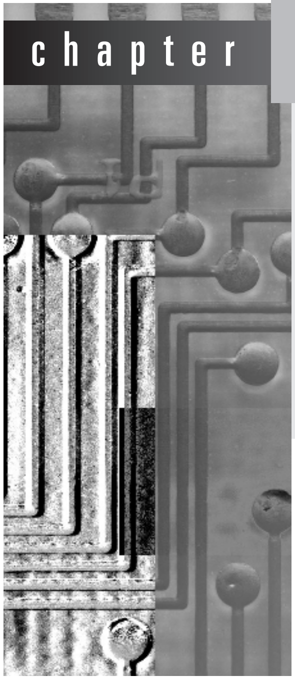
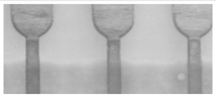
chapter 4

Algorithms: The Basic Methods





Now that we've seen how the inputs and outputs can be represented, it's time to look at the learning algorithms themselves. This chapter explains the basic ideas behind the techniques that are used in practical data mining. We will not delve too deeply into the trickier issues—advanced versions of the algorithms, optimizations that are possible, complications that arise in practice. These topics are deferred to Chapter 6, where we come to grips with real implementations of machine learning methods such as the ones included in data mining toolkits and used for real-world applications. It is important to understand these more advanced issues so that you know what is really going on when you analyze a particular dataset.

In this chapter we look at the basic ideas. One of the most instructive lessons is that simple ideas often work very well, and we strongly recommend the adoption of a "simplicity-first" methodology when analyzing practical datasets. There are many different kinds of simple structure that datasets can exhibit. In one dataset, there might be a single attribute that does all the work and the others may be irrelevant or redundant. In another dataset, the attributes might contribute independently and equally to the final outcome. A third might have a simple logical structure, involving just a few attributes that can be captured by a decision tree. In a fourth, there may be a few independent rules that govern the assignment of instances to different classes. A fifth might exhibit dependencies among different subsets of attributes. A sixth might involve linear dependence among numeric attributes, where what matters is a weighted sum of attribute values with appropriately chosen weights. In a seventh, classifications appropriate to particular regions of instance space might be governed by the distances between the instances themselves. And in an eighth, it might be that no class values are provided: the learning is unsupervised.

In the infinite variety of possible datasets there are many different kinds of structure that can occur, and a data mining tool—no matter how capable—that is looking for one class of structure may completely miss regularities of a different kind, regardless of how rudimentary those may be. The result is a baroque and opaque classification structure of one kind instead of a simple, elegant, immediately comprehensible structure of another.

Each of the eight examples of different kinds of datasets sketched previously leads to a different machine learning method well suited to discovering it. The sections of this chapter look at each of these structures in turn.

4.1 Inferring rudimentary rules

Here's an easy way to find very simple classification rules from a set of instances. Called *1R* for *1-rule*, it generates a one-level decision tree expressed in the form of a set of rules that all test one particular attribute. 1R is a simple, cheap method that often comes up with quite good rules for characterizing the structure in data. It turns out that simple rules frequently achieve surprisingly high accuracy. Perhaps this is because the structure underlying many real-world datasets is quite rudimentary, and just one attribute is sufficient to determine the class of an instance quite accurately. In any event, it is always a good plan to try the simplest things first.

The idea is this: we make rules that test a single attribute and branch accordingly. Each branch corresponds to a different value of the attribute. It is obvious what is the best classification to give each branch: use the class that occurs most often in the training data. Then the error rate of the rules can easily be determined. Just count the errors that occur on the training data, that is, the number of instances that do not have the majority class.

Each attribute generates a different set of rules, one rule for every value of the attribute. Evaluate the error rate for each attribute's rule set and choose the best. It's that simple! Figure 4.1 shows the algorithm in the form of pseudocode.

```
For each attribute,
  For each value of that attribute, make a rule as follows:
    count how often each class appears
    find the most frequent class
    make the rule assign that class to this attribute-value.
  Calculate the error rate of the rules.
Choose the rules with the smallest error rate.
```

Figure 4.1 Pseudocode for 1R.

Table 4.1 **Evaluating the attributes in the weather data.**

	Attribute	Rules	Errors	Total errors
1	outlook	sunny → no	2/5	4/14
		overcast → yes	0/4	
		rainy → yes	2/5	
2	temperature	hot → no*	2/4	5/14
		mild → yes	2/6	
		cool → yes	1/4	
3	humidity	high → no	3/7	4/14
		normal → yes	1/7	
4	windy	false → yes	2/8	5/14
		true → no*	3/6	

*A random choice was made between two equally likely outcomes.

To see the 1R method at work, consider the weather data of Table 1.2 (we will encounter it many times again when looking at how learning algorithms work). To classify on the final column, *play*, 1R considers four sets of rules, one for each attribute. These rules are shown in Table 4.1. An asterisk indicates that a random choice has been made between two equally likely outcomes. The number of errors is given for each rule, along with the total number of errors for the rule set as a whole. 1R chooses the attribute that produces rules with the smallest number of errors—that is, the first and third rule sets. Arbitrarily breaking the tie between these two rule sets gives:

```
outlook: sunny    → no
         overcast → yes
         rainy    → yes
```

We noted at the outset that the game for the weather data is unspecified. Oddly enough, it is apparently played when it is overcast or rainy but not when it is sunny. Perhaps it's an indoor pursuit.

Missing values and numeric attributes

Although a very rudimentary learning method, 1R does accommodate both missing values and numeric attributes. It deals with these in simple but effective ways. *Missing* is treated as just another attribute value so that, for example, if the weather data had contained missing values for the *outlook* attribute, a rule set formed on *outlook* would specify four possible class values, one each for *sunny, overcast,* and *rainy* and a fourth for *missing*.

We can convert numeric attributes into nominal ones using a simple discretization method. First, sort the training examples according to the values of the numeric attribute. This produces a sequence of class values. For example, sorting the numeric version of the weather data (Table 1.3) according to the values of *temperature* produces the sequence

64	65	68	69	70	71	72	72	75	75	80	81	83	85
yes	no	yes	yes	yes	no	no	yes	yes	yes	no	yes	yes	no

Discretization involves partitioning this sequence by placing breakpoints in it. One possibility is to place breakpoints wherever the class changes, producing eight categories:

```
yes | no | yes yes yes | no no | yes yes yes | no | yes yes | no
```

Choosing breakpoints halfway between the examples on either side places them at 64.5, 66.5, 70.5, 72, 77.5, 80.5, and 84. However, the two instances with value 72 cause a problem because they have the same value of *temperature* but fall into different classes. The simplest fix is to move the breakpoint at 72 up one example, to 73.5, producing a mixed partition in which *no* is the majority class.

A more serious problem is that this procedure tends to form a large number of categories. The 1R method will naturally gravitate toward choosing an attribute that splits into many categories, because this will partition the dataset into many classes, making it more likely that instances will have the same class as the majority in their partition. In fact, the limiting case is an attribute that has a different value for each instance—that is, an *identification code* attribute that pinpoints instances uniquely—and this will yield a zero error rate on the training set because each partition contains just one instance. Of course, highly branching attributes do not usually perform well on test examples; indeed, the *identification code* attribute will never predict any examples outside the training set correctly. This phenomenon is known as *overfitting;* we have already described overfitting-avoidance bias in Chapter 1 (page 35), and we will encounter this problem repeatedly in subsequent chapters.

For 1R, overfitting is likely to occur whenever an attribute has a large number of possible values. Consequently, when discretizing a numeric attribute a rule is adopted that dictates a minimum number of examples of the majority class in each partition. Suppose that minimum is set at three. This eliminates all but two of the preceding partitions. Instead, the partitioning process begins

```
yes no yes yes | yes . . .
```

ensuring that there are three occurrences of *yes,* the majority class, in the first partition. However, because the next example is also *yes,* we lose nothing by including that in the first partition, too. This leads to a new division:

```
yes no yes yes yes | no no yes yes yes | no yes yes no
```

where each partition contains at least three instances of the majority class, except the last one, which will usually have less. Partition boundaries always fall between examples of different classes.

Whenever adjacent partitions have the same majority class, as do the first two partitions above, they can be merged together without affecting the meaning of the rule sets. Thus the final discretization is

```
yes no yes yes yes no no yes yes yes | no yes yes no
```

which leads to the rule set

```
temperature: ≤ 77.5 → yes
             > 77.5 → no
```

The second rule involved an arbitrary choice; as it happens, *no* was chosen. If we had chosen *yes* instead, there would be no need for any breakpoint at all—and as this example illustrates, it might be better to use the adjacent categories to help to break ties. In fact this rule generates five errors on the training set and so is less effective than the preceding rule for *outlook*. However, the same procedure leads to this rule for *humidity*:

```
humidity: ≤ 82.5 → yes
          > 82.5 and ≤ 95.5 → no
          > 95.5 → yes
```

This generates only three errors on the training set and is the best "1-rule" for the data in Table 1.3.

Finally, if a numeric attribute has missing values, an additional category is created for them, and the preceding discretization procedure is applied just to the instances for which the attribute's value is defined.

Discussion

In a seminal paper titled "Very simple classification rules perform well on most commonly used datasets" (Holte 1993), a comprehensive study of the performance of the 1R procedure was reported on 16 datasets frequently used by machine learning researchers to evaluate their algorithms. Throughout, the study used *cross-validation,* an evaluation technique that we will explain in Chapter 5, to ensure that the results were representative of what independent test sets would yield. After some experimentation, the minimum number of examples in each partition of a numeric attribute was set at six, not three as used for the preceding illustration.

Surprisingly, despite its simplicity 1R did astonishingly—even embarrassingly—well in comparison with state-of-the-art learning methods, and the rules it produced turned out to be just a few percentage points less accurate, on almost all of the datasets, than the decision trees produced by a state-of-the-art decision tree induction scheme. These trees were, in general, considerably larger than 1R's rules. Rules that test a single attribute are often a viable alternative to more complex structures, and this strongly encourages a simplicity-first methodology in which the baseline performance is established using simple, rudimentary techniques before progressing to more sophisticated learning methods, which inevitably generate output that is harder for people to interpret.

The 1R procedure learns a one-level decision tree whose leaves represent the various different classes. A slightly more expressive technique is to use a different rule for each class. Each rule is a conjunction of tests, one for each attribute. For numeric attributes the test checks whether the value lies within a given interval; for nominal ones it checks whether it is in a certain subset of that attribute's values. These two types of tests—intervals and subset—are learned from the training data pertaining to each class. For a numeric attribute, the endpoints of the interval are the minimum and maximum values that occur in the training data for that class. For a nominal one, the subset contains just those values that occur for that attribute in the training data for the class. Rules representing different classes usually overlap, and at prediction time the one with the most matching tests is predicted. This simple technique often gives a useful first impression of a dataset. It is extremely fast and can be applied to very large quantities of data.

4.2 Statistical modeling

The 1R method uses a single attribute as the basis for its decisions and chooses the one that works best. Another simple technique is to use all attributes and allow them to make contributions to the decision that are *equally important* and *independent* of one another, given the class. This is unrealistic, of course: what

Table 4.2 The weather data with counts and probabilities.

Outlook			Temperature			Humidity			Windy			Play	
	yes	*no*		*yes*	*no*		*yes*	*no*		*yes*	*no*	*yes*	*no*
sunny	2	3	hot	2	2	high	3	4	false	6	2	9	5
overcast	4	0	mild	4	2	normal	6	1	true	3	3		
rainy	3	2	cool	3	1								
sunny	2/9	3/5	hot	2/9	2/5	high	3/9	4/5	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	mild	4/9	2/5	normal	6/9	1/5	true	3/9	3/5		
rainy	3/9	2/5	cool	3/9	1/5								

Table 4.3 A new day.

Outlook	Temperature	Humidity	Windy	Play
sunny	cool	high	true	?

makes real-life datasets interesting is that the attributes are certainly not equally important or independent. But it leads to a simple scheme that again works surprisingly well in practice.

Table 4.2 shows a summary of the weather data obtained by counting how many times each attribute–value pair occurs with each value (*yes* and *no*) for *play*. For example, you can see from Table 1.2 that *outlook* is *sunny* for five examples, two of which have *play* = *yes* and three of which have *play* = *no*. The cells in the first row of the new table simply count these occurrences for all possible values of each attribute, and the *play* figure in the final column counts the total number of occurrences of *yes* and *no*. In the lower part of the table, we rewrote the same information in the form of fractions, or observed probabilities. For example, of the nine days that *play* is yes, *outlook* is *sunny* for two, yielding a fraction of 2/9. For *play* the fractions are different: they are the proportion of days that *play* is *yes* and *no,* respectively.

Now suppose we encounter a new example with the values that are shown in Table 4.3. We treat the five features in Table 4.2—*outlook, temperature, humidity, windy,* and the overall likelihood that *play* is *yes* or *no*—as equally important, independent pieces of evidence and multiply the corresponding fractions. Looking at the outcome *yes* gives:

$$\text{likelihood of } yes = 2/9 \times 3/9 \times 3/9 \times 3/9 \times 9/14 = 0.0053.$$

The fractions are taken from the *yes* entries in the table according to the values of the attributes for the new day, and the final 9/14 is the overall fraction representing the proportion of days on which *play* is *yes*. A similar calculation for the outcome *no* leads to

$$\text{likelihood of } no = 3/5 \times 1/5 \times 4/5 \times 3/5 \times 5/14 = 0.0206.$$

This indicates that for the new day, *no* is more likely than *yes*—four times more likely. The numbers can be turned into probabilities by normalizing them so that they sum to 1:

$$\text{Probability of } yes = \frac{0.0053}{0.0053 + 0.0206} = 20.5\%,$$

$$\text{Probability of } no = \frac{0.0206}{0.0053 + 0.0206} = 79.5\%.$$

This simple and intuitive method is based on Bayes's rule of conditional probability. Bayes's rule says that if you have a hypothesis H and evidence E that bears on that hypothesis, then

$$\Pr[H|E] = \frac{\Pr[E|H]\Pr[H]}{\Pr[E]}.$$

We use the notation that $\Pr[A]$ denotes the probability of an event A and that $\Pr[A|B]$ denotes the probability of A conditional on another event B. The hypothesis H is that *play* will be, say, *yes,* and $\Pr[H|E]$ is going to turn out to be 20.5%, just as determined previously. The evidence E is the particular combination of attribute values for the new day, *outlook* = *sunny, temperature* = *cool, humidity* = *high*, and *windy* = *true*. Let's call these four pieces of evidence E_1, E_2, E_3, and E_4, respectively. Assuming that these pieces of evidence are independent (given the class), their combined probability is obtained by multiplying the probabilities:

$$\Pr[yes|E] = \frac{\Pr[E_1|yes] \times \Pr[E_2|yes] \times \Pr[E_3|yes] \times \Pr[E_4|yes] \times \Pr[yes]}{\Pr[E]}.$$

Don't worry about the denominator: we will ignore it and eliminate it in the final normalizing step when we make the probabilities of *yes* and *no* sum to 1, just as we did previously. The $\Pr[yes]$ at the end is the probability of a *yes* outcome without knowing any of the evidence E, that is, without knowing anything about the particular day referenced—it's called the *prior probability* of the hypothesis H. In this case, it's just 9/14, because 9 of the 14 training examples had a *yes* value for *play*. Substituting the fractions in Table 4.2 for the appropriate evidence probabilities leads to

$$\Pr[yes|E] = \frac{2/9 \times 3/9 \times 3/9 \times 3/9 \times 9/14}{\Pr[E]},$$

just as we calculated previously. Again, the Pr[E] in the denominator will disappear when we normalize.

This method goes by the name of *Naïve Bayes,* because it's based on Bayes's rule and "naïvely" assumes independence—it is only valid to multiply probabilities when the events are independent. The assumption that attributes are independent (given the class) in real life certainly is a simplistic one. But despite the disparaging name, Naïve Bayes works very well when tested on actual datasets, particularly when combined with some of the attribute selection procedures introduced in Chapter 7 that eliminate redundant, and hence nonindependent, attributes.

One thing that can go wrong with Naïve Bayes is that if a particular attribute value does not occur in the training set in conjunction with *every* class value, things go badly awry. Suppose in the example that the training data was different and the attribute value *outlook* = *sunny* had always been associated with the outcome *no*. Then the probability of *outlook* = *sunny* given a *yes,* that is, Pr[*outlook* = *sunny* | yes], would be zero, and because the other probabilities are multiplied by this the final probability of *yes* would be zero no matter how large they were. Probabilities that are zero hold a veto over the other ones. This is not a good idea. But the bug is easily fixed by minor adjustments to the method of calculating probabilities from frequencies.

For example, the upper part of Table 4.2 shows that for *play* = *yes, outlook* is *sunny* for two examples, *overcast* for four, and *rainy* for three, and the lower part gives these events probabilities of 2/9, 4/9, and 3/9, respectively. Instead, we could add 1 to each numerator and compensate by adding 3 to the denominator, giving probabilities of 3/12, 5/12, and 4/12, respectively. This will ensure that an attribute value that occurs zero times receives a probability which is nonzero, albeit small. The strategy of adding 1 to each count is a standard technique called the *Laplace estimator* after the great eighteenth-century French mathematician Pierre Laplace. Although it works well in practice, there is no particular reason for adding 1 to the counts: we could instead choose a small constant μ and use

$$\frac{2+\mu/3}{9+\mu}, \frac{4+\mu/3}{9+\mu}, \text{and } \frac{3+\mu/3}{9+\mu}.$$

The value of μ, which was set to 3, effectively provides a weight that determines how influential the a priori values of 1/3, 1/3, and 1/3 are for each of the three possible attribute values. A large μ says that these priors are very important compared with the new evidence coming in from the training set, whereas a small one gives them less influence. Finally, there is no particular reason for dividing μ into three *equal* parts in the numerators: we could use

$$\frac{2+\mu p_1}{9+\mu}, \frac{4+\mu p_2}{9+\mu}, \text{and } \frac{3+\mu p_3}{9+\mu}$$

instead, where p_1, p_2, and p_3 sum to 1. Effectively, these three numbers are a priori probabilities of the values of the *outlook* attribute being *sunny, overcast*, and *rainy*, respectively.

