Machine Learning and Data Mining

8

If we define AI as is done in Elaine Rich's book [Ric83]:

> Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better.

and if we consider that the computer's learning ability is especially inferior to that of humans, then it follows that research into learning mechanisms, and the development of machine learning algorithms is one of the most important branches of AI.

There is also demand for machine learning from the viewpoint of the software developer who programs, for example, the behavior of an autonomous robot. The structure of the intelligent behavior can become so complex that it is very difficult or even impossible to program close to optimally, even with modern high-level languages such as PROLOG and Python.[1] Machine learning algorithms are even used today to program robots in a way similar to how humans learn (see Chap. 10 or [BCDS08, RGH+06]), often in a hybrid mixture of programmed and learned behavior.

The task of this chapter is to describe the most important machine learning algorithms and their applications. The topic will be introduced in this section, followed by important fundamental learning algorithms in the next sections. Theory and terminology will be built up in parallel to this. The chapter will close with a summary and overview of the various algorithms and their applications. We will restrict ourselves in this chapter to supervised and unsupervised learning algorithms. As an important class of learning algorithms, neural networks will be dealt with in Chap. 9. Due to its special place and important role for autonomous robots, reinforcement learning will also have its own dedicated chapter (Chap. 10).

[1]Python is a modern scripting language with very readable syntax, powerful data types, and extensive standard libraries, which can be used to this end.

W. Ertel, *Introduction to Artificial Intelligence*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-58487-4_8

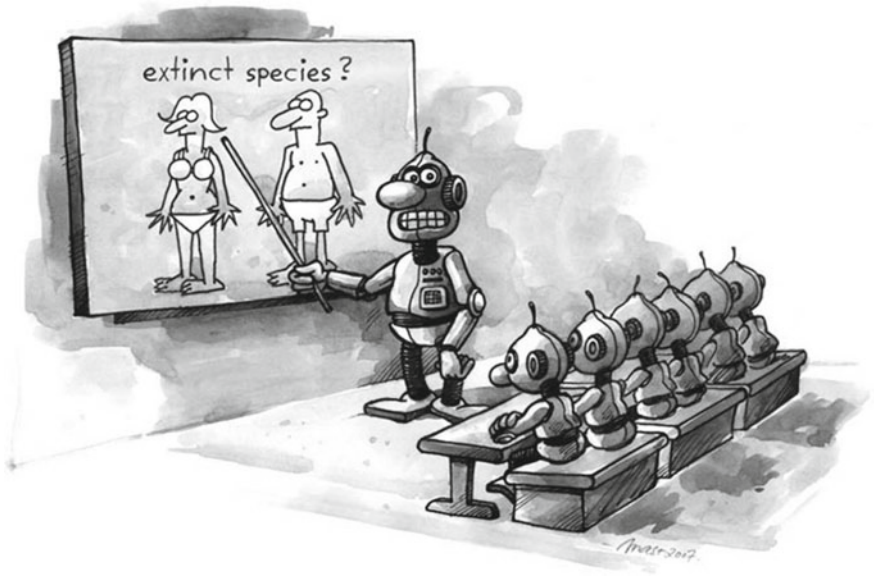


Fig. 8.1 Supervised learning …

What Is Learning? Learning vocabulary of a foreign language, or technical terms, or even memorizing a poem can be difficult for many people. For computers, however, these tasks are quite simple because they are little more than saving text in a file. Thus memorization is uninteresting for AI. In contrast, the acquisition of mathematical skills is usually not done by memorization. For addition of natural numbers this is not at all possible, because for each of the terms in the sum $x + y$ there are infinitely many values. For each combination of the two values x and y, the triple $(x, y, x + y)$ would have to be stored, which is impossible. For decimal numbers, this is downright impossible. This poses the question: how do we learn mathematics? The answer reads: The teacher explains the process and the students practice it on examples until they no longer make mistakes on new examples. After 50 examples the student (hopefully) understands addition. That is, after only 50 examples he can apply what was learned to infinitely many new examples, which to that point were not seen. This process is known as *generalization*. We begin with a simple example.

Example 8.1 A fruit farmer wants to automatically divide harvested apples into the merchandise classes A and B. The sorting device is equipped with sensors to measure two *features*, size and color, and then decide which of the two classes the apple belongs to. This is a typical *classification task*. Systems which are capable of dividing feature vectors into a finite number of classes are called *classifiers*.

To configure the machine, apples are hand-picked by a specialist, that is, they are classified. Then the two measurements are entered together with their class

Table 8.1 Training data for the apple sorting agent

Size [cm]	8	8	6	3	…
Color	0.1	0.3	0.9	0.8	…
Merchandise class	B	A	A	B	…

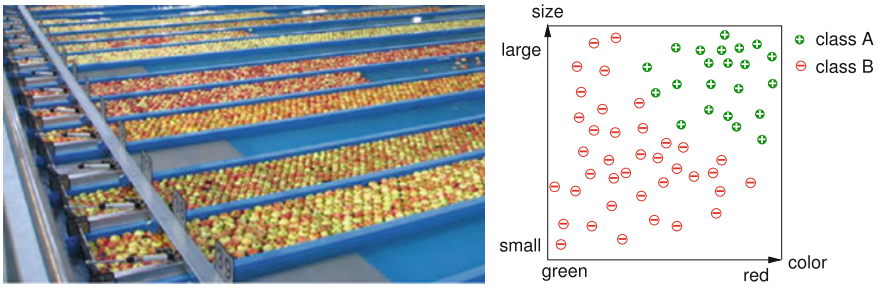


Fig. 8.2 BayWa company apple sorting equipment in Kressbronn and some apples classified into merchandise classes A and B in feature space (Photo: BayWa)

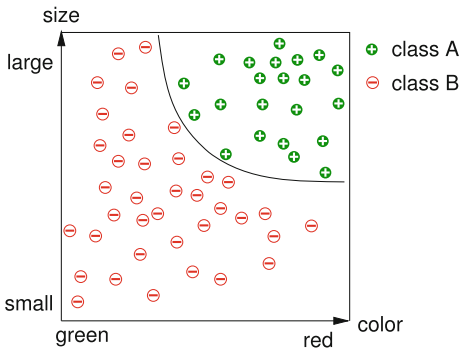


Fig. 8.3 The *curve* drawn in into the diagram divides the classes and can then be applied to arbitrary new apples

label in a table (Table 8.1. The size is given in the form of diameter in centimeters and the color by a numeric value between 0 (for green) and 1 (for red). A visualization of the data is listed as points in a scatterplot diagram in the right of Fig. 8.2.

The task in machine learning consists of generating a function from the collected, classified data which calculates the class value (A or B) for a new apple from the two features size and color. In Fig. 8.3 such a function is shown by the dividing line drawn through the diagram. All apples with a feature vector to the bottom left of the line are put into class B, and all others into class A.

In this example it is still very simple to find such a dividing line for the two classes. It is clearly a more difficult, and above all much less visualizable task, when the objects to be classified are described by not just two, but many features. In practice 30 or more features are usually used. For n features, the task consists of finding an $n - 1$ dimensional hyperplane within the n-dimensional *feature space*

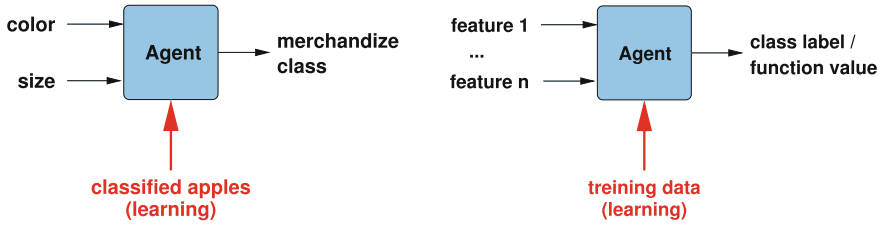


Fig. 8.4 Functional structure of a learning agent for apple sorting (*left*) and in general (*right*)

which divides the classes as well as possible. A "good" division means that the percentage of falsely classified objects is as small as possible.

A classifier maps a feature vector to a class value. Here it has a fixed, usually small, number of alternatives. The desired mapping is also called *target function*. If the target function does not map onto a finite domain, then it is not a classification, but rather an *approximation problem*. Determining the market value of a stock from given features is such an approximation problem. In the following sections we will introduce several learning agents for both types of mappings.

The Learning Agent We can formally describe a learning agent as a function which maps a feature vector to a discrete class value or in general to a real number. This function is not programmed, rather it comes into existence or changes itself during the *learning phase*, influenced by the *training data*. In Fig. 8.4 such an agent is presented in the apple sorting example. During learning, the agent is fed with the already classified data from Table 8.1 on page 177. Thereafter the agent constitutes as good a mapping as possible from the feature vector to the function value (e.g. merchandise class).

We can now attempt to approach a definition of the term "machine learning". Tom Mitchell [Mit97] gives this definition:

Machine Learning is the study of computer algorithms that improve automatically through experience.

Drawing on this, we give

Definition 8.1 An agent is a learning agent if it improves its performance (measured by a suitable criterion) on new, unknown data over time (after it has seen many training examples).

It is important to test the generalization capability of the learning algorithm on unknown data, the *test data*. Otherwise every system that just saved the training data would appear to perform optimally just by calling up the saved data. A learning agent is characterized by the following terms:

Task: the task of the learning algorithm is to learn a mapping. This could for example be the mapping from an apple's size and color to its merchandise class,

Fig. 8.5 Data Mining

but also the mapping from a patient's 15 symptoms to the decision of whether or not to remove his appendix.

Variable agent: (more precisely a class of agents): here we have to decide which learning algorithm will be worked with. If this has been chosen, thus the class of all learnable functions is determined.

Training data: (experience): the training data (sample) contain the knowledge which the learning algorithm is supposed to extract and learn. With the choice of training data one must ensure that it is a representative sample for the task to be learned.

Test data: important for evaluating whether the trained agent can generalize well from the training data to new data.

Performance measure: for the apple sorting device, the number of correctly classified apples. We need it to test the quality of an agent. Knowing the performance measure is usually much easier than knowing the agent's function. For example, it is easy to measure the performance (time) of a 10,000 meter runner. However, this does not at all imply that the referee who measures the time can run as fast. The referee only knows how to measure the performance, but not the "function" of the agent whose performance he is measuring.

What Is Data Mining? The task of a learning machine to extract knowledge from training data. Often the developer or the user wants the learning machine to make the extracted knowledge readable for humans as well. It is still better if the

developer can even alter the knowledge. The process of induction of decision trees in Sect. 8.4 is an example of this type of method.

Similar challenges come from electronic business and knowledge management. A classic problem presents itself here: from the actions of visitors to his web portal, the owner of an Internet business would like to create a relationship between the characteristics of a customer and the class of products which are interesting to that customer. Then a seller will be able to place customer-specific advertisements. This is demonstrated in a telling way at www.amazon.com, where the customer is recommended products which are similar to those seen in the previous visit. In many areas of advertisement and marketing, as well as in customer relationship management (CRM), data mining techniques are coming into use. Whenever large amounts of data are available, one can attempt to use these data for the analysis of customer preferences in order to show customer-specific advertisements. The emerging field of preference learning is dedicated to this purpose.

> The process of acquiring knowledge from data, as well as its representation and application, is called *data mining*. The methods used are usually taken from statistics or machine learning and should be applicable to very large amounts of data at reasonable cost.

In the context of acquiring information, for example on the Internet or in an intranet, text mining plays an increasingly important role. Typical tasks include finding similar text in a search engine or the classification of texts, which for example is applied in spam filters for email. In Sect. 8.7.1 we will introduce the widespread naive Bayes algorithm for the classification of text. A relatively new challenge for data mining is the extraction of structural, static, and dynamic information from graph structures such as social networks, traffic networks, or Internet traffic.

Because the two described tasks of machine learning and data mining are formally very similar, the basic methods used in both areas are for the most part identical. Therefore in the description of the learning algorithms, no distinction will be made between machine learning and data mining.

Because of the huge commercial impact of data mining techniques, there are now many sophisticated optimizations and a whole line of powerful data mining systems, which offer a large palette of convenient tools for the extraction of knowledge from data. Such a system is introduced in Sect. 8.10.

8.1 Data Analysis

Statistics provides a number of ways to describe data with simple parameters. From these we choose a few which are especially important for the analysis of training data and test these on a subset of the LEXMED data from Sect. 7.3. In this example dataset, the symptoms $x_1,\ldots, x_{15}$ of $N = 473$ patients, concisely described in

Table 8.2 Description of variables $x_1, \ldots, x_{16}$. A slightly different formalization was used in Table 7.2 on page 147

Var. num.	Description	Values
1	Age	Continuous
2	Sex (1 = male, 2 = female)	1, 2
3	Pain quadrant 1	0, 1
4	Pain quadrant 2	0, 1
5	Pain quadrant 3	0, 1
6	Pain quadrant 4	0, 1
7	Local muscular guarding	0, 1
8	Generalized muscular guarding	0, 1
9	Rebound tenderness	0, 1
10	Pain on tapping	0, 1
11	Pain during rectal examination	0, 1
12	Axial temperature	Continuous
13	Rectal temperature	Continuous
14	Leukocytes	Continuous
15	Diabetes mellitus	0, 1
16	Appendicitis	0, 1

Table 8.2 on page 181, as well as the class label—that is, the diagnosis (appendicitis positive/negative)—are listed. Patient number one, for example, is described by the vector

$$\boldsymbol{x}^1 = (26, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 37.9, 38.8, 23100, 0, 1)$$

and patient number two by

$$\boldsymbol{x}^2 = (17, 2, 0, 0, 1, 0, 1, 0, 1, 1, 0, 36.9, 37.4, 8100, 0, 0)$$

Patient number two has the leukocyte value $x_{14}^2 = 8100$.

For each variable x_i, its average $\bar{x}_i$ is defined as

$$\bar{x}_i := \frac{1}{N} \sum_{p=1}^{N} x_i^p$$

and the standard deviation s_i as a measure of its average deviation from the average value as

$$s_i := \sqrt{\frac{1}{N-1} \sum_{p=1}^{N} (x_i^p - \bar{x}_i)^2}.$$

The question of whether two variables x_i and x_j are statistically dependent (correlated) is important for the analysis of multidimensional data. For example, the covariance

Table 8.3 Correlation matrix for the 16 appendicitis variables measured in 473 cases

1.	−0.009	0.14	0.037	−0.096	0.12	0.018	0.051	−0.034	−0.041	0.034	0.037	0.05	−0.037	0.37	0.012
−0.009	1.	−0.0074	−0.019	−0.06	0.063	−0.17	0.0084	−0.17	−0.14	−0.13	−0.017	−0.034	−0.14	0.045	−0.2
0.14	−0.0074	1.	0.55	−0.091	0.24	0.13	0.24	0.045	0.18	0.028	0.02	0.045	0.03	0.11	0.045
0.037	−0.019	0.55	1.	−0.24	0.33	0.051	0.25	0.074	0.19	0.087	0.11	0.12	0.11	0.14	−0.0091
−0.096	−0.06	−0.091	−0.24	1.	0.059	0.14	0.034	0.14	0.049	0.057	0.064	0.058	0.11	0.017	0.14
0.12	0.063	0.24	0.33	0.059	1.	0.071	0.19	0.086	0.15	0.048	0.11	0.12	0.063	0.21	0.053
0.018	−0.17	0.13	0.051	0.14	0.071	1.	0.16	0.4	0.28	0.2	0.24	0.36	0.29	−0.0001	0.33
0.051	0.0084	0.24	0.25	0.034	0.19	0.16	1.	0.17	0.23	0.24	0.19	0.24	0.27	0.083	0.084
−0.034	−0.17	0.045	0.074	0.14	0.086	0.4	0.17	1.	0.53	0.25	0.19	0.27	0.27	0.026	0.38
−0.041	−0.14	0.18	0.19	0.049	0.15	0.28	0.23	0.53	1.	0.24	0.15	0.19	0.23	0.02	0.32
0.034	−0.13	0.028	0.087	0.057	0.048	0.2	0.24	0.25	0.24	1.	0.17	0.17	0.22	0.098	0.17
0.037	−0.017	0.02	0.11	0.064	0.11	0.24	0.19	0.19	0.15	0.17	1.	0.72	0.26	0.035	0.15
0.05	−0.034	0.045	0.12	0.058	0.12	0.36	0.24	0.27	0.19	0.17	0.72	1.	0.38	0.044	0.21
−0.037	−0.14	0.03	0.11	0.11	0.063	0.29	0.27	0.27	0.23	0.22	0.26	0.38	1.	0.051	0.44
0.37	0.045	0.11	0.14	0.017	0.21	−0.0001	0.083	0.026	0.02	0.098	0.035	0.044	0.051	1.	−0.0055
0.012	−0.2	0.045	−0.0091	0.14	0.053	0.33	0.084	0.38	0.32	0.17	0.15	0.21	0.44	−0.0055	1.

$$\sigma_{ij} = \frac{1}{N-1} \sum_{p=1}^{N} (x_i^p - \bar{x}_i)(x_j^p - \bar{x}_j)$$

gives information about this. In this sum, the summand returns a positive entry for the pth data vector exactly when the deviations of the ith and jth components from the average both have the same sign. If they have different signs, then the entry is negative. Therefore the covariance $\sigma_{12,13}$ of the two different fever values should be quite clearly positive.

However, the covariance also depends on the absolute value of the variables, which makes comparison of the values difficult. To be able to compare the degree of dependence in the case of multiple variables, we therefore define the *correlation coefficient*

$$K_{ij} = \frac{\sigma_{ij}}{s_i \cdot s_j}$$

for two values x_i and x_j, which is nothing but a normalized covariance. The matrix $\boldsymbol{K}$ of all correlation coefficients contains values between −1 and 1, is symmetric, and all of its diagonal elements have the value 1. The correlation matrix for all 16 variables is given in Table 8.3.

This matrix becomes somewhat more readable when we represent it as a density plot. Instead of the numerical values, the matrix elements in Fig. 8.6 on page 183 are filled with gray values. In the right diagram, the absolute values are shown. Thus we can very quickly see which variables display a weak or strong dependence. We can see, for example, that the variables 7, 9, 10 and 14 show the strongest correlation with the class variable *appendicitis* and therefore are more important for the diagnosis than the other variable. We also see, however, that the variables 9 and 10 are strongly correlated. This could mean that one of these two values is potentially sufficient for the diagnosis.

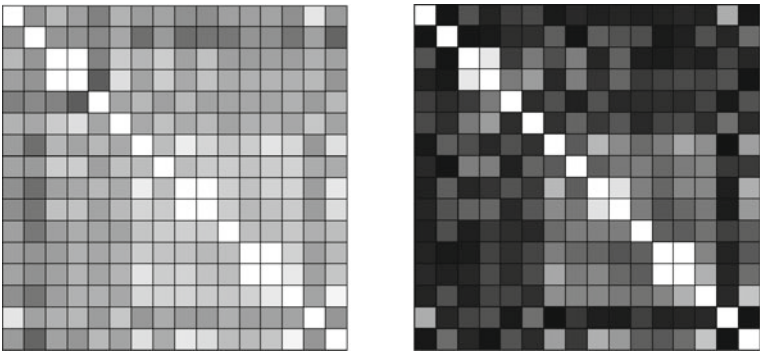


Fig. 8.6 The correlation matrix as a frequency graph. In the *left diagram*, *dark* stands for negative and *light* for positive. In the *right image* the absolute values were listed. Here *black* means $K_{ij} \approx 0$ (uncorrelated) and *white* $|K_{ij}| \approx 1$ (strongly correlated)

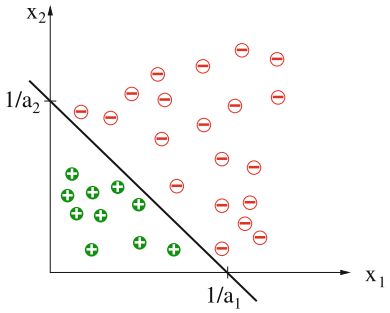


Fig. 8.7 A linearly separable two dimensional data set. The equation for the dividing straight line is $a_1x_1 + a_2x_2 = 1$

8.2 The Perceptron, a Linear Classifier

In the apple sorting classification example, a curved dividing line is drawn between the two classes in Fig. 8.3 on page 177. A simpler case is shown in Fig. 8.7. Here the two-dimensional training examples can be separated by a straight line. We call such a set of training data *linearly separable*. In n dimensions a hyperplane is needed for the separation. This represents a linear subspace of dimension $n - 1$.

