S_{5.})$, $o(R_{12}, v)$, $o(R_{14}, v)$, $o(R_{15}, v)$, $o(R_{21}, v)$, $o(R_{24}, v)$, $o(R_{25}, v)$, $o(R_{41}, v)$, $o(R_{42}, v)$, $o(R_{45}, v)$, $o(R_{51}, v)$, $o(R_{52}, v)$, $o(R_{54}, v)$, $on(t1, R_{12})$, $on(T_{.3}, R_{14})$, $on(T_{.3}, R_{15})$, $on(t1, R_{21})$, $on(T_{.3}, R_{24})$, $on(T_{.3}, R_{25})$, $on(T_{3.}, R_{41})$, $on(T_{3.}, R_{42})$, $on(r3, R_{45})$, $on(T_{3.}, R_{51})$, $on(T_{3.}, R_{52})$, $on(r3, R_{54})$, B where B is the original background, which is always in the lgg, since it occurs in both clauses.

At this point, we can start simplifying the clause by computing its reduced clause. A first point to note is that there is a certain symmetry in the clause: for each R_{ij} there is an R_{ji} that occurs in exactly the same literals, and similarly for $T_{.i}$ and $T_{i.}$, O_{12} and O_{21}, etc. Thus, applying the substitutions O_{21}/O_{12}, R_{21}/R_{12}, $T_{3.}/T_{.3}$, ... yields a clause that is a strict subset of the original one, which gives us a first step towards the reduced clause.

Further, the subset of literals containing R_{14} and R_{15} has exactly the same structure as the subset containing R_{24} and R_{25}. Applying again the appropriate substitutions, we get

$arch(O_{12}) \leftarrow contains(O_{12}, T_{.3})$, $on(T_{.3}, R_{24})$, $on(T_{.3}, R_{25})$, $o(R_{24}, v)$, $o(R_{25}, v)$, $s(R_{24}, r)$, $s(R_{25}, r)$, $s(R_{12}, r)$, $o(R_{12}, v)$, $on(t1, R_{12})$, $s(R_{45}, r)$, $o(R_{45}, v)$, $on(r3, R_{45})$, $s(T_{.3}, S_{.3}), B$

Now note that any literals that are not "chained" to the head of the clause (by a chain of literals that share variables with each other) are redundant in the light of the background knowledge. For instance, collecting all literals with R_{45}, we have

$$s(R_{45}, r), o(R_{45}, v), on(r3, R_{45}),$$

and the background knowledge B contains

$$s(r4, r), o(r4, v), on(r3, r4).$$

Thus, applying the substitution $\{R_{45}/r4\}$ to the clause yields again a subclause. The same holds for the literals with R_{12}. Applying these substitutions gives:

$$arch(O_{12}) \leftarrow contains(O_{12}, T_{.3}), on(T_{.3}, R_{24}), o(R_{24}, v), s(R_{24}, r), s(T_{.3}, S_{.3}), B.$$

Simplifying the variable names a bit more and removing the background B from the clause we finally get

$$arch(O) \leftarrow contains(O, T), s(T, S), on(T, R), o(R, v), s(R, r)$$

which says that an arch contains an object T of any shape S[5] that is on some vertical rectangle R.

Note the condition in Definition 15.19 that the background knowledge B must be expressible as a finite set of facts. Indeed, B may not exist if B contains an infinite number of literals, as the following example shows.

Example 15.39 Given background knowledge consisting of the following definitions of `even` and `odd`, and two facts for the `div4` predicate, the rlgg of the `div4` facts relative to B is undefined:

```
even(0).
even(s(s(X))) :- even(X).
odd(s(0)).
odd(s(s(X))) :- odd(X).

div4(s(s(s(s(0))))).
div4(s(s(s(s(s(s(s(s(0)))))))))).
```

[5]Note that the condition that T has a shape is not dropped. While it may be obvious to the human reader that any object has a shape, this is not logically implied anywhere, so the condition is relevant.

15.5.4 The GOLEM algorithm

GOLEM is an ILP system that, given an input data set consisting of examples, all of which are clauses with a label "positive" or "negative" attached to them, induces a set of clauses that explain all the positive examples and none of the negative examples. It does this by repeatedly computing the lgg of two clauses.

(In practice, GOLEM represents examples as facts and computes the rlgg of these facts given some background B. As shown before, this corresponds to computing the *lgg* of clauses where B has been added to the body. We describe GOLEM here as if it were computing lgg's from clauses with a non-empty body.)

Let $\mathcal{C}$ be the set of all clauses, given a vocabulary of predicates and terms. We can then describe the learning task as follows:

Task description: induction according to GOLEM

Given: a set of positive examples $E^+ \subseteq \mathcal{C}$ (i.e., examples are clauses), and a set of negative examples $E^- \subseteq \mathcal{C}$;
Find: a set of clauses $H \subseteq \mathcal{C}$ such that $\forall e \in E^+ : \exists h \in H : h \preceq_\theta e$; $\forall h \in H, e \in E^- : h \not\preceq_\theta e$; and H is as simple as possible.

While there is an assumption that the "true" theory H has the above listed properties, there may be more than one theory having those properties (in fact, in practice there will usually be many). When we find a theory having all the required properties, there is no guarantee that it is the correct theory. Given this, GOLEM will look for some theory H that fulfills the requirements, and in addition, it will try to look for a H that is *as simple as possible* (following the principle of Occam's razor: why return a complex hypothesis if there is a simpler one that is also correct?). On the other hand, it does not guarantee that it finds the simplest possible H (e.g., the one with the least clauses, or the least number of literals in all clauses together). Giving this guarantee would make the algorithm computationally much heavier, and, obviously, there would still not be any guarantee that the "correct" H is found.

Note the following property, which was already alluded to:

Proposition 15.3 *If the clausal theory H contains a clause c that θ-subsumes two clauses e_1 and e_2, then c also θ-subsumes $lgg(e_1, e_2)$.*

This follows immediately from the definition of the *lgg*.

This means that if GOLEM can find two examples that are subsumed by the same clause in the true theory, then it can safely replace them by their lgg and continue computing the lgg of this new clause and any given clause, repeating this step and thus obtaining ever more general clauses until no further generalization is possible.

Note the condition "if GOLEM can find..." in the above sentence. How can we make sure that when we pick two examples, these are indeed covered by one clause in the true clausal theory?

The answer is: we cannot. We can, in some cases, be certain that they are *not* covered by the same clause: this must be true as soon as their lgg covers negative examples. Indeed, when the lgg of two clauses covers negative examples, then any clause generalizing both clauses will cover these negative examples, which contradicts the problem statement.

Thus, GOLEM will essentially proceed as follows. It follows a covering approach: it learns one rule at a time, and once a rule has been learned, positive examples covered by the rule are removed from the set of uncovered positives. Learning a

function GOLEM(E^+, E^-) **returns** set of clauses:
    $S := E^+$
    $H := \emptyset$
    repeat
        $c :=$ learn-one-clause(S, E^+, E^-)
        $H := H \cup \{c\}$
        $S := S - \{e \in S : c \preceq_\theta e\}$
    until S is empty
    return H

function learn-one-clause(S, E^+, E^-) **returns** clause:
    Select a positive example $e \in S$ at random
    $c := e$
    repeat
        $c^{prev} := c$
        $c :=$ generalize(c^{prev}, E^+, E^-)
    until $c = c^{prev}$
    return c

function generalize(c, E^+, E^-) **returns** clause:
    $c' := c$
    repeat
        Select a positive example $e \in E^+$ for which $c \not\preceq_\theta e$
        $c' :=$ lgg(c, e)
    until
        a c' has been found that covers no negatives ($\forall e \in E^- : c' \not\preceq_\theta e$)
        or a maximum number of attempts have been made
    if c' covers negatives **then return** c
    else return c'

Figure 15.9: The GOLEM algorithm.

single rule is done by first choosing an uncovered positive example, computing its lgg with another positive example, computing the lgg of the result with yet another positive example, etc., and continuing to do this until no further generalization is possible without covering negative examples. As there is a risk that no good lgg is found just because the example selected was badly chosen, the process of computing the lgg of the current clause with a random positive example is executed several times, up to a predefined number of times.