This is now a fully Bayesian formulation where prior probabilities have been assigned to everything in sight. It has the advantage of being completely rigorous, but the disadvantage that it is not usually clear just how these prior probabilities should be assigned. In practice, the prior probabilities make little difference provided that there are a reasonable number of training instances, and people generally just estimate frequencies using the Laplace estimator by initializing all counts to one instead of to zero.

Missing values and numeric attributes

One of the really nice things about the Bayesian formulation is that missing values are no problem at all. For example, if the value of *outlook* were missing in the example of Table 4.3, the calculation would simply omit this attribute, yielding

$$\text{likelihood of } \textit{yes} = 3/9 \times 3/9 \times 3/9 \times 9/14 = 0.0238$$
$$\text{likelihood of } \textit{no} = 1/5 \times 4/5 \times 3/5 \times 5/14 = 0.0343.$$

These two numbers are individually a lot higher than they were before, because one of the fractions is missing. But that's not a problem because a fraction is missing in both cases, and these likelihoods are subject to a further normalization process. This yields probabilities for *yes* and *no* of 41% and 59%, respectively.

If a value is missing in a training instance, it is simply not included in the frequency counts, and the probability ratios are based on the number of values that actually occur rather than on the total number of instances.

Numeric values are usually handled by assuming that they have a "normal" or "Gaussian" probability distribution. Table 4.4 gives a summary of the weather data with numeric features from Table 1.3. For nominal attributes, we calculated counts as before, and for numeric ones we simply listed the values that occur. Then, whereas we normalized the counts for the nominal attributes into probabilities, we calculated the mean and standard deviation for each class and each numeric attribute. Thus the mean value of *temperature* over the *yes* instances is 73, and its standard deviation is 6.2. The mean is simply the average of the preceding values, that is, the sum divided by the number of values. The standard deviation is the square root of the sample variance, which we can calculate as follows: subtract the mean from each value, square the result, sum them together, and then divide by *one less than* the number of values. After we have found this sample variance, find its square root to determine the standard deviation. This is the standard way of calculating mean and standard deviation of a

Table 4.4 The numeric weather data with summary statistics.

Outlook			Temperature			Humidity			Windy			Play	
	yes	*no*		*yes*	*no*		*yes*	*no*		*yes*	*no*	*yes*	*no*
sunny	2	3		83	85		86	85	false	6	2	9	5
overcast	4	0		70	80		96	90	true	3	3		
rainy	3	2		68	65		80	70					
				64	72		65	95					
				69	71		70	91					
				75			80						
				75			70						
				72			90						
				81			75						
sunny	2/9	3/5	*mean*	73	74.6	*mean*	79.1	86.2	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	*std. dev.*	6.2	7.9	*std. dev.*	10.2	9.7	true	3/9	3/5		
rainy	3/9	2/5											

set of numbers (the "one less than" is to do with the number of degrees of freedom in the sample, a statistical notion that we don't want to get into here).

The probability density function for a normal distribution with mean μ and standard deviation σ is given by the rather formidable expression:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{(x-\mu)^2}{2\sigma^2}}.$$

But fear not! All this means is that if we are considering a *yes* outcome when *temperature* has a value, say, of 66, we just need to plug $x = 66$, $\mu = 73$, and $\sigma = 6.2$ into the formula. So the value of the probability density function is

$$f(temperature = 66 \mid yes) = \frac{1}{\sqrt{2\pi} \cdot 6.2} e^{\frac{(66-73)^2}{2 \cdot 6.2^2}} = 0.0340.$$

By the same token, the probability density of a *yes* outcome when *humidity* has value, say, of 90 is calculated in the same way:

$$f(humidity = 90 \mid yes) = 0.0221.$$

The probability density function for an event is very closely related to its probability. However, it is not quite the same thing. If temperature is a continuous scale, the probability of the temperature being *exactly* 66—or *exactly* any other value, such as 63.14159262—is zero. The real meaning of the density function $f(x)$ is that the probability that the quantity lies within a small region around x, say, between $x - \varepsilon/2$ and $x + \varepsilon/2$, is $\varepsilon f(x)$. What we have written above is correct if temperature is measured to the nearest degree and humidity is measured to the nearest percentage point. You might think we ought to factor in the accuracy figure ε when using these probabilities, but that's not necessary. The same ε would appear in both the *yes* and *no* likelihoods that follow and cancel out when the probabilities were calculated.

Using these probabilities for the new day in Table 4.5 yields

$$\text{likelihood of } yes = 2/9 \times 0.0340 \times 0.0221 \times 3/9 \times 9/14 = 0.000036,$$
$$\text{likelihood of } no = 3/5 \times 0.0221 \times 0.0381 \times 3/5 \times 5/14 = 0.000108;$$

which leads to probabilities

$$\text{Probability of } yes = \frac{0.000036}{0.000036 + 0.000108} = 25.0\%,$$

$$\text{Probability of } no = \frac{0.000108}{0.000036 + 0.000108} = 75.0\%.$$

These figures are very close to the probabilities calculated earlier for the new day in Table 4.3, because the *temperature* and *humidity* values of 66 and 90 yield similar probabilities to the *cool* and *high* values used before.

The normal-distribution assumption makes it easy to extend the Naïve Bayes classifier to deal with numeric attributes. If the values of any numeric attributes are missing, the mean and standard deviation calculations are based only on the ones that are present.

Bayesian models for document classification

One important domain for machine learning is document classification, in which each instance represents a document and the instance's class is the document's topic. Documents might be news items and the classes might be domestic news, overseas news, financial news, and sport. Documents are characterized by the words that appear in them, and one way to apply machine learning to document classification is to treat the presence or absence of each word as a Boolean attribute. Naïve Bayes is a popular technique for this application because it is very fast and quite accurate.

However, this does not take into account the number of occurrences of each word, which is potentially useful information when determining the category

Table 4.5 **Another new day.**

Outlook	Temperature	Humidity	Windy	Play
sunny	66	90	true	?

of a document. Instead, a document can be viewed as a *bag of words*—a set that contains all the words in the document, with multiple occurrences of a word appearing multiple times (technically, a *set* includes each of its members just once, whereas a *bag* can have repeated elements). Word frequencies can be accommodated by applying a modified form of Naïve Bayes that is sometimes described as *multinominal* Naïve Bayes.

Suppose $n_1, n_2, \ldots, n_k$ is the number of times word i occurs in the document, and $P_1, P_2, \ldots, P_k$ is the probability of obtaining word i when sampling from all the documents in category H. Assume that the probability is independent of the word's context and position in the document. These assumptions lead to a *multinomial distribution* for document probabilities. For this distribution, the probability of a document E given its class H—in other words, the formula for computing the probability $\Pr[E|H]$ in Bayes's rule—is

$$\Pr[E\,|\,H] \approx N! \times \prod_{i=1}^{k} \frac{P_i^{n_i}}{n_i!}$$

where $N = n_1 + n_2 + \ldots + n_k$ is the number of words in the document. The reason for the factorials is to account for the fact that the ordering of the occurrences of each word is immaterial according to the bag-of-words model. P_i is estimated by computing the relative frequency of word i in the text of all training documents pertaining to category H. In reality there should be a further term that gives the probability that the model for category H generates a document whose length is the same as the length of E (that is why we use the symbol $\approx$ instead of $=$), but it is common to assume that this is the same for all classes and hence can be dropped.

For example, suppose there are only the two words, *yellow* and *blue,* in the vocabulary, and a particular document class H has $\Pr[\textit{yellow}|H] = 75\%$ and $\Pr[\textit{blue}|H] = 25\%$ (you might call H the class of *yellowish green* documents). Suppose E is the document *blue yellow blue* with a length of $N = 3$ words. There are four possible bags of three words. One is {*yellow yellow yellow*}, and its probability according to the preceding formula is

$$\Pr[\{\textit{yellow yellow yellow}\}\,|\,H] \approx 3! \times \frac{0.75^3}{3!} \times \frac{0.25^0}{0!} = \frac{27}{64}$$

The other three, with their probabilities, are

$$\Pr[\{\textit{blue blue blue}\}\,|\,H] = \frac{1}{64}$$

$$\Pr[\{\textit{yellow yellow blue}\}\,|\,H] = \frac{27}{64}$$

$$\Pr[\{\textit{yellow blue blue}\}\,|\,H] = \frac{9}{64}$$

Here, E corresponds to the last case (recall that in a bag of words the order is immaterial); thus its probability of being generated by the *yellowish green* document model is 9/64, or 14%. Suppose another class, *very bluish green* documents (call it H'), has $\Pr[yellow \mid H'] = 10\%$, $\Pr[blue \mid H'] = 90\%$. The probability that E is generated by this model is 24%.

If these are the only two classes, does that mean that E is in the *very bluish green* document class? Not necessarily. Bayes's rule, given earlier, says that you have to take into account the prior probability of each hypothesis. If you know that in fact *very bluish green* documents are twice as rare as *yellowish green* ones, this would be just sufficient to outweigh the preceding 14% to 24% disparity and tip the balance in favor of the *yellowish green* class.

The factorials in the preceding probability formula don't actually need to be computed because—being the same for every class—they drop out in the normalization process anyway. However, the formula still involves multiplying together many small probabilities, which soon yields extremely small numbers that cause underflow on large documents. The problem can be avoided by using logarithms of the probabilities instead of the probabilities themselves.

In the multinomial Naïve Bayes formulation a document's class is determined not just by the words that occur in it but also by the number of times they occur. In general it performs better than the ordinary Naïve Bayes model for document classification, particularly for large dictionary sizes.

Discussion

Naïve Bayes gives a simple approach, with clear semantics, to representing, using, and learning probabilistic knowledge. Impressive results can be achieved using it. It has often been shown that Naïve Bayes rivals, and indeed outperforms, more sophisticated classifiers on many datasets. The moral is, always try the simple things first. Repeatedly in machine learning people have eventually, after an extended struggle, obtained good results using sophisticated learning methods only to discover years later that simple methods such as 1R and Naïve Bayes do just as well—or even better.

There are many datasets for which Naïve Bayes does not do so well, however, and it is easy to see why. Because attributes are treated as though they were completely independent, the addition of redundant ones skews the learning process. As an extreme example, if you were to include a new attribute with the same values as *temperature* to the weather data, the effect of the *temperature* attribute would be multiplied: all of its probabilities would be squared, giving it a great deal more influence in the decision. If you were to add 10 such attributes, then the decisions would effectively be made on *temperature* alone. Dependencies between attributes inevitably reduce the power of Naïve Bayes to discern what is going on. They can, however, be ameliorated by using a subset of the attributes in the decision procedure, making a careful selection of which ones to use. Chapter 7 shows how.

The normal-distribution assumption for numeric attributes is another restriction on Naïve Bayes as we have formulated it here. Many features simply aren't normally distributed. However, there is nothing to prevent us from using other distributions for the numeric attributes: there is nothing magic about the normal distribution. If you know that a particular attribute is likely to follow some other distribution, standard estimation procedures for that distribution can be used instead. If you suspect it isn't normal but don't know the actual distribution, there are procedures for "kernel density estimation" that do not assume any particular distribution for the attribute values. Another possibility is simply to discretize the data first.

4.3 Divide-and-conquer: Constructing decision trees

The problem of constructing a decision tree can be expressed recursively. First, select an attribute to place at the root node and make one branch for each possible value. This splits up the example set into subsets, one for every value of the attribute. Now the process can be repeated recursively for each branch, using only those instances that actually reach the branch. If at any time all instances at a node have the same classification, stop developing that part of the tree.

The only thing left to decide is how to determine which attribute to split on, given a set of examples with different classes. Consider (again!) the weather data. There are four possibilities for each split, and at the top level they produce trees such as those in Figure 4.2. Which is the best choice? The number of *yes* and *no* classes are shown at the leaves. Any leaf with only one class—*yes* or *no*—will not have to be split further, and the recursive process down that branch will terminate. Because we seek small trees, we would like this to happen as soon as possible. If we had a measure of the purity of each node, we could choose the attribute that produces the purest daughter nodes. Take a moment to look at Figure 4.2 and ponder which attribute you think is the best choice.

The measure of purity that we will use is called the *information* and is measured in units called *bits*. Associated with a node of the tree, it represents the expected amount of information that would be needed to specify whether a new instance should be classified *yes* or *no*, given that the example reached that node. Unlike the bits in computer memory, the expected amount of information usually involves fractions of a bit—and is often less than one! We calculate it based on the number of *yes* and *no* classes at the node; we will look at the details of the calculation shortly. But first let's see how it's used. When evaluating the first tree in Figure 4.2, the numbers of *yes* and *no* classes at the leaf nodes are [2,3], [4,0], and [3,2], respectively, and the information values of these nodes are:

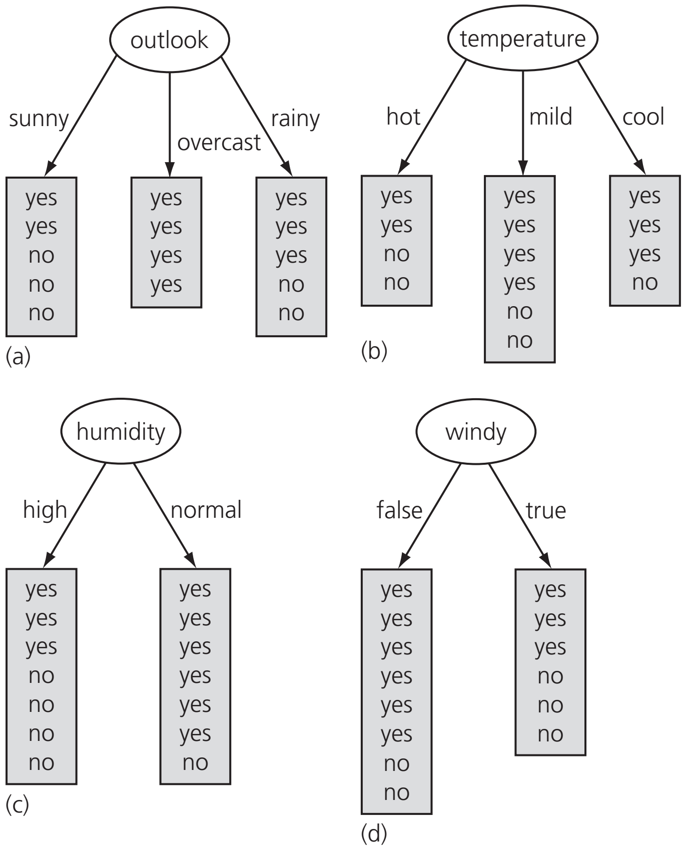


Figure 4.2 Tree stumps for the weather data.

$$\text{info}([2,3]) = 0.971 \text{ bits}$$
$$\text{info}([4,0]) = 0.0 \text{ bits}$$
$$\text{info}([3,2]) = 0.971 \text{ bits}$$

We can calculate the average information value of these, taking into account the number of instances that go down each branch—five down the first and third and four down the second:

$$\text{info}([2,3],[4,0],[3,2]) = (5/14) \times 0.971 + (4/14) \times 0 + (5/14) \times 0.971 = 0.693 \text{ bits.}$$

This average represents the amount of information that we expect would be necessary to specify the class of a new instance, given the tree structure in Figure 4.2(a).

Before we created any of the nascent tree structures in Figure 4.2, the training examples at the root comprised nine *yes* and five *no* nodes, corresponding to an information value of

$$\text{info}([9,5]) = 0.940 \text{ bits.}$$

Thus the tree in Figure 4.2(a) is responsible for an information gain of

$$\text{gain}(\textit{outlook}) = \text{info}([9,5]) - \text{info}([2,3],[4,0],[3,2]) = 0.940 - 0.693 = 0.247 \text{ bits,}$$

which can be interpreted as the informational value of creating a branch on the *outlook* attribute.

The way forward is clear. We calculate the information gain for each attribute and choose the one that gains the most information to split on. In the situation of Figure 4.2,

$$\begin{aligned}\text{gain}(\textit{outlook}) &= 0.247 \text{ bits}\\ \text{gain}(\textit{temperature}) &= 0.029 \text{ bits}\\ \text{gain}(\textit{humidity}) &= 0.152 \text{ bits}\\ \text{gain}(\textit{windy}) &= 0.048 \text{ bits,}\end{aligned}$$

so we select *outlook* as the splitting attribute at the root of the tree. Hopefully this accords with your intuition as the best one to select. It is the only choice for which one daughter node is completely pure, and this gives it a considerable advantage over the other attributes. *Humidity* is the next best choice because it produces a larger daughter node that is almost completely pure.

Then we continue, recursively. Figure 4.3 shows the possibilities for a further branch at the node reached when *outlook* is *sunny*. Clearly, a further split on *outlook* will produce nothing new, so we only consider the other three attributes. The information gain for each turns out to be

$$\begin{aligned}\text{gain}(\textit{temperature}) &= 0.571 \text{ bits}\\ \text{gain}(\textit{humidity}) &= 0.971 \text{ bits}\\ \text{gain}(\textit{windy}) &= 0.020 \text{ bits,}\end{aligned}$$

so we select *humidity* as the splitting attribute at this point. There is no need to split these nodes any further, so this branch is finished.

Continued application of the same idea leads to the decision tree of Figure 4.4 for the weather data. Ideally, the process terminates when all leaf nodes are pure, that is, when they contain instances that all have the same classification. However, it might not be possible to reach this happy situation because there is nothing to stop the training set containing two examples with identical sets of attributes but different classes. Consequently, we stop when the data cannot be split any further.

