Chapter 4
Data Mining

Process mining builds on two pillars: (a) process modeling and analysis (as described in Chap. 3) and (b) data mining. This chapter introduces some basic data mining approaches and structures the field. The motivation for doing so is twofold. On the one hand, some process mining techniques build on classical data mining techniques, e.g., discovery and enhancement approaches focusing on data and resources. On the other hand, ideas originating from the data mining field will be used for the evaluation of process mining results. For example, one can adopt various data mining approaches to measure the quality of the discovered or enhanced process models. Existing data mining techniques are of little use for control-flow discovery, conformance checking, and other process mining tasks. Nevertheless, a basic understanding of data mining is most helpful for fully understanding the process mining techniques presented in subsequent chapters.

4.1 Classification of Data Mining Techniques

In [69] data mining is defined as "the analysis of (often large) data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner". The input data is typically given as a table and the output may be rules, clusters, tree structures, graphs, equations, patterns, etc. The growth of the "digital universe" described in Chap. 2 is the main driver for the popularity of data mining. Initially, the term "data mining" had a negative connotation especially among statisticians. Terms like "data snooping", "fishing", and "data dredging" refer to ad-hoc techniques to extract conclusions from data without a sound statistical basis. However, over time the data mining discipline has become mature as characterized by solid scientific methods and many practical applications [9, 24, 69, 102, 190].

W. van der Aalst, *Process Mining*, DOI 10.1007/978-3-662-49851-4_4

Table 4.1 Data set 1: Data about 860 recently deceased persons to study the effects of drinking, smoking, and body weight on the life expectancy

Drinker	Smoker	Weight	Age
yes	yes	120	44
no	no	70	96
yes	no	72	88
yes	yes	55	52
no	yes	94	56
no	no	62	93
…	…	…	…

4.1.1 Data Sets: Instances and Variables

Let us first look at three example data sets and possible questions. Table 4.1 shows part of a larger table containing information about 860 individuals that have recently deceased. For each person the age of death is recorded (column *age*). Column *drinker* indicates whether the person was drinking alcohol. Column *smoker* indicates whether the person was smoking. Column *weight* indicates the bodyweight of the deceased person. Each row in Table 4.1 corresponds to a person. Questions may be:

- What is the effect of smoking and drinking on a person's bodyweight?
- Do people that smoke also drink?
- What factors influence a person's life expectancy the most?
- Can one identify groups of people having a similar lifestyle?

Table 4.2 shows another data set with information about 420 students that participated in a Bachelor program. Each row corresponds to a student. Students follow different courses. The table lists the highest mark for a particular course, e.g., the first student got a 9 for the course on linear algebra and an 8 for the course on logic. Table 4.2 uses the Dutch grading system, i.e., any mark is in-between 1 (lowest) and 10 (highest). Students who have a 5 or less, fail for the course. A "–" means

Table 4.2 Data set 2: Data about 420 students to investigate relationships among course grades and the student's overall performance in the Bachelor program

Linear algebra	Logic	Programming	Operations research	Workflow systems	…	Duration	Result
9	8	8	9	9	…	36	cum laude
7	6	–	8	8	…	42	passed
–	–	5	4	6	…	54	failed
8	6	6	6	5	…	38	passed
6	7	6	–	8	…	39	passed
9	9	9	9	8	…	38	cum laude
5	5	–	6	6	…	52	failed
…	…	…	…	…	…	…	…

Table 4.3 Data set 3: Data on 240 customer orders in a coffee bar recorded by the cash register

Cappuccino	Latte	Espresso	Americano	Ristretto	Tea	Muffin	Bagel
1	0	0	0	0	0	1	0
0	2	0	0	0	0	1	1
0	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	1	2	0
0	0	0	1	1	0	0	0
...	...	...	...	...	...	...	...

that the course was not taken. The table shows only a selection of courses. Besides mandatory courses there are dozens of elective courses. The last two columns refer to the overall performance. The *duration* column indicates how long the student was enrolled before getting a degree or dropping out. The *result* column shows the final result: cum laude, passed, or failed. The university may be interested in the following questions:

- Are the marks of certain courses highly correlated?
- Which electives do excellent students (cum laude) take?
- Which courses significantly delay the moment of graduation?
- Why do students drop out?
- Can one identify groups of students having a similar study behavior?

The third data set, partly shown in Table 4.3, contains data about 240 orders in a café. Each row corresponds to one customer order. The columns refer to products. For instance, the first customer ordered a cappuccino and a muffin. This example is quite generic and analyzing such a data set is generally referred to as *market basket analysis*. For example, one can think of analyzing the product combinations purchased in a supermarket or in an electronic bookstore. Cafés, supermarkets, bookstores, etc. may be interested in the following questions:

- Which products are frequently purchased together?
- When do people buy a particular product?
- Is it possible to characterize typical customer groups?
- How to promote the sales of products with a higher margin?

Tables 4.1, 4.2, and 4.3 show three typical *data sets* used as input for data mining algorithms. Such a data set is often referred to as *sample* or *table*. The rows in the three tables are called *instances*. Alternative terms are: *individuals*, *entities*, *cases*, *objects*, and *records*. Instances may correspond to deceased persons, students, customers, orders, orderlines, messages, etc. The columns in the three tables are called *variables*. Variables are often referred to as *attributes*, *features*, or *data elements*. The first data set (Table 4.1) has four variables: *drinker*, *smoker*, *weight*, and *age*.

We distinguish between *categorical* variables and *numerical* variables. Categorical variables have a limited set of possible values and can easily be enumerated, e.g.,

a Boolean variable that is either true or false. Numerical variables have an ordering and cannot be enumerated easily. Examples are temperature (e.g., 39.7 degrees centigrade), age (44 years), weight (56.3 kilograms), number of items (3 coffees), and altitude (11 meters below sea level). Categorical variables are typically subdivided into *ordinal* variables and *nominal* variables. Nominal variables have no logical ordering. For example Booleans (true and false), colors (Red, Yellow, Green), and EU countries (Germany, Italy, etc.) have no commonly agreed upon logical ordering. Ordinal variables have an ordering associated to it. For example, the *result* column in Table 4.2 refers to an ordinal variable that can have values "cum laude", "passed", and "failed". For most applications it would make sense to consider the value "passed" in-between "cum laude" and "failed".

Before applying any data mining technique the data is typically preprocessed, e.g., rows and columns may be removed for various reasons. For instance, columns with less relevant information should be removed beforehand to reduce the dimensionality of the problem. Instances that are clearly corrupted should also be removed. Moreover, the value of a variable for a particular instance may be missing or have the wrong type. This may be due to an error while recording the data, but it may also have a particular reason. For example, in Table 4.2 some course grades are missing (denoted by "–"). These missing values are not errors but contain valuable information. For some kinds of analysis, the missing course grade can be treated as "zero", i.e., not taking the course is "lower" than the lowest grade. For other types of analysis it may be that the values in such a column are mapped onto "yes" (participated in the course) and "no" (the entries that now have a "–").

When comparing Tables 4.1, 4.2, and 4.3 with the event log shown in Table 2.1 it becomes obvious that data mining techniques make less assumptions about the format of the input data than process mining techniques. For example, in Table 2.1 there are two notions, events and cases, rather than the single notion of an instance (i.e., row in table). Moreover, events are ordered in time whereas in Tables 4.1, 4.2, and 4.3 the ordering of the rows has no meaning. For particular questions it is possible to convert an event log into a simple data set for data mining. We will refer to this as *feature extraction*. Later, we will use feature extraction for various proposes, e.g., analyzing decisions in a discovered process models and clustering cases before process discovery so that each cluster has a dedicated process model.

After showing the basic input format for data mining and discussing typical questions, we classify data mining techniques into two main categories: *supervised learning* and *unsupervised learning*.

4.1.2 Supervised Learning: Classification and Regression

Supervised learning assumes *labeled data*, i.e., there is a *response variable* that labels each instance. For instance, in Table 4.2 the *result* column could be selected as the response variable. Hence, each student is labeled as "cum laude", "passed", or "failed". The other variables are *predictor variables* and we are interested in

explaining the response variable in terms of the predictor variables. Sometimes the response variable is called the *dependent variable* and the predictor variables are called *independent variables*. The goal is to explain the dependent variable in terms of the independent variables. For example, we would like to predict the final result of a student in terms of the student's course grades.

Techniques for supervised learning can be further subdivided into *classification* and *regression* depending on the type of response variable (categorical or numerical).

Classification techniques assume a *categorical* response variable and the goal is to classify instances based on the predictor variables. Consider for example Table 4.1. We would like to classify people into the class of smokers and the class of non-smokers. Therefore, we select the categorical response variable *smoker*. Through classification we want to learn what the key differences between smokers and non-smokers are. For instance, we could find that most smokers drink and die young. By applying classification to the second data set (Table 4.2) while using column *result* as a response variable, we could find the obvious fact that cum laude students have high grades. In Sect. 4.2, we will show how to construct a so-called *decision tree* using classification.

Regression techniques assume a *numerical* response variable. The goal is to find a function that fits the data with the least error. For example, we could select *age* as response variable for the data set in Table 4.1 and (hypothetically) find the function $age = 124 - 0.8 \times weight$, e.g., a person of 50 kilogram is expected to live until the age of 84 whereas a person of 100 kilogram is expected to live until the age of 44. For the second data set we could find that the mark for the course on workflow systems heavily depends on the mark for linear algebra and logic, e.g., $workflow\ systems = 0.6 + 0.8 \times linear\ algebra + 0.2 \times logic$. For the third data set, we could (again hypothetically) find a function that predicts the number of bagels in terms of the numbers of different drinks.

The most frequently used regression technique is *linear regression*. Given a response variable y and predictor variables $x_1, x_2, \ldots, x_n$ a linear model $\hat{y} = f(x_1, x_2, \ldots, x_n) = a_0 + \sum_{i=1}^{n} a_i x_i$ is learned over the data set. For every instance in the data set there is an error $|y - \hat{y}|$. A popular approach is to minimize the sum of squared errors, i.e., given m instances the goal is to find a function f such that $\sum_{j=1}^{m} (y_j - \hat{y}_j)^2$ is minimal. Other scoring functions are possible and more *general regression models* or even *neural networks* can be used. However, these techniques are out of the scope of this book and the interested reader is referred to [69].

Classification requires a categorical response variable. In some cases it makes sense to transform a numerical response variable into a categorical one. For example, for Table 4.1 one could decide to transform variable *age* into a categorical response variable by mapping values below 70 onto label "young" and values of 70 and above onto label "old". Now a decision tree can be constructed to classify instances into people that die(d) "young" and people that die(d) "old". Similarly, all values in Table 4.3 can be made categorical. For example, positive values are mapped onto "true" (the item was purchased) and value 0 is mapped onto "false" (the item was not purchased). After applying this mapping to Table 4.3, we can

apply classification to the coffee shop data while using e.g. column *muffin* as a response variable. We could, for instance, find that customers who drink lots of tea tend to eat muffins.