This yields the algorithm shown in Figure 15.9.

Exercise 15.5 Apply the GOLEM algorithm to the following dataset. You should end up with a theory of two clauses that cover all positives and no negatives.
Positive examples:

```
likes(garfield, lasagna) <- edible(lasagna)
likes(garfield, odie) <- alive(odie), dog(odie), naive(odie)
likes(garfield, tweety) <- alive(tweety), edible(tweety)
likes(garfield, jon) <- alive(jon), human(jon), naive(jon)
```

Negative examples:

```
likes(garfield, brutus) <- alive(brutus), dog(brutus)
likes(garfield, work) <-
likes(garfield, vet) <- alive(vet), human(vet)
```

15.5.5 The PROGOL algorithm

The PROGOL system has a similar goal as GOLEM, but is more advanced in a number of ways. We can describe the learning task as follows:

Given: a set of positive examples $E^+ \subseteq \mathcal{C}$ (i.e., examples are clauses), a set of negative examples $E^- \subseteq \mathcal{C}$, a background theory $B \subseteq \mathcal{C}$, and a hypothesis space $\mathcal{H} \subseteq \mathcal{P}(\mathcal{C})$ (i.e, hypotheses are sets of clauses),
Find: a hypothesis $H \in \mathcal{H}$ such that $\forall e \in E^+ : H \cup B \models e$ and $\forall e \in E^- : H \cup B \not\models e$, and moreover, $size(H)$ is minimal, where $size(H)$ is defined as the total number of literals occurring in H.

Note the following differences with GOLEM:

- GOLEM used θ-subsumption as the coverage relation: a clause c covers an example e if $c \preceq_\theta e$, and a hypothesis H covers e if at least one of its clauses covers it. PROGOL uses entailment as coverage relation: a hypothesis H covers an example e if $H \cup B \models e$. Entailment is a more complex relationship than θ-subsumption.

- PROGOL explicitly aims to find the most compact hypothesis H, i.e., the hypothesis with smallest size. This is not a crucial difference: GOLEM could in principle be modified to do this too, and PROGOL can in practice be relaxed by introducing heuristics so that it is likely to find a compact theory, without guaranteeing that it is the most compact one.

- PROGOL can use as background knowledge any logic program, whereas GOLEM is necessarily restricted to background knowledge that can be represented as a finite set of facts.

- PROGOL takes as input an explicit description of the hypothesis space $\mathcal{H}$, while GOLEM can construct any clause that is obtainable as a generalization of example clauses. We will see later how the hypothesis space is defined for PROGOL.

The way PROGOL works is as follows. On the highest level, it follows a covering approach just like GOLEM. To learn one rule, first it selects a random positive example e. Next, it constructs a so-called **bottom clause** for the example. This bottom clause, written $\bot_e$, is the *most specific clause in the hypothesis space* $\mathcal{H}$ *that covers this example* (where a hypothesis h covers an example e if $h \cup H' \cup B \models e$, with H' the part of the theory found up till now). The bottom clause of an example can be interpreted as "stating all the properties that the example has". The computation of this bottom clause is an issue that needs some further attention, we will return to it later on.

Once the bottom clause has been constructed, PROGOL conducts a top-down search in a restricted hypothesis space $\mathcal{H}_e$, which contains all the clauses $h \subseteq \bot_e$. Note that since $\bot_e$ is the most specific clause in $\mathcal{H}$ covering e, any clause covering e must be a subset of $\bot_e$. Hence, the hypothesis in which PROGOL searches, $\mathcal{H}_e$, contains all the h that cover e, i.e., all h for which $h \cup H' \cup B \models e$, with H' the part of the hypothesis found up till now.

The search conducted in the standard configuration of PROGOL is an exhaustive search, using the A^* algorithm, for the clause that most compacts the theory. The compaction caused by a clause is defined as the sum of the sizes of the examples covered by the clause, minus the size of the clause itself (where size, as mentioned before, is the number of literals in the clause).

The algorithm is shown in Figure 15.10.

function PROGOL(E^+, E^-, B) **returns** set of clauses:
    $S := E^+$
    $H := \emptyset$
    repeat
        $c :=$ learn-one-clause(S, E^-, B)
        $q :=$ compaction(c, S, B)
        if $q > 0$ **then**
            $H := H \cup \{c\}$
            $S := S - \{e \in S : c \preceq_\theta e\}$
    until S is empty **or** $q \leq 0$
    return H

function learn-one-clause(S, E^-, B) **returns** clause:
    Select a positive example $e \in S$ at random
    $\bot_e :=$ construct_bottom(e)
    perform an A^* search for the $h \subseteq \bot_e$ fulfilling:
        (1) $\forall e \in E^- : B \wedge h \not\models e$
        (2) compaction(h, S, B) is maximal
    return c

function compaction(h, S, B) **returns** integer:
    $S :=$ size(h)
    $C :=$ size($\{e \in S | h \wedge B \models e\}$)
    return $C - S$

Figure 15.10: The PROGOL algorithm

Constructing the bottom clause

The bottom clause is constructed as follows. First, for ease of notation, consider any previously found clauses H' to be currently part of the background B. Given an example e, we are looking for the most specific clause h such that $h \cup B \models e$.

If $h \wedge B \models e$, then $B \wedge \neg e \models \neg h$. Note that if h is a clause (a universally quantified disjunction), $\neg h$ is an existentially quantified conjunction.

Starting from B and $\neg e$, we can compute (e.g., using resolution) facts that follow deductively from it. Since all these facts follow from $B \wedge \neg e$, their conjunction does. Negating this conjunction gives us a clause h for which it holds that $B \wedge \neg e \models \neg h$, hence, $B \wedge h \models e$. Conversely, if $B \wedge h \models e$, then $B \wedge \neg e \models \neg h$: any h that explains e contains only literals the negation of which can be deduced from $B \wedge \neg e$.

Denoting the conjunction of *all* literals that can be deduced from B and $\neg e$ as $\neg \bot_e$, we have that $B \wedge \bot_e \models e$. Moreover, for any clause h such that $B \wedge h \models e$, since h contains only literals the negation of which can be deduced from $B \wedge \neg e$, and $\neg \bot_e$ contains all those literals, we have $h \subseteq \bot_e$.

Thus, any subset of $\bot_e$ entails e, and vice versa, any clause that entails e must be a subset of $\bot_e$. From this it follows that $\bot_e$ is the most specific clause that entails e.

Example 15.40 The following table shows a few examples of bottom clauses as they might be constructed by PROGOL:

```
:- modeh(1, class(+animal,#class))?
:- modeb(1, has_milk(+animal))?
:- modeb(1, has_gills(+animal))?
:- modeb(1, has_covering(+animal,#covering))?
:- modeb(1, has_legs(+animal,#nat))?
:- modeb(1, homeothermic(+animal))?
:- modeb(1, has_eggs(+animal))?
:- modeb(*, habitat(+animal,#habitat))?
```

Figure 15.11: Example of a mode specification for PROGOL.

B	e	$\bot_e$
anim(X) $\leftarrow$ *pet(X)* *pet(X)* $\leftarrow$ *dog(X)*	*nice(X)* $\leftarrow$ *dog(X)*	*nice(X)* $\leftarrow$ *pet(X), dog(X), anim(X)*
hasbeak(X) $\leftarrow$ *bird(X)* *bird(X)* $\leftarrow$ *vulture(X)*	*hasbeak(tweety)*	*hasbeak(tweety)* $\vee$ *bird(tweety)* $\vee$ *vulture(tweety)*

Note that any subset h of $\bot_e$ indeed has the property that together with B it entails e. For instance, for the first row, the clause *nice(X)* $\leftarrow$ *pet(X)* ("all pets are nice"), together with the background clause $pet(X) \leftarrow dog(X)$ (" all dogs are pets") entails the example clause e ("all dogs are nice"). Essentially, knowing that dogs are pets, it explains the fact that dogs are nice by hypothesizing that all pets are nice. (Note that this is the equivalent of a particular inverse resolution operator.)