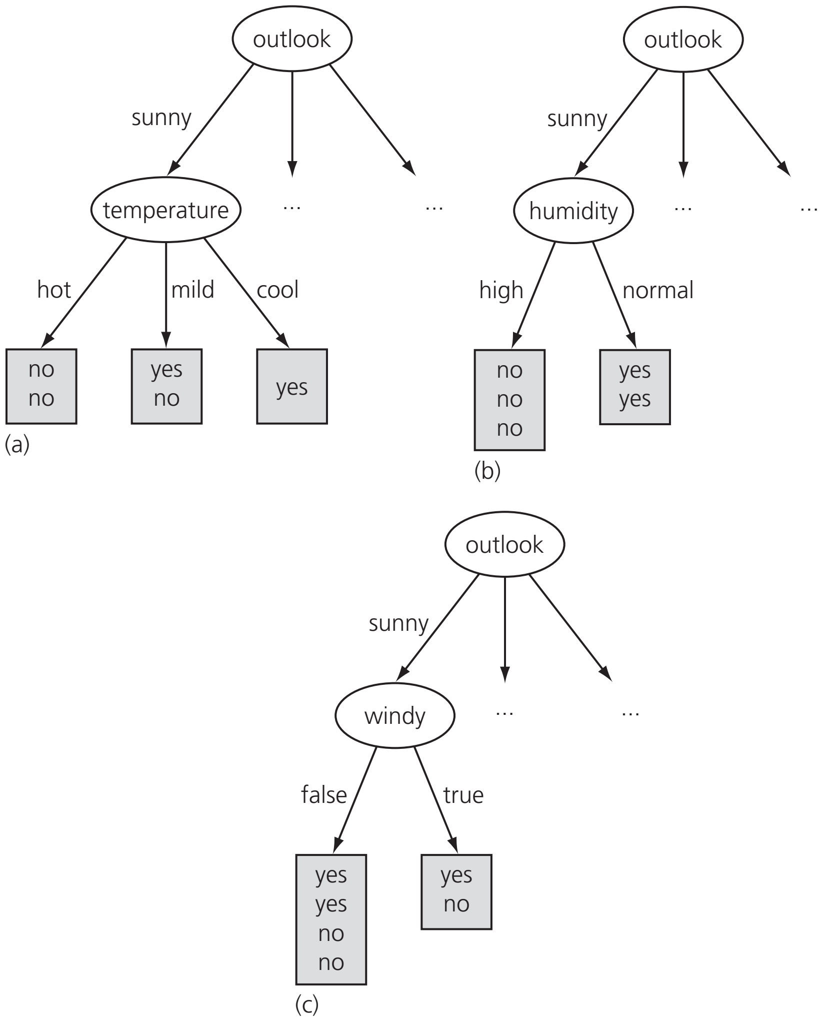


Figure 4.3 Expanded tree stumps for the weather data.

Calculating information

Now it is time to explain how to calculate the information measure that is used as a basis for evaluating different splits. We describe the basic idea in this section, then in the next we examine a correction that is usually made to counter a bias toward selecting splits on attributes with large numbers of possible values.

Before examining the detailed formula for calculating the amount of information required to specify the class of an example given that it reaches a tree node with a certain number of *yes*'s and *no*'s, consider first the kind of properties we would expect this quantity to have:

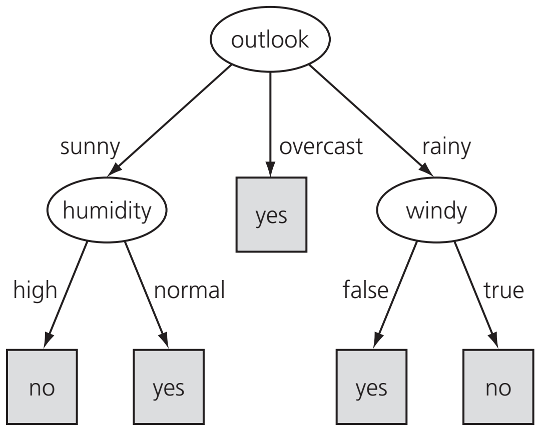


Figure 4.4 Decision tree for the weather data.

1. When the number of either *yes*'s or *no*'s is zero, the information is zero.
2. When the number of *yes*'s and *no*'s is equal, the information reaches a maximum.

Moreover, the measure should be applicable to multiclass situations, not just to two-class ones.

The information measure relates to the amount of information obtained by making a decision, and a more subtle property of information can be derived by considering the nature of decisions. Decisions can be made in a single stage, or they can be made in several stages, and the amount of information involved is the same in both cases. For example, the decision involved in

$$\text{info}([2,3,4])$$

can be made in two stages. First decide whether it's the first case or one of the other two cases:

$$\text{info}([2,7])$$

and then decide which of the other two cases it is:

$$\text{info}([3,4])$$

In some cases the second decision will not need to be made, namely, when the decision turns out to be the first one. Taking this into account leads to the equation

$$\text{info}([2,3,4]) = \text{info}([2,7]) + (7/9) \times \text{info}([3,4]).$$

Of course, there is nothing special about these particular numbers, and a similar relationship must hold regardless of the actual values. Thus we can add a further criterion to the preceding list:

3. The information must obey the multistage property illustrated previously.

Remarkably, it turns out that there is only one function that satisfies all these properties, and it is known as the *information value* or *entropy:*

$$\text{entropy}(p_1, p_2, \ldots, p_n) = -p_1 \log p_1 - p_2 \log p_2 \ldots - p_n \log p_n$$

The reason for the minus signs is that logarithms of the fractions $p_1, p_2, \ldots, p_n$ are negative, so the entropy is actually positive. Usually the logarithms are expressed in base 2, then the entropy is in units called *bits*—just the usual kind of bits used with computers.

The arguments $p_1, p_2, \ldots$ of the entropy formula are expressed as fractions that add up to one, so that, for example,

$$\text{info}([2,3,4]) = \text{entropy}(2/9, 3/9, 4/9).$$

Thus the multistage decision property can be written in general as

$$\text{entropy}(p,q,r) = \text{entropy}(p, q+r) + (q+r) \cdot \text{entropy}\left(\frac{q}{q+r}, \frac{r}{q+r}\right)$$

where $p + q + r = 1$.

Because of the way the log function works, you can calculate the information measure without having to work out the individual fractions:

$$\begin{aligned}\text{info}([2,3,4]) &= -2/9 \times \log 2/9 - 3/9 \times \log 3/9 - 4/9 \times \log 4/9 \\ &= [-2\log 2 - 3\log 3 - 4\log 4 + 9\log 9]/9.\end{aligned}$$

This is the way that the information measure is usually calculated in practice. So the information value for the first leaf node of the first tree in Figure 4.2 is

$$\text{info}([2,3]) = -2/5 \times \log 2/5 - 3/5 \times \log 3/5 = 0.971 \text{ bits},$$

as stated on page 98.

Highly branching attributes

When some attributes have a large number of possible values, giving rise to a multiway branch with many child nodes, a problem arises with the information gain calculation. The problem can best be appreciated in the extreme case when an attribute has a different value for each instance in the dataset—as, for example, an *identification code* attribute might.

Table 4.6 The weather data with identification codes.

ID code	Outlook	Temperature	Humidity	Windy	Play
a	sunny	hot	high	false	no
b	sunny	hot	high	true	no
c	overcast	hot	high	false	yes
d	rainy	mild	high	false	yes
e	rainy	cool	normal	false	yes
f	rainy	cool	normal	true	no
g	overcast	cool	normal	true	yes
h	sunny	mild	high	false	no
i	sunny	cool	normal	false	yes
j	rainy	mild	normal	false	yes
k	sunny	mild	normal	true	yes
l	overcast	mild	high	true	yes
m	overcast	hot	normal	false	yes
n	rainy	mild	high	true	no

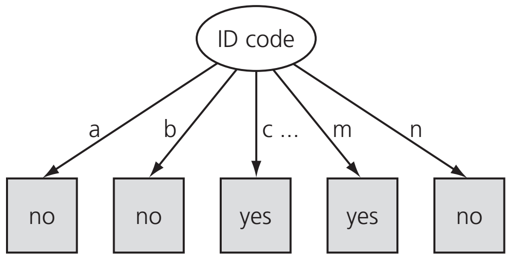


Figure 4.5 Tree stump for the *ID code* attribute.

Table 4.6 gives the weather data with this extra attribute. Branching on *ID code* produces the tree stump in Figure 4.5. The information required to specify the class given the value of this attribute is

$$\text{info}([0,1]) + \text{info}([0,1]) + \text{info}([1,0]) + \ldots + \text{info}([1,0]) + \text{info}([0,1]),$$

which is zero because each of the 14 terms is zero. This is not surprising: the *ID code* attribute identifies the instance, which determines the class without any ambiguity—just as Table 4.6 shows. Consequently, the information gain of this attribute is just the information at the root, info([9,5]) = 0.940 bits. This is greater than the information gain of any other attribute, and so *ID code* will inevitably be chosen as the splitting attribute. But branching on the identification code is no good for predicting the class of unknown instances and tells nothing about the structure of the decision, which after all are the twin goals of machine learning.

The overall effect is that the information gain measure tends to prefer attributes with large numbers of possible values. To compensate for this, a modification of the measure called the *gain ratio* is widely used. The gain ratio is derived by taking into account the number and size of daughter nodes into which an attribute splits the dataset, disregarding any information about the class. In the situation shown in Figure 4.5, all counts have a value of 1, so the information value of the split is

$$\text{info}([1,1,\ldots,1]) = -1/14 \times \log 1/14 \times 14,$$

because the same fraction, 1/14, appears 14 times. This amounts to log 14, or 3.807 bits, which is a very high value. This is because the information value of a split is the number of bits needed to determine to which branch each instance is assigned, and the more branches there are, the greater this value is. The gain ratio is calculated by dividing the original information gain, 0.940 in this case, by the information value of the attribute, 3.807—yielding a gain ratio value of 0.247 for the *ID code* attribute.

Returning to the tree stumps for the weather data in Figure 4.2, *outlook* splits the dataset into three subsets of size 5, 4, and 5 and thus has an intrinsic information value of

$$\text{info}([5,4,5]) = 1.577$$

without paying any attention to the classes involved in the subsets. As we have seen, this intrinsic information value is higher for a more highly branching attribute such as the hypothesized *ID code.* Again we can correct the information gain by dividing by the intrinsic information value to get the gain ratio.

The results of these calculations for the tree stumps of Figure 4.2 are summarized in Table 4.7. *Outlook* still comes out on top, but *humidity* is now a much closer contender because it splits the data into two subsets instead of three. In this particular example, the hypothetical *ID code* attribute, with a gain ratio of 0.247, would still be preferred to any of these four. However, its advantage is

Table 4.7 Gain ratio calculations for the tree stumps of Figure 4.2.

Outlook		Temperature		Humidity		Windy	
info:	0.693	info:	0.911	info:	0.788	info:	0.892
gain: 0.940–0.693	0.247	gain: 0.940–0.911	0.029	gain: 0.940–0.788	0.152	gain: 0.940–0.892	0.048
split info: info([5,4,5])	1.577	split info: info([4,6,4])	1.557	split info: info ([7,7])	1.000	split info: info([8,6])	0.985
gain ratio: 0.247/1.577	0.157	gain ratio: 0.029/1.557	0.019	gain ratio: 0.152/1	0.152	gain ratio: 0.048/0.985	0.049

greatly reduced. In practical implementations, we can use an ad hoc test to guard against splitting on such a useless attribute.

Unfortunately, in some situations the gain ratio modification overcompensates and can lead to preferring an attribute just because its intrinsic information is much lower than that for the other attributes. A standard fix is to choose the attribute that maximizes the gain ratio, provided that the information gain for that attribute is at least as great as the average information gain for all the attributes examined.

Discussion

The divide-and-conquer approach to decision tree induction, sometimes called *top-down induction of decision trees,* was developed and refined over many years by J. Ross Quinlan of the University of Sydney, Australia. Although others have worked on similar methods, Quinlan's research has always been at the very forefront of decision tree induction. The method that has been described using the information gain criterion is essentially the same as one known as ID3. The use of the gain ratio was one of many improvements that were made to ID3 over several years; Quinlan described it as robust under a wide variety of circumstances. Although a robust and practical solution, it sacrifices some of the elegance and clean theoretical motivation of the information gain criterion.

A series of improvements to ID3 culminated in a practical and influential system for decision tree induction called C4.5. These improvements include methods for dealing with numeric attributes, missing values, noisy data, and generating rules from trees, and they are described in Section 6.1.

4.4 Covering algorithms: Constructing rules

As we have seen, decision tree algorithms are based on a divide-and-conquer approach to the classification problem. They work from the top down, seeking at each stage an attribute to split on that best separates the classes; then recursively processing the subproblems that result from the split. This strategy generates a decision tree, which can if necessary be converted into a set of classification rules—although if it is to produce effective rules, the conversion is not trivial.

An alternative approach is to take each class in turn and seek a way of covering all instances in it, at the same time excluding instances not in the class. This is called a *covering* approach because at each stage you identify a rule that "covers" some of the instances. By its very nature, this covering approach leads to a set of rules rather than to a decision tree.

The covering method can readily be visualized in a two-dimensional space of instances as shown in Figure 4.6(a). We first make a rule covering the *a*'s. For

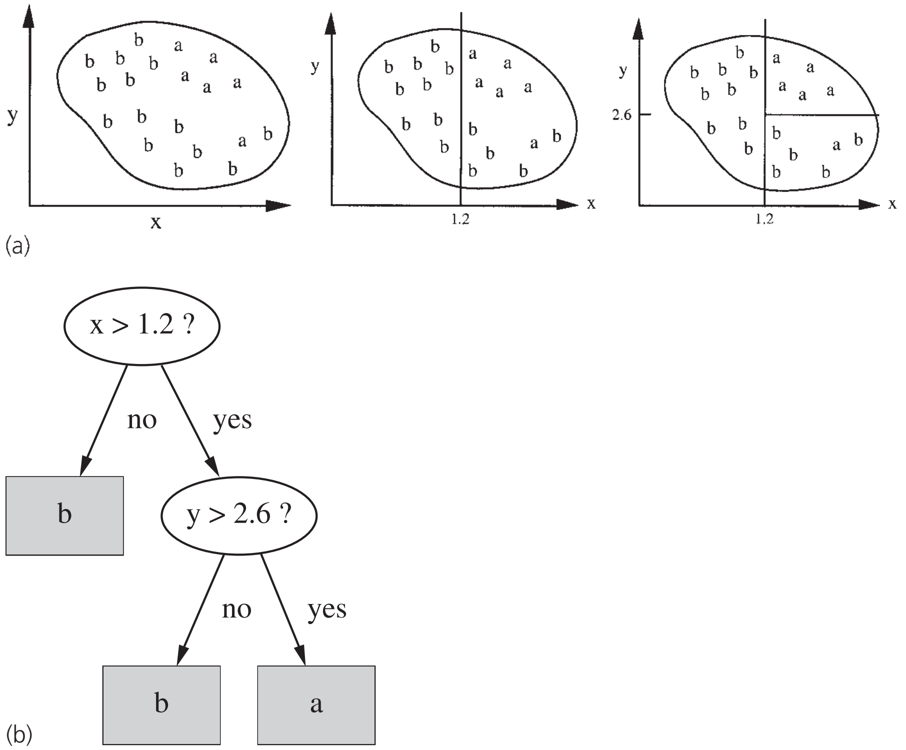


Figure 4.6 Covering algorithm: (a) covering the instances and (b) the decision tree for the same problem.

the first test in the rule, split the space vertically as shown in the center picture. This gives the beginnings of a rule:

```
If x > 1.2 then class = a
```

However, the rule covers many *b*'s as well as *a*'s, so a new test is added to the rule by further splitting the space horizontally as shown in the third diagram:

```
If x > 1.2 and y > 2.6 then class = a
```

This gives a rule covering all but one of the *a*'s. It's probably appropriate to leave it at that, but if it were felt necessary to cover the final *a*, another rule would be necessary—perhaps

```
If x > 1.4 and y < 2.4 then class = a
```

The same procedure leads to two rules covering the *b*'s:

```
If x ≤ 1.2 then class = b
If x > 1.2 and y ≤ 2.6 then class = b
```

Again, one *a* is erroneously covered by these rules. If it were necessary to exclude it, more tests would have to be added to the second rule, and additional rules would need to be introduced to cover the *b*'s that these new tests exclude.

Rules versus trees

A top-down divide-and-conquer algorithm operates on the same data in a manner that is, at least superficially, quite similar to a covering algorithm. It might first split the dataset using the x attribute and would probably end up splitting it at the same place, $x = 1.2$. However, whereas the covering algorithm is concerned only with covering a single class, the division would take both classes into account, because divide-and-conquer algorithms create a single concept description that applies to all classes. The second split might also be at the same place, $y = 2.6$, leading to the decision tree in Figure 4.6(b). This tree corresponds exactly to the set of rules, and in this case there is no difference in effect between the covering and the divide-and-conquer algorithms.

But in many situations there *is* a difference between rules and trees in terms of the perspicuity of the representation. For example, when we described the replicated subtree problem in Section 3.3, we noted that rules can be symmetric whereas trees must select one attribute to split on first, and this can lead to trees that are much larger than an equivalent set of rules. Another difference is that, in the multiclass case, a decision tree split takes all classes into account, trying to maximize the purity of the split, whereas the rule-generating method concentrates on one class at a time, disregarding what happens to the other classes.

A simple covering algorithm

Covering algorithms operate by adding tests to the rule that is under construction, always striving to create a rule with maximum accuracy. In contrast, divide-and-conquer algorithms operate by adding tests to the tree that is under construction, always striving to maximize the separation among the classes. Each of these involves finding an attribute to split on. But the criterion for the best attribute is different in each case. Whereas divide-and-conquer algorithms such as ID3 choose an attribute to maximize the information gain, the covering algorithm we will describe chooses an attribute–value pair to maximize the probability of the desired classification.