Because every $(n - 1)$-dimensional hyperplane in $\mathbb{R}^n$ can be described by an equation

$$\sum_{i=1}^{n} a_i x_i = \theta$$

it makes sense to define linear separability as follows.

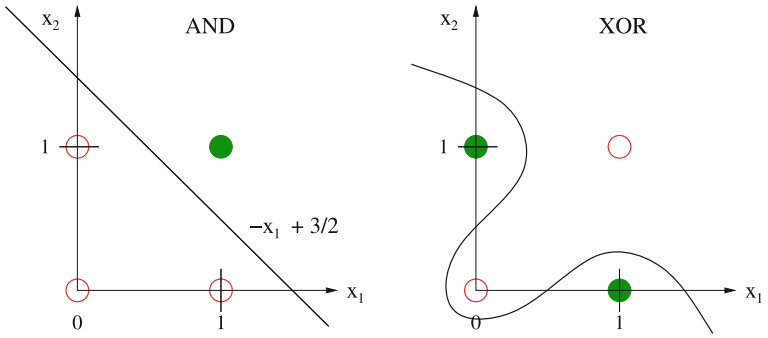


Fig. 8.8 The boolean function AND is linearly separable, but XOR is not (● ≙ true, ○ ≙ false)

Definition 8.2 Two sets $M_1 \subset \mathbb{R}^n$ and $M_2 \subset \mathbb{R}^n$ are called *linearly separable* if real numbers $a_1, \ldots, a_n, \theta$ exist with

$$\sum_{i=1}^{n} a_i x_i > \theta \quad \text{for all } \boldsymbol{x} \in M_1 \quad \text{and} \quad \sum_{i=1}^{n} a_i x_i \leq \theta \quad \text{for all } \boldsymbol{x} \in M_2.$$

The value θ is denoted the threshold.

In Fig. 8.8 we see that the AND function is linearly separable, but the XOR function is not. For AND, for example, the line $-x_1 + 3/2$ separates true and false interpretations of the formula $x_1 \wedge x_2$. In contrast, the XOR function does not have a straight line of separation. Clearly the XOR function has a more complex structure than the AND function in this regard.

With the perceptron, we present a very simple learning algorithm which can separate linearly separable sets.

Definition 8.3 Let $\boldsymbol{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$ be a weight vector and $\boldsymbol{x} \in \mathbb{R}^n$ an input vector. A *perceptron* represents a function P: $\mathbb{R}^n \rightarrow \{0, 1\}$ which corresponds to the following rule:

$$P(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}\, x = \sum_{i=1}^{n} w_i x_i > 0, \\ 0 & \text{else.} \end{cases}$$

The perceptron [Ros58, MP69] is a very simple classification algorithm. It is equivalent to a two-layer neural network with activation by a threshold function, shown in Fig. 8.9 on page 185. As shown in Chap. 9, each node in the network represents a neuron, and every edge a synapse. For now, however, we will only view

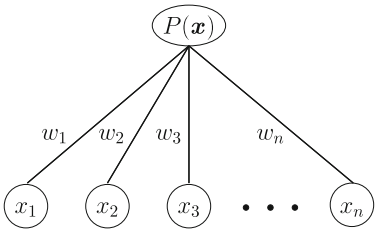


Fig. 8.9 Graphical representation of a perceptron as a two-layer neural network

the perceptron as a learning agent, that is, as a mathematical function which maps a feature vector to a function value. Here the input variables x_i are denoted *features*.

As we can see in the formula $\sum_{i=1}^{n} w_i x_i > 0$, all points $\boldsymbol{x}$ above the hyperplane $\sum_{i=1}^{n} w_i x_i = 0$ are classified as positive $(P(\boldsymbol{x}) = 1)$, and all others as negative $P(\boldsymbol{x}) = 0$. The separating hyperplane goes through the origin because $\theta = 0$. We will use a little trick to show that the absence of an arbitrary threshold represents no restriction of power. First, however, we want to introduce a simple learning algorithm for the perceptron.

8.2.1 The Learning Rule

With the notation M_+ and M_- for the sets of positive and negative training patterns respectively, the perceptron learning rule reads [MP69]

PERCEPTRONLEARNING$[M_+, M_-]$
$\boldsymbol{w}$ = arbitrary vector of real numbers
Repeat
  For all $\boldsymbol{x} \in M_+$
    If $\boldsymbol{w}\,\boldsymbol{x} \leq 0$ **Then** $\boldsymbol{w} = \boldsymbol{w} + \boldsymbol{x}$
  For all $\boldsymbol{x} \in M_-$
    If $\boldsymbol{w}\,\boldsymbol{x} > 0$ **Then** $\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{x}$
Until all $\boldsymbol{x} \in M_+ \cup M_-$ are correctly classified

The perceptron should output the value 1 for all $\boldsymbol{x} \in M_+$. By Definition 8.3 on page 184 this is true when $\boldsymbol{w}\,\boldsymbol{x} > 0$. If this is not the case then $\boldsymbol{x}$ is added to the weight vector $\boldsymbol{w}$, whereby the weight vector is changed in exactly the right direction. We see this when we apply the perceptron to the changed vector $\boldsymbol{w} + \boldsymbol{x}$ because

$$(\boldsymbol{w} + \boldsymbol{x}) \cdot \boldsymbol{x} = \boldsymbol{w}\,\boldsymbol{x} + \boldsymbol{x}^2.$$

If this step is repeated often enough, then at some point the value $\boldsymbol{w}\,\boldsymbol{x}$ will become positive, as it should be. Analogously, we see that, for negative training data, the perceptron calculates an ever smaller value

$$(\boldsymbol{w} - \boldsymbol{x}) \cdot \boldsymbol{x} = \boldsymbol{w}\boldsymbol{x} - \boldsymbol{x}^2$$

which at some point becomes negative.[2]

Example 8.2 A perceptron is to be trained on the sets $M_+ = \{(0, 1.8), (2, 0.6)\}$ and $M_- = \{(-1.2, 1.4), (0.4, -1)\}$. $\boldsymbol{w} = (1, 1)$ was used as an initial weight vector. The training data and the line defined by the weight vector $\boldsymbol{w}\boldsymbol{x} = x_1 + x_2 = 0$ are shown in Fig. 8.10 on page 187 in the first picture in the top row. In addition, the weight vector is drawn as a dashed line. Because $\boldsymbol{w}\boldsymbol{x} = 0$, this is orthogonal to the line.

In the first iteration through the loop of the learning algorithm, the only falsely classified training example is $(-1.2, 1.4)$ because

$$(-1.2, 1.4) \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 0.2 > 0.$$

This results in $\boldsymbol{w} = (1, 1) - (-1.2, 1.4) = (2.2, -0.4)$, as drawn in the second image in the top row in Fig. 8.10 on page 187. The other images show how, after a total of five changes, the dividing line lies between the two classes. The perceptron thus classifies all data correctly. We clearly see in the example that every incorrectly classified data point from M_+ "pulls" the weight vector $\boldsymbol{w}$ in its direction and every incorrectly classified point from M_- "pushes" the weight vector in the opposite direction.

It has been shown [MP69] that the perceptron always converges for linearly separable data. We have

Theorem 8.1 *Let classes M_+ and M_- be linearly separable by a hyperplane $\boldsymbol{w}\boldsymbol{x} = 0$. Then* PerceptronLearning *converges for every initialization of the vector w. The perceptron P with the weight vector so calculated divides the classes M_+ and M_-, that is*:

$$P(\boldsymbol{x}) = 1 \quad \Leftrightarrow \quad \boldsymbol{x} \in \boldsymbol{M}_+$$

and

$$P(\boldsymbol{x}) = 0 \quad \Leftrightarrow \quad \boldsymbol{x} \in \boldsymbol{M}_-.$$

As we can clearly see in Example 8.2, perceptrons as defined above cannot divide arbitrary linearly separable sets, rather only those which are divisible by a line through the origin, or in $\mathbb{R}^n$ by a hyperplane through the origin, because the constant term θ is missing from the equation $\sum_{i=1}^{n} w_i x_i = 0$.

[2] Caution! This is not a proof of convergence for the perceptron learning rule. It only shows that the perceptron converges when the training dataset consists of a single example.

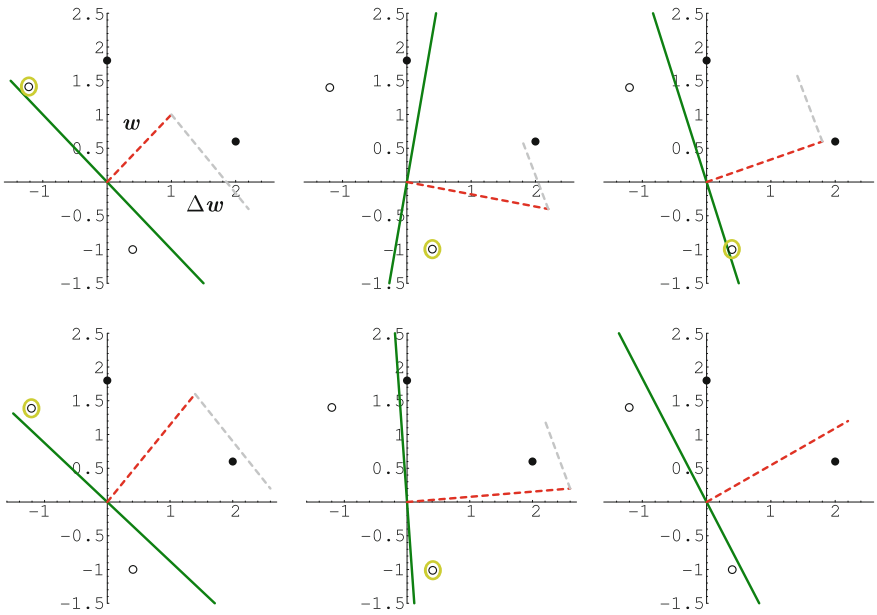


Fig. 8.10 Application of the perceptron learning rule to two positive (•) and two negative (◦) data points. The *solid line* shows the current dividing line $\boldsymbol{w}\boldsymbol{x} = 0$. The *orthogonal dashed line* is the weight vector $\boldsymbol{w}$ and the *second dashed line* the change vector $\Delta\boldsymbol{w} = \boldsymbol{x}$ or $\Delta\boldsymbol{w} = -\boldsymbol{x}$ to be added, which is calculated from the currently active data point surrounded in *light green*

With the following trick we can generate the constant term. We hold the last component x_n of the input vector $\boldsymbol{x}$ constant and set it to the value 1. Now the weight $w_n =: -\theta$ works like a threshold because

$$\sum_{i=1}^{n} w_i x_i = \sum_{i=1}^{n-1} w_i x_i - \theta > 0 \quad \Leftrightarrow \quad \sum_{i=1}^{n-1} w_i x_i > \theta.$$

Such a constant value $x_n = 1$ in the input is called a *bias unit*. Because the associated weight causes a constant shift of the hyperplane, the term "bias" fits well.

In the application of the perceptron learning algorithm, a bit with the constant value 1 is appended to the training data vector. We observe that the weight w_n, or the threshold θ, is learned during the learning process.

Now it has been shown that a perceptron P_θ: $\mathbb{R}^{n-1} \rightarrow \{0, 1\}$

$$P_\theta(x_1, \ldots, x_{n-1}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n-1} w_i x_i > \theta, \\ 0 & \text{else} \end{cases} \tag{8.1}$$

with an arbitrary threshold can be simulated by a perceptron P: $\mathbb{R}^n \rightarrow \{0, 1\}$ with the threshold 0. If we compare (8.1) with the definition of linearly separable, then we see that both statements are equivalent. In summary, we have shown that:

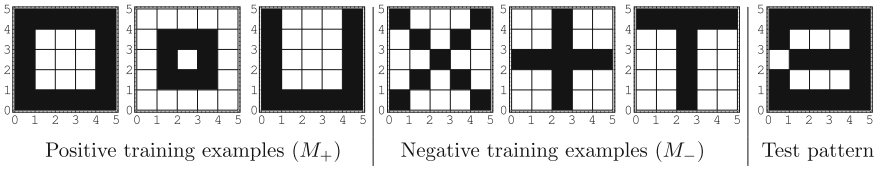


Fig. 8.11 The six patterns used for training. The *whole right pattern* is one of the 22 test patterns for the *first pattern* with a sequence of four inverted bits

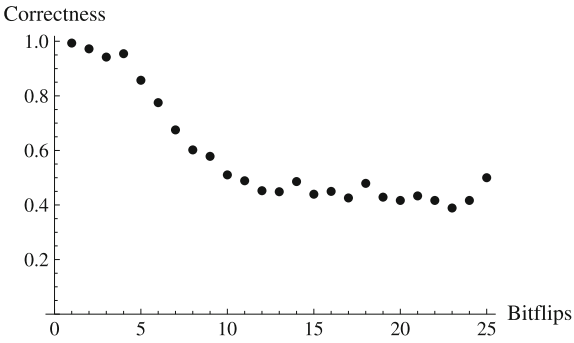


Fig. 8.12 Relative correctness of the perceptron as a function of the number of inverted bits in the test data

Theorem 8.2 *A function* $f\colon \mathbb{R}^n \to \{0, 1\}$ *can by represented by a perceptron if and only if the two sets of positive and negative input vectors are linearly separable.*

Example 8.3 We now train a perceptron with a threshold on six simple, graphical binary patterns, represented in Fig. 8.11, with 5×5 pixels each.

The training data can be learned by PerceptronLearning in four iterations over all patterns. Patterns with a variable number of inverted bits introduced as noise are used to test the system's generalization capability. The inverted bits in the test pattern are in each case in sequence one after the other. In Fig. 8.12 the percentage of correctly classified patterns is plotted as a function of the number of false bits.

After about five consecutive inverted bits, the correctness falls off sharply, which is not surprising given the simplicity of the model. In the next section we will present an algorithm that performs much better in this case.

8.2.2 Optimization and Outlook

As one of the simplest neural-network-based learning algorithms, the two-layer perceptron can only divide linearly separable classes. In Sect. 9.5 we will see that multi-layered networks are significantly more powerful. Despite its simple

structure, the perceptron in the form presented converges very slowly. It can be accelerated by normalization of the weight-altering vector. The formulas $\boldsymbol{w} = \boldsymbol{w} \pm \boldsymbol{x}$ are replaced by $\boldsymbol{w} = \boldsymbol{w} \pm \boldsymbol{x}/|\boldsymbol{x}|$. Thereby every data point has the same weight during learning, independent of its value.

The speed of convergence heavily depends on the initialization of the vector $\boldsymbol{w}$. Ideally it would not need to be changed at all and the algorithm would converge after one iteration. We can get closer to this goal by using the heuristic initialization

$$\boldsymbol{w_0} = \sum_{x \in M_+} \boldsymbol{x} - \sum_{\boldsymbol{x} \in \boldsymbol{M}_-} \boldsymbol{x},$$

which we will investigate more closely in Exercise 8.5 on page 239.

If we compare the perceptron formula with the scoring method presented in Sect. 7.3.1, we immediately see their equivalence. Furthermore, the perceptron, as the simplest neural network model, is equivalent to naive Bayes, the simplest type of Bayesian network (see Exercise 8.17 on page 242). Thus evidently several very different classification algorithms have a common origin.

In Chap. 9 we will become familiar with a generalization of the perceptron in the form of the back-propagation algorithm, which can divide non linearly separable sets through the use of multiple layers, and which possesses a better learning rule.

8.3 The Nearest Neighbor Method

For a perceptron, knowledge available in the training data is extracted and saved in a compressed form in the weights w_i. Thereby information about the data is lost. This is exactly what is desired, however, if the system is supposed to generalize from the training data to new data. Generalization in this case is a time-intensive process with the goal of finding a compact representation of data in the form of a function which classifies new data as good as possible.

Memorization of all data by simply saving them is quite different. Here the learning is extremely simple. However, as previously mentioned, the saved knowledge is not so easily applicable to new, unknown examples. Such an approach is very unfit for learning how to ski, for example. A beginner can never become a good skier just by watching videos of good skiers. Evidently, when learning movements of this type are automatically carried out, something similar happens as in the case of the perceptron. After sufficiently long practice, the knowledge stored in training examples is transformed into an internal representation in the brain.

However, there are successful examples of memorization in which generalization is also possible. During the diagnosis of a difficult case, a doctor could try to remember similar cases from the past. If his memory is sound, then he might hit upon this case, look it up in his files and finally come a similar diagnosis. For this approach the doctor must have a good feeling for *similarity*, in order to remember the most similar case. If he has found this, then he must ask himself whether it is similar enough to justify the same diagnosis.

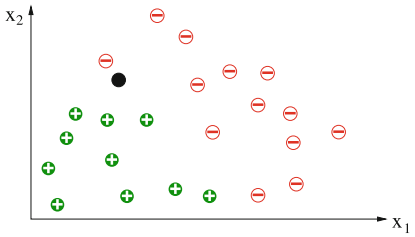


Fig. 8.13 In this example with negative and positive training examples, the nearest neighbor method groups the new point marked in black into the negative class

What does similarity mean in the formal context we are constructing? We represent the training samples as usual in a multidimensional feature space and define: *The smaller their distance in the feature space, the more two examples are similar.*

We now apply this definition to the simple two-dimensional example from Fig. 8.13. Here the next neighbor to the black point is a negative example. Thus it is assigned to the negative class.

The distance $d(\boldsymbol{x}, \boldsymbol{y})$ between two points $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^n$ can for example be measured by the Euclidean distance

$$d(\boldsymbol{x},\boldsymbol{y}) = |\boldsymbol{x} - \boldsymbol{y}| = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}.$$

Because there are many other distance metrics besides this one, it makes sense to think about alternatives for a concrete application. In many applications, certain features are more important than others. Therefore it is often sensible to scale the features differently by weights w_i. The formula then reads

$$d_w(\boldsymbol{x},\boldsymbol{y}) = |\boldsymbol{x} - \boldsymbol{y}| = \sqrt{\sum_{i=1}^{n} w_i(x_i - y_i)^2}.$$

The following simple *nearest neighbor classification* program searches the training data for the nearest neighbor $\boldsymbol{t}$ to the new example $\boldsymbol{s}$ and then classifies $\boldsymbol{s}$ exactly like $\boldsymbol{t}$.[3]

NEARESTNEIGHBOR[$M_+, M_-, \boldsymbol{s}$]
$\boldsymbol{t} = \text{argmin}_{\boldsymbol{x} \in M_+ \cup M_-}\{d(\boldsymbol{s}, \boldsymbol{x})\}$
If $\boldsymbol{t} \in M_+$ **Then Return** („+")
Else Return(„–")

[3]The functionals argmin and argmax determine, similarly to min and max, the minimum or maximum of a set or function. However, rather than returning the value of the maximum or minimum, they give the position, that is, the argument in which the extremum appears.