4.1.3 Unsupervised Learning: Clustering and Pattern Discovery

Unsupervised learning assumes *unlabeled data*, i.e., the variables are *not* split into response and predictor variables. In this chapter, we consider two types of unsupervised learning: *clustering* and *pattern discovery*.

Clustering algorithms examine the data to find groups of instances that are similar. Unlike classification the focus is not on some response variable but on the instance as a whole. For example, the goal could be to find homogeneous groups of students (Table 4.2) or customers (Table 4.3). Well-known techniques for clustering are *k-means clustering* and *agglomerative hierarchical clustering*. These will be briefly explained in Sect. 4.3.

There are many techniques to discover patterns in data. Often the goal is to find rules of the form *IF X THEN Y* where X and Y relate values of different variables. For example, *IF smoker* $=$ *no AND age* ≥ 70 *THEN drinker* $=$ *yes* for Table 4.1 or *IF logic* ≤ 6 *AND duration* > 50 *THEN result* $=$ *failed* for Table 4.2. The most well-known technique is *association rule mining*. This technique will be explained in Sect. 4.4.

Note that decision trees can also be converted into rules. However, a decision tree is constructed for a particular response variable. Hence, rules extracted from a decision tree only say something about the response variable in terms of some of the predictor variables. Association rules are discovered using unsupervised learning, i.e., there is no need to select a response variable.

Data mining results may be both *descriptive* and *predictive*. Decision trees, association rules, regression functions say something about the data set used to learn the model. However, they can also be used to make predictions for new instances, e.g., predict the overall performance of students based on the course grades in the first semester.

In the remainder, we show some of the techniques mentioned in more detail. Moreover, at the end of this chapter we focus on measuring the quality of mining results.

4.2 Decision Tree Learning

Decision tree learning is a supervised learning technique aiming at the classification of instances based on predictor variables. There is one categorical response variable labeling the data and the result is arranged in the form of a tree. Figures 4.1, 4.2, and 4.3 show three decision trees computed for the data sets described earlier in this chapter. Leaf nodes correspond to possible values of the response variable. Non-leaf

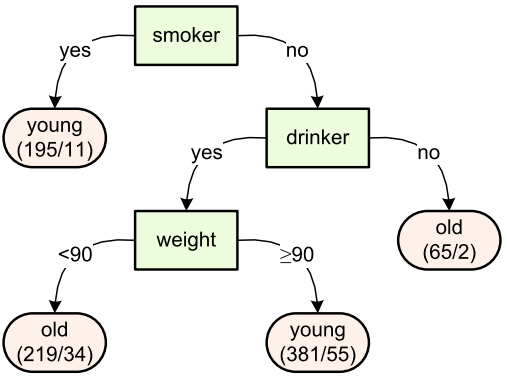


Fig. 4.1 A decision tree derived from Table 4.1. The 860 persons are classified into "young" (died before the age of 70) and "old" (died at 70 or later). People who smoke generally die young (195 persons of which 11 are misclassified). People who do not smoke and do no drink tend to live long (65 persons of which 2 are misclassified). People who only drink but are overweight (≥ 90) also die young (381 persons of which 55 are misclassified)

nodes correspond to predictor variables. In the context of decision tree learning, predictor variables are referred to as *attributes*. Every attribute node splits a set of instances into two or more subsets. The root node corresponds to all instances.

In Fig. 4.1, the root node represents all instances; in this case 860 persons. Based on the attribute *smoker* these instances are split into the ones that are smoking (195 persons) and the ones that not smoking ($860 - 195 = 665$ persons). The smokers are not further split. Based on this information instances are already labeled as "young", i.e., smokers are expected to die before the age of 70. The non-smokers are split into drinkers and non-drinkers. The latter group of people is expected to live long and is thus labeled as "old". All leaf nodes have two numbers. The first number indicates the number of instances classified as such. The second number indicates the number of instances corresponding to the leaf node but wrongly classified. Of the 195 smokers who were classified as "young" 11 people were misclassified, i.e., did not die before 70 while smoking.

The other two decision trees can be read in the same manner. Based on an attribute, a set of instances may also be split into three (or even more) subsets. An attribute may appear multiple times in a tree but not twice on the same path. For example, in Fig. 4.2 there are two nodes referring to the course on linear algebra. However, these are not on the same path and thus refer to disjoint sets of students. As mentioned before there are various ways to handle missing values depending on their assumed semantics. In Fig. 4.2, a missing course grade is treated as a kind of "zero" (see the left-most arc originating from the root node).

Decision trees such as the ones shown in Figs. 4.1, 4.2, and 4.3 can be obtained using a variety of techniques. Most of the techniques use a recursive top-down algorithm that works as follows:

1. Create the root node r and associate all instances to the root node. $X := \{r\}$ is the set of nodes to be traversed.

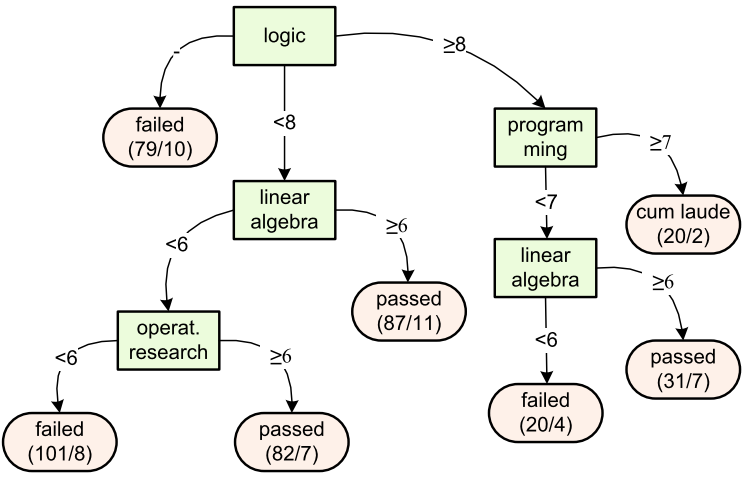
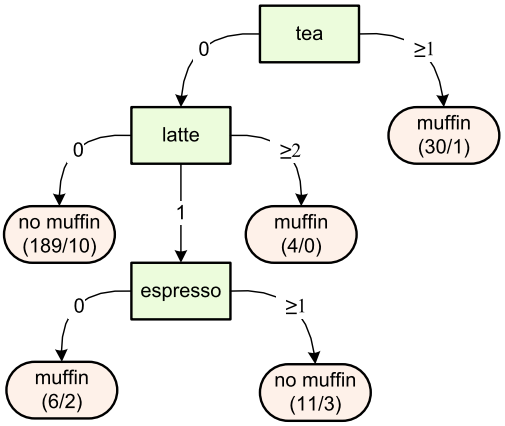


Fig. 4.2 A decision tree derived from Table 4.2. The 420 students are classified into "failed", "passed", and "cum laude" based on study results. Students that do not take the course on logic typically fail (79 students of which 10 are misclassified). Students that have a high mark for logic and programming, typically complete their degree cum laude (20 students of which 2 are misclassified)

Fig. 4.3 A decision tree derived from Table 4.3 after converting response variable *muffin* into a Boolean. Customers who drink tea tend to eat muffins (30 customers of which 1 is misclassified). Customers who do not drink tea or latte typically do not eat muffins (189 customers of which 10 are misclassified)



2. If $X = \emptyset$, then return the tree with root r and end.
3. Select $x \in X$ and remove it from X, i.e., $X := X \setminus \{x\}$. Determine the "score" $s^{old}(x)$ of node x before splitting, e.g., based on entropy.
4. Determine if splitting is possible/needed. If not, go to step 2, otherwise continue with the next step.
5. For all possible attributes $a \in A$, evaluate the effects of splitting on the attribute. Select the attribute a providing the best improvement, i.e., maximize $s_a^{new}(x) - s^{old}(x)$. The same attribute should not appear multiple times on the same path

from the root. Also note that for numerical attributes, so-called "cut values" need to be determined (cf. < 8 and ≥ 8 in Fig. 4.2).
6. If the improvement is substantial enough, create a set of child nodes Y, add Y to X (i.e., $X := X \cup Y$), and connect x to all child nodes in Y.
7. Associate each node in Y to its corresponding set of instances and go to step 2.

Here, we only provide a rough sketch of the generic algorithm. Many design decisions are needed to make a concrete decision tree learner. For example, one needs to decide when to stop adding nodes. This can be based on the improvement of the scoring function or because the tree is restricted to a certain depth. There are also many ways to select attributes. This can be based on entropy (see below), the Gini index of diversity, etc. When selecting a numeric attribute to split on, cut values need to be determined because it is unreasonable/impossible to have a child node for every possible value. For example, a customer can purchase any number of latte's and it would be undesirable to enumerate all possibilities when using this attribute to split. As shown in Fig. 4.2, node *latte* has only three child nodes based on two cut values partitioning the domain of natural numbers in $\{0\}$, $\{1\}$, and $\{2, 3, \ldots\}$.

These are just few of the many ingredients that determine a complete decision tree learning algorithm.

The crucial thing to see is that by *splitting the set of instances in subsets the variation within each subset becomes smaller*. This can be best illustrated using the notion of *entropy*.

Entropy: Encoding uncertainty

Entropy is an information-theoretic measure for the uncertainly in a multi-set of elements. If the multi-set contains many different elements and each element is unique, then variation is maximal and it takes many "bits" to encode the individual elements. Hence, the entropy is "high". If all elements in the multi-set are the same, then actually no bits are needed to encode the individual elements. In this case the entropy is "low". For example, the entropy of the multi-set $[a, b, c, d, e]$ is much higher than the entropy of the multi-set $[a^5]$ even though both multi-sets have the same number of elements (5).

Assume that there is a multi-set X with n elements and there are k possible values, say $v_1, v_2, \ldots, v_k$, i.e., X is a multi-set over $V = \{v_1, v_2, \ldots, v_k\}$ with $|X| = n$. Each value v_i appears c_i times in X, i.e., $X = [(v_1)^{c_1}, (v_2)^{c_2}, \ldots, (v_k)^{c_k}]$. Without loss of generality, we can assume that $c_i \geq 1$ for all i, because values that do not appear in X can be removed from V upfront. The proportion of elements having value v_i is p_i, i.e., $p_i = c_i/n$. The entropy of X is measured in bits of information and is defined by the formula:

$$E = -\sum_{i=1}^{k} p_i \log_2 p_i$$

If all elements in X have the same value, i.e., $k = 1$ and $p_1 = 1$, then $E = -\log_2 1 = 0$. This means that no bits are needed to encode the value of an individual element; they are all the same anyway. If all elements in X are different, i.e., $k = n$ and $p_i = 1/k$, then $E = -\sum_{i=1}^{k}(1/k)\log_2(1/k) = \log_2 k$. For instance, if there are 4 possible values, then $E = \log_2 4 = 2$ bits are needed to encode each individual element. If there are 16 possible values, then $E = \log_2 16 = 4$ bits are needed to encode each individual element.