PROGOL might of course just as well hypothesize that all animals are nice, which would be an even stronger generalization.

Similarly, in the second row, the fact that Tweety has a beak can be explained by any subset of the clause *hasbeak(tweety); bird(tweety); vulture(tweety)*. If Tweety is a vulture, then that explains why it has a beak (since the background knowledge states that vultures are birds and birds have beaks).

Specifying the hypothesis language

PROGOL can find any clause that is a subset of the bottom clause $\bot_e$ of any positive example e; hence, its search space is defined by these bottom clauses. The bottom clause of an example e contains "all" literals for which it holds that the negation of the literal can be deduced from $B \wedge \not e$. But in theory, there may be infinitely many such literals.

In practice, PROGOL uses a so-called **declarative language bias specification**. The **language bias** used by an ILP system during a particular run is simply the hypothesis space that it uses during that run. Many ILP systems allow the user to specify this language bias in a declarative way, using some formalism specially designed for describing a language bias.

The language bias specification formalism that PROGOL uses is based on a number of syntactic and semantic properties that clauses must have in order to be in the search space. More specifically, a language bias specification consists of a number of statements that PROGOL can exploit when building a bottom clause. These statements specify properties of the bottom clause, and hence, indirectly, also properties of all the clauses in the search space.

To describe the language bias, the user can specify *modes*, *types*, *determinacy*, and other properties of predicates or literals. We next discuss these. Figure 15.11 shows an example of a declarative language bias specification for PROGOL.

Modes The *modes* of a predicate are related to the execution strategy used by programming languages like Prolog. Running a program in Prolog is equal to asking

a query, which amounts to asking Prolog to determine the truth value of a literal or, more generally, to determine variable substitutions for the literal such that the substituted literal is true. When the Prolog engine is asked to compute the truth value or answer substitutions for a literal, we say that the literal is *called*. When Prolog computes the truth value of a conjunction or executes a clause, it calls the literals in that conjunction (or in the body of a clause) from left to right.

For some predicates, certain arguments will always be free (i.e., be equal to variables for which substitutions need to be computed) and others will always be instantiated (i.e., be equal to a term, which imposes a condition on the query) when the literal is called.

For instance, if we have a literal *age(P,A)* with P a person and A an integer, we could use it to ask Prolog for a given person the person's age, or to ask for a given number the persons having that age. If we have a literal $A < 12$, however, Prolog cannot compute all the A values for which $A < 12$ holds; this literal can only be called if some specific value for A has already been computed. Thus, Prolog would correctly handle the query

$$? - \texttt{person(P)}, \texttt{age(P, A)}, \texttt{A} < 12,$$

returning all those persons P and their age A for which it holds that $A < 12$. The `person` literal could be said to *generate* persons (values for P), which are then tested for their age. But Prolog cannot answer the query

$$? - \texttt{A} < 12.$$

The $<$ predicate cannot generate values for A. It can therefore only be called with arguments that are either constants (such as 12), or variables that are already bound to some value. (This is a property of Prolog, logically there is no difference.)

The **modes** of a predicate tell us in what way literals of that predicate can be called. Arguments of the literal can be input arguments (denoted `+`), output arguments (denoted `-`), or both. An input argument must be a ground term or must be a variable that occurred earlier as an output argument of another literal. An output argument must be a new variable (one that did not yet occur in the clause).

PROGOL allows the user to indicate whether an argument of a literal should be a ground term (denoted `#`), a variable that already occurs (denoted `+`) or a new variable (denoted `-`). This is done using the `modeb` and `modeh` predicates. `Modeb` indicates that a literal can occur in the body of a clause, `modeh` indicates that it can occur in the head of a clause. See Figure 15.11 for an example of the use of `modeb` and `modeh` in PROGOL. For this example, the clause

```
class(X,mammal) :- has_milk(X)
```

is mode-conform, whereas the clause

```
class(X,mammal) :- has_milk(Y)
```

is not (the variable `Y` is a new variable, which is not allowed by the mode declarations).

Types Besides indicating that variables need to be new or already occurring, we can also indicate the type of certain arguments. Assume, for instance, that we have a literal `age(X,Y)` in a clause. It may make sense to next add a literal `Y<18`, but not a literal `X<18`, since `X` is not a number but refers to a person.

To avoid useless tests, or more generally, the occurrence of variables on places where we know they do not make sense, it is possible to give a type to each argument of a literal. Constants or variables occurring there will automatically be of that type, and a variable can only occur in a position that has the right type.

Determinacy A last indication that is visible in the mode declarations of Figure 15.11 is the determinacy of each literal. When a literal has certain input and output modes, there is the question how many different outputs may correspond to a single input. A predicate is **deterministic** if exactly one output corresponds to each input. The predicate `habitat` in the example of Figure 15.11 is not deterministic: an animal may have multiple habitats (e.g., amphibians live in water as well as on land). The first argument of the `modeb`/ `modeh` declarations tells PROGOL how many different output arguments a literal can at most have for the same input arguments. It is never useful to include a literal with a given input more than this number of times in a clause. `1` indicates that a literal should occur at most once in a clause (for a given input), `*` means that there is no upper bound on how many times the literal can occur.

Thus, the mode declarations in Figure 15.11 allow a clause such as

```
class(X,mammal) :- habitat(X,water), habitat(X,land)
```

but not

```
class(X,mammal) :- has_covering(X,hair), has_covering(X, scales).
```

15.5.6 The FOIL algorithm

FOIL is one of the most widely used ILP systems. It is probably the most efficient and easiest-to-use system. On the other hand, its efficiency comes at a price: FOIL is less powerful than other ILP systems with respect to, for instance, its ability to handle non-ground background knowledge.

FOIL essentially follows a covering approach, learning rules one by one topdown in a manner very similar to that of CN2. The basic algorithm is shown in Figure 15.12. This algorithm is very similar to the covering algorithm used by, e.g., CN2. Yet, in a number of details, which are not visible in the high-level description of the algorithm, FOIL deviates quite a bit from CN2 in that it incorporates a number of particularities for solving problems that do not occur in the propositional setting.

One such particularity is the problem of deterministic literals. In propositional learning, we do not add a condition unless it improves the clause. In the first order logic setting, however, a literal may be useful even if it does not contribute by itself to the quality of the clause, by introducing a logical variable that is relevant for the prediction. But deterministic literals with a variable output argument are always true for each instance, and can therefore never improve the quality of a clause (since they don't reduce the coverage of the clause).

Example 15.41 Assume we need to classify persons, and part of the persons' description is their age, indicated by an *age* predicate. For instance, `age(bob,54)` indicates that Bob is 54 years old. The `age` predicate is determinate: everyone has an age.

Now imagine that the target predicate is of the form

```
can_drive_car(X) :- age(X,A), A > 18.
```

defining that to drive a car one has to be at least 18 years old.

FOIL introduces literals one by one. It cannot introduce the `A>18` literal before the variable A has been introduced by the `age` literal. But from the point of view of improving the accuracy of the rule, the `age` literal will never seem any good: by itself it does not impose any conditions on the rule. Therefore, using a standard heuristic such as entropy or accuracy, FOIL would never consider introducing such a predicate.

```
function FOIL(E+, E−) returns set of clauses:
    H := ∅
    stop := false
    repeat
        r := LearnOneRule(E+, E−)
        if r is good enough
        then add r to H
        else stop := true
    until stop
    return H

function LearnOneRule(E+, E−) returns clause:
    h := p(x1, ..., xD) ←
    stop := false
    repeat
        h' := BestExtension(h, E+, E−)
        if h' is better than h
        then h := h'
        else stop := true
    until stop
    return h
```

Figure 15.12: The FOIL algorithm.