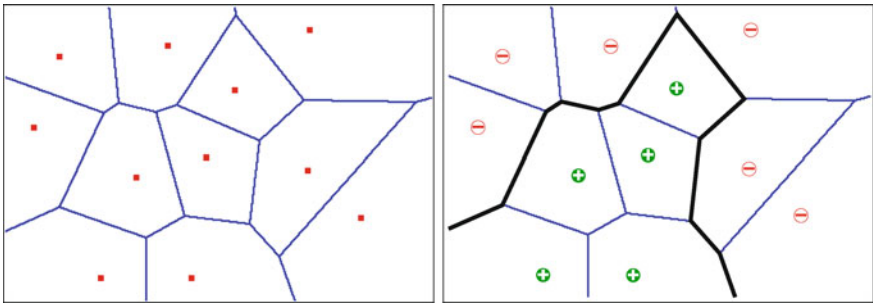


Fig. 8.14 A set of points together with their Voronoi-Diagram (*left*) and the *dividing line* generated for the two classes M_+ and M_-

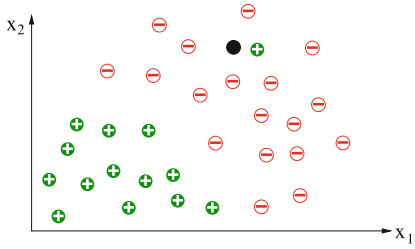


Fig. 8.15 The nearest neighbor method assigns the new point marked in *black* to the wrong (positive) class because the nearest neighbor is most likely classified wrong

In contrast to the perceptron, the nearest neighbor method does not generate a line that divides the training data points. However, an imaginary line separating the two classes certainly exists. We can generate this by first generating the so-called *Voronoi diagram*. In the Voronoi diagram, each data point is surrounded by a convex polygon, which thus defines a neighborhood around it. The Voronoi diagram has the property that for an arbitrary new point, the nearest neighbor among the data points is the data point, which lies in the same neighborhood. If the Voronoi diagram for a set of training data is determined, then it is simple to find the nearest neighbor for a new point which is to be classified. The class membership will then be taken from the nearest neighbor.

In Fig. 8.14 we see clearly that the nearest neighbor method is significantly more powerful than the perceptron. It is capable of correctly realizing arbitrarily complex dividing lines (in general: hyperplanes). However, there is a danger here. A single erroneous point can in certain circumstances lead to very bad classification results. Such a case occurs in Fig. 8.15 during the classification of the black point. The nearest neighbor method may classify this wrong. If the black point is immediately next to a positive point that is an outlier of the positive class, then it will be classified positive rather than negative as would be intended here. An erroneous fitting to random errors (noise) is called *overfitting*.

```
k-NearestNeighbor(M+, M−, s)
    V = {k nearest neighbors in M+ ∪ M−}
    If      |M+ ∩ V| > |M− ∩ V| Then    Return(„+")
    ElseIf  |M+ ∩ V| < |M− ∩ V| Then    Return(„−")
    Else                                Return(Random(„+", „−"))
```

Fig. 8.16 The k-NearestNeighbor Algorithm

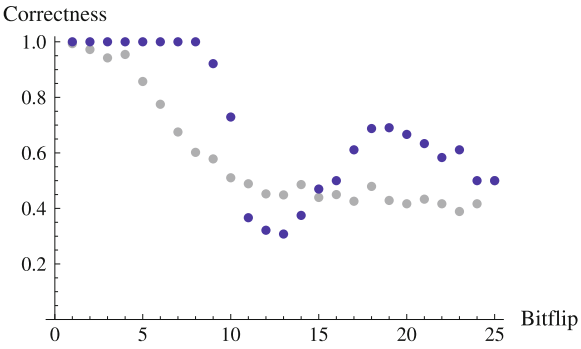


Fig. 8.17 Relative correctness of nearest neighbor classification as a function of the number of inverted bits. The structure of the curve with its minimum at 13 and its maximum at 19 is related to the special structure of the training data. For comparison the values of the perceptron from Example 8.3 on page 188 are shown in *gray*

To prevent false classifications due to single outliers, it is recommended to smooth out the division surface somewhat. This can be accomplished by, for example, with the k-NearestNeighbor algorithm in Fig. 8.16, which makes a majority decision among the k nearest neighbors.

Example 8.4 We now apply NearestNeighbor to Example 8.3 on page 188. Because we are dealing with binary data, we use the Hamming distance as the distance metric.[4] As a test example, we again use modified training examples with n consecutive inverted bits each. In Fig. 8.17 the percentage of correctly classified test examples is shown as a function of the number of inverted bits b. For up to eight inverted bits, all patterns are correctly identified. Past that point, the number of errors quickly increases. This is unsurprising because training pattern number 2 from Fig. 8.11 on page 188 from class M_+ has a hamming distance of 9 to the two training examples, numbers 4 and 5 from the other class. This means that the test pattern is very likely close to the patterns of the other class. Quite clearly we see that nearest neighbor classification is superior to the perceptron in this application for up to eight false bits.

[4]The Hamming distance between two bit vectors is the number of different bits of the two vectors.

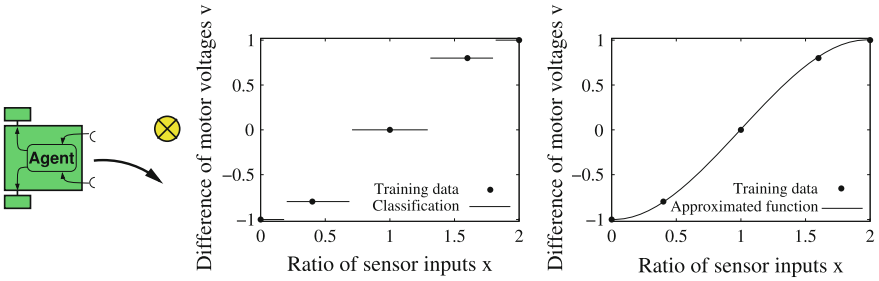


Fig. 8.18 The learning agent, which is supposed to avoid light (*left*), represented as a classifier (*middle*), and as an approximation (*right*)

8.3.1 Two Classes, Many Classes, Approximation

Nearest neighbor classification can also be applied to more than two classes. Just like the case of two classes, the class of the feature vector to be classified is simply set as the class of the nearest neighbor. For the k nearest neighbor method, the class is to be determined as the class with the most members among the k nearest neighbors.

If the number of classes is large, then it usually no longer makes sense to use classification algorithms because the size of the necessary training data grows quickly with the number of classes. Furthermore, in certain circumstances important information is lost during classification of many classes. This will become clear in the following example.

Example 8.5 An autonomous robot with simple sensors similar to the Braitenberg vehicles presented in Fig. 1.1 on page 2 is supposed to learn to move away from light. This means it should learn as optimally as possible to map its sensor values onto a steering signal which controls the driving direction. The robot is equipped with two simple light sensors on its front side. From the two sensor signals (with s_l for the left and s_r for the right sensor), the relationship $x = s_r / s_l$ is calculated. To control the electric motors of the two wheels from this value x, the difference $v = U_r - U_l$ of the two voltages U_r and U_l of the left and right motors, respectively. The learning agent's task is now to avoid a light signal. It must therefore learn a mapping f which calculates the "correct" value $v = f(x)$.[5]

For this we carry out an experiment in which, for a few measured values x, we find as optimal a value v as we can. These values are plotted as data points in Fig. 8.18 and shall serve as training data for the learning agent. During nearest neighbor classification each point in the feature space (that is, on the x-axis) is classified exactly like its nearest neighbor among the training data. The function for steering the motors is then a step function with large jumps (Fig. 8.18 middle). If we want finer steps, then we must provide correspondingly more training data. On

[5]To keep the example simple and readable, the feature vector $\boldsymbol{x}$ was deliberately kept one-dimensional.

the other hand, we can obtain a continuous function if we approximate a smooth function to fit the five points (Fig. 8.18 on page 193 right). Requiring the function f to be continuous leads to very good results, even with no additional data points.

For the approximation of functions on data points there are many mathematical methods, such as polynomial interpolation, spline interpolation, or the method of least squares. The application of these methods becomes problematic in higher dimensions. The special difficulty in AI is that model-free approximation methods are needed. That is, a good approximation of the data must be made without knowledge about special properties of the data or the application. Very good results have been achieved here with neural networks and other nonlinear function approximators, which are presented in Chap. 9.

The k nearest neighbor method can be applied in a simple way to the approximation problem. In the algorithm K-NEARESTNEIGHBOR, after the set $V = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k\}$ is determined, the k nearest neighbors average function value

$$\hat{f}(\boldsymbol{x}) = \frac{1}{k} \sum_{i=1}^{k} f(\boldsymbol{x}_i) \tag{8.2}$$

is calculated and taken as an approximation $\hat{f}$ for the query vector $\boldsymbol{x}$. The larger k becomes, the smoother the function $\hat{f}$ is.

8.3.2 Distance Is Relevant

In practical application of discrete as well as continuous variants of the k nearest neighbor method, problems often occur. As k becomes large, there typically exist more neighbors with a large distance than those with a small distance. Thereby the calculation of $\hat{f}$ is dominated by neighbors that are far away. To prevent this, the k neighbors are weighted such that the more distant neighbors have lesser influence on the result. During the majority decision in the algorithm K-NEARESTNEIGHBOR, the "votes" are weighted with the weight

$$w_i = \frac{1}{1 + \alpha d(\boldsymbol{x}, \boldsymbol{x_i})^2}, \tag{8.3}$$

which decreases with the square of the distance. The constant α determines the speed of decrease of the weights. Equation (8.2) is now replaced by

$$\hat{f}(\boldsymbol{x}) = \frac{\sum_{i=1}^{k} w_i f(\boldsymbol{x}_i)}{\sum_{i=1}^{k} w_i}.$$

For uniformly distributed concentration of points in the feature space, this ensures that the influence of points asymptotically approaches zero as distance increases. Thereby it becomes possible to use many or even all training data to classify or approximate a given input vector.

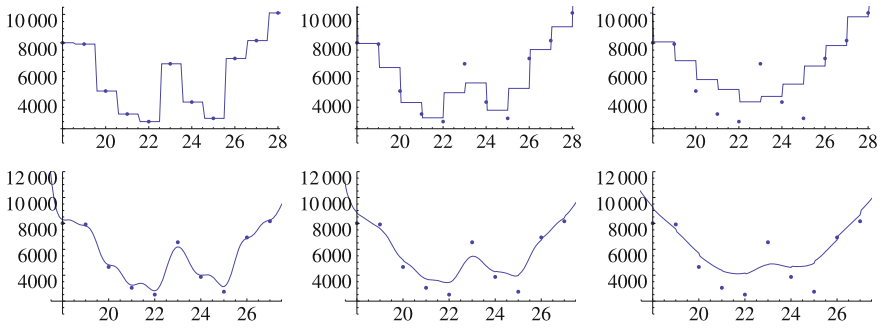


Fig. 8.19 Comparison of the k-nearest neighbor method (*upper row*) with $k = 1$ (*left*), $k = 2$ (*middle*) and $k = 6$ (*right*), to its distance weighted variant (*lower row*) with $\alpha = 20$ (*left*), $\alpha = 4$ (*middle*) and $\alpha = 1$ (*right*) on a one-dimensional dataset

To get a feeling for these methods, in Fig. 8.19 the k-nearest neighbor method (in the upper row) is compared with its distance weighted optimization. Due to the averaging, both methods can generalize, or in other words, cancel out noise, if the number of neighbors for k-nearest neighbor or the parameter α is set appropriately. The diagrams show nicely that the distance weighted method gives a much smoother approximation than k-nearest neighbor. With respect to approximation quality, this very simple method can compete well with sophisticated approximation algorithms such as nonlinear neural networks, support vector machines, and Gaussian processes.

There are many alternatives to the weight function (also called kernel) given in (8.3) on page 194. For example a Gaussian or similar bell-shaped function can be used. For most applications, the results are not very sensible on the selection of the kernel. However, the width parameter α which for all these functions has to be set manually has great influence on the results, as shown in Fig. 8.19. To avoid such an inconvenient manual adaptation, optimization methods have been developed for automatically setting this parameter [SA94, SE10].

8.3.3 Computation Times

As previously mentioned, training is accomplished in all variants of the nearest neighbor method by simply saving all training vectors together with their labels (class values), or the function value $f(\boldsymbol{x})$. Thus there is no other learning algorithm that learns as quickly. However, answering a query for classification or approximation of a vector $\boldsymbol{x}$ can become very expensive. Just finding the k nearest neighbors for n training data requires a cost which grows linearly with n. For classification or approximation, there is additionally a cost which is linear in k. The total computation time thus grows as $\Theta(n + k)$. For large amounts of training data, this can lead to problems.

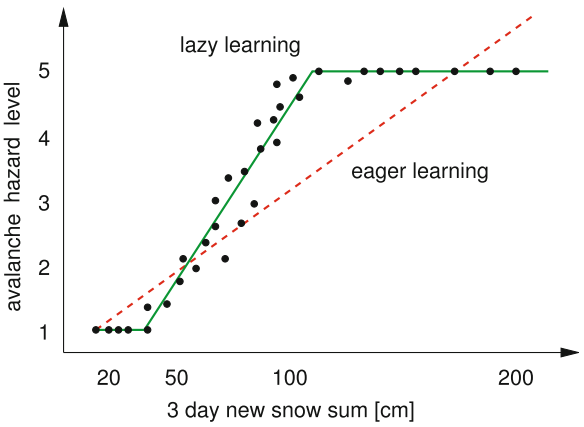


Fig. 8.20 To determine avalanche hazard, a function is approximated from training data. Here for comparison are a local model (*solid line*), and a global model (*dashed line*)

8.3.4 Summary and Outlook

Because nothing happens in the learning phase of the presented nearest neighbor methods, such algorithms are also denoted *lazy learning*, in contrast to *eager learning*, in which the learning phase can be expensive, but application to new examples is very efficient. The perceptron and all other neural networks, decision tree learning, and the learning of Bayesian networks are eager learning methods. Since the lazy learning methods need access to the memory with all training data for approximating a new input, they are also called *memory-based learning*.

To compare these two classes of learning processes, we will use as an example the task of determining the current avalanche hazard from the amount of newly fallen snow in a certain area of Switzerland.[6] In Fig. 8.20 values determined by experts are entered, which we want to use as training data. During the application of a eager learning algorithm which undertakes a linear approximation of the data, the dashed line shown in the figure is calculated. Due to the restriction to a straight line, the error is relatively large with a maximum of about 1.5 hazard levels. During lazy learning, nothing is calculated before a query for the current hazard level arrives. Then the answer is calculated from several nearest neighbors, that is, locally. It could result in the curve shown in the figure, which is put together from line segments and shows much smaller errors. The advantage of the lazy method is its locality. The approximation is taken locally from the current new snow level and not globally. Thus for fundamentally equal classes of functions (for example linear functions), the lazy algorithms are better.

Nearest neighbor methods are well suited for all problem situations in which a good local approximation is needed, but which do not place a high requirement on the speed of the system. The avalanche predictor mentioned here, which runs once per day, is such an application. Nearest neighbor methods are not suitable when a description of the knowledge extracted from the data must be understandable by

[6]The three day total of snowfall is in fact an important feature for determining the hazard level. In practice, however, additional attributes are used [Bra01]. The example used here is simplified.

Feature	Query	Case from case base
Defective part:	Rear light	Front light
Bicycle model:	Marin Pine Mountain	VSF T400
Year:	1993	2001
Power source:	Battery	Dynamo
Bulb condition:	ok	ok
Light cable condition:	?	ok
	Solution	
Diagnosis:	?	Front electrical contact missing
Repair:	?	Establish front electrical contact

Fig. 8.21 Simple diagnosis example for a query and the corresponding case from the case base

humans, which today is the case for many data mining applications (see Sect. 8.4). In recent years these memory-based learning methods are becoming popular, and various improved variants (for example locally weighted linear regression) have been applied [Cle79].

To be able to use the described methods, the training data must be available in the form of vectors of integers or real numbers. They are thus unsuitable for applications in which the data are represented symbolically, for example as first order logic formulas. We will now briefly discuss this.

8.3.5 Case-Based Reasoning

In case-based reasoning (CBR), the nearest neighbor method is extended to symbolic problem descriptions and their solutions. CBR is used in the diagnosis of technical problems in customer service or for telephone hotlines. The example shown in Fig. 8.21 about the diagnosis of a bicycle light going out illustrates this type of situation.

A solution is searched for the query of a customer with a defective rear bicycle light. In the right column, a case similar to the query in the middle column is given. This stems from the case base, which corresponds to training data in the nearest neighbor method. If we simply took the most similar case, as we do in the nearest neighbor method, then we would end up trying to repair the front light when the rear light is broken. We thus need a reverse transformation of the solution to the discovered similar problem back to the query. The most important steps in the solution to a CBR case are carried out in Fig. 8.22 on page 198. The transformation in this example is simple: rear light is mapped to front light.

As beautiful and simple as this methods seems in theory, in practice the construction of CBR diagnostic systems is a very difficult task. The three main difficulties are:

Modeling The domains of the application must be modeled in a formal context. Here logical monotony, which we know from Chap. 4, presents difficulties. Can the developer predict and map all possible special cases and problem variants?

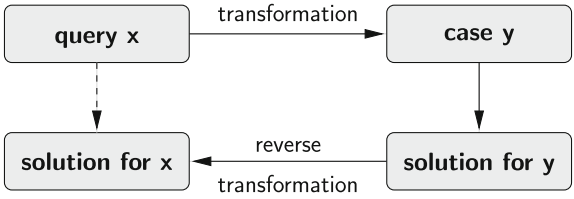


Fig. 8.22 If for a case x a similar case y is found, then to obtain a solution for x, the transformation must be determined and its inverse applied to the discovered case y

Similarity Finding a suitable similarity metric for symbolic, non-numerical features.
Transformation Even if a similar case is found, it is not yet clear how the transformation mapping and its inverse should look.

Indeed there are practical CBR systems for diagnostic applications in use today. However, due to the reasons mentioned, these remain far behind human experts in performance and flexibility. An interesting alternative to CBR are the Bayesian networks presented in Chap. 7. Often the symbolic problem representation can also be mapped quite well to discrete or continuous numerical features. Then the mentioned inductive learning methods such as decision trees or neural networks can be used successfully.

8.4 Decision Tree Learning

Decision tree learning is an extraordinarily important algorithm for AI because it is very powerful, but also simple and efficient for extracting knowledge from data. Compared to the two already introduced learning algorithms, it has an important advantage. The extracted knowledge is not only available and usable as a black box function, but rather it can be easily understood, interpreted, and controlled by humans in the form of a readable decision tree. This also makes decision tree learning an important tool for data mining.

We will discuss function and application of decision tree learning using the *C4.5* algorithm. C4.5 was introduced in 1993 by the Australian Ross Quinlan and is an improvement of its predecessor *ID3* (Iterative Dichotomiser 3, 1986). It is freely available for noncommercial use [Qui93]. A further development, which works even more efficiently and can take into account the costs of decisions, is C5.0 [Qui93].

The *CART* (Classification and Regression Trees, 1984) system developed by Leo Breiman [BFOS84] works similarly to C4.5. It has a convenient graphical user interface, but is very expensive.

Twenty years earlier, in 1964, the CHAID (Chi-square Automatic Interaction Detectors) system, which can automatically generate decision trees, was introduced by J. Sonquist and J. Morgan. It has the noteworthy characteristic that it stops the growth of the tree before it becomes too large, but today it has no more relevance.

Table 8.4 Variables for the skiing classification problem

Variable	Value	Description
Ski (goal variable)	yes, no	Should I drive to the nearest ski resort with enough snow?
Sun (feature)	yes, no	Is there sunshine today?
Snow_Dist (feature)	≤ 100, >100	Distance to the nearest ski resort with good snow conditions (over/under 100 km)
Weekend (feature)	yes, no	Is it the weekend today?

Also interesting is the data mining tool *KNIME* (Konstanz Information Miner), which has a friendly user interface and, using the *WEKA* Java library, also makes induction of decision trees possible. In Sect. 8.10 we will introduce KNIME.

Now we first show in a simple example how a decision tree can be constructed from training data, in order to then analyze the algorithm and apply it to the more complex LEXMED example for medical diagnosis.

8.4.1 A Simple Example

A devoted skier who lives near the high sierra, a beautiful mountain range in California, wants a decision tree to help him decide whether it is worthwhile to drive his car to a ski resort in the mountains. We thus have a two-class problem *ski yes/no* based on the variables listed in Table 8.4.