The proportion p_i can also been seen as a probability. Assume there is random stream of values such that there are four possible values $V = \{a, b, c, d\}$, e.g., a sequence like $bacaabadabaacada\ldots$ is generated. Value a has a probability of $p_1 = 0.5$, value b has a probability of $p_2 = 0.25$, value c has a probability of $p_3 = 0.125$, and value d has a probability of $p_4 = 0.125$. In this case $E = -((0.5\log_2 0.5) + (0.25\log_2 0.25) + (0.125\log_2 0.125) + (0.125\log_2 0.125)) = -((0.5\times -1) + (0.25\times -2) + (0.125\times -3) + (0.125\times -3)) = 0.5 + 0.5 + 0.375 + 0.375 = 1.75$ bits. This means that on average 1.75 bits are needed to encode one element. This is correct. Consider, for example, the following variable length binary encoding $a = 0$, $b = 11$, $c = 100$, and $d = 111$, i.e., a is encoded in one bit, b is encoded in two bits, and c and d are each encoded in three bits. Given the relative frequencies it is easy to see that this is (on average) the most compact encoding. Other encodings are either similar (e.g., $a = 1$, $b = 00$, $c = 011$, and $d = 000$) or require more bits on average. Suppose now that all four values have the same probability, i.e., $p_1 = p_2 = p_3 = p_4 = 0.25$. In this case $E = \log_2 4 = 2$. This is correct because there is no way to improve the encoding $a = 00$, $b = 01$, $c = 10$, and $d = 11$.

The example shows that by using information about the probability of each value, we can reduce the encoding from 2 bits to 1.75 bits on average. If the probabilities are more skewed, further reductions are possible. If value a has a probability of $p_1 = 0.9$, value b has a probability of $p_2 = 0.1$, value c has a probability of $p_3 = 0.05$, and value d has a probability of $p_4 = 0.05$, then $E = 0.901188$. This means that on average less than one bit is needed to encode each element.

Let us now apply the notion of entropy to decision tree learning. Fig. 4.4 shows three steps in the construction of a decision tree for the data set shown in Table 4.1. We label the instances into "old" and "young". Moreover, for simplicity we abstract from the *weight* attribute. In the initial step, the tree consists only of a root. Since the majority of persons in our data set die before 70, we label this node as young. Since of the 860 persons in our data set only 546 actually die before 70, the remaining 314 persons are misclassified. Let us calculate the entropy for the root node: $E = -(((546/860)\log_2(546/860)) + ((314/860)\log_2(314/860))) = 0.946848$. This is a value close to the maximal value of one (in case both groups would have the same size).

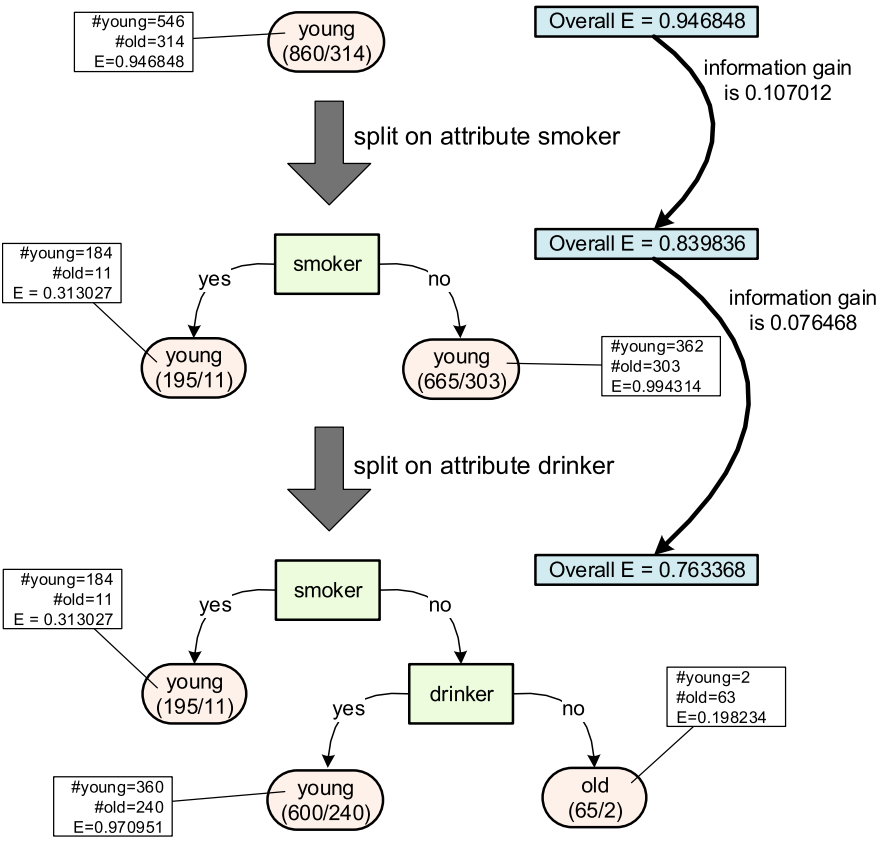


Fig. 4.4 Step-by-step construction of decision tree driven by information gain based on entropy

Next, Fig. 4.4 shows what happens if we split the data set based on attribute *smoker*. Now there are two leaf nodes both bearing the label young. Of the people that smoke (195), most die young (184). Hence, the entropy of this leaf node is very small: $E = -(((184/195)\log_2(184/195)) + ((11/195)\log_2(11/195))) = 0.313027$. This means that the variability is much smaller. The other leaf node is more heterogeneous: about half of the 665 non smokers (362 to be precise) die young. Indeed $E = -(((362/665)\log_2(362/665)) + ((303/665)\log_2(303/665))) = 0.994314$ is higher. However, the overall entropy is still lower. The overall entropy can be found by simply taking the weighted average, i.e., $E = (195/860) \times 0.313027 + (665/860) \times 0.994314 = 0.839836$.

As Fig. 4.4 shows the *information gain* is 0.107012. This is calculated by taking the old overall entropy (0.946848) minus the new overall entropy (0.839836). Note that still all persons are classified as young. However, we gained information by splitting on attribute *smoker*. The information gain, i.e., a reduction in entropy, was obtained because we were able to find a group of persons for which there is

less variability; most smokers die young. The goal is to *maximize the information gain* by selecting a particular attribute to split on. Maximizing the information gain corresponds to minimizing the entropy and heterogeneity in leaf nodes. We could also have chosen the attribute *drinker* first. However, this would have resulted in a smaller information gain.

The lower part of Fig. 4.4 shows what happens if we split the set of non-smokers based on attribute *drinker*. This results in two new leaf nodes. The node that corresponds to persons who do not smoke and do not drink has a low entropy value ($E = 0.198234$). This can be explained by the fact that indeed most of the people associated to this leaf node live long and there are only two exceptions to this rule. The entropy of the other new leaf node (people that drink but do not smoke) is again close two one. However, the overall entropy is clearly reduced. The information gain is 0.076468. Since we abstract from the *weight* attribute we cannot further split the leaf node corresponding to people that drink but do not smoke. Moreover, it makes no sense to split the leaf node with smokers because little can be gained as the entropy is already low.

Note that splitting nodes will always reduce the overall entropy. In the extreme case all the leaf nodes corresponds to single individuals (or individuals having exactly the same attribute values). The overall entropy is then by definition zero. However, the resulting tree is not very useful and probably has little predictive value. It is vital to realize that the decision tree is learned based on *examples*. For instance, if in the data set no customer ever ordered six muffins, this does not imply that this is not possible. A decision tree is "overfitting" if it depends too much on the particularities of the data used to learn it (see also Sect. 4.6). An overfitting decision tree is overly complex and performs poorly on unseen instances. Therefore, it is important to select the right attributes and to stop splitting when little can be gained.

Entropy is just one of several measures that can be used to measure the diversity in a leaf node. Another measure is the *Gini index of diversity* that measures the "impurity" of a data set: $G = 1 - \sum_{i=1}^{k} (p_i)^2$. If all classifications are the same, then $G = 0$. G approaches 1 as there is more and more diversity. Hence, an approach can be to select the attribute that maximizes the reduction of the G value (rather than the E value).

See [9, 24, 69, 190] for more information (and pointers to the extensive literature) on the different strategies to build decision trees.

Decision tree learning is unrelated to process discovery, however it can be used in combination with process mining techniques. For example, process discovery techniques such as the α-algorithm help to locate all decision points in the process (e.g., the XOR/OR-splits discussed in Chap. 3). Subsequently, we can analyze each decision point using decision tree learning. The response variable is the path taken and the attributes are the data elements known at or before the decision point.

4.3 *k*-Means Clustering

Clustering is concerned with grouping instances into *clusters*. Instances in one cluster should be similar to each other and dissimilar to instances in other clusters. Clus-

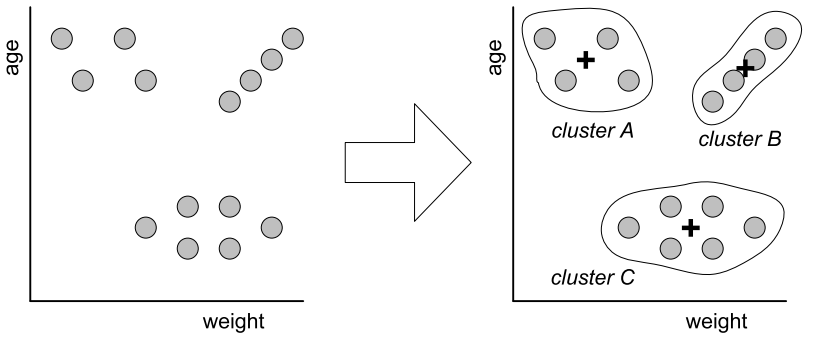


Fig. 4.5 Clustering instances in three clusters using k-means

tering uses unlabeled data and, hence, requires an unsupervised learning technique. Many clustering algorithms exist [9, 24, 69, 102, 190]. Here, we focus on *k-means clustering*.

Figure 4.5 illustrates the basic idea of clustering. Assume we have a data set with only two variables: *age* and *weight*. Such a data set could be obtained by projecting Table 4.1 onto the last two columns. The dots correspond to persons having a particular age and weight. Through a clustering technique like k-means, the three *clusters* shown on the right-hand-side of Fig. 4.5 can be discovered. Ideally, the instances in one cluster are close to one another while being further away from instances in other clusters. Each of the clusters has a *centroid* denoted by a +. The centroid denotes the "center" of the cluster and can be computed by taking the average of the coordinates of the instances in the cluster. Note that Fig. 4.5 shows only two dimensions. This is a bit misleading as typically there will be many dimensions (e.g., the number of courses or products). However, the two dimensional view helps to understand the basic idea.