Figure 4.7 gives a picture of the situation, showing the space containing all the instances, a partially constructed rule, and the same rule after a new term has been added. The new term restricts the coverage of the rule: the idea is to include as many instances of the desired class as possible and exclude as many instances of other classes as possible. Suppose the new rule will cover a total of t instances, of which p are positive examples of the class and $t - p$ are in other

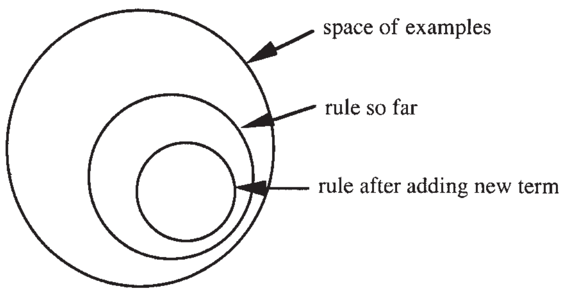


Figure 4.7 The instance space during operation of a covering algorithm.

classes—that is, they are errors made by the rule. Then choose the new term to maximize the ratio *p*/*t*.

An example will help. For a change, we use the contact lens problem of Table 1.1. We will form rules that cover each of the three classes, *hard, soft,* and *none,* in turn. To begin, we seek a rule:

```
If ? then recommendation = hard
```

For the unknown term ?, we have nine choices:

age = young	2/8
age = pre-presbyopic	1/8
age = presbyopic	1/8
spectacle prescription = myope	3/12
spectacle prescription = hypermetrope	1/12
astigmatism = no	0/12
astigmatism = yes	4/12
tear production rate = reduced	0/12
tear production rate = normal	4/12

The numbers on the right show the fraction of "correct" instances in the set singled out by that choice. In this case, *correct* means that the recommendation is *hard.* For instance, *age* = *young* selects eight instances, two of which recommend hard contact lenses, so the first fraction is 2/8. (To follow this, you will need to look back at the contact lens data in Table 1.1 on page 6 and count up the entries in the table.) We select the largest fraction, 4/12, arbitrarily choosing between the seventh and the last choice in the preceding list, and create the rule:

```
If astigmatism = yes then recommendation = hard
```

This rule is an inaccurate one, getting only 4 instances correct out of the 12 that it covers, shown in Table 4.8. So we refine it further:

```
If astigmatism = yes and ? then recommendation = hard
```

Table 4.8 Part of the contact lens data for which *astigmatism* = *yes*.

Age	Spectacle prescription	Astigmatism	Tear production rate	Recommended lenses
young	myope	yes	reduced	none
young	myope	yes	normal	hard
young	hypermetrope	yes	reduced	none
young	hypermetrope	yes	normal	hard
pre-presbyopic	myope	yes	reduced	none
pre-presbyopic	myope	yes	normal	hard
pre-presbyopic	hypermetrope	yes	reduced	none
pre-presbyopic	hypermetrope	yes	normal	none
presbyopic	myope	yes	reduced	none
presbyopic	myope	yes	normal	hard
presbyopic	hypermetrope	yes	reduced	none
presbyopic	hypermetrope	yes	normal	none

Considering the possibilities for the unknown term ? yields the seven choices:

`age = young`	2/4
`age = pre-presbyopic`	1/4
`age = presbyopic`	1/4
`spectacle prescription = myope`	3/6
`spectacle prescription = hypermetrope`	1/6
`tear production rate = reduced`	0/6
`tear production rate = normal`	4/6

(Again, count the entries in Table 4.8.) The last is a clear winner, getting four instances correct out of the six that it covers, and corresponds to the rule

```
If astigmatism = yes and tear production rate = normal
   then recommendation = hard
```

Should we stop here? Perhaps. But let's say we are going for exact rules, no matter how complex they become. Table 4.9 shows the cases that are covered by the rule so far. The possibilities for the next term are now

`age = young`	2/2
`age = pre-presbyopic`	1/2
`age = presbyopic`	1/2
`spectacle prescription = myope`	3/3
`spectacle prescription = hypermetrope`	1/3

We need to choose between the first and fourth. So far we have treated the fractions numerically, but although these two are equal (both evaluate to 1), they have different coverage: one selects just two correct instances and the other

Table 4.9 **Part of the contact lens data for which *astigmatism* = *yes* and *tear production rate* = *normal*.**

Age	Spectacle prescription	Astigmatism	Tear production rate	Recommended lenses
young	myope	yes	normal	hard
young	hypermetrope	yes	normal	hard
pre-presbyopic	myope	yes	normal	hard
pre-presbyopic	hypermetrope	yes	normal	none
presbyopic	myope	yes	normal	hard
presbyopic	hypermetrope	yes	normal	none

selects three. In the event of a tie, we choose the rule with the greater coverage, giving the final rule:

```
If astigmatism = yes and tear production rate = normal
   and spectacle prescription = myope then recommendation = hard
```

This is indeed one of the rules given for the contact lens problem. But it only covers three of the four *hard* recommendations. So we delete these three from the set of instances and start again, looking for another rule of the form:

```
If ? then recommendation = hard
```

Following the same process, we will eventually find that *age* = *young* is the best choice for the first term. Its coverage is seven; the reason for the seven is that 3 instances have been removed from the original set, leaving 21 instances altogether. The best choice for the second term is *astigmatism* = *yes,* selecting 1/3 (actually, this is a tie); *tear production rate* = *normal* is the best for the third, selecting 1/1.

```
If age = young and astigmatism = yes and
   tear production rate = normal then recommendation = hard
```

This rule actually covers three of the original set of instances, two of which are covered by the previous rule—but that's all right because the recommendation is the same for each rule.

Now that all the hard-lens cases are covered, the next step is to proceed with the soft-lens ones in just the same way. Finally, rules are generated for the *none* case—unless we are seeking a rule set with a default rule, in which case explicit rules for the final outcome are unnecessary.

What we have just described is the PRISM method for constructing rules. It generates only correct or "perfect" rules. It measures the success of a rule by the accuracy formula *p*/*t.* Any rule with accuracy less than 100% is "incorrect" in that it assigns cases to the class in question that actually do not have that class. PRISM continues adding clauses to each rule until it is perfect: its accuracy is 100%. Figure 4.8 gives a summary of the algorithm. The outer loop iterates over the classes, generating rules for each class in turn. Note that we reinitialize to the full set of examples each time round. Then we create rules for that class and remove the examples from the set until there are none of that class left. Whenever we create a rule, start with an empty rule (which covers all the examples), and then restrict it by adding tests until it covers only examples of the desired class. At each stage choose the most promising test, that is, the one that maximizes the accuracy of the rule. Finally, break ties by selecting the test with greatest coverage.

Rules versus decision lists

Consider the rules produced for a particular class, that is, the algorithm in Figure 4.8 with the outer loop removed. It seems clear from the way that these rules are produced that they are intended to be interpreted in order, that is, as a decision list, testing the rules in turn until one applies and then using that. This is because the instances covered by a new rule are removed from the instance set as soon as the rule is completed (in the third line from the end of the code in Figure 4.8): thus subsequent rules are designed for instances that are *not* covered by the rule. However, although it appears that we are supposed to check the rules in turn, we do not have to do so. Consider that any subsequent rules generated for this class will have the same effect—they all predict the same class. This means that it does not matter what order they are executed in: either a rule will

```
For each class C
  Initialize E to the instance set
  While E contains instances in class C
    Create a rule R with an empty left-hand side that predicts class C
    Until R is perfect (or there are no more attributes to use) do
      For each attribute A not mentioned in R, and each value v,
        Consider adding the condition A=v to the LHS of R
        Select A and v to maximize the accuracy p/t
          (break ties by choosing the condition with the largest p)
      Add A=v to R
    Remove the instances covered by R from E
```

Figure 4.8 Pseudocode for a basic rule learner.

be found that covers this instance, in which case the class in question is predicted, or no such rule is found, in which case the class is not predicted.

Now return to the overall algorithm. Each class is considered in turn, and rules are generated that distinguish instances in that class from the others. No ordering is implied between the rules for one class and those for another. Consequently, the rules that are produced can be executed independent of order.

As described in Section 3.3, order-independent rules seem to provide more modularity by each acting as independent nuggets of "knowledge," but they suffer from the disadvantage that it is not clear what to do when conflicting rules apply. With rules generated in this way, a test example may receive multiple classifications, that is, rules that apply to different classes may accept it. Other test examples may receive no classification at all. A simple strategy to force a decision in these ambiguous cases is to choose, from the classifications that are predicted, the one with the most training examples or, if no classification is predicted, to choose the category with the most training examples overall. These difficulties do not occur with decision lists because they are meant to be interpreted in order and execution stops as soon as one rule applies: the addition of a default rule at the end ensures that any test instance receives a classification. It is possible to generate good decision lists for the multiclass case using a slightly different method, as we shall see in Section 6.2.

Methods such as PRISM can be described as *separate-and-conquer* algorithms: you identify a rule that covers many instances in the class (and excludes ones not in the class), separate out the covered instances because they are already taken care of by the rule, and continue the process on those that are left. This contrasts nicely with the divide-and-conquer approach of decision trees. The separate step greatly increases the efficiency of the method because the instance set continually shrinks as the operation proceeds.

4.5 Mining association rules

Association rules are like classification rules. You could find them in the same way, by executing a divide-and-conquer rule-induction procedure for each possible expression that could occur on the right-hand side of the rule. But not only might any attribute occur on the right-hand side with any possible value; a single association rule often predicts the value of more than one attribute. To find such rules, you would have to execute the rule-induction procedure once for every possible *combination* of attributes, with every possible combination of values, on the right-hand side. That would result in an enormous number of association rules, which would then have to be pruned down on the basis of their *coverage* (the number of instances that they predict correctly) and their *accuracy* (the same number expressed as a proportion of the number of instances to which the rule applies). This approach is quite infeasible. (Note that, as we mentioned in Section 3.4, what we are calling *coverage* is often called *support* and what we are calling *accuracy* is often called *confidence.*)

Instead, we capitalize on the fact that we are only interested in association rules with high coverage. We ignore, for the moment, the distinction between the left- and right-hand sides of a rule and seek combinations of attribute–value pairs that have a prespecified minimum coverage. These are called *item sets:* an attribute–value pair is an *item.* The terminology derives from market basket analysis, in which the items are articles in your shopping cart and the supermarket manager is looking for associations among these purchases.

Item sets

The first column of Table 4.10 shows the individual items for the weather data of Table 1.2, with the number of times each item appears in the dataset given at the right. These are the one-item sets. The next step is to generate the two-item sets by making pairs of one-item ones. Of course, there is no point in generating a set containing two different values of the same attribute (such as *outlook* = *sunny* and *outlook* = *overcast*), because that cannot occur in any actual instance.

Assume that we seek association rules with minimum coverage 2: thus we discard any item sets that cover fewer than two instances. This leaves 47 two-item sets, some of which are shown in the second column along with the number of times they appear. The next step is to generate the three-item sets, of which 39 have a coverage of 2 or greater. There are 6 four-item sets, and no five-item sets—for this data, a five-item set with coverage 2 or greater could only correspond to a repeated instance. The first row of the table, for example, shows that there are five days when *outlook* = *sunny,* two of which have *temperature* = *mild,* and, in fact, on both of those days *humidity* = *high* and *play* = *no* as well.

Association rules

Shortly we will explain how to generate these item sets efficiently. But first let us finish the story. Once all item sets with the required coverage have been generated, the next step is to turn each into a rule, or set of rules, with at least the specified minimum accuracy. Some item sets will produce more than one rule; others will produce none. For example, there is one three-item set with a coverage of 4 (row 38 of Table 4.10):

```
humidity = normal, windy = false, play = yes
```

This set leads to seven potential rules:

Table 4.10 Item sets for the weather data with coverage 2 or greater.

	One-item sets	Two-item sets	Three-item sets	Four-item sets

1	outlook = sunny (5)	outlook = sunny
		temperature = mild (2)
			outlook = sunny
			temperature = hot
			humidity = high (2)
				outlook = sunny
				temperature = hot
				humidity = high
				play = no (2)

2	outlook = overcast (4)	outlook = sunny
		temperature = hot (2)
			outlook = sunny
			temperature = hot
			play = no (2)
				outlook = sunny
				humidity = high
				windy = false
				play = no (2)

3	outlook = rainy (5)	outlook = sunny
		humidity = normal (2)
			outlook = sunny
			humidity = normal
			play = yes (2)
				outlook = overcast
				temperature = hot
				windy = false
				play = yes (2)

4	temperature = cool (4)	outlook = sunny
		humidity = high (3)
			outlook = sunny
			humidity = high
			windy = false (2)
				outlook = rainy
				temperature = mild
				windy = false
				play = yes (2)

5	temperature = mild (6)	outlook = sunny
		windy = true (2)
			outlook = sunny
			humidity = high
			play = no (3)
				outlook = rainy
				humidity = normal
				windy = false
				play = yes (2)

6	temperature = hot (4)	outlook = sunny
		windy = false (3)
			outlook = sunny
			windy = false
			play = no (2)
				temperature = cool
				humidity = normal
				windy = false
				play = yes (2)

7	humidity = normal (7)	outlook = sunny
		play = yes (2)
			outlook = overcast
			temperature = hot
			windy = false (2)	

8	humidity = high (7)	outlook = sunny
		play = no (3)
			outlook = overcast
			temperature = hot
			play = yes (2)	

9	windy = true (6)	outlook = overcast
		temperature = hot (2)
			outlook = overcast
			humidity = normal
			play = yes (2)	

10	windy = false (8)	outlook = overcast
		humidity = normal (2)
			outlook = overcast
			humidity = high
			play = yes (2)	

11	play = yes (9)	outlook = overcast
		humidity = high (2)
			outlook = overcast
			windy = true
			play = yes (2)	

12	play = no (5)	outlook = overcast
		windy = true (2)
			outlook = overcast
			windy = false
			play = yes (2)	

13		outlook = overcast
		windy = false (2)
			outlook = rainy
			temperature = cool
			humidity = normal (2)	

Table 4.10 **(continued)**

	One-item sets	Two-item sets	Three-item sets	Four-item sets
. . .		. . .	. . .	

38		humidity = normal
		windy = false (4)
			humidity = normal
			windy = false
			play = yes (4)	

39		humidity = normal
		play = yes (6)
			humidity = high
			windy = false
			play = no (2)	

40		humidity = high
		windy = true (3)		

. . .		. . .		

47		windy = false
		play = no (2)		

```
If humidity = normal and windy = false then play = yes         4/4
If humidity = normal and play = yes then windy = false         4/6
If windy = false and play = yes then humidity = normal         4/6
If humidity = normal then windy = false and play = yes         4/7
If windy = false then humidity = normal and play = yes         4/8
If play = yes then humidity = normal and windy = false         4/9
If - then humidity = normal and windy = false and play = yes   4/12
```

The figures at the right show the number of instances for which all three conditions are true—that is, the coverage—divided by the number of instances for which the conditions in the antecedent are true. Interpreted as a fraction, they represent the proportion of instances on which the rule is correct—that is, its accuracy. Assuming that the minimum specified accuracy is 100%, only the first of these rules will make it into the final rule set. The denominators of the fractions are readily obtained by looking up the antecedent expression in Table 4.10 (though some are not shown in the Table). The final rule above has no conditions in the antecedent, and its denominator is the total number of instances in the dataset.

Table 4.11 shows the final rule set for the weather data, with minimum coverage 2 and minimum accuracy 100%, sorted by coverage. There are 58 rules, 3 with coverage 4, 5 with coverage 3, and 50 with coverage 2. Only 7 have two conditions in the consequent, and none has more than two. The first rule comes from the item set described previously. Sometimes several rules arise from the same item set. For example, rules 9, 10, and 11 all arise from the four-item set in row 6 of Table 4.10:

```
temperature = cool, humidity = normal, windy = false, play = yes
```

Table 4.11 Association rules for the weather data.

	Association rule			Coverage	Accuracy
1	humidity = normal windy = false	⇒	play = yes	4	100%
2	temperature = cool	⇒	humidity = normal	4	100%
3	outlook = overcast	⇒	play = yes	4	100%
4	temperature = cool play = yes	⇒	humidity = normal	3	100%
5	outlook = rainy windy = false	⇒	play = yes	3	100%
6	outlook = rainy play = yes	⇒	windy = false	3	100%
7	outlook = sunny humidity = high	⇒	play = no	3	100%
8	outlook = sunny play = no	⇒	humidity = high	3	100%
9	temperature = cool windy = false	⇒	humidity = normal play = yes	2	100%
10	temperature = cool humidity = normal windy = false	⇒	play = yes	2	100%
11	temperature = cool windy = false play = yes	⇒	humidity = normal	2	100%
12	outlook = rainy humidity = normal windy = false	⇒	play = yes	2	100%
13	outlook = rainy humidity = normal play = yes	⇒	windy = false	2	100%
14	outlook = rainy temperature = mild windy = false	⇒	play = yes	2	100%
15	outlook = rainy temperature = mild play = yes	⇒	windy = false	2	100%
16	temperature = mild windy = false play = yes	⇒	outlook = rainy	2	100%
17	outlook = overcast temperature = hot	⇒	windy = false play = yes	2	100%
18	outlook = overcast windy = false	⇒	temperature = hot play = yes	2	100%
19	temperature = hot play = yes	⇒	outlook = overcast windy = false	2	100%
20	outlook = overcast temperature = hot windy = false	⇒	play = yes	2	100%
21	outlook = overcast temperature = hot play = yes	⇒	windy = false	2	100%
22	outlook = overcast windy = false play = yes	⇒	temperature = hot	2	100%
23	temperature = hot windy = false play = yes	⇒	outlook = overcast	2	100%
24	windy = false play = no	⇒	outlook = sunny humidity = high	2	100%
25	outlook = sunny humidity = high windy = false	⇒	play = no	2	100%
26	outlook = sunny windy = false play = no	⇒	humidity = high	2	100%
27	humidity = high windy = false play = no	⇒	outlook = sunny	2	100%
28	outlook = sunny temperature = hot	⇒	humidity = high play = no	2	100%
29	temperature = hot play = no	⇒	outlook = sunny humidity = high	2	100%
30	outlook = sunny temperature = hot humidity = high	⇒	play = no	2	100%
31	outlook = sunny temperature = hot play = no	⇒	humidity = high	2	100%
...	...			...	...
58	outlook = sunny temperature = hot	⇒	humidity = high	2	100%

which has coverage 2. Three subsets of this item set also have coverage 2:

```
temperature = cool, windy = false
temperature = cool, humidity = normal, windy = false
temperature = cool, windy = false, play = yes
```

and these lead to rules 9, 10, and 11, all of which are 100% accurate (on the training data).