The problem of determinate literals is solved by FOIL as follows: whenever a literal is added, the heuristic that is used to evaluate clauses assigns a bonus to the literal for each new variable that the literal introduces. The idea behind this is that introduction of new variables introduces possibly useful information into the clause.

(Note that PROGOL does not need a trick like this, simply because it searches for clauses in an exhaustive way, instead of heuristically. GOLEM does not need it because it works in an entirely different way, computing lgg's instead of using downward refinement.)

A second pitfall that FOIL manages to avoid using some special-purpose improvements is that of recursive clauses. When allowing recursive clauses in the search space, one has to take care that a clause such as `p(X) :- p(X)` is not incorrectly considered a solution. The clause is 100% accurate, in the sense that whenever the body is true, the head is true too; but it is obviously not a useful recursive clause. For details on how such recursive clauses are handled we refer to the literature [7].

15.5.7 The CLAUDIEN algorithm

The CLAUDIEN system is quite different from the earlier discussed approaches with respect to its goals. While the algorithms we saw up till now are supervised learners, and more specifically could be considered concept learners (where a concept is a predicate), CLAUDIEN is an unsupervised learning algorithm. Its problem statement is as follows.

Task description: Clausal discovery

Given: an interpretation I, and a language (set of clauses) $\mathcal{L}$
Find: all clauses in $\mathcal{L}$ that are true in I.

Note as a first difference with the previous approaches that the language $\mathcal{L}$ may contain any kind of clauses, they need not be definite Horn clauses. These clauses express patterns that hold in the data, but are not necessarily usable as part of a predicate definition.

Example 15.42 Given an interpretation

```
male(bart).
male(homer).
female(marge).
female(lisa).
father(homer,bart).
father(homer,lisa).
mother(marge,bart).
mother(marge,lisa).
parent(homer,bart).
parent(marge,bart).
parent(homer,lisa).
parent(marge,lisa).
```

some clauses that hold in these data and that might be returned by CLAUDIEN are:

```
:- male(X), female(X).
parent(X,Y) :- father(X,Y).
:- parent(X,X).
father(X,Y) :- parent(X,Y), male(X).
```

These clauses express the patterns that: no-one is both male and female; if X is the father of Y, X is a parent of Y; no-one is their own parent; and male parents are fathers.

The number of clauses in $\mathcal{L}$ may be huge, and as a result the CLAUDIEN system may return many clauses. The output of CLAUDIEN is therefore restricted by removing redundant clauses from it. For instance, in the above example, the clause $\leftarrow father(X,Y), mother(X,Y)$, which states that no-one can be someone's father and mother at the same time, is implied by the fact that fathers are male, mothers are female, and no-one is both male and female.

Taking this into account, the CLAUDIEN algorithm can be described as shown in Figure 15.13. The algorithm works top-down, starting with the empty clause $\square$, which is always false. It keeps a queue of clauses to be processed. When a clause is taken from the queue, CLAUDIEN tests whether the clause holds in the database. If it does, CLAUDIEN additionally checks whether the clause follows from the clauses already found. If yes, it is disregarded. If no, it is added to the set of solutions. If the clause does not hold in the database, this means it is too general: its refinements $\rho_{\mathcal{L}}(c)$, where $\rho_{\mathcal{L}}$ is a refinement operator under theta-subsumption that generates the whole language $\mathcal{L}$, are then computed and added to the queue.

CLAUDIEN uses a declarative language bias specification formalism known as DLAB. The DLAB formalism differs quite strongly from the mode-based formalism used by PROGOL. Again we refer to the literature for more details on this [11].

15.5.8 Other algorithms

Alternative problem settings

Many other ILP algorithms have been proposed throughout the years. The ones discussed above can be considered the most influential ones, yet all of them are

```
function CLAUDIEN(language L, intepretation I) returns set of clauses
    Q := {←}
    S := ∅
    while Q not empty:
        remove first element c from Q
        if I ⊨ c then
            if S ⊭ c then add c to S else do nothing
        else
            add all elements of ρ_L(c) to Q
    return S
```

Figure 15.13: The CLAUDIEN algorithm.

quite restricted if we look at them from the machine learning point of view.

Note that the goal of inductive logic programming algorithms, as presented in their problem statements, is typically to find a theory H that entails the positive examples and does not entail any negatives. First of all, note that this problem statement does not allow any noise in the data. While most systems have been extended to handle noise in one way or another, this is typically not reflected in the formal description of what the systems do.

Second, the learning of predicates is close to concept learning, which is itself close to classification. In machine learning, this is only one setting: we have also looked at regression, clustering, and other settings. None of the above algorithms are immediately suitable for handling these alternative settings. But a range of ILP systems have been proposed to address exactly this problem. The FORS system [23], for instance, learns clauses in which the head literal contains an argument that will be instantiated to a number when running the clause (this is the argument prediction setting mentioned before). Thus, the system performs regression.

It will not be surprising that, next to classification and regression, also clustering has been studied in a first order logic context.

Looking at all the different tasks and approaches that have been defined in machine learning from attribute-value data, the question naturally comes up whether all of these have their first-order logic counterparts, and whether ILP could profit from the knowledge that has already been gathered in the attribute-value context.

The upgrading story

When we look at attribute-value learning, we see that there is a very broad range of hypothesis representations (rules, decision trees, neural networks, ...), and for all of these representations learning algorithms have been proposed. Compared to this very wide range of algorithms, it is perhaps remarkable that the ILP systems described above are all rule learning systems.

In fact, in the context of logic programming, it is a natural choice to choose a clausal representation for the hypotheses that will be learned: this representation has been studied for decades and is very well-understood. Yet, the plethora of machine learning techniques that exists, and the knowledge that all of them have a different bias and that this bias influences the quality of the learned hypotheses, raises the question whether these some or all of these techniques could be "lifted" to the first order logic setting.

It turns out that this can be done in many cases. A generic "upgrading" methodology for lifting attribute-value learners to the first order logic setting was proposed in the nineties [41]. Nowadays, in ILP, the equivalent of decision tree learning is available, with the TILDE system [1] being probably the best known representative

of this approach. Discovery of association rules has been lifted to the first order setting by Dehaspe and Toivonen with the presentation of their WARMR system. An example of a first-order instance based learner is the RIBL system [16]. Ensemble methods (bagging, boosting, random forests, ...) have been lifted to the first order setting by several researchers; several people have studied clustering in first order logic, etc.

There are at least two research directions where a lot of work is going on right now, and which can be seen as upgrading knowledge from attribute-value learning to the first order logic context. One is the development of methods for probabilistic learning. Essentially, this requires combining methods for probabilistic inference with methods for first order logic inference. This combination is non-trivial: it has been studied by many researchers since the 1990's, and while significant advances have been made, much work remains. Among the most influential methods up till now are the Probabilistic Relational Models proposed end of the nineties by Koller et al. [19], and Markov Logic Networks [36], which have become very popular from around 2005 onwards.

A second important research direction is that of relational reinforcement learning. As argued in Chapter 14, reinforcement learning is set in a somewhat different problem setting, and it is useful to combine it with techniques that perform generalization, such that similarity between different states in which the agent may be can be exploited to improve learning performance. In many practical cases, the states in which the learning agent may be are not naturally described using the attribute-value framework; a structural description may be more appropriate. This leads to the idea of combining ILP, or relational learning, with reinforcement learning. That is the topic studied in the field of relational reinforcement learning, a field that has become popular in the late nineties.

Appendix A

Basics of Probability Theory

In this appendix we introduce some basic terminology and results from probability theory. We do not aim at a complete and mathematically precise treatment of probability theory, but rather at providing intuitions and practical rules that are sufficient for following the main text of these notes, and in particular Chapter 12 (Probabilistic Approaches).