Distance-based clustering algorithms like k-means and agglomerative hierarchical clustering assume a *distance notion*. The most common approach is to consider each instance to be an n-dimensional vector where n is the number of variables and then simply take the Euclidian distance. For this purpose ordinal values but also binary values need to be made numeric, e.g., $true = 1$, $false = 0$, $cum\ laude = 2$, $passed = 1$, $failed = 0$. Note that scaling is important when defining a distance metric. For example, if one variable represents the distance in meters ranging from 10 to 1,000,000 while another variable represents some utilization factor ranging from 0.2 to 0.8, then the distance variable will dominate the utilization variable. Hence, some normalization is needed.

Figure 4.6 shows the basic idea of k-means clustering. Here, we simplified things as much as possible, i.e., $k = 2$ and there are only 10 instances. The approach starts with a random initialization of two centroids denoted by the two + symbols. In Fig. 4.6(a) the centroids are randomly put onto the two dimensional space. Using the selected distance metric, all instances are assigned to the closest centroid. Here we use the standard Euclidian distance. All instances with an open dot are assigned to the centroid on the left whereas all the instances with a closed dot are assigned

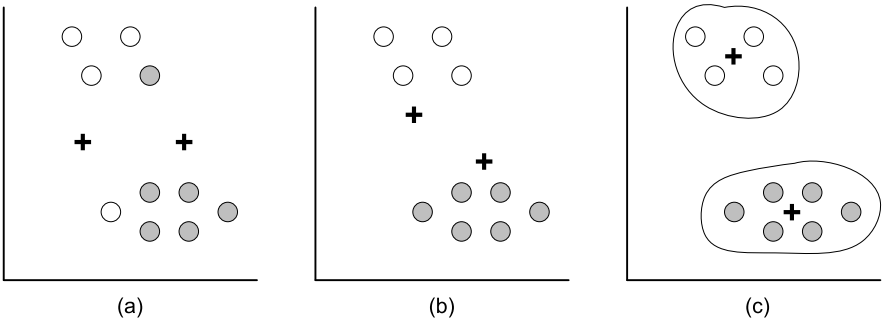


Fig. 4.6 Step-by-step evolution k-means

to the centroid on the right. Based on this assignment we get two initial clusters. Now we compute the real center of each cluster. These form the new positions of the two centroids. The centroids in Fig. 4.6(b) are based on the clusters shown in Fig. 4.6(a). In Fig. 4.6(b) we again assign all instances to the centroid that is closest. This results in the two new clusters shown in Fig. 4.6(b). All instances with an open dot are assigned to one centroid whereas all the instances with a closed dot are assigned to the other one. Now we compute the real centers of these two new clusters. This results in a relocation of the centroids as shown in Fig. 4.6(c). Again we assign the instances to the centroid that is closest. However, now nothing changes and the location of the centroids remains the same. After converging the k-means algorithm outputs the two clusters and related statistics.

The quality of a particular clustering can be defined as the average distance from an instance to its corresponding centroid. k-means clustering is only a heuristic and does not guarantee that it finds the k clusters that minimize the average distance from an instance to its corresponding centroid. In fact, the result depends on the initialization. Therefore, it is good to repeatedly execute the algorithm with different initializations and select the best one.

There are many variants of the algorithm just described. However, we refer to standard literature for details [9, 24, 69, 102, 190]. One of the problems when using the k-means algorithm is determining the number of clusters k. For k-means this is fixed from the beginning. Note that the average distance from an instance to its corresponding centroid decreases as k is increased. In the extreme case every instance has its own cluster and the average distance from an instance to its corresponding centroid is zero. This is not very useful. Therefore, a frequently used approach is to start with a small number of clusters and then gradually increase k as long as there are significant improvements.

Another popular clustering technique is *Agglomerative Hierarchical Clustering* (AHC). Here, a variable number of clusters is generated. Figure 4.7 illustrates the idea. The approach works as follows. Assign each instance to a dedicated singleton cluster. Now search for the two clusters that are closest to one another. Merge these two clusters into a new cluster. For example, the initial clusters consisting of just a and just b are merged into a new cluster ab. Now search again for the two clusters

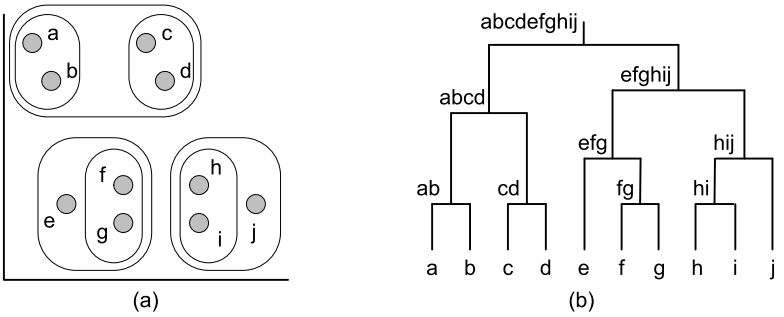


Fig. 4.7 Agglomerative hierarchical clustering: (**a**) clusters and (**b**) dendrogram

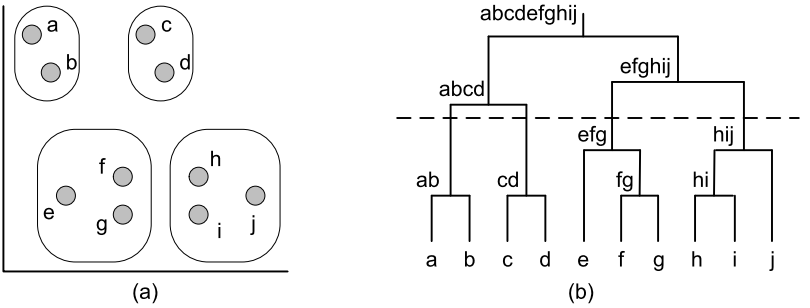


Fig. 4.8 Any horizontal line in dendrogram corresponds to a concrete clustering at a particular level of abstraction

that are closest to one another and merge them. This is repeated until all instances are in the same cluster. Figure 4.7(a) shows all intermediate clusters, i.e., all except the initial singleton clusters and the final overall cluster. Because of the hierarchical nature of the agglomerative hierarchical clustering we can visualize the clusters using a so-called *dendrogram* as shown in Fig. 4.7(b).

Any horizontal line cutting through the dendrogram corresponds to a concrete clustering. For example, Fig. 4.8(b) shows such a horizontal line. The clusters resulting from this are shown in Fig. 4.8(a). Moving the line to the bottom of the dendrogram results in many singleton clusters. Moving the line all the way up results in a single cluster containing all instances. By moving the horizontal line, the user can vary the abstraction level.

Clustering is only indirectly related to process discovery as described in Chap. 2. Nevertheless, clustering can be used as a preprocessing step for process mining [13, 62, 78]. By grouping similar cases together it may be possible to construct partial process models that are easier to understand. If the process model discovered for all cases is too complex to comprehend, then it may be useful to first identify clusters and then discover simpler models per cluster.

4.4 Association Rule Learning

Decision trees can be used to predict the value of some response variable that has been identified as being important. Driven by the response variable, rules like "people who drink and smoke die before 70" can be found. *Association rule learning* aims at finding similar rules but now *without* focusing on a particular response variable. The goal is to find rules of the form *IF* X *THEN* Y where X is often called the *antecedent* and Y the *consequent*. Such rules are also denoted as $X \Rightarrow Y$. X and Y can be any conjunction of "*variable* = *value*" terms. The only requirement is that X and Y are nonempty and any variable appears at most once in X and Y. Examples are *IF smoker* = *no AND age* ≥ 70 *THEN drinker* = *yes* for Table 4.1 or *IF logic* ≤ 6 *AND duration* > 50 *THEN result* = *failed* for Table 4.2. Typically, only categorical variables are considered. However, there are various techniques to transform numerical variables in categorical ones.

When learning association rules of the form $X \Rightarrow Y$, three metrics are frequently used: *support*, *confidence*, and *lift*. Let N_X be the number of instances for which X holds. N_Y is the number of instances for which Y holds. $N_{X \wedge Y}$ is the number of instances for which both X and Y hold. N is the total number of instances. The support of a rule $X \Rightarrow Y$ is defined as

$$support(X \Rightarrow Y) = N_{X \wedge Y}/N$$

The support indicates the applicability of the approach, i.e., the fraction of instances for which with both antecedent and consequent hold. Typically a rule with high support is more useful than a rule with low support.

The confidence of a rule $X \Rightarrow Y$ is defined as

$$confidence(X \Rightarrow Y) = N_{X \wedge Y}/N_X$$

A rule with high confidence, i.e., a value close to 1, indicates that the rule is very reliable, i.e., if X holds, then Y will also hold. A rule with high confidence is definitely more useful than a rule with low confidence.

The lift of a rule $X \Rightarrow Y$ is defined as

$$lift(X \Rightarrow Y) = \frac{N_{X \wedge Y}/N}{(N_X/N)\ (N_Y/N)} = \frac{N_{X \wedge Y}\ N}{N_X\ N_Y}$$

If X and Y are independent, then the lift will be close to 1. If $lift(X \Rightarrow Y) > 1$, then X and Y correlate positively. For example $lift(X \Rightarrow Y) = 5$ means that X and Y happen five times more together than what would be the case if they were independent. If $lift(X \Rightarrow Y) < 1$, then X and Y correlate negatively (i.e., the occurrence of X makes Y less likely and vice versa). Rules with a higher lift value are generally considered to be more interesting. However, typically lift values are only considered if certain thresholds with respect to support and confidence are met.

In the remainder of this section, we restrict ourselves to a special form of association rule learning known as *market basket analysis*. Here we only consider binary

variables that should be interpreted as present or not. For example, let us consider the first two columns in Table 4.1. This data set can be rewritten to so called *item-sets*: [{*drinker*, *smoker*}, { }, {*drinker*}, {*drinker*, *smoker*}, {*smoker*}, { }, ...]. If we ignore the number of items ordered in Table 4.3, then it is also straightforward to rewrite this data set in terms of item-sets: [{*cappuccino*, *muffin*}, {*latte*, *muffin*, *bagel*}, {*espresso*}, {*cappuccino*}, {*tea*, *muffin*}, {*americano*, *ristretto*}, ...]. The latter illustrates why the term "market basket" analysis is used for systematically analyzing such input. Based on item-sets, the goal is to generate rules of the form $X \Rightarrow Y$ where X and Y refer to disjoint non-empty sets of items. For example, $smoker \Rightarrow drinker$, $tea \wedge latte \Rightarrow muffin$, and $tea \Rightarrow muffin \wedge bagel$. Recall that there are $N = 240$ customer orders in Table 4.3. Assume that $N_{tea} = 50$ (i.e., 50 orders included at least one cup of tea), $N_{latte} = 40$, $N_{muffin} = 40$, $N_{tea \wedge latte} = 20$, and $N_{tea \wedge latte \wedge muffin} = 15$ (i.e., 15 orders included at least one tea, at least one latte, and at least one muffin). Let us consider the rule $tea \wedge latte \Rightarrow muffin$, i.e., $X = tea \wedge latte$ and $Y = muffin$. Given the numbers indicated we can easily compute the three metrics defined earlier:

$$support(X \Rightarrow Y) = N_{X \wedge Y}/N = N_{tea \wedge latte \wedge muffin}/N = 15/240 = 0.0625$$

$$confidence(X \Rightarrow Y) = N_{X \wedge Y}/N_X = N_{tea \wedge latte \wedge muffin}/N_{tea \wedge latte} = 15/20 = 0.75$$

$$lift(X \Rightarrow Y) = \frac{N_{X \wedge Y}\ N}{N_X\ N_Y} = \frac{N_{tea \wedge latte \wedge muffin}\ N}{N_{tea \wedge latte}\ N_{muffin}} = \frac{15 \times 240}{20 \times 40} = 4.5$$

Hence the $tea \wedge latte \Rightarrow muffin$ has a support of 0.0625, a confidence of 0.75, and a lift of 4.5.