Generating rules efficiently

We now consider in more detail an algorithm for producing association rules with specified minimum coverage and accuracy. There are two stages: generating item sets with the specified minimum coverage, and from each item set determining the rules that have the specified minimum accuracy.

The first stage proceeds by generating all one-item sets with the given minimum coverage (the first column of Table 4.10) and then using this to generate the two-item sets (second column), three-item sets (third column), and so on. Each operation involves a pass through the dataset to count the items in each set, and after the pass the surviving item sets are stored in a hash table—a standard data structure that allows elements stored in it to be found very quickly. From the one-item sets, candidate two-item sets are generated, and then a pass is made through the dataset, counting the coverage of each two-item set; at the end the candidate sets with less than minimum coverage are removed from the table. The candidate two-item sets are simply all of the one-item sets taken in pairs, because a two-item set cannot have the minimum coverage unless both its constituent one-item sets have minimum coverage, too. This applies in general: a three-item set can only have the minimum coverage if all three of its two-item subsets have minimum coverage as well, and similarly for four-item sets.

An example will help to explain how candidate item sets are generated. Suppose there are five three-item sets—(A B C), (A B D), (A C D), (A C E), and (B C D)—where, for example, A is a feature such as *outlook* = *sunny.* The union of the first two, (A B C D), is a candidate four-item set because its other three-item subsets (A C D) and (B C D) have greater than minimum coverage. If the three-item sets are sorted into lexical order, as they are in this list, then we need only consider pairs whose first two members are the same. For example, we do not consider (A C D) and (B C D) because (A B C D) can also be generated from (A B C) and (A B D), and if these two are not candidate three-item sets then (A B C D) cannot be a candidate four-item set. This leaves the pairs (A B C) and (A B D), which we have already explained, and (A C D) and (A C E). This second pair leads to the set (A C D E) whose three-item subsets do not all have the minimum coverage, so it is discarded. The hash table assists with this check: we simply remove each item from the set in turn and check that the remaining three-item set is indeed present in the hash table. Thus in this example there is only one candidate four-item set, (A B C D). Whether or not it actually has minimum coverage can only be determined by checking the instances in the dataset.

The second stage of the procedure takes each item set and generates rules from it, checking that they have the specified minimum accuracy. If only rules with a single test on the right-hand side were sought, it would be simply a matter of considering each condition in turn as the consequent of the rule, deleting it from the item set, and dividing the coverage of the entire item set by the coverage of the resulting subset—obtained from the hash table—to yield the accuracy of the corresponding rule. Given that we are also interested in association rules with multiple tests in the consequent, it looks like we have to evaluate the effect of placing each *subset* of the item set on the right-hand side, leaving the remainder of the set as the antecedent.

This brute-force method will be excessively computation intensive unless item sets are small, because the number of possible subsets grows exponentially with the size of the item set. However, there is a better way. We observed when describing association rules in Section 3.4 that if the double-consequent rule

```
If windy = false and play = no then outlook = sunny
                                    and humidity = high
```

holds with a given minimum coverage and accuracy, then both single-consequent rules formed from the same item set must also hold:

```
If humidity = high and windy = false and play = no
   then outlook = sunny
If outlook = sunny and windy = false and play = no
   then humidity = high
```

Conversely, if one or other of the single-consequent rules does not hold, there is no point in considering the double-consequent one. This gives a way of building up from single-consequent rules to candidate double-consequent ones, from double-consequent rules to candidate triple-consequent ones, and so on. Of course, each candidate rule must be checked against the hash table to see if it really does have more than the specified minimum accuracy. But this generally involves checking far fewer rules than the brute force method. It is interesting that this way of building up candidate $(n + 1)$-consequent rules from actual n-consequent ones is really just the same as building up candidate $(n + 1)$-item sets from actual n-item sets, described earlier.

Discussion

Association rules are often sought for very large datasets, and efficient algorithms are highly valued. The method described previously makes one pass through the dataset for each different size of item set. Sometimes the dataset is too large to read in to main memory and must be kept on disk; then it may be worth reducing the number of passes by checking item sets of two consecutive sizes in one go. For example, once sets with two items have been generated, all sets of three items could be generated from them before going through the instance set to count the actual number of items in the sets. More three-item sets than necessary would be considered, but the number of passes through the entire dataset would be reduced.

In practice, the amount of computation needed to generate association rules depends critically on the minimum coverage specified. The accuracy has less influence because it does not affect the number of passes that we must make through the dataset. In many situations we will want to obtain a certain number of rules—say 50—with the greatest possible coverage at a prespecified minimum accuracy level. One way to do this is to begin by specifying the coverage to be rather high and to then successively reduce it, reexecuting the entire rule-finding algorithm for each coverage value and repeating this until the desired number of rules has been generated.

The tabular input format that we use throughout this book, and in particular a standard ARFF file based on it, is very inefficient for many association-rule problems. Association rules are often used when attributes are binary—either present or absent—and most of the attribute values associated with a given instance are absent. This is a case for the sparse data representation described in Section 2.4; the same algorithm for finding association rules applies.

4.6 Linear models

The methods we have been looking at for decision trees and rules work most naturally with nominal attributes. They can be extended to numeric attributes either by incorporating numeric-value tests directly into the decision tree or rule induction scheme, or by prediscretizing numeric attributes into nominal ones. We will see how in Chapters 6 and 7, respectively. However, there are methods that work most naturally with numeric attributes. We look at simple ones here, ones that form components of more complex learning methods, which we will examine later.

Numeric prediction: Linear regression

When the outcome, or class, is numeric, and all the attributes are numeric, linear regression is a natural technique to consider. This is a staple method in statistics. The idea is to express the class as a linear combination of the attributes, with predetermined weights:

$$x = w_0 + w_1a_1 + w_2a_2 + \ldots + w_ka_k$$

where x is the class; $a_1, a_2, \ldots, a_k$ are the attribute values; and $w_0, w_1, \ldots, w_k$ are weights.

The weights are calculated from the training data. Here the notation gets a little heavy, because we need a way of expressing the attribute values for each training instance. The first instance will have a class, say $x^{(1)}$, and attribute values $a_1^{(1)}, a_2^{(1)}, \ldots, a_k^{(1)}$, where the superscript denotes that it is the first example. Moreover, it is notationally convenient to assume an extra attribute a_0 whose value is always 1.

The predicted value for the first instance's class can be written as

$$w_0a_0^{(1)} + w_1a_1^{(1)} + w_2a_2^{(1)} + \ldots + w_ka_k^{(1)} = \sum_{j=0}^{k} w_ja_j^{(1)}.$$

This is the predicted, not the actual, value for the first instance's class. Of interest is the difference between the predicted and the actual values. The method of linear regression is to choose the coefficients w_j—there are $k + 1$ of them—to minimize the sum of the squares of these differences over all the training instances. Suppose there are n training instances; denote the ith one with a superscript (i). Then the sum of the squares of the differences is

$$\sum_{i=1}^{n}\left(x^{(i)} - \sum_{j=0}^{k} w_ja_j^{(i)}\right)^2$$

where the expression inside the parentheses is the difference between the ith instance's actual class and its predicted class. This sum of squares is what we have to minimize by choosing the coefficients appropriately.

This is all starting to look rather formidable. However, the minimization technique is straightforward if you have the appropriate math background. Suffice it to say that given enough examples—roughly speaking, more examples than attributes—choosing weights to minimize the sum of the squared differences is really not difficult. It does involve a matrix inversion operation, but this is readily available as prepackaged software.

Once the math has been accomplished, the result is a set of numeric weights, based on the training data, which we can use to predict the class of new instances. We saw an example of this when looking at the CPU performance data, and the actual numeric weights are given in Figure 3.7(a). This formula can be used to predict the CPU performance of new test instances.

Linear regression is an excellent, simple method for numeric prediction, and it has been widely used in statistical applications for decades. Of course, linear models suffer from the disadvantage of, well, linearity. If the data exhibits a nonlinear dependency, the best-fitting straight line will be found, where "best" is interpreted as the least mean-squared difference. This line may not fit very well.

However, linear models serve well as building blocks for more complex learning methods.

Linear classification: Logistic regression

Linear regression can easily be used for classification in domains with numeric attributes. Indeed, we can use *any* regression technique, whether linear or nonlinear, for classification. The trick is to perform a regression for each class, setting the output equal to one for training instances that belong to the class and zero for those that do not. The result is a linear expression for the class. Then, given a test example of unknown class, calculate the value of each linear expression and choose the one that is largest. This method is sometimes called *multiresponse linear regression.*

One way of looking at multiresponse linear regression is to imagine that it approximates a numeric *membership function* for each class. The membership function is 1 for instances that belong to that class and 0 for other instances. Given a new instance we calculate its membership for each class and select the biggest.

Multiresponse linear regression often yields good results in practice. However, it has two drawbacks. First, the membership values it produces are not proper probabilities because they can fall outside the range 0 to 1. Second, least-squares regression assumes that the errors are not only statistically independent, but are also normally distributed with the same standard deviation, an assumption that is blatantly violated when the method is applied to classification problems because the observations only ever take on the values 0 and 1.

A related statistical technique called *logistic regression* does not suffer from these problems. Instead of approximating the 0 and 1 values directly, thereby risking illegitimate probability values when the target is overshot, logistic regression builds a linear model based on a transformed target variable.

Suppose first that there are only two classes. Logistic regression replaces the original target variable

$$\Pr[1|a_1,a_2,\ldots,a_k],$$

which cannot be approximated accurately using a linear function, with

$$\log(\Pr[1|a_1,a_2,\ldots,a_k])/(1-\Pr[1|a_1,a_2,\ldots,a_k]).$$

The resulting values are no longer constrained to the interval from 0 to 1 but can lie anywhere between negative infinity and positive infinity. Figure 4.9(a) plots the transformation function, which is often called the *logit transformation.*

The transformed variable is approximated using a linear function just like the ones generated by linear regression. The resulting model is

$$\Pr[1|a_1,a_2,\ldots,a_k]=1/(1+\exp(-w_0-w_1a_1-\ldots-w_ka_k)),$$

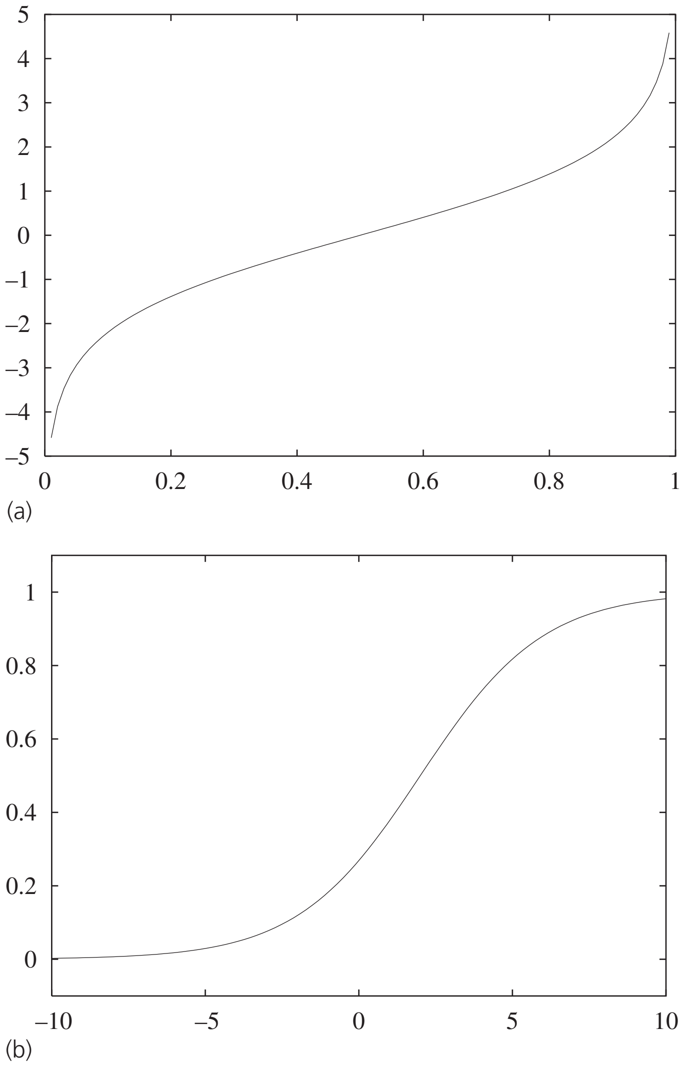


Figure 4.9 Logistic regression: (a) the logit transform and (b) an example logistic regression function.

with weights w. Figure 4.9(b) shows an example of this function in one dimension, with two weights $w_0 = 0.5$ and $w_1 = 1$.

Just as in linear regression, weights must be found that fit the training data well. Linear regression measures the goodness of fit using the squared error. In logistic regression the *log-likelihood* of the model is used instead. This is given by

$$\sum_{i=1}^{n}(1-x^{(i)})\log(1-\Pr[1|a_1^{(i)},a_2^{(i)},\ldots,a_k^{(i)}])+x^{(i)}\log(\Pr[1|a_1^{(i)},a_2^{(i)},\ldots,a_k^{(i)}])$$

where the $x^{(i)}$ are either zero or one.

The weights w_i need to be chosen to maximize the log-likelihood. There are several methods for solving this maximization problem. A simple one is to iteratively solve a sequence of weighted least-squares regression problems until the log-likelihood converges to a maximum, which usually happens in a few iterations.

To generalize logistic regression to several classes, one possibility is to proceed in the way described previously for multiresponse linear regression by performing logistic regression independently for each class. Unfortunately, the resulting probability estimates will not sum to one. To obtain proper probabilities it is necessary to couple the individual models for each class. This yields a joint optimization problem, and there are efficient solution methods for this.

A conceptually simpler, and very general, way to address multiclass problems is known as *pairwise classification.* Here a classifier is built for every pair of classes, using only the instances from these two classes. The output on an unknown test example is based on which class receives the most votes. This method generally yields accurate results in terms of classification error. It can also be used to produce probability estimates by applying a method called *pairwise coupling,* which calibrates the individual probability estimates from the different classifiers.

If there are k classes, pairwise classification builds a total of $k(k-1)/2$ classifiers. Although this sounds unnecessarily computation intensive, it is not. In fact, if the classes are evenly populated pairwise classification is at least as fast as any other multiclass method. The reason is that each of the pairwise learning problem only involves instances pertaining to the two classes under consideration. If n instances are divided evenly among k classes, this amounts to $2n/k$ instances per problem. Suppose the learning algorithm for a two-class problem with n instances takes time proportional to n seconds to execute. Then the run time for pairwise classification is proportional to $k(k-1)/2 \times 2n/k$ seconds, which is $(k-1)n$. In other words, the method scales linearly with the number of classes. If the learning algorithm takes more time—say proportional to n^2—the advantage of the pairwise approach becomes even more pronounced.

The use of linear functions for classification can easily be visualized in instance space. The decision boundary for two-class logistic regression lies where the prediction probability is 0.5, that is:

$$\Pr[1|a_1,a_2,\ldots,a_k]=1/(1+\exp(-w_0-w_1a_1-\ldots-w_ka_k))=0.5.$$

This occurs when

$$-w_0 - w_1a_1 - \ldots - w_ka_k = 0.$$

Because this is a linear equality in the attribute values, the boundary is a linear plane, or *hyperplane,* in instance space. It is easy to visualize sets of points that cannot be separated by a single hyperplane, and these cannot be discriminated correctly by logistic regression.

Multiresponse linear regression suffers from the same problem. Each class receives a weight vector calculated from the training data. Focus for the moment on a particular pair of classes. Suppose the weight vector for class 1 is

$$w_0^{(1)} + w_1^{(1)}a_1 + w_2^{(1)}a_2 + \ldots + w_k^{(1)}a_k$$

and the same for class 2 with appropriate superscripts. Then, an instance will be assigned to class 1 rather than class 2 if

$$w_0^{(1)} + w_1^{(1)}a_1 + \ldots + w_k^{(1)}a_k > w_0^{(2)} + w_1^{(2)}a_1 + \ldots + w_k^{(2)}a_k$$

In other words, it will be assigned to class 1 if

$$\left(w_0^{(1)} - w_0^{(2)}\right) + \left(w_1^{(1)} - w_1^{(2)}\right)a_1 + \ldots + \left(w_k^{(1)} - w_k^{(2)}\right)a_k > 0.$$

This is a linear inequality in the attribute values, so the boundary between each pair of classes is a hyperplane. The same holds true when performing pairwise classification. The only difference is that the boundary between two classes is governed by the training instances in those classes and is not influenced by the other classes.