Figure 8.23 on page 200 shows a *decision tree* for this problem. A decision tree is a tree whose inner nodes represent features (attributes). Each edge stands for an attribute value. At each leaf node a class value is given.

The data used for the construction of the decision tree are shown in Table 8.5 on page 200. Each row in the table contains the data for one day and as such represents a sample. Upon closer examination we see that row 6 and row 7 contradict each other. Thus no deterministic classification algorithm can correctly classify all of the data. The number of falsely classified data must therefore be ≥ 1. The tree in Fig. 8.23 on page 200 thus classifies the data optimally.

How is such a tree created from the data? To answer this question we will at first restrict ourselves to discrete attributes with finitely many values. Because the number of attributes is also finite and each attribute can occur at most once per path, there are finitely many different decision trees. A simple, obvious algorithm for the construction of a tree would simply generate all trees, then for each tree calculate the number of erroneous classifications of the data, and at the end choose the tree with the minimum number of errors. Thus we would even have an optimal algorithm (in the sense of errors for the training data) for decision tree learning.

The obvious disadvantage of this algorithm is its unacceptably high computation time, as soon as the number of attributes becomes somewhat larger. We will now develop a heuristic algorithm which, starting from the root, recursively builds

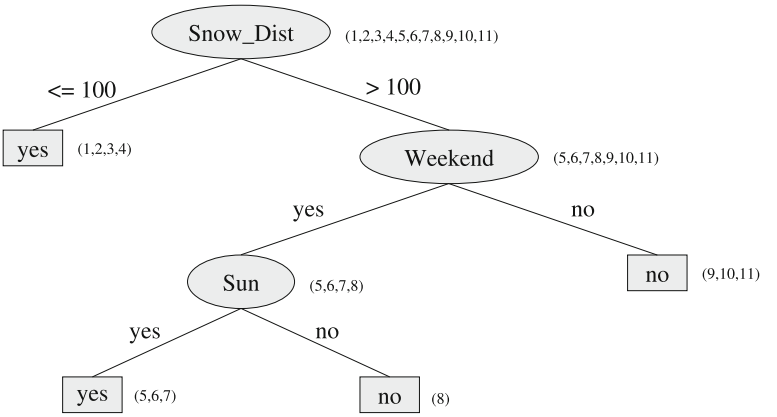


Fig. 8.23 Decision tree for the skiing classification problem. In the lists to the *right* of the nodes, the numbers of the corresponding training data are given. Notice that of the leaf nodes *sunny* = *yes* only two of the three examples are classified correctly

Table 8.5 Data set for the skiing classification problem

Day	*Snow_Dist*	*Weekend*	*Sun*	*Skiing*
1	≤100	yes	yes	yes
2	≤100	yes	yes	yes
3	≤100	yes	no	yes
4	≤100	no	yes	yes
5	>100	yes	yes	yes
6	>100	yes	yes	yes
7	>100	yes	yes	no
8	>100	yes	no	no
9	>100	no	yes	no
10	>100	no	yes	no
11	>100	no	no	no

a decision tree. First the attribute with the highest information gain (*Snow_ Dist*) is chosen for the root node from the set of all attributes. For each attribute value (≤100, >100) there is a branch in the tree. Now for every branch this process is repeated recursively. During generation of the nodes, the attribute with the highest information gain among the attributes which have not yet been used is always chosen, in the spirit of a greedy strategy.

8.4.2 Entropy as a Metric for Information Content

The described top-down algorithm for the construction of a decision tree, at each step selects the attribute with the highest information gain. We now introduce the

entropy as *the* metric for the information content of a set of training data D. If we only look at the binary variable *skiing* in the above example, then D can be described as

$$D = (\text{yes, yes, yes, yes, yes, yes, no, no, no, no, no})$$

with estimated probabilities

$$p_1 = P(\text{yes}) = 6/11 \quad \text{and} \quad p_2 = P(\text{no}) = 5/11.$$

Here we evidently have a probability distribution $\boldsymbol{p} = (6/11, 5/11)$. In general, for an n class problem this reads

$$\boldsymbol{p} = (p_1, \ldots, p_n)$$

with

$$\sum_{i=1}^{n} p_i = 1.$$

To introduce the information content of a distribution we observe two extreme cases. First let

$$\boldsymbol{p} = (1, 0, 0, \ldots, 0). \tag{8.4}$$

That is, the first one of the n events will certainly occur and all others will not. The uncertainty about the outcome of the events is thus minimal. In contrast, for the uniform distribution

$$\boldsymbol{p} = \left(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right) \tag{8.5}$$

the uncertainty is maximal because no event can be distinguished from the others. Here Claude Shannon asked himself how many bits would be needed to encode such an event. In the certain case of (8.4) zero bits are needed because we know that the case $i = 1$ always occurs. In the uniformly distributed case of (8.5) there are n equally probable possibilities. For binary encodings, $\log_2 n$ bits are needed here. Because all individual probabilities are $p_i = 1/n$, $\log_2 \frac{1}{p_i}$ bits are needed for this encoding.

In the general case $\boldsymbol{p} = (p_1, \ldots, p_n)$, if the probabilities of the elementary events deviate from the uniform distribution, then the expectation value H for the number of bits is calculated. To this end we will weight all values $\log_2 \frac{1}{p_i} = -\log_2 p_i$ with their probabilities and obtain

$$H = \sum_{i=1}^{n} p_i(-\log_2 p_i) = -\sum_{i=1}^{n} p_i \log_2 p_i.$$

The more bits we need to encode an event, clearly the higher the uncertainty about the outcome. Therefore we define:

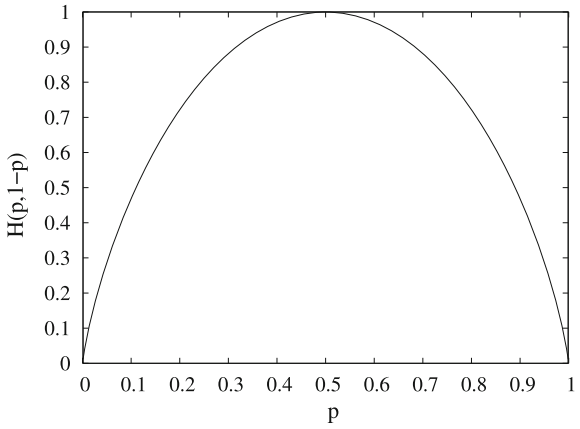


Fig. 8.24 The entropy function for the case of two classes. We see the maximum at $p = 1/2$ and the symmetry with respect to swapping p and $1 - p$

Definition 8.4 The *Entropy* H as a metric for the uncertainty of a probability distribution is defined by[7]

$$H(\boldsymbol{p}) = H(p_1, \ldots, p_n) := -\sum_{i=1}^{n} p_i \log_2 p_i.$$

A detailed derivation of this formula is found in [SW76]. If we substitute the certain event $\boldsymbol{p} = (1, 0, 0, \ldots, 0)$, then $0 \log_2 0$, an undefined expression results. We solve this problem by the definition $0 \log_2 0 := 0$ (see Exercise 8.10 on page 240).

Now we can calculate $H(1, 0, \ldots, 0) = 0$. We will show that the entropy in the hypercube $[0, 1]^n$ under the constraint $\sum_{i=1}^{n} p_i = 1$ takes on its maximum value with the uniform distribution $(\frac{1}{n}, \ldots, \frac{1}{n})$. In the case of an event with two possible outcomes, which correspond to two classes, the result is

$$H(\boldsymbol{p}) = H(p_1, p_2) = H(p_1, 1 - p_1) = -(p_1 \log_2 p_1 + (1 - p_1) \log_2 (1 - p_1)).$$

This expression is shown as a function of p_1 in Fig. 8.24 with its maximum at $p_1 = 1/2$.

Because each classified dataset D is assigned a probability distribution $\boldsymbol{p}$ by estimating the class probabilities, we can extend the concept of entropy to data by the definition

[7] In (7.9) on page 138 the natural logarithm rather than $\log_2$ is used in the definition of entropy. Because here, and also in the case of the MaxEnt method, entropies are only being compared, this difference does not play a role. (see Exercise 8.12 on page 240).

$$H(D) = H(\boldsymbol{p}).$$

Now, since the information content $I(D)$ of the dataset D is meant to be the opposite of uncertainty. Thus we define:

Definition 8.5 The information content of a dataset is defined as

$$I(D) := 1 - H(D). \tag{8.6}$$

8.4.3 Information Gain

If we apply the entropy formula to the example, the result is

$$H(6/11, 5/11) = 0.994$$

During construction of a decision tree, the dataset is further subdivided by each new attribute. The more an attribute raises the information content of the distribution by dividing the data, the better that attribute is. We define accordingly:

Definition 8.6 The *information gain* $G(D, A)$ through the use of the attribute A is determined by the difference of the average information content of the dataset $D = D_1 \cup D_2 \cup \cdots \cup D_n$ divided by the n-value attribute A and the information content $I(D)$ of the undivided dataset, which yields

$$G(D,A) = \sum_{i=1}^{n} \frac{|D_i|}{|D|} I(D_i) - I(D).$$

With (8.6) we obtain from this

$$\begin{aligned} G(D,A) &= \sum_{i=1}^{n} \frac{|D_i|}{|D|} I(D_i) - I(D) = \sum_{i=1}^{n} \frac{|D_i|}{|D|} (1 - H(D_i)) - (1 - H(D)) \\ &= 1 - \sum_{i=1}^{n} \frac{|D_i|}{|D|} H(D_i) - 1 + H(D) \\ &= H(D) - \sum_{i=1}^{n} \frac{|D_i|}{|D|} H(D_i). \end{aligned} \tag{8.7}$$

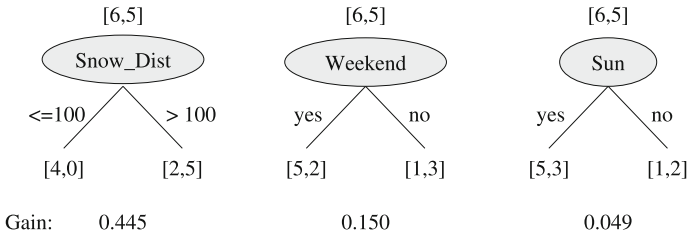


Fig. 8.25 The calculated gain for the various attributes reflects whether the division of the data by the respective attribute results in a better class division. The more the distributions generated by the attribute deviate from the uniform distribution, the higher the information gain

Applied to our example for the attribute *Snow_Dist*, this yields

$$\begin{aligned} G(D, Snow\_Dist) &= H(D) - \left(\frac{4}{11} H(D_{\leq 100}) + \frac{7}{11} H(D)_{>100}\right) \\ &= 0.994 - \left(\frac{4}{11} \cdot 0 + \frac{7}{11} \cdot 0.863\right) = 0.445. \end{aligned}$$

Analogously we obtain

$$G(D, Weekend) = 0.150$$

and

$$G(D, Sun) = 0.049.$$

The attribute *Snow_Dist* now becomes the root node of the decision tree. The situation of the selection of this attribute is once again clarified in Fig. 8.25.

The two attribute values $\leq$100 and >100 generate two edges in the tree, which correspond to the subsets $D_{\leq 100}$ and $D_{>100}$. For the subset $D_{\leq 100}$ the classification is clearly *yes*. thus the tree terminates here. In the other branch $D_{>100}$ there is no clear result. Thus the algorithm repeats recursively. From the two attributes still available, *Sun* and *Weekend*, the better one must be chosen. We calculate

$$G(D_{>100}, Weekend) = 0.292$$

and

$$G(D_{>100}, Sun) = 0.170.$$

The node thus gets the attribute *Weekend* assigned. For *Weekend* = *no* the tree terminates with the decision *Ski* = *no*. A calculation of the gain here returns the value 0. For *Weekend* = *yes*, *Sun* results in a gain of 0.171. Then the construction of the tree terminates because no further attributes are available, although example

number 7 is falsely classified. The finished tree is already familiar from Fig. 8.23 on page 200.

8.4.4 Application of C4.5

The decision tree that we just generated can also be generated by C4.5. The training data are saved in a data file `ski.data` in the following format:

```
<=100, yes,  yes,  yes
<=100, yes,  yes,  yes
<=100, yes,  no,   yes
<=100, no,   yes,  yes
>100,  yes,  yes,  yes
>100,  yes,  yes,  yes
>100,  yes,  yes,  no
>100,  yes,  no,   no
>100,  no,   yes,  no
>100,  no,   yes,  no
>100,  no,   no,   no
```

The information about attributes and classes is stored in the file `ski.names` (lines beginning with "|" are comments):

```
|Classes: no: do not ski, yes: go skiing
|
no,yes.
|
|Attributes
|
Snow_Dist:        <=100,>100.
Weekend:          no,yes.
Sun:              no,yes.
```

C4.5 is then called from the Unix command line and generates the decision tree shown below, which is formatted using indentations. The option `-f` is for the name of the input file, and the option `-m` specifies the minimum number of training data points required for generating a new branch in the tree. Because the number of training data points in this example is extremely small, `-m 1` is sensible here. For larger datasets, a value of at least `-m 10` should be used.

```
unixprompt> c4.5 -f ski -m 1

C4.5 [release 8] decision tree generator

                                        Wed Aug 23 10:44:49 2010
----------------------------------------
    Options:
        File stem <ski>
        Sensible test requires 2 branches with >=1 cases

Read 11 cases (3 attributes) from ski.data

Decision Tree:

Snow_Dist = <=100: ja (4.0)
Snow_Dist = >100:
|   Weekend = no: no (3.0)
|   Weekend = yes:
|   | Sun = no: no (1.0)
|   | Sun = yes: yes (3.0/1.0)

Simplified Decision Tree:

Snow_Dist = <=100: yes (4.0/1.2)
Snow_Dist = >100: no (7.0/3.4)

Evaluation on training data (11 items):

       Before Pruning              After Pruning
       ----------------    --------------------------
       Size     Errors      Size     Errors    Estimate

        7     1(9.1%)        3      2(18.2%)   (41.7%) <<
```

Additionally, a simplified tree with only one attribute is given. This tree, which was created by pruning (see Sect. 8.4.7), will be important for an increasing amount of training data. In this little example it does not yet makes much sense. The error rate for both trees on the training data is also given. The numbers in parentheses after the decisions give the size of the underlying dataset and the number of errors. For example, the line `Sun = yes: yes (3.0/1.0)` in the top tree indicates that for this leaf node `Sun = yes`, three training examples exist, one of which is falsely classified. The user can thus read from this whether the decision is statistically grounded and/or certain.

In Fig. 8.26 on page 207 we can now give the schema of the learning algorithm for generating a decision tree.

We are now familiar with the foundations of the automatic generation of decision trees. For the practical application, however, important extensions are needed. We will introduce these using the already familiar LEXMED application.

GENERATEDECISIONTREE(*Data,Node*)

A_{max} = Attribute with maximum information gain
If $G(A_{max}) = 0$
  Then *Node* becomes leaf node with most frequent class in *Data*
  Else assign the attribute A_{max} to *Node*
    For each value $a_1, \ldots, a_n$ of A_{max}, generate
      a successor node: $K_1, \ldots, K_n$
    Divide *Data* into $D_1, \ldots, D_n$ with $D_i = \{x \in Data | A_{max}(x) = a_i\}$
    For all $i \in \{1, \ldots, n\}$
      If all $x \in D_i$ belong to the same class C_i
        Then generate leaf node K_i of class C_i
        Else GENERATEDECISIONTREE(D_i, K_i)

Fig. 8.26 Algorithm for the construction of a decision tree

8.4.5 Learning of Appendicitis Diagnosis

In the research project LEXMED, an expert system for the diagnosis of appendicitis was developed on top of a database of patient data [ES99, SE00]. The system, which works with the method of maximum entropy, is described in Sect. 7.3.

We now use the LEXMED database to generate a decision tree for diagnosing appendicitis with C4.5. The symptoms used as attributes are defined in the file `app.names`:

```
|Definition of the classes and attributes
|
|Classes 0=appendicitis negative
| 1=appendicitis positive
0,1.
|
|Attributes
|
Age: continuous.
Sex_(1=m___2=w): 1,2.
Pain_Quadrant1_(0=no__1=yes): 0,1.
Pain_Quadrant2_(0=no__1=yes): 0,1.
Pain_Quadrant3_(0=no__1=yes): 0,1.
Pain_Quadrant4_(0=no__1=yes): 0,1.
Local_guarding_(0=no__1=yes): 0,1.
Generalized_guarding_(0=no__1=yes): 0,1.
Rebound_tenderness_(0=no__1=yes): 0,1.
Pain_on_tapping_(0=no__1=yes): 0,1.
```

```
Pain_during_rectal_examination_(0=no__1=yes): 0,1.
Temp_axial: continuous.
Temp_rectal: continuous.
Leukocytes: continuous.
Diabetes_mellitus_(0=no__1=yes): 0,1
```

We see that, besides many binary attributes such as the various pain symptoms, continuous symptoms such as age and fever temperature also occur. In the following training data file, `app.data`, in each line a case is described. In the first line is a 19-year-old male patient with pain in the third quadrant (lower right, where the appendix is), the two fever values 36.2 and 37.8 degree Celsius a leukocyte value of 13400 and a positive diagnosis, that is, an inflamed appendix.

```
19,1,0,0,1,0,1,0,1,1,0,362,378,13400,0,1
13,1,0,0,1,0,1,0,1,1,1,383,385,18100,0,1
32,2,0,0,1,0,1,0,1,1,0,364,374,11800,0,1
18,2,0,0,1,1,0,0,0,0,0,362,370,09300,0,0
73,2,1,0,1,1,1,0,1,1,1,376,380,13600,1,1
30,1,1,1,1,1,0,1,1,1,1,377,387,21100,0,1
56,1,1,1,1,1,0,1,1,1,0,390,?,14100,0,1
36,1,0,0,1,0,1,0,1,1,0,372,382,11300,0,1
36,2,0,0,1,0,0,0,1,1,1,370,379,15300,0,1
33,1,0,0,1,0,1,0,1,1,0,367,376,17400,0,1
19,1,0,0,1,0,0,0,1,1,0,361,375,17600,0,1
12,1,0,0,1,0,1,0,1,1,0,364,370,12900,0,0
...
```