If we also assume that $N_{tea \wedge muffin} = 25$, then we can deduce that the rule $tea \Rightarrow muffin$ has a support of 0.104167, a confidence of 0.5, and a lift of 3. Hence, this more compact rule has a better support but lower confidence and lift.

Let us also assume that $N_{latte \wedge muffin} = 35$. This implies that the rule $tea \Rightarrow latte \wedge muffin$ has a support of 0.0625, a confidence of 0.3, and a lift of 2.057. This rule has a rather poor performance compared to the original rule $tea \wedge latte \Rightarrow muffin$: the support is the same, but the confidence and lift are much lower.

To systematically *generate* association rules, one typically defines two parameters: *minsup* and *minconf*. The support of any rule $X \Rightarrow Y$ should be above the threshold *minsup*, i.e., $support(X \Rightarrow Y) \geq minsup$. Similarly the confidence of any rule $X \Rightarrow Y$ should be above the threshold *minconf*, i.e., $confidence(X \Rightarrow Y) \geq minconf$. Association rules can now be generated as follows:

1. Generate all *frequent item-sets*, i.e., all sets Z such that $N_Z/N \geq minsup$ and $|Z| \geq 2$.
2. For each frequent item-set Z consider all partitionings of Z into two non-empty subsets X and Y. If $confidence(X \Rightarrow Y) \geq minconf$, then keep the rule $X \Rightarrow Y$. If $confidence(X \Rightarrow Y) < minconf$, then discard the rule.
3. Output the rules found.

This simple algorithm has two problems. First of all, there is a computational problem related to the first step. If there are m variables, then there are $2^m - m - 1$

possible item-sets. Hence, for 100 products ($m = 100$) there are

$$1267650600228229401496703205275$$

candidate frequent item-sets. The second problem is that many uninteresting rules are generated. For example, after presenting the rule *tea* $\wedge$ *latte* $\Rightarrow$ *muffin*, there is no point in also showing *tea* $\Rightarrow$ *latte* $\wedge$ *muffin* even when it meets the *minsup* and *minconf* thresholds. Many techniques have been developed to speed-up the generation of association rules and to select the most interesting rules. Here we only sketch the seminal *Apriori algorithm*.

Apriori: Efficiently generating frequent item-sets
The Apriori algorithm is one of the best known algorithms in computer science. The algorithm, initially developed by Agrawal and Srikant [7], is able to speed up the generation of association rules by exploiting the following two observations:

1. If an item-set is *frequent* (i.e., an item-set with a support above the threshold), then all of its non-empty subsets are also frequent. Formally, for any pair of non-empty item-sets X, Y: if $Y \subseteq X$ and $N_X/N \geq minsup$, then $N_Y/N \geq minsup$.
2. If, for any k, I_k is the set of all frequent item-sets with cardinality k and $I_l = \emptyset$ for some l, then $I_k = \emptyset$ for all $k \geq l$.

These two properties can be used to dramatically reduce the search-space when constructing the set of frequent item-sets. For example, if item-set $\{a, b\}$ is infrequent, then it does not make any sense to look at item-sets containing both a and b. The Apriori algorithm works as follows:

1. Create I_1. This is the set of singleton frequent item-sets, i.e., item-sets with a support above the threshold *minsup* containing just one element.
2. $k := 1$.
3. If $I_k = \emptyset$, then output $\bigcup_{i=1}^{k} I_i$ and end. If $I_k \neq \emptyset$, continue with the next step.
4. Create C_{k+1} from I_k. C_{k+1} is the candidate set containing item-sets of cardinality $k + 1$. Note that one only needs to consider elements that are the union of two item-sets A and B in I_k such that $|A \cap B| = k$ and $|A \cup B| = k + 1$.
5. For each candidate frequent item-set $c \in C_{k+1}$: examine all subsets of c with k elements; delete c from C_{k+1} if any of the subsets is not a member of I_k.
6. For each item-set c in the pruned candidate frequent item-set C_{k+1}, check whether c is indeed frequent. If so, add c to I_{k+1}. Otherwise, discard c.

7. $k := k + 1$ and return to Step 3.

The algorithm only considers candidates for I_{k+1} that are not ruled out by evidence in I_k. This way the number of traversals through the data set is reduced dramatically.

Association rules are related to process discovery. Recall that the α-algorithm also traverses the event log looking for patterns. However, association rules do not consider the ordering of activities and do not aim to build an overall process model.

4.5 Sequence and Episode Mining

The Apriori algorithm uses the monotonicity property that all subsets of a frequent item-set are also frequent. Many other pattern or rule discovery problems have similar monotonicity properties, thus enabling efficient implementations. A well-known example is the *mining of sequential patterns*. After introducing sequence mining, we also describe an approach to *discover frequent episodes* and mention some other data mining techniques relevant for process mining.

4.5.1 Sequence Mining

The Apriori algorithm does not consider the ordering of events. Sequence mining overcomes this problem by analyzing sequences of item-sets. One of the early approaches was developed by Srikant and Agrawal [131]. Here we sketch the essence of this approach. To explain the problem addressed by sequence mining, we consider the data set shown in Table 4.4. Each line corresponds to a customer ordering a set of items, e.g., at 9.02 on January 2nd 2011, Wil ordered a cappuccino, one day later he orders an espresso and a muffin. Per customer there is a sequence of orders. Orders have a sequence number, a timestamp, and an item-set. A more compact representation of the first customer sequence is $\langle\{cappuccino\}, \{espresso, muffin\}, \{americano, cappuccino\}, \{espresso, muffin\}, \{cappuccino\}, \{americano, cappuccino\}\rangle$. The goal is to find frequent sequences defined by a pattern like $\langle\{cappuccino\}, \{espresso, muffin\}, \{cappuccino\}\rangle$. A sequence is frequent if the pattern is contained in a predefined proportion of the customer sequences in the data set.

A sequence $\langle a_1, a_2, \ldots, a_n\rangle$ is a *subsequence* of another sequence $\langle b_1, b_2, \ldots, b_m\rangle$ if there exist integers $i_1 < i_2 < \cdots < i_n$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, $\ldots$, $a_n \subseteq b_{i_n}$. For example, the sequence $\langle\{x\}, \{x, y\}, \{y\}\rangle$ is a subsequence of $\langle\{z\}, \{x\}, \{z\}, \{x, y, z\}, \{y, z\}, \{z\}\rangle$ because $\{x\} \subseteq \{x\}$, $\{x, y\} \subseteq \{x, y, z\}$, and $\{y\} \subseteq \{y, z\}$. However, $\langle\{x\}, \{y\}\rangle$ is not a subsequence of $\langle\{x, y\}\rangle$ and vice versa. The

Table 4.4 A fragment of a data set used for sequence mining: each line corresponds to an order

Customer	Seq. number	Timestamp	Items
Wil	1	02-01-2011:09.02	{*cappuccino*}
	2	03-01-2011:10.06	{*espresso*, *muffin*}
	3	05-01-2011:15.12	{*americano*, *cappuccino*}
	4	06-01-2011:11.18	{*espresso*, *muffin*}
	5	07-01-2011:14.24	{*cappuccino*}
	6	07-01-2011:14.24	{*americano*, *cappuccino*}
Mary	1	30-12-2010:11.32	{*tea*}
	2	30-12-2010:12.12	{*cappuccino*}
	3	30-12-2010:14.16	{*espresso*, *muffin*}
	4	05-01-2011:11.22	{*bagel*, *tea*}
Bill	1	30-12-2010:14.32	{*cappuccino*}
	2	30-12-2010:15.06	{*cappuccino*}
	3	30-12-2010:16.34	{*bagel*, *espresso*, *muffin*}
	4	06-01-2011:09.18	{*ristretto*}
	5	06-01-2011:12.18	{*cappuccino*}
...	...	...	...

support of a sequence s is the fraction of sequences in the data set that has s as a subsequence. A sequence is *frequent* if its support meets some threshold *minsup*. Consider, for example, the data sets consisting of just the three visible customer sequences in Table 4.4. Pattern $\langle\{tea\}, \{bagel, tea\}\rangle$ has a support of $1/3$ as it is only a subsequence of Mary's sequence. Pattern $\langle\{espresso\}, \{cappuccino\}\rangle$ has a support of $2/3$ as it is a subsequence of both Wil's and Bill's subsequences, but not a subsequence of Mary's sequence. Pattern $\langle\{cappuccino\}, \{espresso, muffin\}\rangle$ has a support of $3/3 = 1$.

In principle, there is an infinite number of potential patterns. However, just like in the Apriori algorithm a monotonicity property can be exploited: if a sequence is frequent, then its subsequences are also frequent. Therefore, it is possible to efficiently generate patterns. Frequent sequences can also be used to create rules of the form $X \Rightarrow Y$ where X is a pattern and Y is an extension or continuation of the pattern. Consider for example $X = \langle\{cappuccino\}, \{espresso\}\rangle$ and $Y = \langle\{cappuccino\}, \{espresso\}, \{latte, muffin\}\rangle$. Suppose that X has a support of 0.05 and Y has a support of 0.04. Then the confidence of $X \Rightarrow Y$ is $0.04/0.05 = 0.8$, i.e., 80% of the customer that ordered a cappuccino followed by an espresso later also order a muffin and latte.

In [131] several extensions of the above approach have been proposed. For example, it is possible to add taxonomies, sliding windows, and time constraints. For practical applications it is important to relax the strict subsequence requirement such that a one-to-one matching of item-sets is no longer needed.

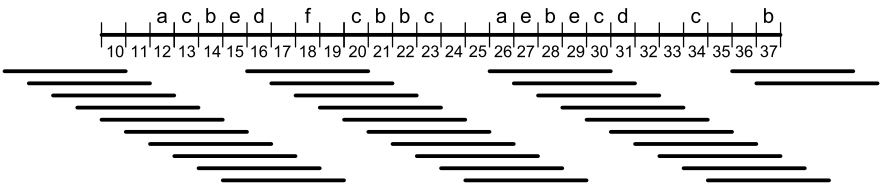


Fig. 4.9 A timed sequence of events and the corresponding time windows

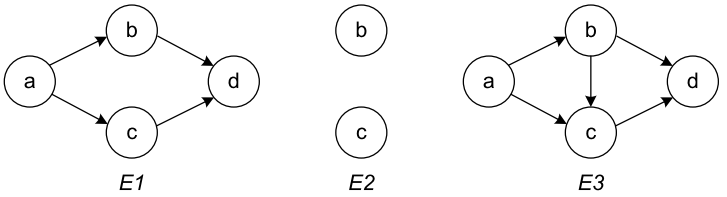


Fig. 4.10 Three episodes

4.5.2 Episode Mining

Another problem that can be solved using an Apriori-like approach is the *discovery of frequent episodes* [94]. Here a sliding window is used to analyze how frequent an *episode* is appearing. An episode defines a partial order. The goal is to discover frequent episodes.