A.1 Experiments, outcomes, events, and probabilities

We call a **stochastic experiment**, any process that yields a result, or **outcome**, in an non-deterministic way; that is, exactly the same process may give a different result when it is repeated. An **event** is a set of outcomes. We say that an **event occurs** when the stochastic experiment yields an outcome that is a member of the event.

If we conduct a series of n experiments, we can count the number of times n_E that some event E occurs. The **probability** of an event is the limit of the proportion n_E/n when n approaches infinity. When we speak of the probability of a single outcome e, we mean the probability of the event $E = \{e\}$, i.e., the singleton event containing only outcome e. We denote the probability of an event E as $Pr(E)$.

The interpretation of the probability of an event as the proportion of experiment executions where the event occurs is a very important one; it provides good intuitions for the rules of probability theory.

Example A.1 Rolling a die is a stochastic experiment. It has 6 possible outcomes (indicated 1, 2, 3, 4, 5, 6). When we say the probability of throwing a 3 is 1/6, we mean that when we throw this die very often, we expect that a 3 will occur in about 1/6 of the cases. Obviously, for a low number of die rolls, the 3 will not occur in exactly 1 out of 6 cases: if we roll once, it will occur in either 0 or 100% of the cases. But if we roll many times, we expect the proportion of threes to become very close to 1/6.

For a fair die, it is reasonable (for reasons of symmetry) to assume that no number has a higher probability than any other number. In that case, each outcome has a probability of 1/6. Generally, when we have n outcomes, all of which are equally probable, we can define the probability of an outcome as $1/n$, and the probability of an event E is then $|E|/n$, with $|E|$ the number of outcomes in E. This gives us a second intuitive meaning for probabilities, which we will also use in the remainder of this text. It is important to see, though, that the intuition of

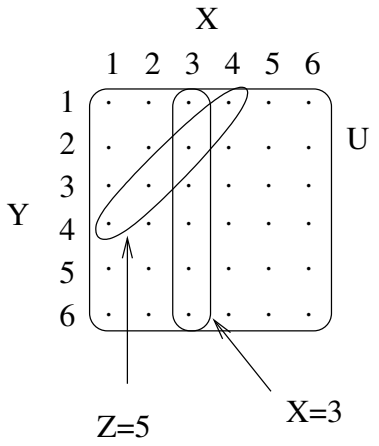

Figure A.1: The table shows the set U containing all possible outcomes of the experiment consisting of throwing two dice. For fair dice, each outcome has a probability of 1/36. If we call the numbers shown by the dice X and Y, and $Z = X + Y$, then the events "$X = 3$" and "$Z = 5$" are defined by the sets shown in the figure.

probability as "the proportion of outcomes that are a member of the event" is only valid if all outcomes are equally probable according to our first interpretation of probability (i.e., when you repeat the experiment many times, all of them occur equally frequently).

Example A.2 Consider the experiment of throwing two dice at the same time. The first die can show any number from 1 to 6, and these can occur in combination with any number from 1 to 6 on die 2. So we have $6 \times 6 = 36$ possible outcomes, which we will denote (i, j) with i the number shown by the first die and j that of the second die. For symmetry reasons, we can assume that each outcome is equally probable, hence each outcome has a probability of 1/36.

Let us use X to denote the outcome of the first die, Y to denote that of the second die, and Z to denote their sum: $Z = X + Y$. The event $X = 3$ corresponds to all outcomes where the first die is 3. The event $Z = 5$ is the set of all outcomes where $X + Y = 5$, which is $\{(1, 4), (2, 3), (3, 2), (4, 1)\}$.

Figure A.1 shows an overview of all possible outcomes of the experiment, and a few events that can be defined for it.

If A and B are events, their union and intersection are also events. The union of A and B, denoted $A \cup B$, contains all outcomes that are in A or B, and can hence be interpreted as "A **or** B". Similarly, the intersection $A \cap B$ contains all elements in both A and B and can be interpreted as "A **and** B". The complement of a set A, $\bar{A}$, contains all outcomes not in A and is interpreted as **not** A.

If A is a finite set, the probability of A is the sum of the probabilities of all the outcomes in A. (We will see later that for infinite sets the situation may be more complex.) This is easy to understand intuitively. Consider an event with two outcomes $A = \{a_1, a_2\}$. In a sequence of n outcomes of the experiment, let n_1 be the number of outcomes equal to a_1, and n_2 the number of outcomes equal to n_2; then the number of times A occurs is $n_A = n_1 + n_2$. From this we get $n_A/n = n_1/n + n_2/n$, and since this equality is valid for any n, in the limit for n going towards infinity we get $Pr(A) = Pr(\{a_1\}) + Pr(\{a_2\})$.

With this interpretation, it is easy to see that the following rule holds:

$$Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B) \tag{A.1}$$

Indeed, to obtain the probability of $A \cup B$, we can list all the outcomes in A and next all the outcomes in B, and add their probabilities; but any outcome that is in both A and B is then listed twice, and hence its probability has been added twice. To compensate, we need to subtract the probability of each element in the intersection of A and B once. This corresponds to subtracting $Pr(A \cap B)$.

Example A.3 Consider again Figure A.1. If we have fair dice, each outcome is equally likely, and there are 36 outcomes, so each outcome has a probability of 1/36. The probability of $X = 2$ is then obtained by listing all the outcomes where $X = 2$, which is $\{(2,1),(2,2),(2,3),(2,4),(2,5),(2,6)\}$, and adding their probabilities, to obtain $6/36 = 1/6$. Similarly, $Pr(Y = 5) = 1/6$. The probability of $X = 2$ **or** $Y = 5$ can be similarly obtained by counting all the outcomes in $X = 2 \cup Y = 5$ once, which gives 11/36. The same outcome is obtained by adding $Pr(X = 2)$ and $Pr(Y = 5)$ and subtracting $Pr(X = 2 \cap Y = 5)$, which is 1/36 as it contains only one outcome, (2,5): $6/36 + 6/36 - 1/36 = 11/36$.

Note that the empty set is also an event. It is an impossible event, because it can never occur that an outcome is observed that is a member of the empty set, and hence its probability is zero.

The set of all possible outcomes is also an event. Obviously, this event occurs each time the experiment is performed; hence, it has a probability of one.

Two events are called **mutually exclusive** if their intersection is empty. For mutually exclusive events, the probability of their union is the sum of their probabilities (since their intersection has zero probability).

Since a set A and its complement $\bar{A}$ always have an empty intersection, and the set of all outcomes has probability 1, we have $Pr(A) + Pr(\bar{A}) = Pr(A \cup \bar{A}) = 1$, and hence

$$Pr(\bar{A}) = 1 - Pr(A). \tag{A.2}$$

We know how to obtain the probability of the union of two events, based on the probability of the events themselves and that of their intersection. One might wonder whether the probability of the intersection of two events is also computable from the probability of the events themselves. Generally, this is not the case.

Take the events $X = 1$ and $X = 2$. Both have a probability of 1/6, but the probability of $X = 1$ **and** $X = 2$ is zero: one cannot throw a one and a two with the same die at the same time.

Now take the events $X = 2$ and $Y = 5$. Both have probability 1/6, and the probability of $X = 2$ **and** $Y = 5$ is 1/36, which is $1/6 \times 1/6$. In this case, the probability of the intersection of two events is the product of the probabilities of the events.

Finally, consider the events $X = 2$ and $Z = 4$. $Pr(X = 2) = 1/6$, $Pr(Z = 4) = 1/12$ (this is easily checked on Figure A.1). $Pr(X = 2 \cap Z = 4) = 1/18$.

Clearly the probability of the intersection of two events is generally not obtainable from that of the events themselves. Some events are exclusive and have no intersection, others overlap partially, in still other cases one set is a subset of the other, ... The particularities of the events must be taken into account when calculating the probability of an intersection.