Without going into detail about the database, it is important to mention that only patients who were suspected of having appendicitis upon arrival at the hospital and were then operated upon are included in the database. We see in the seventh row that C4.5 can also deal with missing values. The data contain 9764 cases.

```
unixprompt> c4.5 -f app -u -m 100

C4.5 [release 8] decision tree generator
                                        Wed Aug 23 13:13:15 2006
----------------------------------------

Read 9764 cases (15 attributes) from app.data

Decision Tree:
```

```
Leukocytes <= 11030 :
| Rebound_tenderness = 0:
| | Temp_rectal > 381 : 1 (135.9/54.2)
| | Temp_rectal <= 381 :
| | | Local_guarding = 0: 0 (1453.3/358.9)
| | | Local_guarding = 1:
| | | | Sex_(1=m___2=w) = 1: 1 (160.1/74.9)
| | | | Sex_(1=m___2=w) = 2: 0 (286.3/97.6)
| Rebound_tenderness = 1:
| | Leukocytes <= 8600 :
| | | Temp_rectal > 378 : 1 (176.0/59.4)
| | | Temp_rectal <= 378 :
| | | | Sex_(1=m___2=w) = 1:
| | | | | Local_guarding = 0: 0 (110.7/51.7)
| | | | | Local_guarding = 1: 1 (160.6/68.5)
| | | | Sex_(1=m___2=w) = 2:
| | | | | Age <= 14 : 1 (131.1/63.1)
| | | | | Age > 14 : 0 (398.3/137.6)
| | Leukocytes > 8600 :
| | | Sex_(1=m___2=w) = 1: 1 (429.9/91.0)
| | | Sex_(1=m___2=w) = 2:
| | | | Local_guarding = 1: 1 (311.2/103.0)
| | | | Local_guarding = 0:
| | | | | Temp_rectal <= 375 : 1 (125.4/55.8)
| | | | | Temp_rectal > 375 : 0 (118.3/56.1)
Leukocytes > 11030 :
| Rebound_tenderness = 1: 1 (4300.0/519.9)
| Rebound_tenderness = 0:
| | Leukocytes > 14040 : 1 (826.6/163.8)
| | Leukocytes <= 14040 :
| | | Pain_on_tapping = 1: 1 (260.6/83.7)
| | | Pain_on_tapping = 0:
| | | | Local_guarding = 1: 1 (117.5/44.4)
| | | | Local_guarding = 0:
| | | | | Temp_axial <= 368 : 0 (131.9/57.4)
| | | | | Temp_axial > 368 : 1 (130.5/57.8)

Simplified Decision Tree:

Leukocytes > 11030 : 1 (5767.0/964.1)
Leukocytes <= 11030 :
| Rebound_tenderness = 0:
| | Temp_rectal > 381 : 1 (135.9/58.7)
| | Temp_rectal <= 381 :
| | | Local_guarding = 0: 0 (1453.3/370.9)
| | | Local_guarding = 1:
| | | | Sex_(1=m___2=w) = 1: 1 (160.1/79.7)
| | | | Sex_(1=m___2=w) = 2: 0 (286.3/103.7)
| Rebound_tenderness = 1:
| | Leukocytes > 8600 : 1 (984.7/322.6)
| | Leukocytes <= 8600 :
```

```
| | | Temp_rectal > 378 : 1 (176.0/64.3)
| | | Temp_rectal <= 378 :
| | | | Sex_(1=m___2=w) = 1:
| | | | | Local_guarding = 0: 0 (110.7/55.8)
| | | | | Local_guarding = 1: 1 (160.6/73.4)
| | | | Sex_(1=m___2=w) = 2:
| | | | | Age <= 14 : 1 (131.1/67.6)
| | | | | Age > 14 : 0 (398.3/144.7)

Evaluation on training data (9764 items):

         Before Pruning           After Pruning
        ----------------   --------------------------
        Size      Errors   Size      Errors   Estimate
          37  2197(22.5%)     21  2223(22.8%)    (23.6%)   <<

Evaluation on test data (4882 items):

         Before Pruning           After Pruning
        ----------------   --------------------------
        Size      Errors   Size      Errors   Estimate
          37  1148(23.5%)     21  1153(23.6%)    (23.6%)   <<

          (a)  (b)      <-classified as
         ----  ----
          758  885      (a): class 0
          268 2971      (b): class 1
```

8.4.6 Continuous Attributes

In the trees generated for the appendicitis diagnosis there is a node `Leukocytes > 11030` which clearly comes from the continuous attribute `Leukocytes` by setting a threshold at the value 11030. C4.5 thus has made a binary attribute `Leukocytes > 11030` from the continuous attribute `Leukocytes`. The threshold $\Theta_{D,A}$ for an attribute A is determined by the following algorithm: for all values v which occur in the training data D, the binary attribute $A > v$ is generated and its information gain is calculated. The threshold $\Theta_{D,A}$ is then set to the value v with the maximum information gain, that is:

$$\Theta_{D,A} = \text{argmax}_v\{G(D, A > v)\}.$$

For an attribute such as the leukocyte value or the patient's age, a decision based on a binary discretization is presumably too imprecise. Nevertheless there is no need to discretize finer because each continuous attribute is tested on each newly generated node and can thus occur repeatedly in one tree with a different threshold $\Theta_{D,A}$. Thus we ultimately obtain a very good discretization whose fineness fits the problem.

8.4.7 Pruning—Cutting the Tree

Since the time of Aristotle it has been stipulated that of two scientific theories which explain the same situation equally well, the simpler one is preferred. This law of economy, also now known as *Occam's razor*, is of great importance for machine learning and data mining.

A decision tree is a theory for describing the training data. A different theory for describing the data is the data themselves. If the tree classifies all data without any errors, but is much more compact and thus more easily understood by humans, then it is preferable according to Occam's razor. The same is true for two decision trees of different sizes. Thus the goal of every algorithm for generating a decision tree must be to generate the smallest possible decision tree for a given error rate. Among all trees with a fixed error rate, the smallest tree should always be selected.

Up until now we have not defined the term error rate precisely. As already mentioned several times, it is important that the learned tree not just memorize the training data, rather that it generalizes well. To test the ability of a tree to generalize, we divide the available data into a set of *training data* and a set of v. The test data are hidden from the learning algorithm and only used for testing. If a large dataset is available, such as the appendicitis data, then we can for example use two-thirds of the data for learning and the remaining third for testing.

Aside from better comprehensibility, Occam's razor has another important justification: generalization ability. The more complex the model (here a decision tree), the more details are represented, but to the same extent the less is the model transferable to new data. This relationship is illustrated in Fig. 8.27. Decision trees of various sizes were trained against the appendicitis data. In the graph, classification errors on both the training data and on the test data are given. The error rate on the training data decreases monotonically with the size of the tree. Up to a tree size of 55 nodes, the error rate on test data also decreases. If the tree grows further, however, then the error rate starts to increase again! This effect, which we have already seen in the nearest neighbor method, is called *overfitting*.

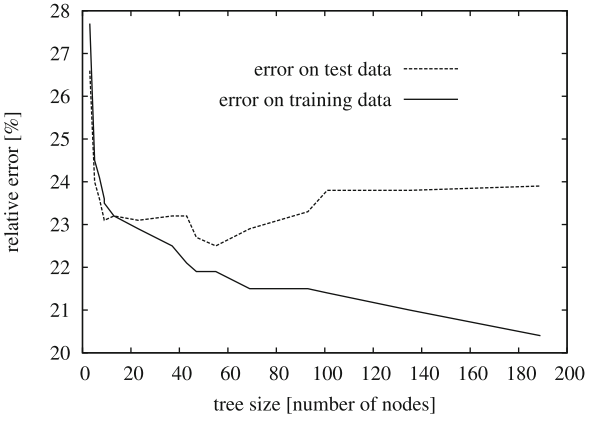


Fig. 8.27 Learning curve of C4.5 on the appendicitis data. We clearly see the overfitting of trees with more than 55 nodes

We will give this concept, which is important for nearly all learning processes, a general definition taken from [Mit97]:

Definition 8.7 Let a specific learning algorithm, that is, a learning agent, be given. We call an agent A overfit to the training data if there is another agent A' whose error on the training data is larger than that of A, but whose error on the whole distribution of data is smaller than the error of A.

How can we now find this point of minimum error on the test data? The most obvious algorithm is called *cross-validation*. During construction of the tree, the error on the test data is measured in parallel. As soon as the error rises significantly, the tree with the minimum error is saved. This algorithm is used by the CART system mentioned earlier.

C4.5 works somewhat differently. First, using the algorithm GenerateDecisionTree from Fig. 8.26 on page 207, it generates a tree which is usually overfit. Then, using *pruning*, it attempts to cut away nodes of the tree until the error on the test data, estimated by the error on the training data, begins to rise.[8] Like the construction of the tree, this is also a greedy algorithm. This means that once a node is pruned, it cannot be re-inserted, even if this later turns out to be better.

8.4.8 Missing Values

Frequently individual attribute values are missing from the training data. In the Lexmed dataset, the following entry occurs:

$$56, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 390, ?, 14100, 0, 1,$$

in which one of the fever values is missing. Such data can nevertheless be used during construction of the decision tree. We can assign the attribute the most frequent value from the whole dataset or the most frequent of all data points from the same class. It is even better to substitute the probability distribution of all attribute values for the missing attribute value and to split the training example into branches according to this distribution. This is incidentally a reason for the occurrence of non-integer values in the expressions in parentheses next to the leaf nodes of the C4.5 tree.

Missing values can occur not only during learning, but also during classification. These are handled in the same way as during learning.

[8] It would be better to use the error on the test data directly. At least when the amount of training data is sufficient to justify a separate testing set.

8.4.9 Summary

Learning of decision trees is a favorite approach to classification tasks. The reasons for this are its simple application and speed. On a dataset of about 10 000 LEXMED data points with 15 attributes each, C4.5 requires about 0.3 s for learning. This is very fast compared to other learning algorithms.

For the user it is also important, however, that the decision tree as a learned model can be understood and potentially changed. It is also not difficult to automatically transform a decision tree into a series of if-then-else statements and thus efficiently build it into an existing program.

Because a greedy algorithm is used for construction of the tree as well as during pruning, the trees are in general suboptimal. The discovered decision tree does usually have a relatively small error rate. However, there is potentially a better tree, because the heuristic greedy search of C4.5 prefers small trees and attributes with high information gain at the top of the tree. For attributes with many values, the presented formula for the information gain shows weaknesses. Alternatives to this are given in [Mit97].

8.5 Cross-Validation and Overfitting

As discussed in Sect. 8.4.7, many learning algorithms have the problem of overfitting. Powerful learning algorithms, such as for example decision tree learning, can adapt the complexity of the learned model to the complexity of the training data. This leads to overfitting if there is noise in the data.

With cross-validation one attempts to optimize the model complexity such that it minimizes the classification or approximation error on an unknown test data set. This requires the model complexity to be controllable by a parameter γ. For decision trees this is, for example, the size of the tree. For the k nearest neighbor method from Sect. 8.3, the parameter is k, the number of nearest neighbors, while for neural networks it is the number of hidden neurons (see Chap. 9).

We vary a parameter γ while training the algorithm on a training data set and choose the value of γ that minimizes the error on an independent testing data set. k-ary cross-validation works according to the following schema:

CROSSVALIDATION($\boldsymbol{X}, k$)
Partition data into k equally sized blocks $\boldsymbol{X} = \boldsymbol{X}_1 \cup \ldots \cup \boldsymbol{X}_k$
For all $\gamma \in \{\gamma_{min}, \ldots, \gamma_{max}\}$
  For all $i \in \{1, \ldots, k\}$
    Train a model of complexity γ on $\boldsymbol{X} \backslash \boldsymbol{X}_i$
    Compute the error $E(\gamma, \boldsymbol{X}_i)$ on the test set $\boldsymbol{X}_i$
  Compute the mean error $E(\gamma) = \frac{1}{k} \sum_{i=1}^{k} E(\gamma, \boldsymbol{X}_i)$
Choose the value $\gamma_{opt} = \text{argmin}_\gamma E(\gamma)$ with smallest mean error
Train the final model with complexity γ_{opt} on the whole data set $\boldsymbol{X}$

The whole data set $\boldsymbol{X}$ is divided into k equally sized blocks. Then the algorithm is trained k times on $k-1$ blocks and tested on the remaining block. The k computed errors are averaged and the value γ_{opt} with the smallest mean error is chosen to train the final model on the whole data set $\boldsymbol{X}$.

If the training set $\boldsymbol{X}$ is large, it may be divided, for example, into $k = 3$ or $k = 10$ blocks. The resulting increase in computational complexity is usually acceptable. For small training sets, it is preferable to train on all n feature vectors if possible. Then one can choose $k = n$, which yields what is called leave-one-out cross-validation.

Cross-validation is the most important and most used automatic optimization method for model complexity. It solves the overfitting problem. We can gain additional insight into this problem from short look at the so called *bias variance tradeoff*. If an overly simple model is used, this forces the non-optimal approximation of the data in a certain direction (bias). On the other hand, if an overly complex model is used, it will tend to overfit any data set. Thus, for a new data sample from the same distribution, it may learn a very different model. The model therefore varies greatly for a change in the data (variance).

This relationship can be illustrated by the example of function approximation using polynomials. If one chooses the degree of the polymonial as the complexity parameter γ, degree 1 approximates a line, which is not a good approximation for non-trivial data. If, on the other hand, the polynomial degree equals $n-1$ for n points, it tends to hit every point and reduce the error on the training data to zero. This can be seen in Fig. 8.28 left on page 215, in which eight data points have been generated using

$$y(x) = \sin(x) + g(0, 0.2),$$

where $g(0,\ 0.2)$ generates normally distributed noise with zero mean and a standard deviation of 0.2.

In the left image, in addition to the data points, the underlying sine function (black) is inscribed, as well as a straight line (red) approximated using the method of least squares (see Sect. 9.4 oder [Ert15]). In addition, a seventh degree polynomial (green) is interpolated through the points, hitting each point, as well as a fourth degree polynomial (blue), which comes closest to the sine curve. In the right image, the same approximations were carried out again on eight new data points. One can see very nicely that despite differing data, both straight lines deviate greatly from the data (large bias), but have a similar function graph (very small variance). Both seventh degree polynomials, on the other hand, fit the data perfectly (zero bias), but they have very different function graphs (very large variance). This is an obvious overfitting effect. The fourth degree polynomial presents a good compromise. Cross-validation would presumably yield degree four as the optimal solution.

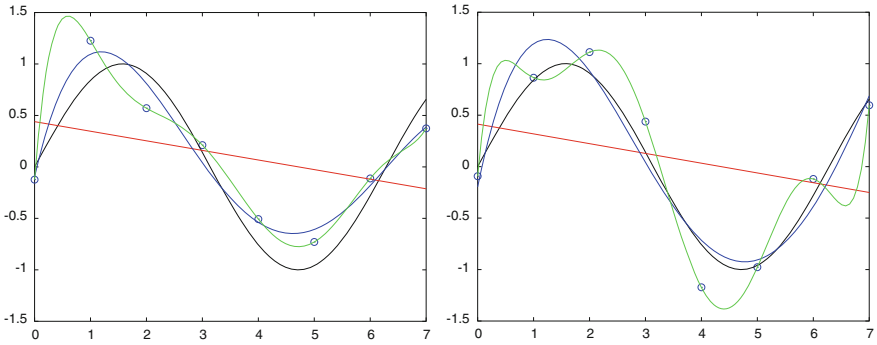


Fig. 8.28 Two different data sets approximated by polynomials of degree 1, 4 and 7. The bias variance tradeoff is clearly visible

8.6 Learning of Bayesian Networks

In Sect. 7.4, it was shown how to build a Bayesian network manually. Now we will introduce algorithms for the induction of Bayesian networks. Similar to the learning process described previously, a Bayesian network is automatically generated from a file containing training data. This process is typically decomposed into two parts.

1. *Learning the network structure:* For given variables, the network topology is generated from the training data. This first step is by far the more difficult one and will be given closer attention later.
2. *Learning the conditional probabilities:* For known network topologies, the CPTs must be filled with values. If enough training data is available, all necessary conditional probabilities can be estimated by counting the frequencies in the data. This step can be automated relatively easily.

We will now explain how Bayesian networks learn using a simple algorithm from [Jen01].

8.6.1 Learning the Network Structure

During the development of a Bayesian network (see Sect. 7.4.6), the causal dependency of the variables must be taken into account in order to obtain a simple network of good quality. The human developer relies on background knowledge, which is unavailable to the machine. Therefore, this procedure cannot be easily automated.

Finding an optimal structure for a Bayesian network can be formulated as a classic search problem. Let a set of variables $V_1, \ldots, V_n$ and a file with training data

be given. We are looking for a set of directed edges without cycles between the nodes $V_1, \ldots, V_n$, that is, a directed acyclic graph (DAG) which reproduces the underlying data as well as possible.

First we observe the search space. The number of different DAGs grows more than exponentially with the number of nodes. For five nodes there are 29281 and for nine nodes about 10^{15} different DAGs [MDBM00]. Thus an uninformed combinatorial search (see Sect. 6.2) in the space of all graphs with a given set of variables is hopeless if the number of variables grows. Therefore heuristic algorithms must be used. This poses the question of an evaluation function for Bayesian networks. It is possible to measure the classification error of a network during application to a set of test data, as is done, for example, in C4.5 (see Sect. 8.4). For this, however, the probabilities calculated by the Bayesian network must be mapped to a decision.

A direct measurement of the quality of a network can be taken over the probability distribution. We assume that, before the construction of the network from the data, we could determine (estimate) the distribution. Then we begin the search in the space of all DAGs, estimate the value of the CPTs for each DAG (that is, for each Bayesian network) using the data, and from that we calculate the distribution and compare it to the distribution known from the data. For the comparison of distributions we will obviously need a distance metric.

Let us consider the weather prediction example from Exercise 7.3 on page 172 with the three variables *Sky*, *Bar*, *Prec*, and the distribution

$$\boldsymbol{P}(Sky, Bar, Prec) = (0.40, 0.07, 0.08, 0.10, 0.09, 0.11, 0.03, 0.12).$$

In Fig. 8.29 on page 217 two Bayesian networks are presented, which we will now compare with respect to their quality. Each of these networks makes an assumption of independence, which is validated in that we determine the distribution of the network and then compare this with the original distribution (see Exercise 8.16 on page 241).

Because, for constant predetermined variables, the distribution is clearly represented by a vector of constant length, we can calculate the Euclidian norm of the difference of the two vectors as a distance between distributions. We define

$$d_q(\boldsymbol{x}, \boldsymbol{y}) = \sum_i (x_i - y_i)^2$$

as the sum of the squares of the distances of the vector components and calculate the distance $d_q(\boldsymbol{P}_a, \boldsymbol{P}) = 0.0029$ of the distribution $\boldsymbol{P}_a$ of network 1 from the original distribution. For network 2 we calculate $d_q(\boldsymbol{P}_b, \boldsymbol{P}) = 0.014$. Clearly network 1 is a better approximation of the distribution. Often, instead of the square distance, the so-called Kullback–Leibler distance

$$d_k(\boldsymbol{x}, \boldsymbol{y}) = \sum_i y_i (\log_2 y_i - \log_2 x_i),$$

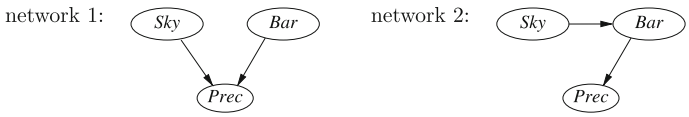


Fig. 8.29 Two Bayesian networks for modeling the weather prediction example from Exercise 7.3 on page 172

an information theory metric, is used. With it we calculate $d_k(\boldsymbol{P}_a, \boldsymbol{P}) = 0.017$ and $d_k(\boldsymbol{P}_b, \boldsymbol{P}) = 0.09$ and come to the same conclusion as before. It is to be expected that networks with many edges approximate the distribution better than those with few edges. If all edges in the network are constructed, then it becomes very confusing and creates the risk of overfitting, as is the case in many other learning algorithms. To avoid overfitting, we give small networks a larger weight using a heuristic evaluation function

$$f(N) = Size(N) + w \cdot d_k(\boldsymbol{P}_N, \boldsymbol{P}).$$

Here $Size(N)$ is the number of entries in the CPTs and $\boldsymbol{P}_N$ is the distribution of network N. w is a weight factor, which must be manually fit.

The learning algorithm for Bayesian networks thus calculates the heuristic evaluation $f(N)$ for many different networks and then chooses the network with the smallest value. As previously mentioned, the difficulty consists of the reduction of the search space for the network topology we are searching for. As a simple algorithm it is possible, starting from a (for example causal) ordering of the variables $V_1, \ldots, V_n$, to include in the graph only those edges for which $i < j$. We start with the maximal model which fulfills this condition. This network is shown in Fig. 8.30 for five ordered variables.

Now, for example in the spirit of a greedy search (compare Sect. 6.3.1), one edge after another is removed until the value f no longer decreases.

This algorithm is not practical for larger networks in this form. The large search space, the manual tuning of the weight w, and the necessary comparison with a goal distribution $\boldsymbol{P}$ are reasons for this, because these can simply become too large, or the available dataset could be too small.