Input for episode mining is a time sequence as shown in Fig. 4.9. The timed sequence starts at time 10 and ends at time 37. The sequence consists of discrete time points, and, as shown in Fig. 4.9, at some points in time an event occurs. An event has a type (e.g., the activity that happened) and a timestamp. For example, an event of type a occurs at time *12*, an event of type c occurs at time *13*, etc. Figure 4.9 also shows 32 *time windows* of length 5. These are all the windows (partially) overlapping with the timed sequence. The length 5 is a predefined parameter of the algorithm used to discover frequent patterns. An episode *occurs* in a time window if the partial order is "embedded" in it.

Figure 4.10 shows three episodes. An episode is described by a directed acyclic graph. The nodes refer to event types and the arcs define a partial order. For example, episode $E1$ defines that a should be followed by b and c, b should be followed by d, and c should be followed by d. Episode $E2$ merely states that b and c should both happen at least once. Episode $E3$ states that a should be followed by b and c, b should be followed by c, b should be followed by d, and c should be followed by d. This episode contains two redundant arcs: the arc from a to c and the arc from b to d can be removed without changing the requirements. An episode *occurs* in a time window if it is possible to assign events to nodes in the episode such that the ordering relations are satisfied. Note that the episode only defines the minimal set of events, i.e., there may be all kinds of additional events. The key requirement is that the episode is embedded.

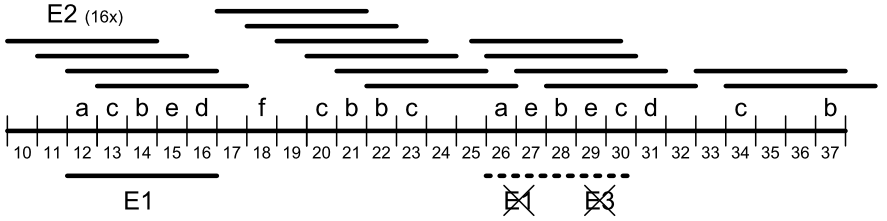


Fig. 4.11 Occurrences of episodes $E1$ and $E2$

To illustrate the notion of "occurring in a time window", we use Fig. 4.11. Consider episode $E1$ and slide a window of length 5 from left to right. There are 32 possible positions. However, just one of the 32 windows embeds episode $E1$. This is the window starting at time 12 shown below the timed sequence in Fig. 4.11. Here we find the sequence $\langle a, c, b, e, d\rangle$. Clearly all the requirements are met in the sequence: a is followed by b and this b is followed by d, the same a is also followed by c and this c is followed by the same d.

Now consider episode $E2$ and again slide a window of length 5 from left to right. This pattern is much more frequent. Figure 4.11 shows all time windows in which the pattern occurs. In total there are 16 windows where $E2$ is embedded. Note that the only requirement is that both b and c occur: no ordering relation is defined.

Episode $E3$ does not occur in the time sequence if we use a window length of 5. There is no window of length of 5 where the sequence $\langle a, b, c, d\rangle$ is embedded. If the window length is extended to 6, $E3$ occurs once. The corresponding window starts at time 26. Here we find the sequence $\langle a, e, b, e, c, d\rangle$.

The *support* of an episode is the fraction of windows in which the episode occurs. For a window size of 5 time units, the support of $E1$ is $1/32$, the support of $E2$ is $16/32 = 0.5$, and the support of $E3$ is $0/32 = 0$. Like for sequence mining and association rule learning, we define a threshold for the support. All episodes having a support of at least this threshold are *frequent*. For example, if the threshold is 0.2 then $E2$ is frequent whereas $E1$ and $E3$ are not.

The goal is to generate all frequent episodes. Note that there are typically many potential candidates (all partial orders over the set of event types). Fortunately, like in the Apriori algorithm, we can exploit a monotonicity property to quickly rule out bad candidates. To explain this property we need to define the notion of a *subepisode*. $E1$ is a subepisode of $E3$ because $E1$ is a subgraph of $E3$, i.e., the nodes and arcs of $E1$ are contained in $E3$. $E2$ is a subepisode of both $E1$ and $E3$. It is easy to see that, if an episode E is frequent, then also all of its subepisodes are frequent. This monotonicity property can be used to speed-up the search process.

Frequent episodes can also be used to create rules of the form $X \Rightarrow Y$ where X is a subepisode of Y. As before the confidence of such a rule can be computed. In our example, rule $E1 \Rightarrow E3$ has a confidence of $0/1 = 0$, i.e., a very poor rule. Rule $E2 \Rightarrow E1$ has a confidence of $1/16$.

Episode mining and sequence mining can be seen as variants of association rule learning. Because they take into account the ordering of events, they are related to

process discovery. However, there are many differences with process mining algorithms. First of all, *only local patterns* are considered, i.e., no overall process model is created. Second, the focus is on frequent behavior without trying to generate models that also *exclude* behavior. Consider, for example, episode $E1$ in Fig. 4.10. Also the time window $\langle a, b, d, c, d \rangle$ contains the episode despite the two occurrences of d. Therefore, episodes cannot be read as if they are process models. Moreover, episodes *cannot model choices, loops,* etc. Finally, episode mining and sequence mining *cannot handle concurrency* well. Sequence mining searches for sequential patterns only. Episode mining runs into problems if there are concurrent episodes, because it is unclear what time window to select to get meaningful episodes.

4.5.3 Other Approaches

In the data mining and machine learning communities several other techniques have been developed to analyze sequences of events. Applications are in text mining (sequences of letters and words), bio-informatics (analysis of DNA sequences), speech recognition, web analytics, etc. Examples of techniques that are used for this purpose are *neural networks* and *hidden Markov models* [9, 102].

Artificial neural networks try to mimic the human brain in order to learn complex tasks. An artificial neural network is an interconnected group of nodes, akin to the vast network of neurons in the human brain. Different learning paradigms can be used to train the neural network: supervised learning, unsupervised learning, and reinforcement learning [9, 102]. Advantages are that neural networks can exploit parallel computing and that they can be used to solve ill-defined tasks, e.g., image and speech recognition. The main drawback is that the resulting model (e.g., a multi-layer perceptron), is typically not human readable. Hence there is no resulting process model in the sense of Chap. 3 (e.g., a WF-net or BPMN model).

Hidden Markov models are an extension of ordinary Markov processes. A hidden Markov model has a set of states and transition probabilities. Moreover, unlike standard Markov models, in each state an observation is possible, but the state itself remains hidden. Observations have probabilities per state as shown in Fig. 4.12. Three fundamental problems have been investigated for hidden Markov models [9]:

- Given an observation sequence, how to compute the probability of the sequence given a hidden Markov model?
- Given an observation sequence and a hidden Markov model, how to compute the most likely "hidden path" in the model?
- Given a set of observation sequences, how to derive the hidden Markov model that maximizes the probability of producing these sequences?

The last problem is most related to process mining but also the most difficult problem. The well-known Baum–Welch algorithm [9] is a so-called Expectation-Maximization (EM) algorithm that solves this problem iteratively for a fixed number of states. Although hidden Markov models are versatile and relevant for process

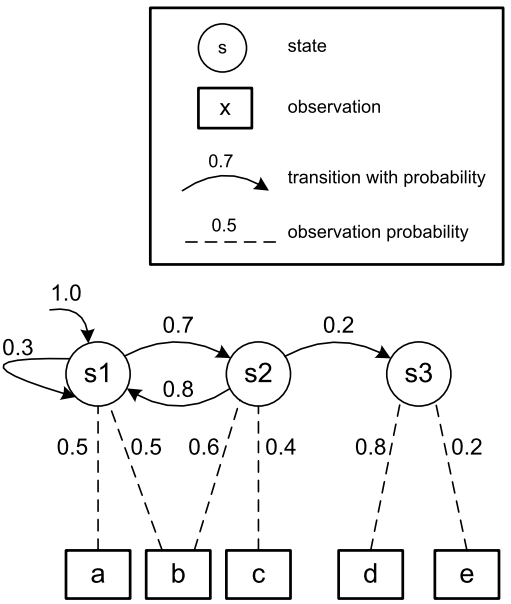


Fig. 4.12 A hidden Markov model with three states: $s1$, $s2$, and $s3$. The arcs have state transition probabilities as shown, e.g., in state $s2$ the probability of moving to state $s3$ is 0.2 and the probability of moving to state $s1$ is 0.8. Each visit to a state generates an observation. The observation probabilities are also given. When visiting $s2$ the probability of observing b is 0.6 and the probability of observing c is 0.4. Possible observation sequences are $\langle a, b, c, d \rangle$, $\langle a, b, b, c \rangle$, and $\langle a, b, c, b, b, a, c, e \rangle$. For the observation sequence $\langle a, b, c, d \rangle$ it is clear what the hidden sequence is: $\langle s1, s2, s2, s3 \rangle$. For the other two observation sequences multiple hidden sequences are possible

mining, there are several complications. First of all, there are many computational challenges due to the time consuming iterative procedures. Second, one needs to guess an appropriate number of states as this is input to the algorithm. Third, the resulting hidden Markov model is typically not very accessible for the end user, i.e., accurate models are typically large and even for small examples the interpretation of the states is difficult. Clearly, hidden Markov models are at a lower abstraction level than the notations discussed in Chap. 3.

4.6 Quality of Resulting Models

This chapter provided an overview of the mainstream data mining techniques most relevant for process mining. Although some of these techniques can be exploited for process mining, they cannot be used for important process mining tasks such as process discovery, conformance checking, and process enhancement. However, there is an additional reason for showing a variety of data mining techniques. Like in data mining it is non-trivial to analyze the quality of process mining results. Here one can

		predicted class		
		failed	passed	cum laude
actual class	failed	178	22	0
	passed	21	175	2
	cum laude	1	3	18

Fig. 4.13 Confusion matrix for the decision tree shown in Fig. 4.2. Of the 200 students who failed, 178 are classified as failed and 22 are classified as passed. None of the failing students was classified as cum laude. Of the 198 students who passed, 175 are classified correctly, 21 were classified as failed, and 2 as cum laude

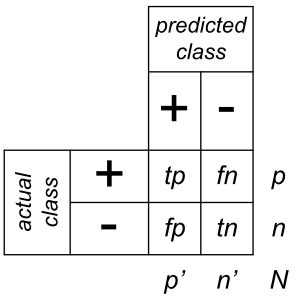
benefit from experiences in the data mining field. Therefore, we discuss some of the validation and evaluation techniques developed for the algorithms presented in this chapter. First, we focus on the quality of classification results, e.g., obtained through a decision tree. Second, we describe general techniques for cross-validation. Here, we concentrate on k-fold cross-validation. Finally, we conclude with a more general discussion on Occam's razor.