A.2 Conditional probabilities

Up till now we defined the probability of an event as the proportion of cases, out of an infinite series of experiments, where the event occurs. We can similarly define

the concept of conditional probability. The **conditional probability of an event** A **given an event** B, denoted $Pr(A|B)$, is defined as follows. Imagine we repeat some experiment (for which A and B are meaningful events) n times, and we look at only those results where B occurred. $Pr(A|B)$ is the proportion of cases where A occurred among these (actually, the limit of this proportion for n going to infinity).

Example A.4 Imagine someone rolls the two dice and you don't see the answer; then the probability that an 11 was rolled is $2/36 = 1/18$ (see Figure A.1). But what if you accidentally saw that one of the dice was a 5? Then you have more information: you already know that the outcome is one of $\{(5,1), (5,2), (5,3), (5,4), (5,5), (5,6)\}$. The probability that a 11 was rolled, is now $1/6$, since one out of the six outcomes that are still possible, namely (5,6), is a member of the event $Z = 11$. This $1/6$ is called the conditional probability of $Z = 11$, given that $X = 5$.

Generally, the conditional probability of an event A given an event B is written $Pr(A|B)$. The above example already showed how such a conditional probability is computed: to find $Pr(A|B)$, list all the outcomes in B, and check which of these are also in A. Or, put differently: compute the probability of B and check what part of this is also in A, i.e., is in $A \cap B$. This gives:

$$Pr(A|B) = Pr(A \cap B)/Pr(B) \tag{A.3}$$

which can be seen as the definition of conditional probability.

From this definition immediately follows an alternative way to compute $Pr(A \cap B)$:

$$Pr(A \cap B) = Pr(A|B)Pr(B) \tag{A.4}$$

This rule is often used to compute $Pr(A \cap B)$. Note that this is not in contradiction with what was earlier said about not being able to compute $Pr(A \cap B)$ from $Pr(A)$ and $Pr(B)$. If we want to use this formula to compute $Pr(A \cap B)$, we need $Pr(A|B)$, which is itself related to the particular relationship between the events A and B. $Pr(A \cap B)$ and $Pr(A|B)$ are interchangeable in this sense: if we know one, we can compute the other (assuming $Pr(A)$ and $Pr(B)$ are known). But one of the two has to somehow follow from the problem statement itself.

A.3 Bayes' rule

Note that there is a certain symmetry in the $Pr(A \cap B)$ rule: since $A \cap B = B \cap A$, we can write

$$Pr(A \cap B) = Pr(A|B)Pr(B)$$

just as well as

$$Pr(B \cap A) = Pr(B|A)Pr(A)$$

and this implies that

$$Pr(A|B)Pr(B) = Pr(B|A)Pr(A)$$

which can be solved for $Pr(A|B)$:

$$Pr(A|B) = \frac{Pr(B|A)Pr(A)}{Pr(B)}. \tag{A.5}$$

This formula is called **Bayes' rule**. It gives us a method for "reversing the conditioning": we can compute the conditional probability of A given B from the conditional probability of B given A.

Example A.5 Consider again the example of the die rolls. It was intuitively quite easy to compute $Pr(Z = 11|X = 5)$, the conditional probability that $Z = 11$ given that $X = 5$. The reverse question would be: given that the experimenter tells us that an 11 was thrown, what is the probability that the first die, X, shows a five? We can compute this again from the table in Figure A.1: there are two cases where the sum is 11 and in one of these two cases, $X = 5$, so the requested probability is 1/2. The alternative is to use Bayes' rule:

$$Pr(X = 5|Z = 11) = Pr(Z = 11|X = 5)Pr(X = 5)/Pr(Z = 11) = \frac{1/6 \cdot 1/6}{2/36} = 1/2$$

Bayes' rule is important because in some practical circumstances, it may be obvious what $Pr(A|B)$ is, but not what $Pr(B|A)$ is. Bayes' rule tells us how to compute one from the other.

A.4 Rule of total probability

A final rule of probability is the so-called rule of total probability. This rule allows us to turn conditional probabilities into an unconditional one.

Consider the following problem. We have two boxes: Box 1 contains 5 white and 5 black balls, and Box 2 contains 2 white and 8 black balls. The boxes look the same from the outside. Someone chooses a box at random (i.e., each box is chosen with a probability of 1/2) and takes a random ball out of it. What is the probability that it looks white?

The probability of drawing a white ball if we draw it from Box 1 is 5/10. The probability of drawing a white ball from Box 2 is 2/10. If we consider the following events:

- W: drawing a white ball;
- B_1: draws a ball from Box 1;
- B_2: draws a ball from Box 2

then the probability of drawing a white ball given that we draw a ball from Box i is denoted as $Pr(W|B_i)$. So we have $Pr(W|B_1) = 5/10$ and $Pr(W|B_2) = 2/10$.

What is now $Pr(W)$, the probability of drawing a white ball from a randomly chosen box? To compute this, note that whenever we draw a ball, we draw it from either Box 1 or Box 2, and we never draw it from both at the same time. That is: W is the union of the event "drawing a white ball from Box 1", which is denoted $W \cap B_1$, and the event "drawing a white ball from Box 2". Hence we can write

$$W = (W \cap B_1) \cup (W \cap B_2)$$

and since the intersection between $W \cap B_1$ and $W \cap B_2$ is empty:

$$Pr(W) = Pr(W \cap B_1) + Pr(W \cap B_2) = Pr(W|B_1)Pr(B_1) + Pr(W|B_2)Pr(B_2).$$

We see that the unconditional probability of drawing a white ball is equal to a weighted average of the conditional probabilities of drawing a white ball given that it is drawn from a certain box, where the weights are the probabilities of choosing exactly that box.

This formula can be extended to the case where we have k events B_i, $i = 1, ..., k$ that are exhaustive and mutually exclusive, i.e., exactly one of them must happen when we perform the experiment, and we know for all i the conditional probability

of A given B_i, as well as the unconditional probability of each B_i. In that case we can compute the unconditional probability of A as

$$Pr(A) = \sum_i Pr(A|B_i)Pr(B_i). \tag{A.6}$$

This is known as the **rule of total probability**. Note that it requires that the B_i be exhaustive ($A \subseteq B_1 \cup \cdots \cup B_k$) and mutually exclusive ($B_i \cap B_j = \emptyset$ for all $i \neq j$).

Exercise A.1 Consider again our two boxes: Box 1 contains 7 white and 3 black balls, and Box 2 contains 3 white and 7 black balls. Someone chooses a box at random and takes a random ball out of it, and the ball looks white. Given this result, what is the probability that the person has chosen Box 1? (Combine the rule of total probability with Bayes' rule to obtain the answer.)

A.5 Generalizing the above rules

A conditional probability behaves just like any other probability. We call it "conditional" only when we want to make the conditions under which we describe the probability of an event explicit. That means that the above rules can be trivially extended to take into account explicit conditions. For instance, the rule of total probability

$$Pr(A) = \sum_i Pr(A|B_i)Pr(B_i). \tag{A.7}$$

holds just as well when the context under which these probabilities are computed fulfills some condition C. We can make this condition explicit by adding it to the condition part of all probabilities, which gives:

$$Pr(A|C) = \sum_i Pr(A|C, B_i)Pr(B_i|C). \tag{A.8}$$

Similarly, we can write:

$$Pr(\bar{A}|C) = 1 - Pr(A|C) \tag{A.9}$$

$$Pr(A \cap B|C) = Pr(A|B, C)Pr(B|C) \tag{A.10}$$

$$Pr(A|B, C) = \frac{Pr(B|A, C)Pr(A|C)}{Pr(B|C)} \tag{A.11}$$

A.6 Continuous variables

Until now we have considered discrete outcomes. We can also consider stochastic processes where the outcome is described as a continuous variable taking some value. Often, no probability is then assigned to a single value; only intervals are assigned probabilities.