In fact, research into learning of Bayesian networks is still in full swing, and there is a large number of suggested algorithms, for example the EM algorithm (see Sect. 8.9.2), Markov chain Monte Carlo methods, and Gibbs sampling [DHS01, Jor99, Jen01, HTF09]. Besides batch learning, which has been presented here, in which the network is generated once from the whole dataset, there are also incremental algorithms, in which each individual new case is used to improve the

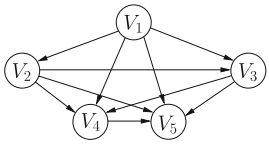


Fig. 8.30 The maximal network with five variables and edges (V_i, V_j) which fulfill the condition $i < j$

network. Implementations of these algorithms also exist, such as Hugin (www.hugin.com) and Bayesware (www.bayesware.com).

8.7 The Naive Bayes Classifier

In Fig. 7.14 on page 166 the diagnosis of appendicitis was modeled as a Bayesian network. Because directed edges start at a diagnosis node and none end there, Bayes' formula must be used to answer a diagnosis query. For the symptoms $S_1, \ldots, S_n$ and the k-value diagnosis D with the values $b_1, \ldots, b_k$ we calculate the probability

$$P(D|S_1, \ldots, S_n) = \frac{P(S_1, \ldots, S_n|D) \cdot P(D)}{P(S_1, \ldots, S_n)},$$

for the diagnosis given the patient's symptoms. In the worst case, that is, if there were no independent variables, all combinations of all symptoms and D would need to be determined for all 20 643 840 probabilities of the distribution $\boldsymbol{P}(S_1, \ldots, S_n, D)$. This would require an enormous database. In the case of LEXMED's Bayesian network, the number of necessary values (in the CPTs) is reduced to 521. The network can be further simplified, however, in that we assume all symptom variables are conditionally independent given D, that is:

$$P(S_1, \ldots, S_n|D) = P(S_1|D) \cdot \cdots \cdot P(S_n|D).$$

The Bayesian network for appendicitis is then simplified to the star shown in Fig. 8.31 on page 219.
Thus we obtain the formula

$$P(D|S_1, \ldots, S_n) = \frac{P(D) \prod_{i=1}^{n} P(S_i|D)}{P(S_1, \ldots, S_n)}. \tag{8.8}$$

The computed probabilities are transformed into a decision by a simple naive Bayes classifier, which chooses the maximum $P(D = d_i|S_1, \ldots, S_n)$ from all values d_i in D. That is, it determines

$$d_{Naive-Bayes} = \text{argmax}_{i \in \{1,\ldots,k\}} P(D = d_i|S_1, \ldots, S_n).$$

Because the denominator in (8.8) is constant, it can be omitted during maximization, which results in the *naive Bayes formula*

$$d_{Naive-Bayes} = \text{argmax}_{i \in \{1,\ldots,k\}} P(D = d_i) \prod_{j=1}^{n} P(S_j|D).$$

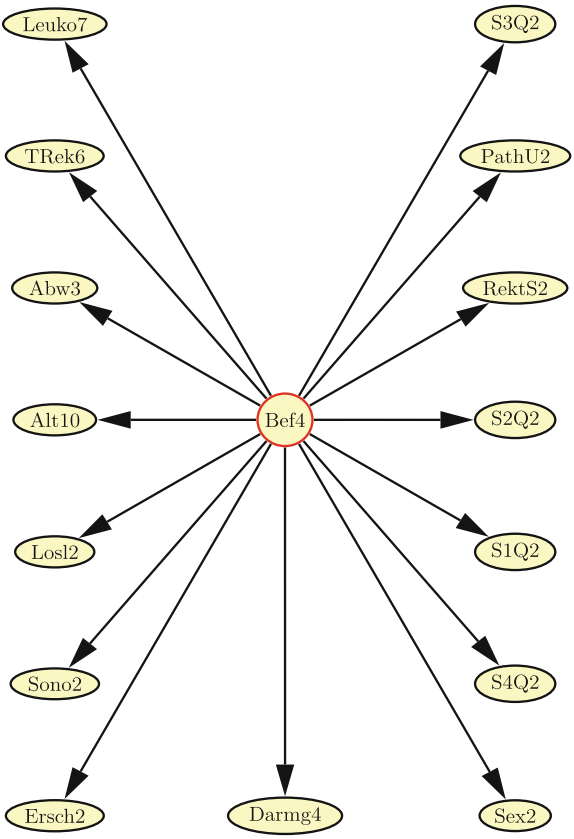


Fig. 8.31 Bayesian network for the LEXMED application with the assumption that all symptoms are conditionally independent given the diagnosis

Because several nodes now have less ancestors, the number of values necessary to describe the LEXMED distribution in the CPTs decreases, according to (7.24) on page 165, to

$$6 \cdot 4 + 5 \cdot 4 + 2 \cdot 4 + 9 \cdot 4 + 3 \cdot 4 + 10 \cdot (1 \cdot 4) + 3 = 143.$$

For a medical diagnostic system like LEXMED this simplification would not be acceptable. But for tasks with many independent variables, naive Bayes is partly or even very well suited, as we will see in the text classification example.

By the way, naive Bayes classification is equally expressive as the linear score system described in Sect. 7.3.1 (see Exercise 8.17 on page 242). That is, all scores share the underlying assumption that all symptoms are conditionally independent given the diagnosis. Nevertheless, scores are still used in medicine today. Despite the fact that it was generated from a better, representative database, the Ohmann score compared with LEXMED in Fig. 7.10 on page 156 has a worse quality of diagnosis. Its limited expressiveness is certainly a reason for this. For example, as with naive Bayes, it is not possible to model dependencies between symptoms using scores.

Estimation of Probabilities If we observe the naive Bayes formula in (8.8) on page 218, we see that the whole expression becomes zero as soon as one of the factors $P(S_i|D)$ on the right side becomes zero. Theoretically there is nothing wrong here. In practice, however, this can lead to very uncomfortable effects if the $P(S_i|D)$ are small, because these are estimated by counting frequencies and substituting them into

$$P(S_i = x \,|\, D = y) = \frac{|S_i = x \wedge D = y|}{|D = y|}.$$

Assume that for the variables S_i: $P(S_i = x \,|\, D = y) = 0.01$ and that there are 40 training cases with $D = y$. Then with high probability there is no training case with $S_i = x$ and $D = y$, and we estimate $P(S_i = x \,|\, D = y) = 0$. For a different value $D = z$, assume that the relationships are similarly situated, but the estimate results in values greater than zero for all $P(S_i = x \,|\, D = z)$. Thus the value $D = z$ is always preferred, which does not reflect the actual probability distribution. Therefore, when estimating probabilities, the formula

$$P(A|B) \approx \frac{|A \wedge B|}{|B|} = \frac{n_{AB}}{n_B}$$

is replaced by

$$P(A|B) \approx \frac{n_{AB} + mp}{n_B + m},$$

where $p = P(A)$ is the a priori probability for A, and m is a constant which can be freely chosen and is known as the "equivalent data size". The larger m becomes, the larger the weight of the a priori probability compared to the value determined from the measured frequency.

8.7.1 Text Classification with Naive Bayes

Naive Bayes is very successful and prolific today in text classification. Its primary, and at the same time very important, application is the automatic filtering of email into desired and undesired, or spam emails. In spam filters such as SpamAssassin [Sch04], among other methods a naive Bayes classifier is used that learns to separate desired emails from spam. SpamAssassin is a hybrid system which performs an initial filtering using black and white lists. Black lists are lists of blocked email addresses from spam senders whose emails are always deleted, and white lists are those with senders whose emails are always delivered. After this prefiltering, the remaining emails are classified by the naive Bayes classifier according to their actual content, in other words, according to the text. The detected class value is then evaluated by a score, together with other attributes from the header of the email such as the sender's domain, the MIME type, etc., and then finally filtered.

Here the learning capability of the naive Bayes filter is quite important. For this the user must at first manually classify a large number of emails as desired or spam. Then the filter is trained. To stay up to date, the filter must be regularly retrained. For this the user should correctly classify all emails which were falsely classified by the filter, that is, put them in the appropriate folders. The filter is then continually retrained with these emails.

Beside spam filtering, there are many other applications for automatic text classification. Important applications include filtering of undesired entries in Internet discussion forums, and tracking websites with criminal content such as militant or terrorist activities, child pornography or racism. It can also be used to customize search engines to fit the user's preferences in order to better classify the search results. In the industrial and scientific setting, company-wide search in databases or in the literature is in the foreground of research. Through its learning ability, a filter can adapt to the habits and wishes of each individual user.

We will introduce the application of naive Bayes to text analysis on a short example text by Alan Turing from [Tur50]:

> "We may hope that machines will eventually compete with men in all purely intellectual fields. But which are the best ones to start with? Even this is a difficult decision. Many people think that a very abstract activity, like the playing of chess, would be best. It can also be maintained that it is best to provide the machine with the best sense organs that money can buy, and then teach it to understand and speak English. This process could follow the normal teaching of a child. Things would be pointed out and named, etc. Again I do not know what the right answer is, but I think both approaches should be tried."

Suppose that texts such as the one given should be divided into two classes: "I" for interesting and "$\neg I$ for uninteresting. Suppose also that a database exists of texts which are already classified. Which attributes should be used? In a classical approach to the construction of a Bayesian network, we define a set of attributes such as the length of the text, the average sentence length, the relative frequency of specific punctuation marks, the frequency of several important words such as "I", "machines", etc. During classification using naive Bayes, in contrast, a surprisingly primitive algorithm is selected. For each of the n word positions in the text, an attribute s_i is defined. All words which occur in the text are allowed as possible values for all positions s_i. Now for the classes I and $\neg I$ the values

$$P(I|s_1, \ldots, s_n) = c \cdot P(I) \prod_{i=1}^{n} P(s_i|I) \tag{8.9}$$

and $P(\neg I|s_1, \ldots, s_n)$ must be calculated and then the class with the maximum value selected. In the above example with a total of 113 words, this yields

$$\begin{aligned} &P(I|s_1, \ldots, s_n) \\ &\quad = c \cdot P(I) \cdot P(s_1 = \text{"We"}|I) \cdot P(s_2 = \text{"may"}|I) \ldots . P(s_{113} = \text{"tried"}|I) \end{aligned}$$

and

$$\begin{aligned} P(\neg I|s_1,\ldots,s_n) \\ = c \cdot (P\neg) \cdot P(s_1 = \text{“We”}|\neg I) \\ \cdot P(s_2 = \text{“may”}|\neg I) \cdot \cdots \cdot P(s_{113} = \text{“tried”}|\neg I). \end{aligned}$$

The learning here is quite simple. The conditional probabilities $P(s_i \mid I)$, $P(s_i \mid \neg I)$ and the a priori probabilities $P(I)$, $P(\neg I)$ must simply be calculated. We now additionally assume that the $P(s_i \mid I)$ are not dependent on position in the text. This means, for example, that

$$P(s_{61} = \text{“}and\text{”}|I) = P(s_{69} = \text{“}and\text{”}|I) = P(s_{86} = \text{“}and\text{”}|I).$$

We could thus use the expression $P(and|I)$, with the new binary variable *and*, as the probability of the occurrence of “and” at an arbitrary position.

The implementation can be accelerated somewhat if we find the frequency n_i of every word w_i which occurs in the text and use the formula

$$P(I|s_1,\ldots,s_n) = c \cdot P(I) \prod_{i=1}^{l} P(w_i|I)^{n_i} \tag{8.10}$$

which is equivalent to (8.9) on page 221. Please note that the index i in the product only goes to the number l of different words which occur in the text.

Despite its simplicity, naive Bayes delivers excellent results for text classification. Spam filters which work with naive Bayes achieve error rates of well under one percent. The systems DSPAM and CRM114 can even be so well trained that they only incorrectly classify one in 7000 or 8000 emails respectively. This corresponds to a correctness of nearly 99.99%.

8.8 One-Class Learning

Classification tasks in supervised learning require all training data to be given class labels. However, there are applications for which only one class of labels is available. A classic example is detecting errors in complex technical systems. The task is to recognize whether a device is defective or not. This sounds like an ordinary two-class problem. In the training data, the state of the device (good or defective) must be provided manually. In practice, the classifier training happens during deployment of the device or later on demand while it is being used in production. Data capture in the error-free condition is not a problem. Collection of data from the defective system however is problematic due to the following reasons:

- Measurement on production equipment that has intentionally been made defective is associated with high costs because the measurement leads to expensive downtime.

- Measurement using defective equipment with errors that actually occur in practice is often impossible because, during the deployment of the equipment, the later occurring potential errors may be unknown.
- No engineer knows in advance exactly which kind of errors will occur in a new piece of equipment. If now, during training, measurements with a few types of errors are taken, this may lead to poor classification results. If the errors used for training are not representative for the device, i.e. if some types of errors are missing in the training data, then the learned class-separating hypersurface in the feature space will often lead to false-positive classifications.

In such cases, it is not possible to train a two-class classifier in the standard way. Instead, one can use one-class learning, which gets by with data from one class during training.

Assume that there is no negative data are available. Positive data, i.e. data from error-free operation, however, are available in sufficient amount. Thus a learning algorithm is needed which can, based on the error-free data, capture all error-free operational states of the device and classify all others as negative.

For this purpose, there are a number of different algorithms known as one-class learning algorithms [Tax01], such as nearest neighbor data description (NNDD), support vector data description (SVDD, see also Sect. 9.6), etc. In statistics, these and related algorithms fall under the category of outlier detection.

One of the best known algorithms today is the local outlier factor (LOF, [BKNS00]), which calculates an outlier score for a point that is to be classified based on its density estimate. For large data sets with millions of points, and for high-dimensional data points with thousands of dimensions, the algorithms discussed so far are not suitable. The EXPoSE algorithm presented in [SEP16] has a failure rate about as low as that of LOF, but it is suitable for large high-dimensional data sets due to its constant computation time. With the simple example of NNDD we will now briefly introduce the one-class learning principle.

8.8.1 Nearest Neighbor Data Description

Similar to the nearest neighbor method from Sect. 8.3, the nearest neighbor data description algorithm belongs to the category of lazy learning algorithms. During learning, the only things that happen are normalization and storage of the feature vectors. Thus the qualifier "lazy" learning. Normalization of each individual feature is necessary in order to give each feature the same weight. Without normalization, a feature in the range $[0, 10^{-4}]$ would be lost in the noise of another feature in the range $[-10000, +10000]$. Thus all features are linearly scaled onto the interval [0, 1].[9] The actual nearest neighbor algorithm first comes into play during classification.

[9]Feature scaling is necessary or advantageous for many machine learning algorithms.

Let $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be a training set consiting of n feature vectors. A new point $\boldsymbol{q}$, which is yet to be classified, is accepted if

$$D(\boldsymbol{q}, \mathrm{NN}(\boldsymbol{q})) \leq \gamma \bar{D}, \tag{8.11}$$

that is, if the distance to a nearest neighbor is no larger than $\gamma\bar{D}$. Here $D(\boldsymbol{x}, \boldsymbol{y})$ is a distance metric, for example, as used here, the Euclidean distance. The function

$$\mathrm{NN}(\boldsymbol{q}) = \underset{\boldsymbol{x} \in \boldsymbol{X}}{\operatorname{argmin}} \{D(\boldsymbol{q}, \boldsymbol{x})\}$$

returns a nearest neighbor of $\boldsymbol{q}$ and

$$\bar{D} = \frac{1}{n} \sum_{i=1}^{n} D(\boldsymbol{x}_i, \mathrm{NN}(\boldsymbol{x}_i))$$

is the mean distance of the nearest neighbors to all data points in $\boldsymbol{X}$. To calculate $\bar{D}$ the distance of each point to its nearest neighbor is calculated. $\bar{D}$ is then the arithmetic mean of these calculated distances. One could use $\gamma = 1$ in Eq. 8.11, but would then obtain suboptimal classification results. It is better to determine the parameter γ using cross-validation (Sect. 8.5) such that the NNDD classifier's error rate is as small as possible.

The NNDD algorithm used here is a modification of the NNDD_T algorithm used in [Tax01], in which a point $\boldsymbol{q}$ is accepted if the distance to its nearest neighbor in the training data is no larger than the distance of the discovered nearest neighbor to its own nearest neighbor in the training data. Formally this reads

$$D(\boldsymbol{q}, \mathrm{NN}(\boldsymbol{q})) \leq D(\mathrm{NN}(\boldsymbol{q}), \mathrm{NN}(\mathrm{NN}(\boldsymbol{q}))). \tag{8.12}$$

This formula has several disadvantages compared to Inequality 8.5. First, the use of NNDD_T (Inequality 8.12) results in intuitively ugly lines of class separation, as shown in Fig. 8.32 right on page 225 in a simple two-dimensional example. In the center of Fig. 8.32 on page 225 we can see that the bottom left data point defines a large area of influence, despite or directly because of the fact that it is far from all other data points. NNDD with Eq. 8.5 on page 201 on the other hand yields the circular class partitions of a constant radius shown in Fig. 8.32 left on page 225. This intuitive conjecture is confirmed by practical experiments.

8.9 Clustering

If we search in a search engine for the term "mars", we will get results like "the planet mars" and "Chocolate, confectionery and beverage conglomerate" which are semantically quite different. In the set of discovered documents there are two

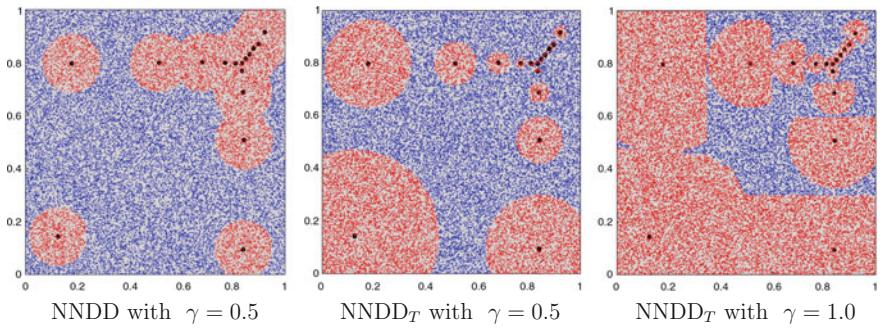


Fig. 8.32 NNDD and NNDD$_T$ applied to a set of 16 selected two-dimensional data points (black points). In each case, class membership has been determined for 10,000 randomly selected points. The points marked in red were classified as positive, i.e. assigned the training set's class, while the blue points are classified as negative

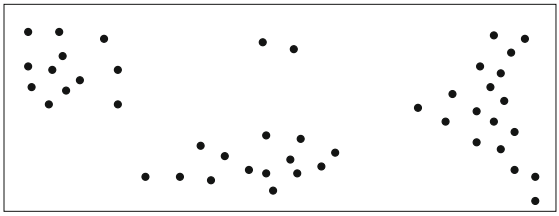


Fig. 8.33 Simple two-dimensional example with four clearly separated clusters

noticeably different *clusters*. Google, for example, still lists the results in an unstructured way. It would be better if the search engine separated the clusters and presented them to the user accordingly because the user is usually interested in only one of the clusters.

The distinction of *clustering* in contrast to supervised learning is that the training data are unlabeled. Thus the pre-structuring of the data by the supervisor is missing. Rather, finding structures is the whole point of clustering. In the space of training data, accumulations of data such as those in Fig. 8.33 are to be found. In a cluster, the distance of neighboring points is typically smaller than the distance between points of different clusters. Therefore the choice of a suitable distance metric for points, that is, for objects to be grouped and for clusters, is of fundamental importance. As before, we assume in the following that every data object is described by a vector of numerical attributes.