4.6.1 Measuring the Performance of a Classifier

In Sect. 4.2, we showed how to construct a decision tree. As discussed, there are many design decisions when developing a decision tree learner (e.g., selection of attributes to split on, when to stop splitting, and determining cut values). The question is how to evaluate the performance of a decision tree learner. This is relevant for judging the trustworthiness of the resulting decision tree and for comparing different approaches. A complication is that one can only judge the performance based on *seen* instances although the goal is also to predict good classifications for *unseen* instances. However, for simplicity, let us first assume that we first want to judge the result of a classifier (like a decision tree) on a given data set.

Given a data set consisting of N instances we know for each instance what the actual class is and what the predicted class is. For example, for a particular person that smokes, we may predict that the person will die young (predicted class is "young"), even though the person dies at age 104 (actual class is "old"). This can be visualized using a so-called *confusion matrix*. Figure 4.13 shows the confusion matrix for the data set shown in Table 4.2 and the decision tree shown in Fig. 4.2. The decision tree classifies each of the 420 students into an actual class and a predicted class. All elements on the diagonal are predicted correctly, i.e., $178 + 175 + 18 = 371$ of the 420 students are classified correctly (approximately 88%).

There are several performance measures based on the confusion matrix. To define these let us consider a data set with only two classes: "positive" (+) and "negative" (−). Figure 4.14(a) shows the corresponding 2×2 confusion matrix. The following entries are shown:



(a)

name	formula
error	*(fp+fn)/N*
accuracy	*(tp+tn)/N*
tp-rate	*tp/p*
fp-rate	*fp/n*
precision	*tp/p'*
recall	*tp/p*

(b)

Fig. 4.14 Confusion matrix for two classes and some performance measures for classifiers

- *tp* is the number of *true positives*, i.e., instances that are correctly classified as positive.
- *fn* is the number of *false negatives*, i.e., instances that are predicted to be negative but should have been classified as positive.
- *fp* is the number of *false positives*, i.e., instances that are predicted to be positive but should have been classified as negative.
- *tn* is the number of *true negatives*, i.e., instances that are correctly classified as negative.

Figure 4.14(a) also shows the sums of rows and columns, e.g., $p = tp + fn$ is the number of instances that are actually positive, $n' = fn + tn$ is the number of instances that are classified as negative by the classifier. $N = tp + fn + fp + tn$ is the total number of instances in the data set. Based on this it is easy to define the measures shown in Fig. 4.14(b). The *error* is defined as the proportion of instances misclassified: $(fp + fn)/N$. The *accuracy* measures the fraction of instances on the diagonal of the confusion matrix. The "true positive rate", *tp-rate*, also known as "hit rate", measures the proportion of positive instances indeed classified as positive. The "false positive rate", *fp-rate*, also known as "false alarm rate", measures the proportion of negative instances wrongly classified as positive. The terms *precision* and *recall* originate from information retrieval. Precision is defined as tp/p'. Here, one can think of p' as the number of documents that have been retrieved based on some search query and *tp* as the number of documents that have been retrieved and also should have been retrieved. Recall is defined as tp/p where p can be interpreted as the number of documents that should have been retrieved based on some search query. It is possible to have high precision and low recall; few of the documents searched for are returned by the query, but those that are returned are indeed relevant. It is also possible to have high recall and low precision; many documents are returned (including the ones relevant), but also many irrelevant documents are returned. Note that recall is the same as tp-rate. There is another frequently used metric not shown in Fig. 4.14(b): the so-called *F1 score*. The F1 score takes the harmonic mean of precision and recall: $(2 \times precision \times recall)/(precision + recall)$. If either the precision or recall is really poor (i.e., close to 0), then the F1 score is also

Fig. 4.15 Two confusion matrices for the decision trees in Fig. 4.4

		predicted class: young	predicted class: old
actual class	young	546	0
	old	314	0

(a)

		predicted class: young	predicted class: old
actual class	young	544	2
	old	251	63

(b)

close to 0. Only if both precision and recall are really good, the F1 score is close to 1.

To illustrate the different metrics let us consider the three decision trees depicted in Fig. 4.4. In the first two decision trees, all instances are classified as young. Note that even after splitting the root node based on the attribute *smoker*, still all instances are predicted to die before 70. Figure 4.15(a) shows the corresponding confusion matrix assuming "young = positive" and "old = negative". $N = 860$, $tp = p = 546$, and $fp = n = 314$. Note that $n' = 0$ because all are classified as young. The error is $(314 + 0)/860 = 0.365$, the tp-rate is $546/546 = 1$, the fp-rate is $314/314 = 1$, precision is $546/860 = 0.635$, recall is $546/546 = 1$, and the F1 score is 0.777. Figure 4.15(b) shows the confusion matrix for the third decision tree in Fig. 4.4. The error is $(251 + 2)/860 = 0.292$, the tp-rate is $544/546 = 0.996$, the fp-rate is $251/314 = 0.799$, precision is $544/795 = 0.684$, recall is $544/546 = 0.996$, and the F1 score is 0.811. Hence, as expected, the classification improved: the error and fp rate decreased considerably and the tp-rate, precision and F1 score increased. Note that the recall went down slightly because of the two persons that are now predicted to live long but do not (despite not smoking nor drinking).

4.6.2 Cross-Validation

The various performance metrics computed using the confusion matrix in Fig. 4.15(b) are based on the same data set as the data set used to learn the third decision tree in Fig. 4.4. Therefore, the confusion matrix is only telling something about *seen* instances, i.e., instances used to learn the classifier. In general, it is trivial to provide classifiers that score perfectly (i.e., precision, recall and F1 score are all 1) on seen instances. (Here we assume that instances are unique or instances with identical attributes belong to the same class.) For example, if students have a unique registration number, then the decision tree could have a leaf node per student thus perfectly encoding the data set. However, this does not say anything about *unseen* instances, e.g., the registration number of a new student carries no information about expected performance of this student.

The most obvious criterion to estimate the performance of a classifier is its predictive accuracy on unseen instances. The number of unseen instances is potentially

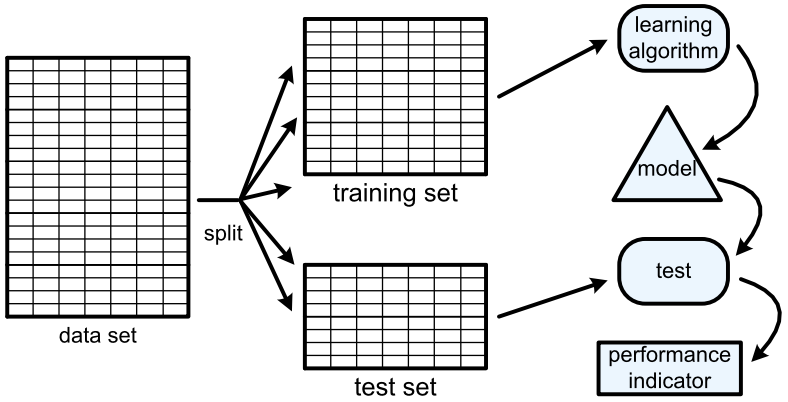


Fig. 4.16 Cross-validation using a test and training set

very large (if not infinite), therefore an estimate needs to be computed on a test set. This is commonly referred to as *cross-validation*. The data set is split into a *training set* and a *test set*. The training set is used to learn a model whereas the test set is used to evaluate this model based on unseen examples.

It is important to realize that cross-validation is not limited to classification but can be used for any data mining technique. The only requirement for cross-validation is that the performance of the result can be measured in some way. For classification we defined measures such as *precision*, *recall*, *F1 score*, and *error*.

For regression also various measures can be defined. In the context of linear regression the *mean square error* is a standard indicator of quality. If $y_1, y_2, \ldots, y_n$ are the actual values and $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ the predicted values according to the linear regression model, then $(\sum_{i=1}^{n}(y_i - \hat{y}_i)^2)/n$ is the mean square error.

Clustering is typically used in a more descriptive or explanatory manner, and rarely used to make direct predictions about unseen instances. Nevertheless, the clusters derived for a training set could also be tested on a test set. Assign all instances in the test set to the closest centroid. After doing this, the *average distance* of each instance to its centroid can be used as a performance measure.

In the context of association rule mining, we defined metrics such as *support*, *confidence*, and *lift*. One can learn association rules using a training set and then test the discovered rules using the test set. The confidence metric then indicates the proportion of instances for which the rule holds while being applicable. Later, we will also define such metrics for process mining. For example, given an event log that serves as a test set and a Petri net model, one can look at the proportion of instances that can be replayed by the model.

Figure 4.16 shows the basic setting for cross-validation. The data set is split into a test and training set. Based on the training set a model is generated (e.g., a decision tree or regression model). Then the performance is analyzed using the test set. If just one number is generated for the performance indicator, then this does not give an indication of the reliability of the result. For example, based on some test set the F1

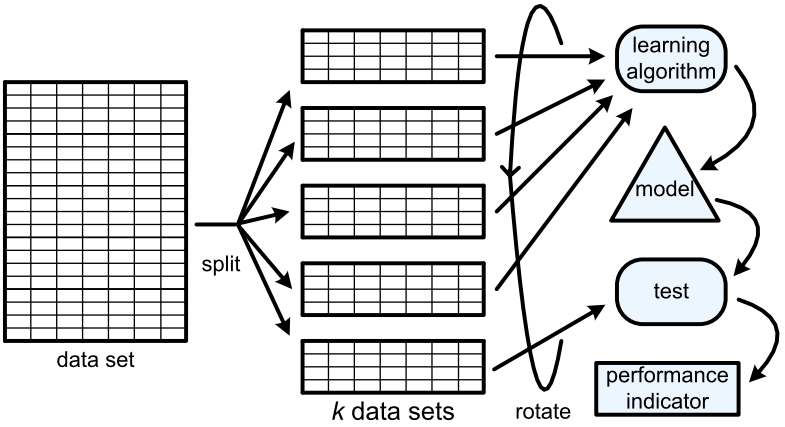


Fig. 4.17 k-fold cross-validation

score is 0.811. However, based on another test set the F1 score could be completely different even if the circumstances did not change. Therefore, one often wants to calculate a *confidence interval* for such a performance indicator. Confidence intervals can only be computed over multiple measurements. Here, we discuss two possibilities.