Assume for instance that today the weather is sunny, and X indicates how long it will take before the first raindrop falls on our head. We can assign a probability to the event that this raindrop falls within a week; within a day; within 1 and 2 hours from now; or even to the event that would fall within 34899 and 34900 seconds from now. These probabilities will clearly be ever smaller: the probability that the first drop falls exactly during that one second between the 34899th and the 34900 second is very small. The probability that the first raindrop will fall *exactly 34899* seconds from how, and not a fraction of a microsecond sooner or later, must be zero.

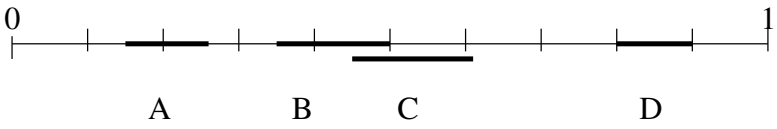

Figure A.2: When choosing a number between 0 and 1 totally randomly, without any preference for higher or lower numbers, intervals of equal length have an equal probability that the chosen number lies in them. More generally, the probability of an interval is proportional to its length. If the total interval (which has probability 1) has length 1, then the probability of each interval is equal to its length. Intervals A and D are of equal length and hence have the same probability (0.1), B and C have a probability of 0.15. The probability of $B \cup C$ is $Pr(B)+Pr(C)-Pr(B \cap C) = 0.25$, which is indeed the length of $B \cup C$.

But assigning a probability zero to this is a bit strange, in the sense that it suggests that it is impossible that the drop falls exactly on that timespot. And since this holds for any imaginable timespot, a logical conclusion would be that it can never rain!

The problem with this kind of reasoning is that while each single outcome has probability 0, there are infinitely many such outcomes, and if these form a continuous interval, they may "sum up" to a non-zero number. Mathematically, however, this kind of reasoning is difficult to describe. The cleanest solution is to simply not assign a probability to any particular outcome, but only to intervals.

When assigning probabilities to intervals, the usual rules from probability theory apply. Assume for instance that X is a variable that takes a random value between 0 and 1, uniformly. Uniformly here means that there is no preference for high values, low values, values in the middle, or whatever. So if we take any two intervals with the same width l, say $(a, a+l)$ and $(b, b+l)$, with $0 \leq a < a+l \leq 1$ and $0 \leq b < b+l \leq 1$, then the probability that X falls in the first interval is the same as the probability that it falls in the second interval. (If not, i.e., if the second interval would have a higher probability than the first, this would imply that somehow higher values are more likely than lower values, which contradicts our assumptions.) In this case, for an interval of width l, the probability that X takes a value in that interval is simply l. See Figure A.2 for an illustration.

Since intervals are sets of outcomes, just like other events, the rules we have seen for computing the probability of events apply for intervals just as well. For instance, the probability that X is a member of the union $I_1 \cup I_2$ of two intervals I_1 and I_2 is the sum of their probabilities minus the probability of their intersection:

$$Pr(I_1 \cup I_2) = Pr(I_1) + Pr(I_2) - Pr(I_1 \cap I_2).$$

The probability that X falls outside such an interval I_1 is $Pr(\bar{I_1}) = 1 - Pr(I_1)$. Etc.

Cumulative probability distribution, probability density

The probability of the interval $(-\infty, x]$, written $Pr((-\infty, x]))$, or $Pr(X < x)$ (where X refers to the outcome of the process), is called the **cumulative probability distribution** of the variable X, and written $F_X(x)$. The derivative of F_X is usually written f_X and is called the **probability density function** of X. This probability density function does not return a probability in itself, but one can obtain the probability of an interval $(a, b]$ by integrating f_X over that interval. Indeed, $\int_a^b f_X(x) = F_X(b) - F_X(a) = P(X \leq b) - P(X \leq a)$.

While probability densities are not probabilities, they behave in pretty much the same way for certain purposes, as we will see next.

A.7 Joint probability distributions

The **probability distribution** p_X of a variable X is a function that any value x to the probability that X takes the value x (if X is discrete), or to the probability density of X in x if X is continuous. Note that p_X is a function, and that $p_X(x)$ may be a probability or a probability density, depending on whether X is discrete or continuous. When it is clear what variable we are talking about, we will often drop the X subscript and just write $p(x)$ instead of $p_X(x)$.

Note that $p(x)$ can be interpreted as $Pr(X = x)$ when X is discrete and as $\lim_{\epsilon \to 0} Pr(x \leq X < x + \epsilon)/\epsilon$ when X is continuous.

If we have multiple variables $X_1, \ldots, X_D$, we define the **joint probability distribution** of $X_1, \ldots X_D$, denoted $p_{X_1, X_2, \ldots, X_D}$, or simply p when the subscript is clear from the context, as the function that maps any vector $\mathbf{x} = [x_1, \ldots, x_D]$ to the probability (density) of $X_1, \ldots X_D$ in $\mathbf{x}$. If all the X_i are discrete, this is easiest to interpret: then $p(\mathbf{x}) = Pr(X_1 = x_1, \ldots, X_D = x_D)$. If some X_i are continuous, then for each continuous X_i the condition $X_i = x_i$ is replaced by $x_i \leq X_i \leq X_i + \epsilon$, the probability is divided by ϵ^k with k the number of continuous variables, and the limit of this expression for $\epsilon \to 0$ is taken:

$$p(\mathbf{x}) = lim_{\epsilon \to 0} Pr(X_1 = x_1, x_2 \leq X_2 \leq x_2 + \epsilon, \ldots, X_D)/\epsilon^k$$

The rules for computing probabilities that we saw before remain valid for probability distributions and densities. This is obvious for discrete variables, where the $p(\mathbf{x})$ are probabilities, but it holds also for continuous and mixed distributions, under the condition that continuous variables are integrated instead of summed.

Take a simple two-dimensional distribution over variables X and Y. If X and Y are discrete, then the standard rule of total probability, written in terms of probability distributions, looks as follows:

$$p(x) = \sum_y p(x|y)p(y)$$

(Note that, strictly speaking, the three p's in the above formula are three different functions: in $p(x)$, p is in fact p_X, the distribution over X; in $p(y)$, p is p_Y, and in $p(x|y)$, p is $p_{X|Y}$, the conditional distribution of X given Y. We drop the subscripts because they are clear from how we write the arguments.)

If X is continuous, the same formula holds, though $p(x)$ and $p(x|y)$ are then probability densities rather than probabilities.

If Y is continuous, the sum over all possible values becomes an integral, and we get

$$p(x) = \int_{-\infty}^{\infty} p(x|y)p(y)dy$$

Using $p(x|y)p(y) = p(x, y)$, these rules of total probability can be rewritten as

$$p(x) = \sum_y p(x, y)$$

$$p(x) = \int_{-\infty}^{\infty} p(x, y)dy$$

$p_{X,Z}$		X						p_Z
		1	2	3	4	5	6	
	2	1/36	0	0	0	0	0	1/36
	3	1/36	1/36	0	0	0	0	1/18
	4	1/36	1/36	1/36	0	0	0	1/12
	5	1/36	1/36	1/36	1/36	0	0	1/9
	6	1/36	1/36	1/36	1/36	1/36	0	5/36
Z	7	1/36	1/36	1/36	1/36	1/36	1/36	1/6
	8	0	1/36	1/36	1/36	1/36	1/36	5/36
	9	0	0	1/36	1/36	1/36	1/36	1/9
	10	0	0	0	1/36	1/36	1/36	1/12
	11	0	0	0	0	1/36	1/36	1/18
	12	0	0	0	0	0	1/36	1/36
p_X		1/6	1/6	1/6	1/6	1/6	1/6	1

Figure A.3: Joint distribution of the variables X and Z from Example A.6. The marginal distributions of X and Z are visible in the margins of the table.

This shows how from a joint distribution over multiple variables, a distribution over a subset of the variables (in this case the single variable X) can be obtained by summing away, or integrating away, the variables that we are not interested in (in the above case, Y). The resulting distribution is called a **marginal distribution**, as opposed to the joint distribution we obtained it from.