8.9.1 Distance Metrics

Accordingly for each application, the various distance metrics are defined for the distance d between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\mathbb{R}^n$. The most common is the Euclidean distance

$$d_e(\boldsymbol{x},\boldsymbol{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}.$$

Somewhat simpler is the sum of squared distances

$$d_q(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{n}(x_i - y_i)^2,$$

which, for algorithms in which only distances are compared, is equivalent to the Euclidean distance (Exercise 8.20 on page 242). Also used are the aforementioned Manhattan distance

$$d_m(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{n}|x_i - y_i|$$

as well as the distance of the maximum component

$$d_\infty(\boldsymbol{x},\boldsymbol{y}) = \max_{i=1,\ldots,n}|x_i - y_i|$$

which is based on the maximum norm. During text classification, the normalized projection of the two vectors on each other, that is, the normalized scalar product

$$\frac{\boldsymbol{x}\,\boldsymbol{y}}{|\boldsymbol{x}|\,|\boldsymbol{y}|}$$

is frequently calculated, where $|\boldsymbol{x}|$ is the Euclidian norm of $\boldsymbol{x}$. Because this formula is a metric for the similarity of the two vectors, as a distance metric the inverse

$$d_s(\boldsymbol{x},\boldsymbol{y}) = \frac{|\boldsymbol{x}|\,|\boldsymbol{y}|}{\boldsymbol{x}\,\boldsymbol{y}}$$

can be used, or “>” and “<” can be swapped for all comparisons. In the search for a text, the attributes $x_1, \ldots, x_n$ are calculated similarly to naive Bayes as components of the vector $\boldsymbol{x}$ as follows. For a dictionary with 50,000 words, the value x_i equals the frequency of the ith dictionary word in the text. Since normally almost all components are zero in such a vector, during the calculation of the scalar product, nearly all terms of the summation are zero. By exploiting this kind of information, the implementation can be sped up significantly (Exercise 8.21 on page 243).

8.9.2 k-Means and the EM Algorithm

Whenever the number of clusters is already known in advance, the *k-means* algorithm can be used. As its name suggests, k clusters are defined by their average

value. First the k cluster midpoints $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ are initialized to the coordinates of k randomly or manually selected data points. (Note: Selection of arbitrary points (which are not data points) as cluster centers may lead to empty clusters.) Then the following two steps are repeatedly carried out:

- Classification of all data to their nearest cluster midpoint
- Recomputation of the cluster midpoint.

The following scheme results as an algorithm:

K-MEANS$(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, k)$
initialize cluster centers $\boldsymbol{\mu}_1 = \boldsymbol{x}_{i_1}, \ldots, \boldsymbol{\mu}_k = \boldsymbol{x}_{i_k}$ (e.g. randomly)
Repeat
  classify $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ to each's nearest $\boldsymbol{\mu}_i$
  recalculate $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$
Until no change in $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$
Return$(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k)$

The calculation of the cluster midpoint $\boldsymbol{\mu}$ for points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l$ is done by

$$\boldsymbol{\mu} = \frac{1}{l} \sum_{i=1}^{l} x_i.$$

The execution on an example is shown in Fig. 8.34 for the case of two classes. We see how after three iterations, the class centers, which were first randomly chosen, stabilize. While this algorithm does not guarantee convergence, it usually converges very quickly. This means that the number of iteration steps is typically much smaller than the number of data points. Its complexity is $O(ndkt)$, where n is

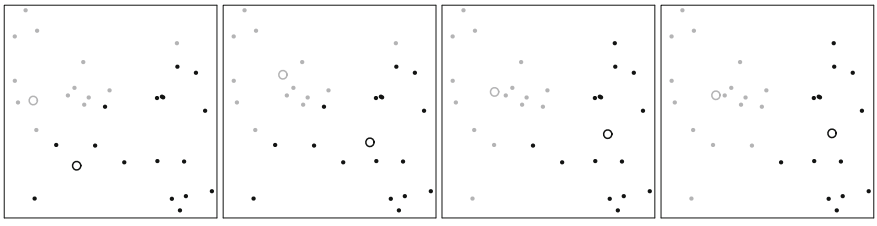


Fig. 8.34 k-means with two classes ($k = 2$) applied to 30 data points. *Far left* is the dataset with the initial centers, and to the *right* is the cluster after each iteration. After three iterations convergence is reached

the total number of points, d the dimensionality of the feature space, and t the number of iteration steps.

In many cases, the necessity of giving the number of classes in advance poses an inconvenient limitation. Therefore we will next introduce an algorithm which is more flexible.

Before that, however, we will mention the *EM algorithm*, which is a continuous variant of k-means, for it does not make a firm assignment of the data to classes, rather, for each point it returns the probability of it belonging to the various classes. Here we must assume that the type of probability distribution is known. Often the normal distribution is used. The task of the EM algorithm is to determine the parameters (mean $\boldsymbol{\mu}_i$ and covariance matrix Σ_i of the k multi-dimensional normal distributions) for each cluster. Similarly to k-means, the two following steps are repeatedly executed:

Expectation: For each data point the probability $P(C_j \,|\, \boldsymbol{x}_i)$ that it belongs to each cluster is calculated.

Maximization: Using the newly calculated probabilities, the parameters of the distribution are recalculated.

Thereby a softer clustering is achieved, which in many cases leads to better results. This alternation between expectation and maximization gives the algorithm its name. In addition to clustering, for example, the EM algorithm is used to learn Bayesian networks [DHS01].

8.9.3 Hierarchical Clustering

In hierarchical clustering we begin with n clusters consisting of one point each. Then the nearest neighbor clusters are combined until all points have been combined into a single cluster, or until a termination criterion has been reached. We obtain the scheme

HIERARCHICALCLUSTERING$(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, k)$
initialize $C_1 = \{\boldsymbol{x}_1\}, \ldots, C_n = \{\boldsymbol{x}_n\}$
Repeat
  Find two clusters C_i and C_j with the smallest distance
  Combine C_i and C_j
Until Termination condition reached
Return(tree with clusters)

The termination condition could be chosen as, for example, a desired number of clusters or a maximum distance between clusters. In Fig. 8.35 on page 229 this algorithm is represented schematically as a binary tree, in which from bottom to top

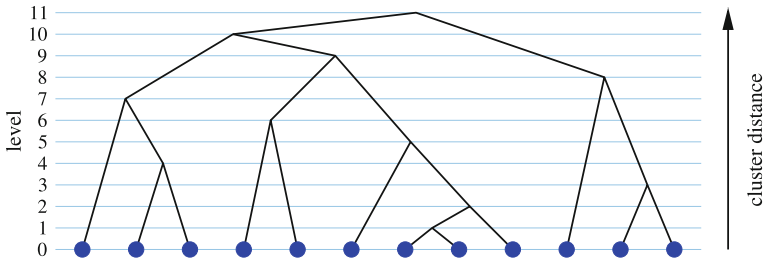


Fig. 8.35 In hierarchical clustering, the two clusters with the smallest distance are combined in each step

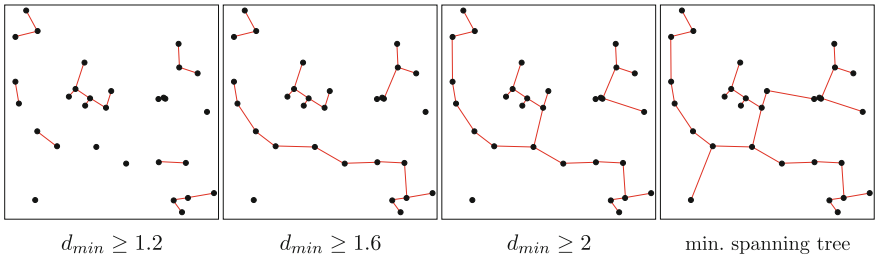


Fig. 8.36 The nearest neighbor algorithm applied to the data from Fig. 8.34 on page 227 at different levels with 12, 6, 3, 1 clusters

in each step, that is, at each level, two subtrees are connected. At the top level all points are unified into one large cluster.

It is so far unclear how the distances between the clusters are calculated. Indeed, in the previous section we defined various distance metrics for points, but these cannot be used on clusters. A convenient and often used metric is the distance between the two closest points in the two clusters C_i and C_j:

$$d_{min}(C_i, C_j) = \min_{\boldsymbol{x} \in \boldsymbol{C}_i,\, \boldsymbol{y} \in \boldsymbol{C}_j} d(\boldsymbol{x}, \boldsymbol{y}).$$

Thus we obtain the *nearest neighbor algorithm*, whose application is shown in Fig. 8.36.[10] We see that this algorithm generates a minimum spanning tree.[11] The example furthermore shows that the two described algorithms generate quite different clusters. This tells us that for graphs with clusters which are not clearly separated, the result depends heavily on the algorithm or the chosen distance metric.

For an efficient implementation of this algorithm, we first create an adjacency matrix in which the distances between all points is saved, which requires $O(n^2)$ time

[10] The nearest neighbor algorithm is not to be confused with the nearest neighbor method for classification from Sect. 8.3.

[11] A minimum spanning tree is an acyclic, undirected graph with the minimum sum of edge lengths.

and memory. If the number of clusters does not have an upper limit, the loop will iterate $n - 1$ times and the asymptotic computation time becomes $O(n^3)$.

To calculate the distance between two clusters, we can also use the distance between the two farthest points

$$d_{max}(C_i, C_j) = \max_{\boldsymbol{x} \in C_i,\, \boldsymbol{y} \in C_j} d(\boldsymbol{x}, \boldsymbol{y})$$

and obtain the *farthest neighbor algorithm*. Alternatively, the distance of the cluster's midpoint $d_\mu\,(C_i, C_j) = d(\boldsymbol{\mu}_i, \boldsymbol{\mu}_j)$ is used. Besides the clustering algorithm presented here, there are many others, for which we direct the reader to [DHS01] for further study.

8.9.4 How is the Number of Clusters Determined?

In all of the clustering algorithms discussed so far, the user must specify the number of clusters. In many cases, users have no idea what a sensible number of clusters would be, rather they simply want a "good" partitioning, i.e. separation of their n data points into clusters. Because there is, unfortunately, no absolute and generally accepted standard for the quality of a partition, we present a approved heuristic method for evaluating a clustering using the **silhouette width criterion** presented in [Rou87].

Before we discuss this criterion, let us first show how it can be applied. Assume that we had a program that could combinatorially enumerate all possible partitions for a set of n data points. Then we could simply apply the silhouette width criterion to each partition and use the maximum to determine the best partition. But because the number of partitions grows quickly with the number of data points, this method is not practical. We could, however, simply apply one of the presented clustering algorithms (for example k-means), let it run for k from 1 to n, and then use the silhouette width criterion to determine the best k and its respective partition.

What we are still missing is such a criterion that measures the quality of a partition. The idea, very roughly, is as follows: the mean distance between two arbitrary points within the same cluster should be smaller than the distance between two arbitrary points that are in different neighboring clusters. The ratio of the mean distance between points in neighboring clusters and the mean distance between points within the cluster should be maximized.

Let the data points $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be given, as well as a clustering function

$$c : \boldsymbol{x}_i \mapsto c(\boldsymbol{x}_i),$$

which assigns each point to a cluster. Let $\bar{d}(i, \ell)$ be the mean distance from $\boldsymbol{x}_i$ to all points $(\neq \boldsymbol{x}_i)$ in cluster ℓ. Then $a(i) = \bar{d}(i, c(\boldsymbol{x}_i))$ is the mean distance from $\boldsymbol{x}_i$ to all other points in its own cluster. If $\boldsymbol{x}_i$ is the only point in the cluster, we set $a(i) = 0$.

$$b(i) = \min_{j \neq c(\boldsymbol{x}_i)} \{\bar{d}(i, j)\}$$

is the smallest mean distance from point $\boldsymbol{x}_i$ to a cluster to which $\boldsymbol{x}_i$ does not belong.

We now define the function

$$s(i) = \begin{cases} 0 & \text{if } a(i) = 0 \\ \dfrac{b(i) - a(i)}{\max\{a(i), b(i)\}} & \text{otherwise} \end{cases}$$

which measures how well the point $\boldsymbol{x}_i$ fits into a cluster. Then we have $-1 \leq s(i) \leq 1$ and

$$s(i) = \begin{cases} 1 & \text{if } x_i \text{ is in the middle of a clearly delineated cluster.} \\ 0 & \text{if } x_i \text{ lies on the border of two clusters.} \\ -1 & \text{if } x_i \text{ is in the "wrong" cluster.} \end{cases}$$

Reflecting the idea formulated above, we now seek a partition that maximizes the mean

$$S = \frac{1}{n} \sum_{i=1}^{n} s(i)$$

of $s(i)$ over all points, called **silhouette width criterion**. For k-means clustering, this can be done with the OMRk algorithm presented in [BJCdC14]:

```
OMRk(x_1, ..., x_n, p, k_max)
S* = −∞
For k = 2 To k_max
    For i=1 To p
        Generate a random partition with k clusters (initialization)
        Obtain a partition P with k-means
        Determine S for P
        If S > S* Then
            S* = S; k* = k; P* = P
Return (k*, P*)
```

This algorithm repeatedly applies k-means for different values of k. Because the result of k-means depends heavily on its initialization, for every k, p different random initializations are tried in the inner loop, and then the function returns the optimal k^* and the corresponding best partition P^*. The OMRk algorithm can also be used with other clustering algorithms such as the EM algorithm and hierarchical clustering.

Example 4 The top left diagram in Fig. 8.37 on page 232 shows a set of two-dimensional data points with four obvious clusters. The OMRk algorithm was run on this data with $p = 30$ and $k_{max} = 9$. In the following eight diagrams, the figure shows the best partition together with its quality S for each k. The algorithm finds the maximum value $S = 0.786$ at $k = 5$. This does not reflect the natural (to

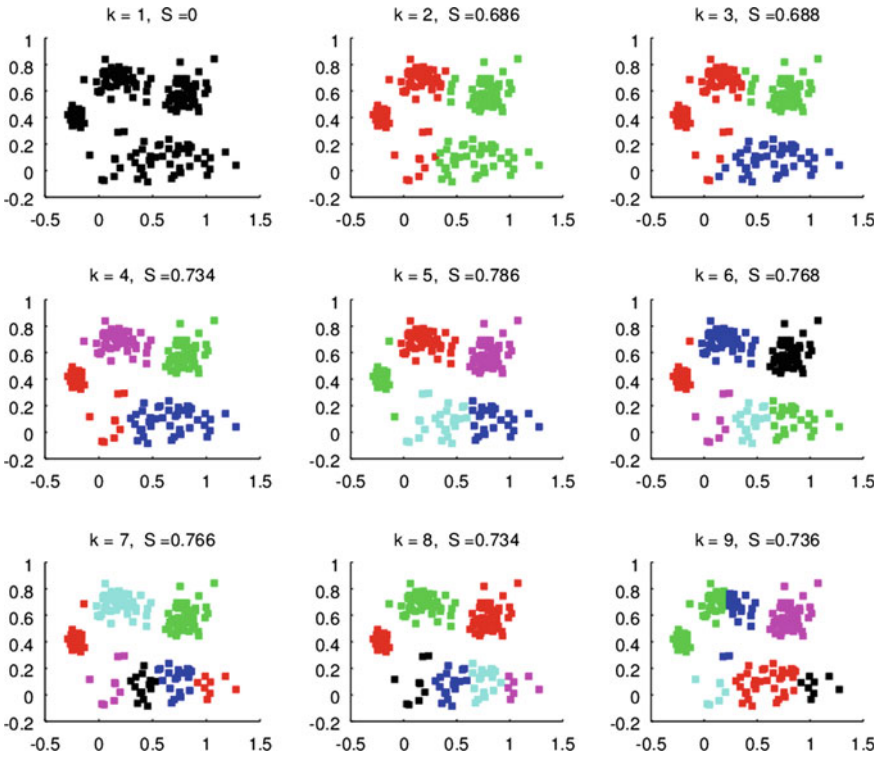


Fig. 8.37 Results of the OMRk algorithm for $k = 2$ to 9. The best value of $S = 0.786$ was found with $k = 5$

the human eye) grouping of the points into four clusters. In the partition found for $k = 4$, several points which should belong to the blue cluster are assigned to the red cluster. This is because k-means minimizes the distance to the cluster center point, and the points are closer to the center of the red cluster. The higher density of points in the red cluster is not taken into account.

The EM algorithm, which can approximate the difference in point density by using a normal distribution, performs significantly better. As shown in Fig. 8.38 on page 233, the EM algorithm finds almost exactly the same aforementioned natural distribution for $k = 4$.

Finally we should reiterate that all of the methods we have described are only heuristic greedy search algorithms which do not explore the entire space of all partitions. The silhouette width criterion described is only a heuristic estimation of a partition's "quality". There can be no absolute measure of quality for partitions because, as we have shown in even such a simple two-dimensional example, different people can in certain cases prefer very different groupings. We should also mention that there are many other interesting clustering algorithms such as the density-based DBSCAN algorithm.

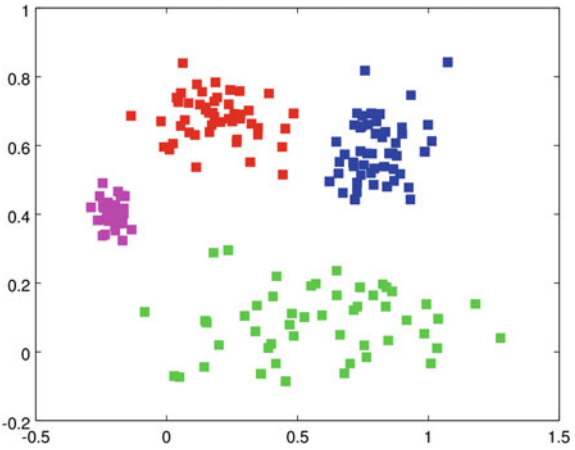


Fig. 8.38 A partition generated by the EM algorithm with $k = 4$

8.10 Data Mining in Practice

All the learning algorithms presented so far can be used as tools for data mining. For the user it is, however, sometimes quite troublesome to get used to new software tools for each application and furthermore to put the data to be analyzed into the appropriate format for each particular case.

A number of data mining systems address these problems. Most of these systems offer a convenient graphical user interface with diverse tools for visualization of the data, for preprocessing such as manipulation of missing values, and for analysis. For analysis, the learning algorithms presented here are used, among others.

The comprehensive open-source Java library WEKA deserves a special mention. It offers a large number of algorithms and simplifies the development of new algorithms.

The freely available system KNIME, which we will briefly introduce in the following section, offers a convenient user interface and all the types of tools mentioned above. KNIME also uses WEKA modules. Furthermore it offers a simple way of controlling the data flow of the chosen visualization, preprocessing, and analysis tools with a graphical editor. A large number of other systems meanwhile offer quite similar functionality, such as the open-source project RapidMiner (www.rapidminer.com), the system Clementine (www.spss.com/clementine) sold by SPSS, and the KXEN analytic framework (www.kxen.com).

8.10.1 The Data Mining Tool KNIME

Using the LEXMED data, we will now show how to extract knowledge from data using *KNIME* (Konstanz Information Miner, www.knime.org). First we generate a decision tree as shown in Fig. 8.39 on page 234. After creating a new project, a workflow is built graphically. To do this, the appropriate tools are simply taken out of the node repository with the mouse and dragged into the main workflow window.

The training and test data from the C4.5 file can be read in with the two file reader nodes without any trouble. These nodes can, however, also be quite easily configured for other file formats. The sideways traffic light under the node shows its status (not ready, configured, executed). Then node J48 is selected from the WEKA library [WF01], which contains a Java implementation of C4.5. The configuration for this is quite simple. Now a predictor node is chosen, which applies the generated tree to the test data. It inserts a new column into the test data table "Prediction" with the classification generated by the tree. From there the scorer node calculates the confusion matrix shown in the figure, which gives the number of correctly classified cases for both classes in the diagonal, and additionally the number of false positive and false negative data points.

Once the flow is completely built and all nodes configured, then an arbitrary node can be executed. It automatically ensures that predecessor nodes are executed, if necessary. The J48 node generates the view of the decision tree, shown in the right of the figure. This tree is identical with the one generated by C4.5 in Sect. 8.4.5, although here the node `TRekt<=378` is shown collapsed.