Linear classification using the perceptron

Logistic regression attempts to produce accurate probability estimates by maximizing the probability of the training data. Of course, accurate probability estimates lead to accurate classifications. However, it is not necessary to perform probability estimation if the sole purpose of the model is to predict class labels. A different approach is to learn a hyperplane that separates the instances pertaining to the different classes—let's assume that there are only two of them. If the data can be separated perfectly into two groups using a hyperplane, it is said to be *linearly separable.* It turns out that if the data is linearly separable, there is a very simple algorithm for finding a separating hyperplane.

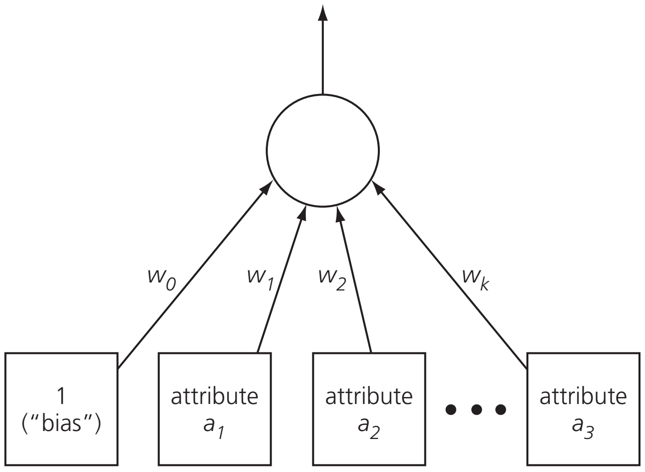
The algorithm is called the *perceptron learning rule.* Before looking at it in detail, let's examine the equation for a hyperplane again:

$$w_0a_0 + w_1a_1 + w_2a_2 + \ldots + w_ka_k = 0.$$

Here, a_1, a_2, . . ., a_k are the attribute values, and w_0, w_1, . . ., w_k are the weights that define the hyperplane. We will assume that each training instance a_1, a_2, . . . is extended by an additional attribute a_0 that always has the value 1 (as we did in the case of linear regression). This extension, which is called the *bias,* just

```
Set all weights to zero
Until all instances in the training data are classified correctly
  For each instance I in the training data
    If I is classified incorrectly by the perceptron
      If I belongs to the first class add it to the weight vector
      else subtract it from the weight vector
```

(a)



(b)

Figure 4.10 The perceptron: (a) learning rule and (b) representation as a neural network.

means that we don't have to include an additional constant element in the sum. If the sum is greater than zero, we will predict the first class; otherwise, we will predict the second class. We want to find values for the weights so that the training data is correctly classified by the hyperplane.

Figure 4.10(a) gives the perceptron learning rule for finding a separating hyperplane. The algorithm iterates until a perfect solution has been found, but it will only work properly if a separating hyperplane exists, that is, if the data is linearly separable. Each iteration goes through all the training instances. If a misclassified instance is encountered, the parameters of the hyperplane are changed so that the misclassified instance moves closer to the hyperplane or maybe even across the hyperplane onto the correct side. If the instance belongs to the first class, this is done by adding its attribute values to the weight vector; otherwise, they are subtracted from it.

To see why this works, consider the situation after an instance *a* pertaining to the first class has been added:

$$(w_0 + a_0)a_0 + (w_1 + a_1)a_1 + (w_2 + a_2)a_2 + \ldots + (w_k + a_k)a_k.$$

This means the output for *a* has increased by

$$a_0 \times a_0 + a_1 \times a_1 + a_2 \times a_2 + \ldots + a_k \times a_k.$$

This number is always positive. Thus the hyperplane has moved in the correct direction for classifying instance *a* as positive. Conversely, if an instance belonging to the second class is misclassified, the output for that instance decreases after the modification, again moving the hyperplane to the correct direction.

These corrections are incremental and can interfere with earlier updates. However, it can be shown that the algorithm converges in a finite number of iterations if the data is linearly separable. Of course, if the data is not linearly separable, the algorithm will not terminate, so an upper bound needs to be imposed on the number of iterations when this method is applied in practice.

The resulting hyperplane is called a *perceptron,* and it's the grandfather of neural networks (we return to neural networks in Section 6.3). Figure 4.10(b) represents the perceptron as a graph with nodes and weighted edges, imaginatively termed a "network" of "neurons." There are two layers of nodes: input and output. The input layer has one node for every attribute, plus an extra node that is always set to one. The output layer consists of just one node. Every node in the input layer is connected to the output layer. The connections are weighted, and the weights are those numbers found by the perceptron learning rule.

When an instance is presented to the perceptron, its attribute values serve to "activate" the input layer. They are multiplied by the weights and summed up at the output node. If the weighted sum is greater than 0 the output signal is 1, representing the first class; otherwise, it is −1, representing the second.

Linear classification using Winnow

The perceptron algorithm is not the only method that is guaranteed to find a separating hyperplane for a linearly separable problem. For datasets with binary attributes there is an alternative known as *Winnow,* shown in Figure 4.11(a). The structure of the two algorithms is very similar. Like the perceptron, Winnow only updates the weight vector when a misclassified instance is encountered—it is *mistake driven.*

The two methods differ in how the weights are updated. The perceptron rule employs an additive mechanism that alters the weight vector by adding (or subtracting) the instance's attribute vector. Winnow employs multiplicative updates and alters weights individually by multiplying them by the user-specified parameter α (or its inverse). The attribute values a_i are either 0 or 1 because we

```
While some instances are misclassified
      for every instance a
        classify a using the current weights
        if the predicted class is incorrect
          if a belongs to the first class
            for each $a_i$ that is 1, multiply $w_i$ by $\alpha$
            (if $a_i$ is 0, leave $w_i$ unchanged)
          otherwise
            for each $a_i$ that is 1, divide $w_i$ by $\alpha$
            (if $a_i$ is 0, leave $w_i$ unchanged)
```

(a)

```
While some instances are misclassified
  for every instance a
    classify a using the current weights
    if the predicted class is incorrect
      if a belongs to the first class
        for each $a_i$ that is 1,
          multiply $w_i^+$ by $\alpha$
          divide $w_i^-$ by $\alpha$
        (if $a_i$ is 0, leave $w_i^+$ and $w_i^-$ unchanged)
      otherwise for
        for each $a_i$ that is 1,
          multiply $w_i^-$ by $\alpha$
          divide $w_i^+$ by $\alpha$
        (if $a_i$ is 0, leave $w_i^+$ and $w_i^-$ unchanged)
```

(b)

Figure 4.11 The Winnow algorithm: (a) the unbalanced version and (b) the balanced version.

are working with binary data. Weights are unchanged if the attribute value is 0, because then they do not participate in the decision. Otherwise, the multiplier is α if that attribute helps to make a correct decision and $1/\alpha$ if it does not.

Another difference is that the threshold in the linear function is also a user-specified parameter. We call this threshold θ and classify an instance as belonging to class 1 if and only if

$$w_0a_0 + w_1a_1 + w_2a_2 + \ldots + w_ka_k > \theta.$$

The multiplier α needs to be greater than one. The w_i are set to a constant at the start.

The algorithm we have described doesn't allow negative weights, which—depending on the domain—can be a drawback. However, there is a version, called *Balanced Winnow,* which does allow them. This version maintains two weight vectors, one for each class. An instance is classified as belonging to class 1 if:

$$\left(w_0^+ - w_0^-\right)a_0 + \left(w_1^+ - w_1^-\right)a_1 + \ldots + \left(w_k^+ - w_k^-\right)a_k > \theta$$

Figure 4.11(b) shows the balanced algorithm.

Winnow is very effective in homing in on the relevant features in a dataset—therefore it is called an *attribute-efficient* learner. That means that it may be a good candidate algorithm if a dataset has many (binary) features and most of them are irrelevant. Both winnow and the perceptron algorithm can be used in an online setting in which new instances arrive continuously, because they can incrementally update their hypotheses as new instances arrive.

4.7 Instance-based learning

In instance-based learning the training examples are stored verbatim, and a distance function is used to determine which member of the training set is closest to an unknown test instance. Once the nearest training instance has been located, its class is predicted for the test instance. The only remaining problem is defining the distance function, and that is not very difficult to do, particularly if the attributes are numeric.

The distance function

Although there are other possible choices, most instance-based learners use Euclidean distance. The distance between an instance with attribute values $a_1^{(1)}$, $a_2^{(1)}, \ldots, a_k^{(1)}$ (where k is the number of attributes) and one with values $a_1^{(2)}$, $a_2^{(2)}, \ldots, a_k^{(2)}$ is defined as

$$\sqrt{\left(a_1^{(1)} - a_1^{(2)}\right)^2 + \left(a_2^{(1)} - a_2^{(2)}\right)^2 + \ldots + \left(a_k^{(1)} - a_k^{(2)}\right)^2}.$$

When comparing distances it is not necessary to perform the square root operation; the sums of squares can be compared directly. One alternative to the Euclidean distance is the Manhattan or city-block metric, where the difference between attribute values is not squared but just added up (after taking the absolute value). Others are obtained by taking powers higher than the square. Higher powers increase the influence of large differences at the expense of small differences. Generally, the Euclidean distance represents a good compromise. Other distance metrics may be more appropriate in special circumstances. The key is to think of actual instances and what it means for them to be separated by a certain distance—what would twice that distance mean, for example?

Different attributes are measured on different scales, so if the Euclidean distance formula were used directly, the effects of some attributes might be completely dwarfed by others that had larger scales of measurement. Consequently, it is usual to normalize all attribute values to lie between 0 and 1, by calculating

$$a_i = \frac{v_i - \min v_i}{\max v_i - \min v_i}$$

where v_i is the actual value of attribute *i*, and the maximum and minimum are taken over all instances in the training set.

These formulae implicitly assume numeric attributes. Here, the difference between two values is just the numerical difference between them, and it is this difference that is squared and added to yield the distance function. For nominal attributes that take on values that are symbolic rather than numeric, the difference between two values that are not the same is often taken to be one, whereas if the values are the same the difference is zero. No scaling is required in this case because only the values 0 and 1 are used.

A common policy for handling missing values is as follows. For nominal attributes, assume that a missing feature is maximally different from any other feature value. Thus if either or both values are missing, or if the values are different, the difference between them is taken as one; the difference is zero only if they are not missing and both are the same. For numeric attributes, the difference between two missing values is also taken as one. However, if just one value is missing, the difference is often taken as either the (normalized) size of the other value or one minus that size, whichever is larger. This means that if values are missing, the difference is as large as it can possibly be.

Finding nearest neighbors efficiently

Although instance-based learning is simple and effective, it is often slow. The obvious way to find which member of the training set is closest to an unknown test instance is to calculate the distance from every member of the training set

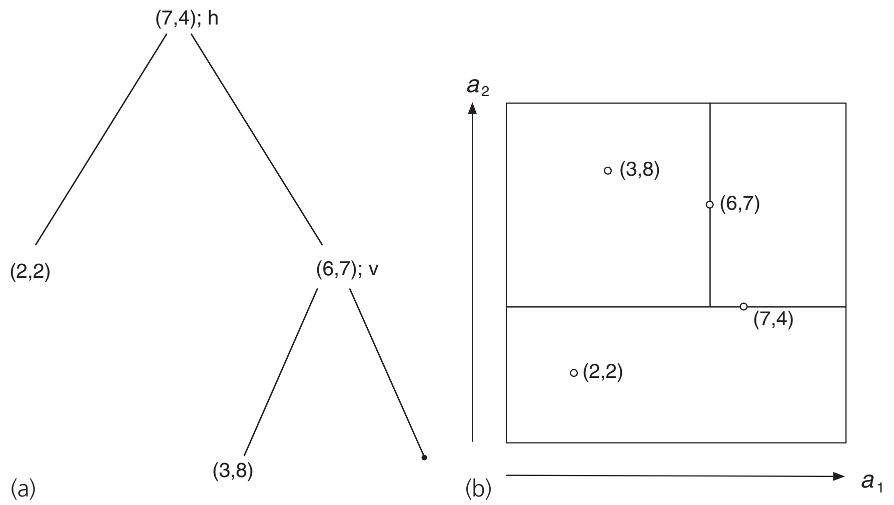


Figure 4.12 A *k*D-tree for four training instances: (a) the tree and (b) instances and splits.

and select the smallest. This procedure is linear in the number of training instances: in other words, the time it takes to make a single prediction is proportional to the number of training instances. Processing an entire test set takes time proportional to the product of the number of instances in the training and test sets.

Nearest neighbors can be found more efficiently by representing the training set as a tree, although it is not quite obvious how. One suitable structure is a *kD-tree.* This is a binary tree that divides the input space with a hyperplane and then splits each partition again, recursively. All splits are made parallel to one of the axes, either vertically or horizontally, in the two-dimensional case. The data structure is called a *kD-tree* because it stores a set of points in *k*-dimensional space, *k* being the number of attributes.

Figure 4.12(a) gives a small example with $k = 2$, and Figure 4.12(b) shows the four training instances it represents, along with the hyperplanes that constitute the tree. Note that these hyperplanes are *not* decision boundaries: decisions are made on a nearest-neighbor basis as explained later. The first split is horizontal (*h*), through the point (7,4)—this is the tree's root. The left branch is not split further: it contains the single point (2,2), which is a leaf of the tree. The right branch is split vertically (*v*) at the point (6,7). Its left child is empty, and its right child contains the point (3,8). As this example illustrates, each region contains just one point—or, perhaps, no points. Sibling branches of the tree—for example, the two daughters of the root in Figure 4.12(a)—are not necessarily developed to the same depth. Every point in the training set corresponds to a single node, and up to half are leaf nodes.

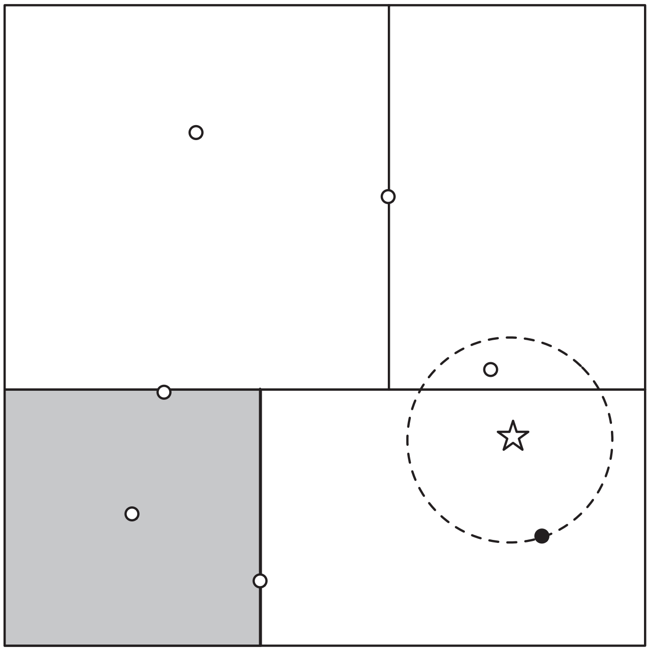


Figure 4.13 Using a *k*D-tree to find the nearest neighbor of the star.

How do you build a *k*D-tree from a dataset? Can it be updated efficiently as new training examples are added? And how does it speed up nearest-neighbor calculations? We tackle the last question first.

To locate the nearest neighbor of a given target point, follow the tree down from its root to locate the region containing the target. Figure 4.13 shows a space like that of Figure 4.12(b) but with a few more instances and an extra boundary. The target, which is not one of the instances in the tree, is marked by a star. The leaf node of the region containing the target is colored black. This is not necessarily the target's closest neighbor, as this example illustrates, but it is a good first approximation. In particular, any nearer neighbor must lie closer—within the dashed circle in Figure 4.13. To determine whether one exists, first check whether it is possible for a closer neighbor to lie within the node's sibling. The black node's sibling is shaded in Figure 4.13, and the circle does not intersect it, so the sibling cannot contain a closer neighbor. Then back up to the parent node and check *its* sibling—which here covers everything above the horizontal line. In this case it *must* be explored, because the area it covers intersects with the best circle so far. To explore it, find its daughters (the original point's two aunts), check whether they intersect the circle (the left one does not, but the right one does), and descend to see whether it contains a closer point (it does).

In a typical case, this algorithm is far faster than examining all points to find the nearest neighbor. The work involved in finding the initial approximate nearest neighbor—the black point in Figure 4.13—depends on the depth of the tree, given by the logarithm of the number of nodes, $\log_2 n$. The amount of work involved in backtracking to check whether this really is the nearest neighbor depends a bit on the tree, and on how good the initial approximation is. But for a well-constructed tree whose nodes are approximately square, rather than long skinny rectangles, it can also be shown to be logarithmic in the number of nodes.

How do you build a good tree for a set of training examples? The problem boils down to selecting the first training instance to split at and the direction of the split. Once you can do that, apply the same method recursively to each child of the initial split to construct the entire tree.

To find a good direction for the split, calculate the variance of the data points along each axis individually, select the axis with the greatest variance, and create a splitting hyperplane perpendicular to it. To find a good place for the hyperplane, locate the median value along that axis and select the corresponding point. This makes the split perpendicular to the direction of greatest spread, with half the points lying on either side. This produces a well-balanced tree. To avoid long skinny regions it is best for successive splits to be along different axes, which is likely because the dimension of greatest variance is chosen at each stage. However, if the distribution of points is badly skewed, choosing the median value may generate several successive splits in the same direction, yielding long, skinny hyperrectangles. A better strategy is to calculate the mean rather than the median and use the point closest to that. The tree will not be perfectly balanced, but its regions will tend to be squarish because there is a greater chance that different directions will be chosen for successive splits.

An advantage of instance-based learning over most other machine learning methods is that new examples can be added to the training set at any time. To retain this advantage when using a *k*D-tree, we need to be able to update it incrementally with new data points. To do this, determine which leaf node contains the new point and find its hyperrectangle. If it is empty, simply place the new point there. Otherwise split the hyperrectangle, splitting it along its longest dimension to preserve squareness. This simple heuristic does not guarantee that adding a series of points will preserve the tree's balance, nor that the hyperrectangles will be well shaped for nearest-neighbor search. It is a good idea to rebuild the tree from scratch occasionally—for example, when its depth grows to twice the best possible depth.