Example A.6 Consider again the example of the die rolls, where X and Y are the numbers rolled and $Z = X + Y$. We can consider the joint probability distribution p_{XYZ} of X, Y, Z, or, for instance, its marginal distribution p_{XZ} over X and Z, which is itself a joint distribution with marginal distributions $p_X(x)$ and $p_Z(z)$. Figure A.3 shows $p_{XZ}(x, z)$ in tabular form, as well as its marginal distributions $p_X(x)$ and $p_Z(z)$, which are computed by summing up the rows, respectively columns, of $p_{XZ}(x, z)$.

Just like the total probability rule, Bayes' rule can also be rewritten in terms of probability distributions, giving:

$$p(x|y) = p(y|x)p(x)/p(y)$$

This formula follows immediately from the previous version of Bayes' rule if X and Y are discrete, but it holds just as well when X is continuous (hence $p(x)$ and $p(x|y)$ are densities), Y is continuous ($p(y|x)$ and $p(y)$ are densities), or both are continous (all p's are densities).

Further we have that for discrete variables the probabilities of all possible values sum to 1:

$$\sum_x p(x) = 1,$$

and in the case of continuous variables, that the integral over all values is 1:

$$\int_{-\infty}^{\infty} p(x)dx = 1.$$

While the above rules were illustrated with two variables X and Y, they hold also for more variables.

Example A.7 Take 5 die rolls X_1, X_2, X_3, X_4, X_5. The probability of rolling (1,5,3,6,2) is $1/61/61/61/6 = 1/6^5$. Generally, $p_{X_1,\ldots,X_5}(x_1, x_2, x_3, x_4, x_5) = 1/6^5$ for each

possible value of x_i from 1 to 6. The probability that $X_1 = 1, X_3 = 3, X_4 = 6$ is the sum of all $p_{X_1,\ldots,X_5}(1, x_2, 3, 6, x_5)$ for all values of x_2 and x_5 from 1 to 6. Thus $p_{X_1,X_3,X_4}(1, 3, 6) = \sum_{x_2=1\ldots6} \sum_{x_5=1\ldots6} p_{X_1,\ldots,X_5}(1, x_2, 3, 6, x_5)$.

Appendix B

List of Symbols

Set notation:	
$\mathbb{N}$	set of natural numbers
$\mathbb{R}$	set of real numbers
$[a, b]$	closed interval: $\{x \in R \vert a \leq x \leq b\}$
$(a, b]$	half-open interval: $\{x \in R \vert a < x \leq b\}$
$\mathbb{B}$	set of booleans ($\{true, false\}$)
2^S	powerset of S (set of all its subsets): $\{T \vert T \subseteq S\}$
$S \to T$	set of all functions from S to T
$x \in S$	x is an element of the set S
$S \subseteq T$	S is a subset of T (all elements of S are also in T)
$S \cup T$	the union of S and T
$S \cap T$	the intersection of S and T

Logic:	
$\vee$	disjunction ("or")
$\wedge$	conjunction ("and")
$\neg$	negation ("not")
$\to$	implication ("implies"); $x \to y$ is equivalent to $\neg x \vee y$
$\leftrightarrow$	equivalence ($x \leftrightarrow y$ is equivalent to $(x \wedge y) \vee (\neg x \wedge \neg y)$)
$\exists x \in S : Q(x)$	there exists an x in S for which Q holds
$\forall x \in S : Q(x)$	for all x in S, Q holds

Frequently used by convention:	
$\mathcal{X}$	instance space
$\mathbf{x}$	instance
y	scalar target value
$\mathbf{y}$	non-scalar target value
$\mathcal{Y}$	target space
$\mathbf{u}$	labeled instance
$\mathcal{U}$	target space
T	training set
$\mathcal{C}$	concept class (set of concepts)
H	hypothesis space
C	concept (subset of $\mathcal{X}$)
$\mathcal{I}$	input space of a learning system
$\mathcal{O}$	output space of a learning system

Probability theory:	
$Pr(A)$	probability of an event A
$Pr_{\mathbf{x} \sim \mathcal{D}}(A)$	probability of A when $\mathbf{x}$ is drawn from distribution $\mathcal{D}$

Bibliography

[1] H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.

[2] H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of clustering trees. In *Proceedings of the 15th International Conference on Machine Learning*, pages 55–63, 1998.

[3] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[4] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[5] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

[6] C.J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[7] R.M. Cameron-Jones and J.R. Quinlan. Avoiding pitfalls when learning recursive theories. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1050–1055. Morgan Kaufmann, 1993.

[8] B. Cestnik. Estimating probabilities: A crucial task in machine learning. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 147–149, London, 1990. Pitman.

[9] P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–284, 1989.

[10] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and other Kernel Based Methods*. Cambridge University Press, Cambridge, UK, 2000.

[11] L. Dehaspe and L. De Raedt. DLAB: A declarative language bias formalism. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems*, volume 1079 of *Lecture Notes in Artificial Intelligence*, pages 613–622. Springer-Verlag, 1996.

[12] N.M. Dempster, A.P. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society B*, 39:185–197, 1977.

[13] T. G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1924, 1998.

[14] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez. Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1-2):31–71, 1997.

[15] P. Domingos. Rule induction and instance-based learning: A unified approach. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1226–1232. Morgan Kaufmann, 1995.

[16] W. Emde and D. Wettschereck. Relational instance-based learning. In L. Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 122–130. Morgan Kaufmann, 1996.

[17] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987.

[18] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In L. Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1996.

[19] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1300–1307, Stockholm, Sweden, July 31 – August 6 1999. Morgan Kaufmann.

[20] J.A. Hartigan. *Clustering Algorithms*. Wiley New York, 1975.

[21] L. Hyafil and R.L. Rivest. Constructing optimal binary trees is np-complete. *Information processing letters*, 5:15–17, 1976.

[22] T. Joachims. *Learning to Classify Text using Support Vector Machines: Methods, Theory, and Algorithms*. Springer, 2002.

[23] A. Karalič and I. Bratko. First order regression. *Machine Learning*, 26:147–176, 1997.

[24] R. Kosala, H. Blockeel, M. Bruynooghe, and J. Van den Bussche. Information extraction from structured documents using k-testable tree automaton inference. *Data and Knowledge Engineering*, 58(2):129–158, 2006.

[25] P. Langley. *Elements of Machine Learning*. Morgan Kaufmann, 1996.

[26] T. Mitchell. *Machine Learning*. McGraw-Hill, 1996.

[27] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[28] R. Neapolitan. *Learning Bayesian networks*. Prentice Hall, Upper Saddle River, NJ, USA, 2003.

[29] D.J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases, 1998.

[30] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 2. edition, 1991.

[31] J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.

[32] G. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[33] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[34] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in Machine Learning. Morgan Kaufmann, 1993.

[35] Stefan Raeymaekers, Maurice Bruynooghe, and Jan Van den Bussche. Learning (k,l)-contextual tree languages for information extraction. In *ECML 2005, 16th European Conference on Machine Learning, Proceedings*, volume 3720 of *Lecture Notes in Computer Science*, pages 305–316. Springer, 2005. Best student paper award.

[36] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1–2):107–136, 2006.

[37] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14:465–471, 1978.

[38] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

[39] M. Teyssier and D. Koller. Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In *Proceedings of the 21st conference on Uncertainty in Artificial Intelligence*, pages 584–590. AUAI Press, 2005.

[40] K.M. Ting and I.H. Witten. Issues in stacked generalization. *Journal of Artificial Intelligence Research*, 10:271–289, 1999.

[41] W. Van Laer and L. De Raedt. How to upgrade propositional learners to first order logic: A case study. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 235–261. Springer-Verlag, 2001.

[42] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279 – 292, 1992.

[43] Christopher Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge., 1989.

[44] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005. 2nd Edition.

[45] D.H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[46] H. Zantema and H.L. Bodlaender. Finding small equivalent decision trees is hard. *International Journal of Foundations of Computer Science*, 11(2):343–354, 2000.