For comparison, a project for learning a multilayer perceptron (see Sect. 9.5) is shown in Fig. 8.40 on page 235. This works similarly to the previously introduced

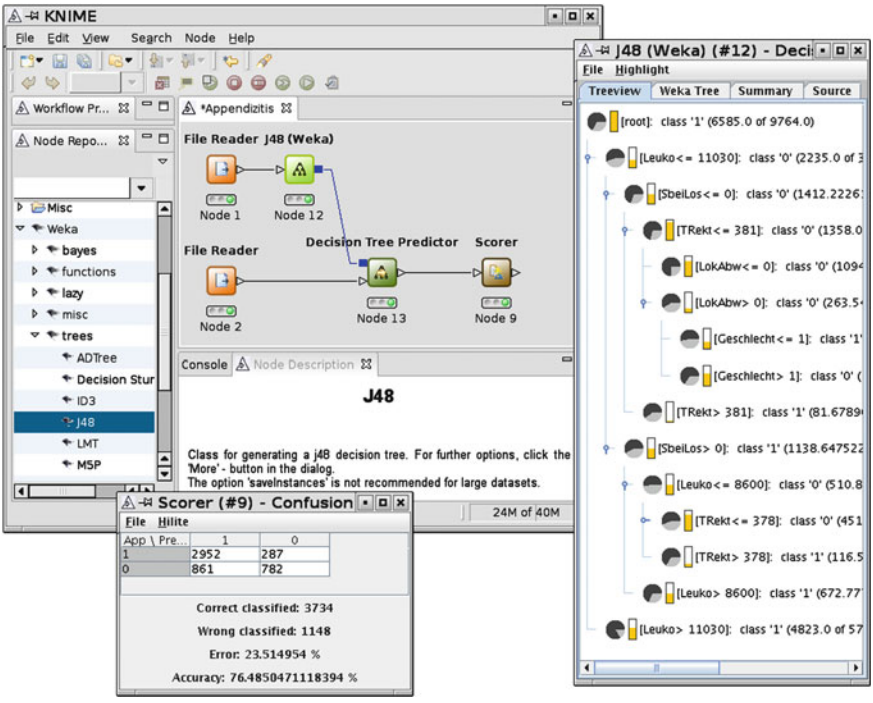


Fig. 8.39 The KNIME user interface with two additional views, which show the decision tree and the confusion matrix

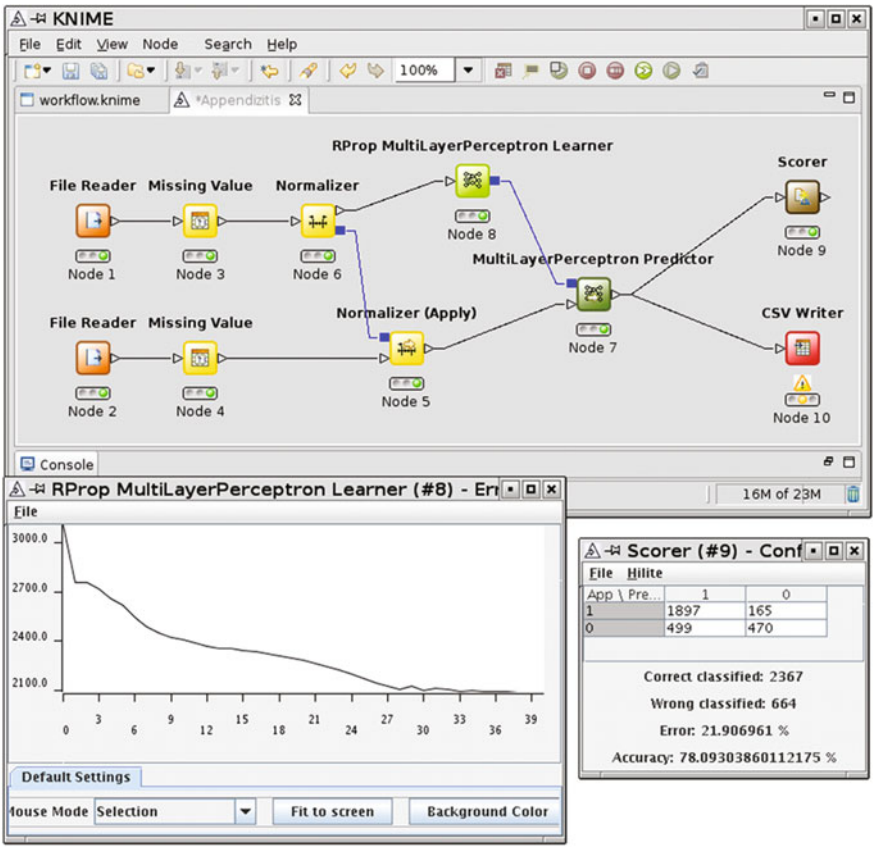


Fig. 8.40 The KNIME user interface with the workflow window, the learning curve, and the confusion matrix

linear perceptron, but it can also divide non linearly separable classes. Here the flow is somewhat more complex. An extra node is needed for error handling missing values when preprocessing each of the two files. We set it so that those lines will be deleted. Because neural networks cannot deal with arbitrary values, the values of all variables are scaled linearly into the interval [0, 1] using the "normalizer" node.

After applying the RProp learner, an improvement on backpropagation (see Sect. 9.5), we can analyze the progression in time of the approximation error in the learning curve shown. In the confusion matrix, the scorer outputs the analysis of the test data. The "CSV Writer" node at the bottom right serves to export the result files, which can then be used externally to generate the ROC curve shown in Fig. 7.10 on page 156, for which there is unfortunately no KNIME tool (yet).

In summary, we can say the following about KNIME (and similar tools): for projects with data analysis requirements which are not too exotic, it is worthwhile to work with such a powerful workbench for analysis of data. The user already saves a lot of time with the preprocessing stage. There is a large selection of easily usable "mining tools", such as nearest neighbor classifiers, simple Bayesian network learning algorithms, as well as the k-means clustering algorithm (Sect. 8.9.2). Evaluation of the results, for example cross validation, can be easily carried out. It remains to be mentioned, that besides those shown, there are many other tools for visualization of the data. Furthermore, the developers of KNIME have made an extension to KNIME available, with which the user can program his own tools in Java or Python.

However, it should also be mentioned that the user of this type of data mining system should bring along solid prior knowledge of machine learning and the use of data mining techniques. The software alone cannot analyze data, but in the hand of a specialist it becomes a powerful tool for the extraction of knowledge from data. For the beginner in the fascinating field of machine learning, such a system offers an ideal and simple opportunity to test one's knowledge practically and to compare the various algorithms. The reader may verify this in Exercise 8.22 on page 243.

8.11 Summary

We have thoroughly covered several algorithms from the established field of supervised learning, including decision tree learning, Bayesian networks, and the nearest neighbor method. These algorithms are stable and efficiently usable in various applications and thus belong to the standard repertoire in AI and data mining. The same is true for the clustering algorithms, which work without a "supervisor" and can be found, for example, in search engine applications. Reinforcement learning as another field of machine learning uses no supervisor either. In contrast to supervised learning, where the learner receives the correct actions or answers as the labels in the training data, in reinforcement learning only now and then positive or negative feedback is received from the environment. In Chap. 10 we will show how this works. Not quite as hard is the task in semi-supervised learning, a young sub-area of machine learning, where only very few out of a large number of training data are labeled.

Supervised learning is now a well established area with lots of successful applications. For supervised learning of data with continuous labels any function approximation algorithm can be employed. Thus there are many algorithms from various areas of mathematics and computer science. In Chap. 9 we will introduce various types of neural networks, least squares algorithms and support vector machines, which are all function approximators. Nowadays, Gaussian processes are very popular because they are very universal and easy to apply and provide the user with an estimate of the uncertainty of the output values [RW06].

The following taxonomy gives an overview of the most important learning algorithms and their classification.

Supervised learning

- Lazy learning
  - k nearest neighbor method (classification + approximation)
  - Locally weighted regression (approximation)
  - Case-based learning (classification + approximation)
- Eager learning
  - Decision trees induction (classification)
  - Learning of Bayesian networks (classification + approximation)
  - Neural networks (classification + approximation)
  - Gaussian processes (classification + approximation)
  - Support vector machines
  - Wavelets, splines, radial basis functions, …

Unsupervised learning (*clustering*)

- Nearest neighbor algorithm
- Farthest neighbor algorithm
- k-means
- Neural networks

Reinforcement learning

- Value iteration
- Q learning
- TD learning
- Policy gradient methods
- Neural networks

What has been said about supervised learning is only true, however, when working with a fixed set of known attributes. An interesting, still open field under intensive research is automatic feature selection. In Sect. 8.4, for learning with decision trees, we presented an algorithm for the calculation of the information gain of attributes that sorts the attributes according to their relevance and uses only those which improve the quality of the classification. With this type of method it is possible to automatically select the relevant attributes from a potentially large base set. This base set, however, must be manually selected.

Still open and also not clearly defined is the question of how the machine can find new attributes. Let us imagine a robot which is supposed to pick apples. For this he must learn to distinguish between ripe and unripe apples and other objects. Traditionally we would determine certain attributes such as the color and form of pixel regions and then train a learning algorithm using manually classified images. It is also possible that for example a neural network could be trained directly with all pixels of the image as input, which for high resolution is linked with severe computation time problems, however. Approaches which automatically make suggestions for relevant features would be desired here.

Clustering provides one approach to feature selection. Before training the apple recognition machine, we let clustering run on the data. For (supervised) learning of the classes *apple* and *non apple*, the input is no longer all of the pixels, rather only the classes found during clustering, potentially together with other attributes. Clustering at any rate can be used for automatic, creative "discovery" of features. It is, however, uncertain whether the discovered features are relevant. The relatively young but successful deep learning algorithms, which combine large, complex neural networks with unsupervised pre-processing, represent a breakthrough in feature generation, and therefore a milestone in AI. We will cover this topic in Sect. 9.7.

The following problem is yet more difficult: assume that the video camera used for apple recognition only transmits black and white images. The task can no longer be solved well. It would be nice if the machine would be creative on its own account, for example by suggesting that the camera should be replaced with a color camera. This would be asking for too much today.

In addition to specialized works about all subfields of machine learning, there are very good textbooks such as [Mit97, Bis06, DHS01, HTF09, Alp04]. Above all, I recommend the excellent book by Peter Flach [Fla12], which paves the way to a deep understanding of the concepts and algorithms of machine learning with very good explanations and examples. For current research results, a look into the freely available Journal of Machine Learning Research (http://jmlr.csail.mit.edu), the Machine Learning Journal, as well as the proceedings of the International Conference on Machine Learning (ICML) is recommended. For every developer of learning algorithms, the Machine Learning Repository [DNM98] of the University of California at Irvine (UCI) is interesting, with its large collection of training and test data for learning algorithms and data mining tools. MLOSS, which stands for machine learning open source software, is an excellent directory of links to freely available software (www.mloss.org).

8.12 Exercises

8.12.1 Introduction

Exercise 8.1

(a) Specify the task of an agent which should predict the weather for the next day given measured values for temperature, air pressure, and humidity. The weather should be categorized into one of the three classes: sunny, cloudy, and rainy.

(b) Describe the structure of a file with the training data for this agent.

Exercise 8.2 Show that the correlation matrix is symmetric and that all diagonal elements are equal to 1.

8.12.2 The Perceptron

Exercise 8.3 Apply the perceptron learning rule to the sets

$$M_+ = \{(0, 1.8), (2, 0.6)\} \quad \text{and} \quad M_- = \{(-1.2, 1.4), (0.4, -1)\}$$

from Example 8.2 on page 186 and give the result of the values of the weight vector.

Exercise 8.4 Given the table to the right with the training data:

(a) Using a graph, show that the data are linearly separable.
(b) Manually determine the weights w_1 and w_2, as well as the threshold Θ of a perceptron (with threshold) which correctly classifies the data.
(c) Program the perceptron learning rule and apply your program to the table. Compare the discovered weights with the manually calculated ones.

Num.	x_1	x_2	Class
1	6	1	0
2	7	3	0
3	8	2	0
4	9	0	0
5	8	4	1
6	8	6	1
7	9	2	1
8	9	5	1

Exercise 8.5

(a) Give a visual interpretation of the heuristic initialization

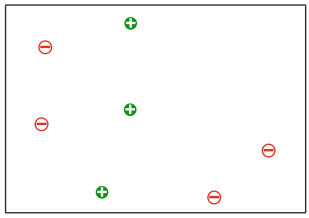
$$\boldsymbol{w_0} = \sum_{x_i \in M_+} x_i - \sum_{x_i \in M_-} x_i,$$

of the weight vector described in Sect. 8.2.2.
(b) Give an example of a linearly separable dataset for which this heuristic does not produce a dividing line.

8.12.3 Nearest Neighbor Method

Exercise 8.6

(a) Show the Voronoi diagram for neighboring sets of points.
(b) Then draw in the class division lines.



Exercise 8.7 Let the table with training data from Exercise 8.4 be given. In the following, use the Manhattan distance $d(\boldsymbol{a}, \boldsymbol{b})$, defined as $d(\boldsymbol{a}, \boldsymbol{b}) = |a_1 - b_1| +$

$|a_2 - b_2|$, to determine the distance d between two data points $\boldsymbol{a} = (a_1, a_2)$ and $\boldsymbol{b} = (b_1, b_2)$.

(a) Classify the vector $\boldsymbol{v} = (8, 3.5)$ with the nearest neighbor method.
(b) Classify the vector $\boldsymbol{v} = (8, 3.5)$ with the k nearest neighbor method for $k = 2, 3, 5$.

❄**Exercise 8.8**

(a) Show that in a two-dimensional feature space it is reasonable, as claimed in (8.3) on page 194, to weight the k nearest neighbors by the inverse of the squared distance.
(b) Why would a weighting using $w_i' = \frac{1}{1 + \alpha d(\boldsymbol{x}, \boldsymbol{x_i})}$ make less sense?

Exercise 8.9

(a) Write a program implementing the k-Nearest-Neighbor-Method for classification.
(b) Apply this program for $k = 1$ (i.e. one nearest neighbor) to the Lexmed-data from http://www.hs-weingarten.de/~ertel/kibuch/uebungen/. Use for training the file app1.data and for testing the file app1.test. Do not forget to normalize all features before training.
(c) Apply leave-one-out cross-validation on the whole data set app1.data ∪ app1.test to determine the optimal number of nearest neighbors k and determine the classification error.
(d) Repeat the cross-validation with not normalized data.
(e) Compare your results with those given in Fig. 9.14 on page 266 for the least squares method and RProp.

8.12.4 Decision Trees

Exercise 8.10 Show that the definition $0 \log_2 0 := 0$ is reasonable, in other words, that the function $f(x) = x \log_2 x$ thereby becomes continuous in the origin.

Exercise 8.11 Determine the entropy for the following distributions.

(a) $(1, 0, 0, 0, 0)$ (b) $\left(\frac{1}{2}, \frac{1}{2}, 0, 0, 0\right)$ (c) $\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{4}, 0, 0\right)$
(d) $\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}\right)$ (e) $\left(\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}\right)$ (f) $\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \ldots\right)$

Exercise 8.12

(a) Show that the two different definitions of entropy from (7.9) on page 138 and Definition 8.4 on page 202 only differ by a constant factor, that is, that

$$\sum_{i=1}^{n} p_i \log_2 p_i = c \sum_{i=1}^{n} p_i \ln p_i$$

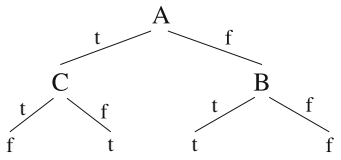
and give the constant c.

(b) Show that for the MaxEnt method and for decision trees it makes no difference which of the two formulas we use.

Exercise 8.13 Develop a decision tree for the dataset D from Exercise 8.4 on page 239.

(a) Treat both attributes as discrete.
(b) Now treat attribute x_2 as continuous and x_1 as discrete.
(c) Have C4.5 generate a tree with both variants. Use `-m 1 -t 10` as parameters in order to get different suggestions.

Exercise 8.14 Given the following decision tree and tables for both training and test data:



Training data

A	B	C	Class
t	t	f	t
t	f	f	t
t	t	t	f
f	f	t	f
f	f	f	f
f	t	t	t

Test data

A	B	C	Class
t	t	f	t
t	f	f	t
f	t	f	f
f	f	t	f

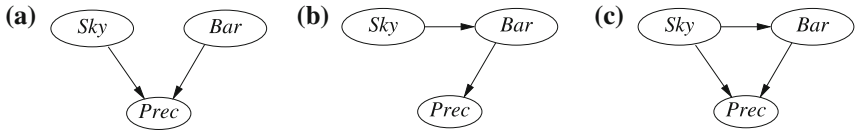
(a) Give the correctness of the tree for the training and test data.
(b) Give a propositional logic formula equivalent to the tree.
(c) Carry out pruning on the tree, draw the resulting tree, and give its correctness for the training and test data.

❄**Exercise 8.15**

(a) When determining the current attribute, the algorithm for generating decision trees (Fig. 8.26 on page 207) does not eliminate the attributes which have already been used further up in the tree. Despite this, a discrete attribute occurs in a path at most once. Why?
(b) Why can continuous attributes occur multiple times?

8.12.5 Learning of Bayesian Networks

Exercise 8.16 Use the distribution given in Exercise 7.3 on page 172 and determine the CPTs for the three Bayesian networks:



(d) Determine the distribution for the two networks from (a) and (b) and compare these with the original distribution. Which network is "better"?
(e) Now determine the distribution for network (c). What does occur to you? Justify!

❄❄ **Exercise 8.17** Show that for binary variables $S_1, \ldots, S_n$ and binary class variable K, a linear score of the form

$$\text{decision} = \begin{cases} \text{positive} & \text{if } w_1 S_1 + \cdots + w_n S_n > \Theta \\ \text{negative} & \text{else} \end{cases}$$

is equally expressive in relation to the perceptron and to the naive Bayes classifier, which both decide according to the formula

$$\text{decision} = \begin{cases} \text{positive} & \text{if } P(K|S_1, \ldots, S_n) > 1/2, \\ \text{negative} & \text{else} \end{cases}$$

Exercise 8.18 In the implementation of text classification with naive Bayes, exponent underflow can happen quickly because the factors $P(w_i\,|K)$ (which appear in (8.10) on page 222) are typically all very small, which can lead to extremely small results. How can we mitigate this problem?

❄❄ **Exercise 8.19** Write a program for naive Bayes text analysis. Then train and test it on text benchmarks using a tool of your choice. Counting the frequency of words in the text can be done easily in Linux with the command

```
cat <datei> | tr -d "[:punct:]" | tr -s "[:space:]" "\n" | sort | uniq -ci
```

Obtain the Twenty Newsgroups data by Tom Mitchell in the UCI machine learning benchmark collection (Machine Learning Repository) [DNM98]. There you will also find a reference to a naive Bayes program for text classification by Mitchell.

8.12.6 Clustering

Exercise 8.20 Show that for algorithms which only compare distances, applying a strictly monotonically increasing function f to the distance makes no difference. In other words you must show that the distance $d_1(x, y)$ and the distance $d_2(x, y)$: = $f(d_1(x,y))$ lead to the same result with respect to the ordering relation.

Exercise 8.21 Determine the distances d_s (scalar product) of the following texts to each other.

x_1: We will introduce the application of naive Bayes to text analysis on a short example text by Alan Turing from [Tur50].

x_2: We may hope that machines will eventually compete with men in all purely intellectual fields. But which are the best ones to start with?

x_3: Again I do not know what the right answer is, but I think both approaches should be tried.

8.12.7 Data Mining

Exercise 8.22 Use KNIME (www.knime.de) and

(a) Load the example file with the Iris data from the KNIME directory and experiment with the various data representations, especially with the scatter-plot diagrams.

(b) First train a decision tree for the three classes, and then train an RProp network.

(c) Load the appendicitis data on this book's website. Compare the classification quality of the k nearest neighbor method to that of an RProp network. Optimize k as well as the number of hidden neurons of the RProp network.

(d) Obtain a dataset of your choice from the UCI data collection for data mining at http://kdd.ics.uci.edu or for machine learning at http://mlearn.ics.uci.edu/MLRepository.html and experiment with it.