As we have seen, *k*D-trees are good data structures for finding nearest neighbors efficiently. However, they are not perfect. Skewed datasets present a basic conflict between the desire for the tree to be perfectly balanced and the desire for regions to be squarish. More importantly, rectangles—even squares—are not the best shape to use anyway, because of their corners. If the dashed circle in

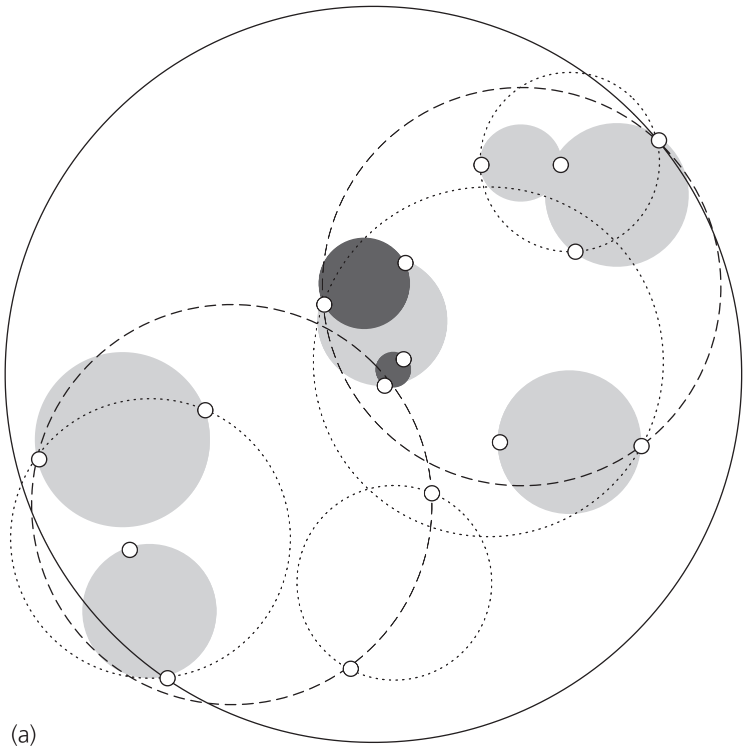
Figure 4.13 were any bigger, which it would be if the black instance were a little further from the target, it would intersect the lower right-hand corner of the rectangle at the top left and then that rectangle would have to be investigated, too—despite the fact that the training instances that define it are a long way from the corner in question. The corners of rectangular regions are awkward.

The solution? Use hyperspheres, not hyperrectangles. Neighboring spheres may overlap whereas rectangles can abut, but this is not a problem because the nearest-neighbor algorithm for *k*D-trees described previously does not depend on the regions being disjoint. A data structure called a *ball tree* defines *k*-dimensional hyperspheres ("balls") that cover the data points, and arranges them into a tree.

Figure 4.14(a) shows 16 training instances in two-dimensional space, overlaid by a pattern of overlapping circles, and Figure 4.14(b) shows a tree formed from these circles. Circles at different levels of the tree are indicated by different styles of dash, and the smaller circles are drawn in shades of gray. Each node of the tree represents a ball, and the node is dashed or shaded according to the same convention so that you can identify which level the balls are at. To help you understand the tree, numbers are placed on the nodes to show how many data points are deemed to be inside that ball. But be careful: this is not necessarily the same as the number of points falling within the spatial region that the ball represents. The regions at each level sometimes overlap, but points that fall into the overlap area are assigned to only one of the overlapping balls (the diagram does not show which one). Instead of the occupancy counts in Figure 4.14(b) the nodes of actual ball trees store the center and radius of their ball; leaf nodes record the points they contain as well.

To use a ball tree to find the nearest neighbor to a given target, start by traversing the tree from the top down to locate the leaf that contains the target and find the closest point to the target in that ball. This gives an upper bound for the target's distance from its nearest neighbor. Then, just as for the *k*D-tree, examine the sibling node. If the distance from the target to the sibling's center exceeds its radius plus the current upper bound, it cannot possibly contain a closer point; otherwise the sibling must be examined by descending the tree further. In Figure 4.15 the target is marked with a star and the black dot is its closest currently known neighbor. The entire contents of the gray ball can be ruled out: it cannot contain a closer point because its center is too far away. Proceed recursively back up the tree to its root, examining any ball that may possibly contain a point nearer than the current upper bound.

Ball trees are built from the top down, and as with *k*D-trees the basic problem is to find a good way of splitting a ball containing a set of data points into two. In practice you do not have to continue until the leaf balls contain just two points: you can stop earlier, once a predetermined minimum number is reached—and the same goes for *k*D-trees. Here is one possible splitting method.



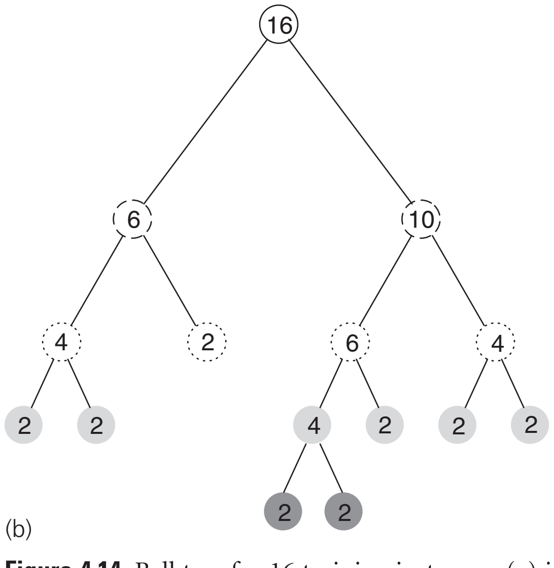


Figure 4.14 Ball tree for 16 training instances: (a) instances and balls and (b) the tree.

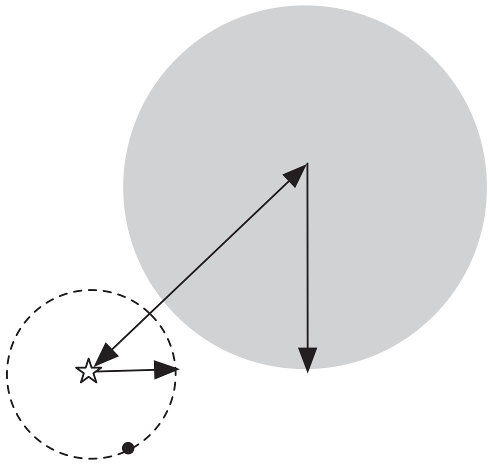


Figure 4.15 Ruling out an entire ball (gray) based on a target point (star) and its current nearest neighbor.

Choose the point in the ball that is farthest from its center, and then a second point that is farthest from the first one. Assign all data points in the ball to the closest one of these two cluster centers, then compute the centroid of each cluster and the minimum radius required for it to enclose all the data points it represents. This method has the merit that the cost of splitting a ball containing *n* points is only linear in *n*. There are more elaborate algorithms that produce tighter balls, but they require more computation. We will not describe sophisticated algorithms for constructing ball trees or updating them incrementally as new training instances are encountered.

Discussion

Nearest-neighbor instance-based learning is simple and often works very well. In the method described previously each attribute has exactly the same influence on the decision, just as it does in the Naïve Bayes method. Another problem is that the database can easily become corrupted by noisy exemplars. One solution is to adopt the *k*-nearest-neighbor strategy, where some fixed, small, number *k* of nearest neighbors—say five—are located and used together to determine the class of the test instance through a simple majority vote. (Note that we used *k* to denote the number of attributes earlier; this is a different, independent usage.) Another way of proofing the database against noise is to choose the exemplars that are added to it selectively and judiciously; improved procedures, described in Chapter 6, address these shortcomings.

The nearest-neighbor method originated many decades ago, and statisticians analyzed *k*-nearest-neighbor schemes in the early 1950s. If the number of training instances is large, it makes intuitive sense to use more than one nearest neighbor, but clearly this is dangerous if there are few instances. It can be shown that when k and the number n of instances both become infinite in such a way that $k/n \rightarrow 0$, the probability of error approaches the theoretical minimum for the dataset. The nearest-neighbor method was adopted as a classification method in the early 1960s and has been widely used in the field of pattern recognition for more than three decades.

Nearest-neighbor classification was notoriously slow until *k*D-trees began to be applied in the early 1990s, although the data structure itself was developed much earlier. In practice, these trees become inefficient when the dimension of the space increases and are only worthwhile when the number of attributes is small—up to 10. Ball trees were developed much more recently and are an instance of a more general structure sometimes called a *metric tree.* Sophisticated algorithms can create metric trees that deal successfully with thousands of dimensions.

Instead of storing all training instances, you can compress them into regions. A very simple technique, mentioned at the end of Section 4.1, is to just record the range of values observed in the training data for each attribute and category. Given a test instance, you work out which ranges the attribute values fall into and choose the category with the greatest number of correct ranges for that instance. A slightly more elaborate technique is to construct intervals for each attribute and use the training set to count the number of times each class occurs for each interval on each attribute. Numeric attributes can be discretized into intervals, and "intervals" consisting of a single point can be used for nominal ones. Then, given a test instance, you can determine which intervals it resides in and classify it by voting, a method called *voting feature intervals.* These methods are very approximate, but very fast, and can be useful for initial analysis of large datasets.

4.8 Clustering

Clustering techniques apply when there is no class to be predicted but rather when the instances are to be divided into natural groups. These clusters presumably reflect some mechanism at work in the domain from which instances are drawn, a mechanism that causes some instances to bear a stronger resemblance to each other than they do to the remaining instances. Clustering naturally requires different techniques to the classification and association learning methods we have considered so far.

As we saw in Section 3.9, there are different ways in which the result of clustering can be expressed. The groups that are identified may be exclusive so that any instance belongs in only one group. Or they may be overlapping so that an instance may fall into several groups. Or they may be probabilistic, whereby an instance belongs to each group with a certain probability. Or they may be hierarchical, such that there is a crude division of instances into groups at the top level, and each of these groups is refined further—perhaps all the way down to individual instances. Really, the choice among these possibilities should be dictated by the nature of the mechanisms that are thought to underlie the particular clustering phenomenon. However, because these mechanisms are rarely known—the very existence of clusters is, after all, something that we're trying to discover—and for pragmatic reasons too, the choice is usually dictated by the clustering tools that are available.

We will examine an algorithm that forms clusters in numeric domains, partitioning instances into disjoint clusters. Like the basic nearest-neighbor method of instance-based learning, it is a simple and straightforward technique that has been used for several decades. In Chapter 6 we examine newer clustering methods that perform incremental and probabilistic clustering.

Iterative distance-based clustering

The classic clustering technique is called *k-means.* First, you specify in advance how many clusters are being sought: this is the parameter *k*. Then *k* points are chosen at random as cluster centers. All instances are assigned to their closest cluster center according to the ordinary Euclidean distance metric. Next the centroid, or mean, of the instances in each cluster is calculated—this is the "means" part. These centroids are taken to be new center values for their respective clusters. Finally, the whole process is repeated with the new cluster centers. Iteration continues until the same points are assigned to each cluster in consecutive rounds, at which stage the cluster centers have stabilized and will remain the same forever.

This clustering method is simple and effective. It is easy to prove that choosing the cluster center to be the centroid minimizes the total squared distance from each of the cluster's points to its center. Once the iteration has stabilized, each point is assigned to its nearest cluster center, so the overall effect is to minimize the total squared distance from all points to their cluster centers. But the minimum is a local one; there is no guarantee that it is the global minimum. The final clusters are quite sensitive to the initial cluster centers. Completely different arrangements can arise from small changes in the initial random choice. In fact, this is true of all practical clustering techniques: it is almost always infeasible to find globally optimal clusters. To increase the chance of finding a global minimum people often run the algorithm several times with different initial choices and choose the best final result—the one with the smallest total squared distance.

It is easy to imagine situations in which *k*-means fails to find a good clustering. Consider four instances arranged at the vertices of a rectangle in two-dimensional space. There are two natural clusters, formed by grouping together the two vertices at either end of a short side. But suppose that the two initial cluster centers happen to fall at the midpoints of the *long* sides. This forms a stable configuration. The two clusters each contain the two instances at either end of a long side—no matter how great the difference between the long and the short sides.

Faster distance calculations

The *k*-means clustering algorithm usually requires several iterations, each involving finding the distance of *k* cluster centers from every instance to determine its cluster. There are simple approximations that speed this up considerably. For example, you can project the dataset and make cuts along selected axes, instead of using the arbitrary hyperplane divisions that are implied by choosing the nearest cluster center. But this inevitably compromises the quality of the resulting clusters.

Here's a better way of speeding things up. Finding the closest cluster center is not so different from finding nearest neighbors in instance-based learning. Can the same efficient solutions—*k*D-trees and ball trees—be used? Yes! Indeed they can be applied in an even more efficient way, because in each iteration of *k*-means all the data points are processed together, whereas in instance-based learning test instances are processed individually.

First, construct a *k*D-tree or ball tree for all the data points, which will remain static throughout the clustering procedure. Each iteration of *k*-means produces a set of cluster centers, and all data points must be examined and assigned to the nearest center. One way of processing the points is to descend the tree from the root until reaching a leaf and check each individual point in the leaf to find its closest cluster center. But it may be that the region represented by a higher interior node falls entirely within the domain of a single cluster center. In that case all the data points under that node can be processed in one blow!

The aim of the exercise, after all, is to find new positions for the cluster centers by calculating the centroid of the points they contain. The centroid can be calculated by keeping a running vector sum of the points in the cluster, and a count of how many there are so far. At the end, just divide one by the other to find the centroid. Suppose that with each node of the tree we store the vector sum of the points within that node and a count of the number of points. If the whole node falls within the ambit of a single cluster, the running totals for that cluster can be updated immediately. If not, look inside the node by proceeding recursively down the tree.

Figure 4.16 shows the same instances and ball tree as Figure 4.14, but with two cluster centers marked as black stars. Because all instances are assigned to the closest center, the space is divided in two by the thick line shown in Figure 4.16(a). Begin at the root of the tree in Figure 4.16(b), with initial values for the vector sum and counts for each cluster; all initial values are zero. Proceed recursively down the tree. When node A is reached, all points within it lie in cluster 1, so cluster 1's sum and count can be updated with the sum and count for node A, and we need descend no further. Recursing back to node B, its ball straddles the boundary between the clusters, so its points must be examined individually. When node C is reached, it falls entirely within cluster 2; again, we can update cluster 2 immediately and need descend no further. The tree is only examined down to the frontier marked by the dashed line in Figure 4.16(b), and the advantage is that the nodes below need not be opened—at least, not on this particular iteration of *k*-means. Next time, the cluster centers will have changed and things may be different.

Discussion

Many variants of the basic *k*-means procedure have been developed. Some produce a hierarchical clustering by applying the algorithm with $k = 2$ to the overall dataset and then repeating, recursively, within each cluster.

How do you choose *k?* Often nothing is known about the likely number of clusters, and the whole point of clustering is to find out. One way is to try different values and choose the best. To do this you need to learn how to evaluate the success of machine learning, which is what Chapter 5 is about. We return to clustering in Section 6.6.

4.9 Further reading

The 1R scheme was proposed and thoroughly investigated by Holte (1993). It was never really intended as a machine learning "method": the point was more to demonstrate that very simple structures underlie most of the practical datasets being used to evaluate machine learning methods at the time and that putting high-powered inductive inference methods to work on simple datasets was like using a sledgehammer to crack a nut. Why grapple with a complex decision tree when a simple rule will do? The method that generates one simple rule per class is the result of work by Lucio de Souza Coelho of Brazil and Len Trigg of New Zealand, and it has been dubbed *hyperpipes.* A very simple algorithm, it has the advantage of being extremely fast and is quite feasible even with an enormous number of attributes.

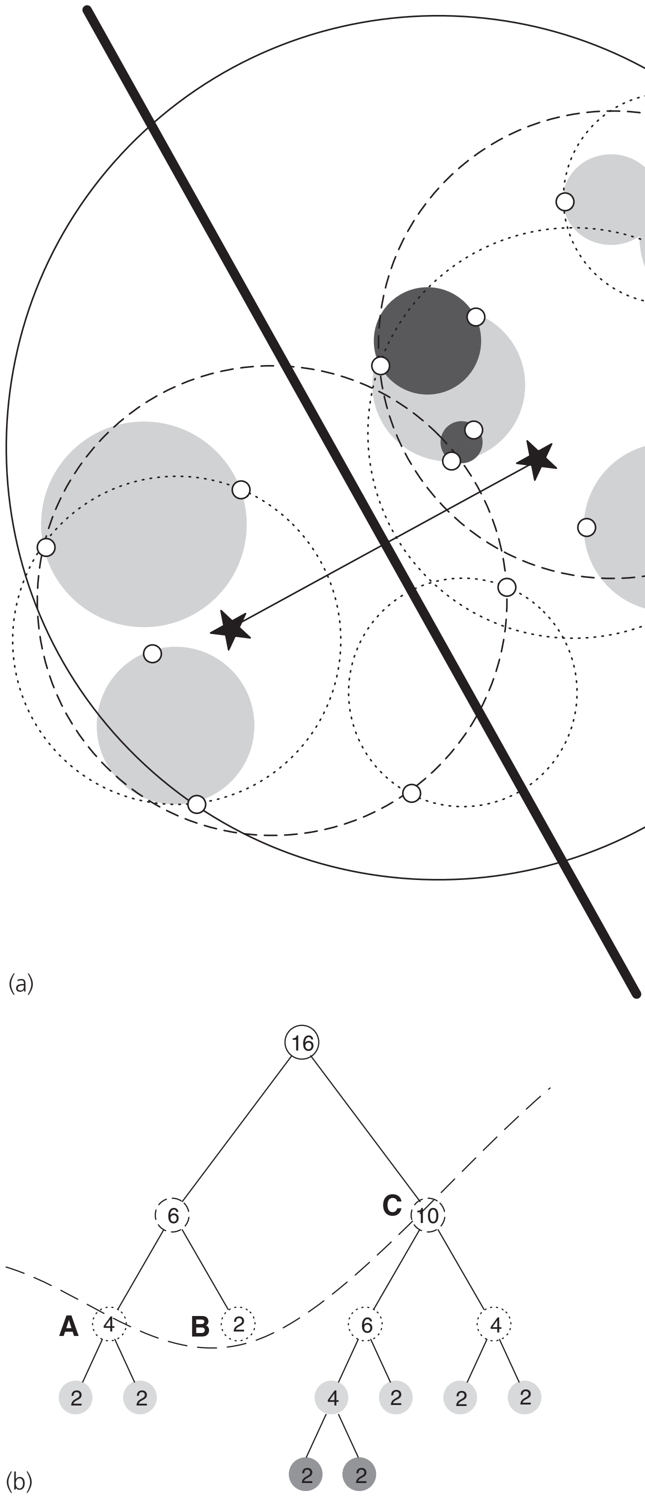


Figure 4.16 A ball tree: (a) two cluster centers and their dividing line and (b) the corresponding tree.