The first possibility is that one is measuring a performance indicator that is the average over of a large set of independent measurements. Consider for example classification. The test set consists of N instances that are mutually independent. Hence, each classification $1 \leq i \leq N$ can be seen as a separate test x_i where $x_i = 1$ means that the classification is wrong and $x_i = 0$ means that the classification is good. These tests can be seen as samples from a Bernoulli distribution with parameter p (p is the probability of a wrong classification). This distribution has an expected value p and variance $p(1-p)$. If we assume that N is large, then the average error $(\sum_{i=1}^{n} x_i)/N$ is approximately normal distributed. This is due to the central limit theorem, also known as the "law of large numbers". Using this assumption we find the *95% confidence interval* which is $[p - \alpha_{0.95}\sqrt{p(1-p)/N}, p + \alpha_{0.95}\sqrt{p(1-p)/N}]$, i.e., with 95% certainty the real average error will lie within $p - \alpha_{0.95}\sqrt{p(1-p)/N}$ and $p + \alpha_{0.95}\sqrt{p(1-p)/N}$. $\alpha_{0.95} = 1.96$ is a standard value that can be found in any statistics textbook, p is the measured average error rate, and N is the number of tests. For calculating the 90% or 99% confidence interval one can use $\alpha_{0.90} = 1.64$ respectively $\alpha_{0.99} = 2.58$. Note that it is only possible to calculate such an interval if there are many independent measurements possible based on a single test run.

The second possibility is *k-fold cross-validation*. This approach is used when there are relatively few instances in the data set or when the performance indicator is defined on a set of instances rather than a single instance. For example, the F1 score cannot be defined for one instance in isolation. Figure 4.17 illustrates the idea behind k-fold cross-validation. The data set is split into k equal parts, e.g., $k = 10$. Then k tests are done. In each test, one of the subsets serves as a test set whereas

the other $k-1$ subsets serve together as the training set. If subset $i \in \{1, 2, \ldots, k\}$ is used as a test set, then the union of subsets $\{1, 2, \ldots, i-1, i+1, \ldots k\}$ is used as the training set. One can inspect the individual tests or take the average of the k folds.

There are two advantages associated to k-fold cross-validation. First of all, all data is used both as training data and test data. Second, if desired, one gets k tests of the desired performance indicator rather than just one. Formally, the tests cannot be considered to be independent as the training sets used in the k folds overlap considerably. Nevertheless, the k folds make it possible to get more insight into the reliability.

An extreme variant of k-fold cross-validation is "leave-one-out" cross-validation, also known as jack-knifing. Here $k = N$, i.e., the test sets contain only one element at a time. See [9, 102] for more information on the various forms of cross-validation.

4.6.3 Occam's Razor

Evaluating the quality of data mining results is far from trivial. In this subsection, we discuss some additional complications that are also relevant for process mining.

Learning is typically an "ill posed problem", i.e., only examples are given. Some examples may rule out certain solutions, however, typically many possible models remain. Moreover, there is typically a *bias* in both the target representation and the learning algorithm. Consider, for example, the sequence $2, 3, 5, 7, 11, \ldots$. What is the next element in this sequence? Most readers will guess that it is 13, i.e., the next prime number, but there are infinitely many sequences that start with $2, 3, 5, 7, 11$. Yet, there seems to be preference for hypothesizing about some solutions. The term *inductive bias* refers to a preference for one solution rather than another which cannot be determined by the data itself but which is driven by external factors.

A *representational bias* refers to choices that are implicitly made by selecting a particular representation. For example, in Sect. 4.2, we assumed that in a decision tree the same attribute may appear only once on a path. This representational bias rules out certain solutions, e.g., a decision tree where closer to the root a numerical attribute is used in a coarse-grained manner and in some subtrees it is used in a fine-grained manner. Linear regression also makes assumptions about the function used to best fit the data. The function is assumed to be linear although there may be other non-linear functions that fit the data much better. Note that a representational bias is not necessarily bad, e.g., linear regression has been successfully used in many application domains. However, it is important to realize that the search space is limited by the representation used. The limitations can guide the search process, but also exclude good solutions.

A *learning bias* refers to strategies used by the algorithm that give preference to particular solutions. For example, in Fig. 4.4, we used the criterion of information gain (reduction of entropy) to select attributes. However, we could also have used the Gini index of diversity G rather than entropy E to select attributes, thus resulting in different decision trees.

Both factors also play a role in process mining. Consider, for example, Fig. 2.6 in the first chapter. This process model was discovered using the α-algorithm [157] based on the set of traces $\{\langle a,b,d,e,h\rangle, \langle a,d,c,e,g\rangle, \langle a,c,d,e,f,b,d,e,g\rangle, \langle a,d,b,e,h\rangle, \langle a,c,d,e,f,d,c,e,f,b,d,e,h\rangle, \langle a,c,d,e,g\rangle\}$. Clearly, there is a representational bias. The assumption is that the process can be presented by a Petri net where every transition bears a unique and visible label. Many processes cannot be represented by such a Petri net. The α-algorithm also has a learning bias as it is focusing on "direct succession". If a is directly followed by b in the event log, then this information is used. However, an observation such as "a is eventually followed by b in the event log" is not exploited by the α-algorithm.

An inductive bias is not necessarily bad. In fact it is often needed to come to a solution. However, the analyst should be aware of this and reflect on the implicit choices made.

Curse of dimensionality

Some data sets have many variables. However, for most data mining problems the amount of data needed to maintain a specific level of accuracy is exponential in the number of parameters [69]. High-dimensional problems, i.e., analyzing a data set with many variables, may be computationally intractable or lead to incorrect conclusions. This is the *"curse of dimensionality"* that many real-life applications of data mining are confronted with. Consider, for example, a supermarket selling 1000 products. In this case, there are $2^{1000}-1$ potential item-sets. Although the Apriori algorithm can quickly rule out many irrelevant candidates, the generation of association rules in such a setting is likely to encounter performance problems. Moreover, the interpretation of the results is typically difficult due to an excessive number of potential rules. In a supermarket having hundreds or thousands of products, there are many customers that purchase a unique combination of products. If there are 1000 different products, then there are $2^{1000}-1 \approx 1.07 \times 10^{301}$ possible shopping lists (ignoring quantities). Although the probability that two customers purchase the same is small, the number of potential rules is very large. This problem is not restricted to association rule learning. Clustering or regression in a 1000 dimensional space will suffer from similar problems. Typical approaches to address this problem are *variable selection* and *transformation* [69]. The goal of variable selection is to simply remove irrelevant of redundant variables. For example, the student's registration number and address are irrelevant when predicting study progress. Sometimes the data set can be transformed to reduce dimensionality, e.g., taking the average mark rather than individual marks per course.

Another problem is the delicate balance between *overfitting* and *underfitting*. A learned model is overfitting if it is too specific and too much driven by accidental information in the data set. For example, when constructing a decision tree

for a training set without conflicting input (i.e., instances with identical attributes belong to the same class), it is easy to construct a decision tree with a perfect F1 score. This tree can be obtained by continuing to split nodes until each leaf node corresponds to instances belonging to the same class. However, it is obvious that such a decision tree is too specific and has little predictive value.

A learned model is underfitting if it is too general and allows for things not "supported by evidence" in the data set. Whereas overfitting can be characterized by a lack of generalization, underfitting has the opposite problem: too much generalization. Consider, for example, the generation of association rules. Generating many detailed rules due to very low settings of *minsup* and *minconf*, corresponds to overfitting. Many rules are found, but these are probably rather specific for the training set. Generating very few rules due to very high settings of *minsup* and *minconf*, corresponds to underfitting. In the extreme case no association rules are found. Note that the model with no rules fits any data set and, hence, carries no information.

Underfitting is particularly problematic if the data set contains *no negative examples*. Consider, for example, the confusion matrix in Fig. 4.14(a). Suppose that we have a training set with only positive examples, i.e., $n = 0$ in the training set. How to construct a decision tree without negative examples? Most algorithms will simply classify everything as positive. This shows that classification assumes both positive and negative examples. This is not the case for association rule learning. Consider, for example, the data set shown in Table 4.3. Suppose that the item-set $\{latte, tea, bagel\}$ does not appear in the data set. This implies that no customer ordered these three items together in the training set. Can we conclude from this that it is not possible to order these three items together? Of course not! Therefore, association rule learning focuses on positive examples that are somehow frequent. Nevertheless, for some applications it would be useful to be able to discover "negative rules" such as the rule that customers are not allowed to order latte's, teas, and bagels in a single order.

A good balance between overfitting and underfitting is of the utmost importance for process discovery. Consider again the Petri net model shown in Fig. 2.6. The model allows for the behavior seen in the event log. It also generalizes as it allows for more sequences than present in the training set. In the event log there is no trace $\langle a, h \rangle$, i.e., the scenario in which a request is registered and immediately rejected does not appear in the log. This does not necessarily imply that this is not possible. However, constructing a model that allows for $\langle a, h \rangle$ although it is not in the log would result in a model that is clearly underfitting. This dilemma is caused by the lack of negative examples in the event log. The traces in the event log show what has happened and not what could not happen. This problem will be addressed in later chapters.

We conclude this chapter with *Occam's Razor*, a principle attributed to the 14th-century English logician William of Ockham. The principle states that "one should not increase, beyond what is necessary, the number of entities required to explain anything", i.e., one should look for the "simplest model" that can explain what is observed in the data set. This principle is related to finding a natural balance between overfitting and underfitting. The *Minimal Description Length* (MDL) principle tries

to operationalize Occam's Razor [63, 190]. According to the MDL paradigm, model quality is no longer only based on predicting performance (e.g., F1 score), but also on the simplicity of the model. Moreover, it does not aim at cross-validation in the sense of Sect. 4.6.2. In MDL performance is judged on the training data alone and not measured against new, unseen instances. The basic idea is that the "best" model is the one that *minimizes the encoding of both model and data set*. Here the insight is used that any regularity in the data can be used to compress the data, i.e., to describe it using fewer symbols than the number of symbols needed to describe the data literally. The more regularities there are, the more the data can be *compressed*. Equating "learning" with "finding regularity", implies that the more we are able to compress the data, the more we have learned about the data [63]. Obviously, a data set can be encoded more compactly if valuable knowledge about the data set is captured in the model. However, encoding such knowledge also requires space. A complex and overfitting model helps to reduce the encoding of the data set. A simple and underfitting model can be stored compactly, but does not help in reducing the encoding of the data set. Note that this idea is related to the notion of entropy in decision tree learning. When building the decision tree, the goal is to find homogeneous leaf nodes that can be encoded compactly. However, when discussing algorithms for decision tree learning in Sect. 4.2 there was no penalty for the complexity of the decision tree itself. The goal of MDL is to minimize the entropy of (a) the data set encoded using the learned model and (b) the encoding of the model itself. To balance between overfitting and underfitting, variable weights may be associated to both encodings.

Applying Occam's Razor is not easy. Extracting reliable and meaningful insights from complex data is far from trivial. In fact, it is much easier to transform complex data sets into "impressive looking garbage" by abusing the techniques presented in this chapter. However, when used wisely, data mining can add tremendous value. Moreover, process mining adds the "process dimension" to data and can be used to dissect event data from a more holistic perspective. As will be shown in the remainder, process mining creates a solid bridge between process modeling and analysis on the one hand and data mining on the other.