Bayes was an eighteenth-century English philosopher who set out his theory of probability in “An essay towards solving a problem in the doctrine of chances,” published in the *Philosophical Transactions of the Royal Society of London* (Bayes 1763); the rule that bears his name has been a cornerstone of probability theory ever since. The difficulty with the application of Bayes’s rule in practice is the assignment of prior probabilities. Some statisticians, dubbed Bayesians, take the rule as gospel and insist that people make serious attempts to estimate prior probabilities accurately—although such estimates are often subjective. Others, non-Bayesians, prefer the kind of prior-free analysis that typically generates statistical confidence intervals, which we will meet in the next chapter. With a particular dataset, prior probabilities are usually reasonably easy to estimate, which encourages a Bayesian approach to learning. The independence assumption made by the Naïve Bayes method is a great stumbling block, however, and some attempts are being made to apply Bayesian analysis without assuming independence. The resulting models are called *Bayesian networks* (Heckerman et al. 1995), and we describe them in Section 6.7.

Bayesian techniques had been used in the field of pattern recognition (Duda and Hart 1973) for 20 years before they were adopted by machine learning researchers (e.g., see Langley et al. 1992) and made to work on datasets with redundant attributes (Langley and Sage 1994) and numeric attributes (John and Langley 1995). The label *Naïve Bayes* is unfortunate because it is hard to use this method without feeling simpleminded. However, there is nothing naïve about its use in appropriate circumstances. The multinomial Naïve Bayes model, which is particularly appropriate for text classification, was investigated by McCallum and Nigam (1998).

The classic paper on decision tree induction is by Quinlan (1986), who describes the basic ID3 procedure developed in this chapter. A comprehensive description of the method, including the improvements that are embodied in C4.5, appears in a classic book by Quinlan (1993), which gives a listing of the complete C4.5 system, written in the C programming language. PRISM was developed by Cendrowska (1987), who also introduced the contact lens dataset.

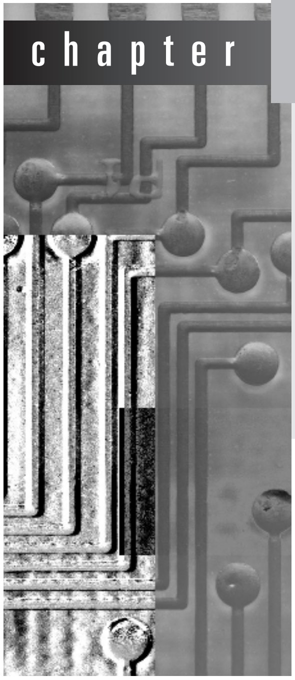
Association rules are introduced and described in the database literature rather than in the machine learning literature. Here the emphasis is very much on dealing with huge amounts of data rather than on sensitive ways of testing and evaluating algorithms on limited datasets. The algorithm introduced in this chapter is the Apriori method developed by Agrawal and his associates (Agrawal et al. 1993a, 1993b; Agrawal and Srikant 1994). A survey of association-rule mining appears in an article by Chen et al. (1996).

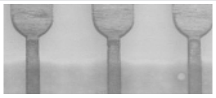
Linear regression is described in most standard statistical texts, and a particularly comprehensive treatment can be found in a book by Lawson and Hanson (1995). The use of linear models for classification enjoyed a great deal of popularity in the 1960s; Nilsson (1965) provides an excellent reference. He defines a *linear threshold unit* as a binary test of whether a linear function is greater or less than zero and a *linear machine* as a set of linear functions, one for each class, whose value for an unknown example is compared and the largest chosen as its predicted class. In the distant past, perceptrons fell out of favor on publication of an influential book that showed they had fundamental limitations (Minsky and Papert 1969); however, more complex systems of linear functions have enjoyed a resurgence in recent years in the form of neural networks, described in Section 6.3. The Winnow algorithms were introduced by Nick Littlestone in his PhD thesis in 1989 (Littlestone 1988, 1989). Multiresponse linear classifiers have found a new application recently for an operation called *stacking* that combines the output of other learning algorithms, described in Chapter 7 (see Wolpert 1992). Friedman (1996) describes the technique of pairwise classification, Fürnkranz (2002) further analyzes it, and Hastie and Tibshirani (1998) extend it to estimate probabilities using pairwise coupling.

Fix and Hodges (1951) performed the first analysis of the nearest-neighbor method, and Johns (1961) pioneered its use in classification problems. Cover and Hart (1967) obtained the classic theoretical result that, for large enough datasets, its probability of error never exceeds twice the theoretical minimum; Devroye et al. (1996) showed that k-nearest neighbor is asymptotically optimal for large k and n with $k/n \rightarrow 0$. Nearest-neighbor methods gained popularity in machine learning through the work of Aha (1992), who showed that instance-based learning can be combined with noisy exemplar pruning and attribute weighting and that the resulting methods perform well in comparison with other learning methods. We take this up again in Chapter 6.

The kD-tree data structure was developed by Friedman et al. (1977). Our description closely follows an explanation given by Andrew Moore in his PhD thesis (Moore 1991), who, along with Omohundro (1987), pioneered its use in machine learning. Moore (2000) describes sophisticated ways of constructing ball trees that perform well even with thousands of attributes. We took our ball tree example from lecture notes by Alexander Gray of Carnegie-Mellon University. The voting feature intervals method mentioned in the *Discussion* subsection at the end of Section 4.7 is described by Demiroz and Guvenir (1997).

The k-means algorithm is a classic technique, and many descriptions and variations are available (e.g., see Hartigan 1975). The clever use of kD-trees to speed up k-means clustering, which we chose to illustrate using ball trees instead, was pioneered by Moore and Pelleg (2000) in their X-means clustering algorithm. That algorithm also contains some other innovations, described in Section 6.6.





chapter 5

Credibility: Evaluating What's Been Learned

Evaluation is the key to making real progress in data mining. There are lots of ways of inferring structure from data: we have encountered many already and will see further refinements, and new methods, in the next chapter. But to determine which ones to use on a particular problem we need systematic ways to evaluate how different methods work and to compare one with another. Evaluation is not as simple as it might appear at first sight.

What's the problem? We have the training set; surely we can just look at how well different methods do on that. Well, no: as we will see very shortly, performance on the training set is definitely not a good indicator of performance on an independent test set. We need ways of predicting performance bounds in practice, based on experiments with whatever data can be obtained.

When a vast supply of data is available, this is no problem: just make a model based on a large training set, and try it out on another large test set. But although data mining sometimes involves "big data"—particularly in marketing, sales, and customer support applications—it is often the case that data, quality data, is scarce. The oil slicks mentioned in Chapter 1 (pages 23–24) had to be detected and marked manually—a skilled and labor-intensive process—before being used as training data. Even in the credit card application (pages 22–23), there turned out to be only 1000 training examples of the appropriate type. The electricity supply data (pages 24–25) went back 15 years, 5000 days—but only 15 Christmas Days and Thanksgivings, and just 4 February 29s and presidential elections. The electromechanical diagnosis application (pages 25–26) was able to capitalize on 20 years of recorded experience, but this yielded only 300 usable examples of faults. Marketing and sales applications (pages 26–28) certainly involve big data, but many others do not: training data frequently relies on specialist human expertise—and that is always in short supply.

The question of predicting performance based on limited data is an interesting, and still controversial, one. We will encounter many different techniques, of which one—repeated cross-validation—is gaining ascendance and is probably the evaluation method of choice in most practical limited-data situations. Comparing the performance of different machine learning methods on a given problem is another matter that is not so easy as it sounds: to be sure that apparent differences are not caused by chance effects, statistical tests are needed. So far we have tacitly assumed that what is being predicted is the ability to classify test instances accurately; however, some situations involve predicting the class probabilities rather than the classes themselves, and others involve predicting numeric rather than nominal values. Different methods are needed in each case. Then we look at the question of cost. In most practical data mining situations the cost of a misclassification error depends on the type of error it is—whether, for example, a positive example was erroneously classified as negative or vice versa. When doing data mining, and evaluating its performance, it is often essential to take these costs into account. Fortunately, there are simple techniques to make most learning schemes cost sensitive without grappling with the internals of the algorithm. Finally, the whole notion of evaluation has fascinating philosophical connections. For 2000 years philosophers have debated the question of how to evaluate scientific theories, and the issues are brought into sharp focus by data mining because what is extracted is essentially a "theory" of the data.

5.1 Training and testing

For classification problems, it is natural to measure a classifier's performance in terms of the *error rate.* The classifier predicts the class of each instance: if it is correct, that is counted as a *success;* if not, it is an *error.* The error rate is just the proportion of errors made over a whole set of instances, and it measures the overall performance of the classifier.

Of course, what we are interested in is the likely future performance on new data, not the past performance on old data. We already know the classifications of each instance in the training set, which after all is why we can use it for training. We are not generally interested in learning about those classifications—although we might be if our purpose is data cleansing rather than prediction. So the question is, is the error rate on old data likely to be a good indicator of the error rate on new data? The answer is a resounding no—not if the old data was used during the learning process to train the classifier.

This is a surprising fact, and a very important one. Error rate on the training set is *not* likely to be a good indicator of future performance. Why? Because the classifier has been learned from the very same training data, any estimate of performance based on that data will be optimistic, and may be hopelessly optimistic.

We have already seen an example of this in the labor relations dataset. Figure 1.3(b) was generated directly from the training data, and Figure 1.3(a) was obtained from it by a process of pruning. The former is likely to be more accurate on the data that was used to train the classifier but will probably perform less well on independent test data because it is overfitted to the training data. The first tree will look good according to the error rate on the training data, better than the second tree. But this does not reflect how they will perform on independent test data.

The error rate on the training data is called the *resubstitution error,* because it is calculated by resubstituting the training instances into a classifier that was constructed from them. Although it is not a reliable predictor of the true error rate on new data, it is nevertheless often useful to know.

To predict the performance of a classifier on new data, we need to assess its error rate on a dataset that played no part in the formation of the classifier. This independent dataset is called the *test set.* We assume that both the training data and the test data are representative samples of the underlying problem.

In some cases the test data might be distinct in nature from the training data. Consider, for example, the credit risk problem from Section 1.3. Suppose the bank had training data from branches in New York City and Florida and wanted to know how well a classifier trained on one of these datasets would perform in a new branch in Nebraska. It should probably use the Florida data as test data to evaluate the New York-trained classifier and the New York data to evaluate the Florida-trained classifier. If the datasets were amalgamated before training, performance on the test data would probably not be a good indicator of performance on future data in a completely different state.

It is important that the test data was not used *in any way* to create the classifier. For example, some learning methods involve two stages, one to come up with a basic structure and the second to optimize parameters involved in that structure, and separate sets of data may be needed in the two stages. Or you might try out several learning schemes on the training data and then evaluate them—on a fresh dataset, of course—to see which one works best. But none of this data may be used to determine an estimate of the future error rate. In such situations people often talk about three datasets: the *training* data, the *validation* data, and the *test* data. The training data is used by one or more learning methods to come up with classifiers. The validation data is used to optimize parameters of those classifiers, or to select a particular one. Then the test data is used to calculate the error rate of the final, optimized, method. Each of the three sets must be chosen independently: the validation set must be different from the training set to obtain good performance in the optimization or selection stage, and the test set must be different from both to obtain a reliable estimate of the true error rate.

It may be that once the error rate has been determined, the test data is bundled back into the training data to produce a new classifier for actual use. There is nothing wrong with this: it is just a way of maximizing the amount of data used to generate the classifier that will actually be employed in practice. What is important is that error rates are not quoted based on any of this data. Also, once the validation data has been used—maybe to determine the best type of learning scheme to use—then it can be bundled back into the training data to retrain that learning scheme, maximizing the use of data.

If lots of data is available, there is no problem: we take a large sample and use it for training; then another, independent large sample of different data and use it for testing. Provided that both samples are representative, the error rate on the test set will give a true indication of future performance. Generally, the larger the training sample the better the classifier, although the returns begin to diminish once a certain volume of training data is exceeded. And the larger the test sample, the more accurate the error estimate. The accuracy of the error estimate can be quantified statistically, as we will see in the next section.

The real problem occurs when there is not a vast supply of data available. In many situations the training data must be classified manually—and so must the test data, of course, to obtain error estimates. This limits the amount of data that can be used for training, validation, and testing, and the problem becomes how to make the most of a limited dataset. From this dataset, a certain amount is held over for testing—this is called the *holdout* procedure—and the remainder is used for training (and, if necessary, part of that is set aside for validation). There's a dilemma here: to find a good classifier, we want to use as much of the data as possible for training; to obtain a good error estimate, we want to use as much of it as possible for testing. Sections 5.3 and 5.4 review widely used methods for dealing with this dilemma.

5.2 Predicting performance

Suppose we measure the error of a classifier on a test set and obtain a certain numeric error rate—say 25%. Actually, in this section we refer to success rate rather than error rate, so this corresponds to a success rate of 75%. Now, this is only an estimate. What can you say about the *true* success rate on the target population? Sure, it's expected to be close to 75%. But how close—within 5%? Within 10%? It must depend on the size of the test set. Naturally, we would be more confident of the 75% figure if it was based on a test set of 10,000 instances rather than on a test set of 100 instances. But how much more confident would we be?

To answer these questions, we need some statistical reasoning. In statistics, a succession of independent events that either succeed or fail is called a *Bernoulli process.* The classic example is coin tossing. Each toss is an independent event. Let's say we always predict heads; but rather than "heads" or "tails," each toss is considered a "success" or a "failure." Let's say the coin is biased, but we don't know what the probability of heads is. Then, if we actually toss the coin 100 times and 75 of them are heads, we have a situation much like the one described previously for a classifier with an observed 75% success rate on a test set. What can we say about the true success probability? In other words, imagine that there is a Bernoulli process—a biased coin—whose true (but unknown) success rate is p. Suppose that out of N trials, S are successes: thus the observed success rate is $f = S/N$. The question is, what does this tell you about the true success rate p?

The answer to this question is usually expressed as a confidence interval; that is, p lies within a certain specified interval with a certain specified confidence. For example, if $S = 750$ successes are observed out of $N = 1000$ trials, this indicates that the true success rate must be around 75%. But how close to 75%? It turns out that with 80% confidence, the true success rate p lies between 73.2% and 76.7%. If $S = 75$ successes are observed out of $N = 100$ trials, this also indicates that the true success rate must be around 75%. But the experiment is smaller, and the 80% confidence interval for p is wider, stretching from 69.1% to 80.1%.

These figures are easy to relate to qualitatively, but how are they derived quantitatively? We reason as follows: the mean and variance of a single Bernoulli trial with success rate p are p and $p(1 - p)$, respectively. If N trials are taken from a Bernoulli process, the expected success rate $f = S/N$ is a random variable with the same mean p; the variance is reduced by a factor of N to $p(1 - p)/N$. For large N, the distribution of this random variable approaches the normal distribution. These are all facts of statistics: we will not go into how they are derived.

The probability that a random variable X, with zero mean, lies within a certain confidence range of width $2z$ is

$$\Pr[-z \le X \le z] = c.$$

For a normal distribution, values of c and corresponding values of z are given in tables printed at the back of most statistical texts. However, the tabulations conventionally take a slightly different form: they give the confidence that X will lie outside the range, and they give it for the upper part of the range only:

$$\Pr[X \geq z].$$

This is called a *one-tailed* probability because it refers only to the upper "tail" of the distribution. Normal distributions are symmetric, so the probabilities for the lower tail

$$\Pr[X \leq -z]$$

are just the same.

Table 5.1 gives an example. Like other tables for the normal distribution, this assumes that the random variable X has a mean of zero and a variance of one. Alternatively, you might say that the z figures are measured in *standard deviations from the mean.* Thus the figure for $\Pr[X \geq z] = 5\%$ implies that there is a 5% chance that X lies more than 1.65 standard deviations above the mean. Because the distribution is symmetric, the chance that X lies more than 1.65 standard deviations from the mean (above or below) is 10%, or

$$\Pr[-1.65 \leq X \leq 1.65] = 90\%.$$

All we need do now is reduce the random variable f to have zero mean and unit variance. We do this by subtracting the mean p and dividing by the standard deviation $\sqrt{p(1-p)/N}$. This leads to

$$\Pr\left[-z < \frac{f-p}{\sqrt{p(1-p)/N}} < z\right] = c.$$

Now here is the procedure for finding confidence limits. Given a particular confidence figure c, consult Table 5.1 for the corresponding z value. To use the table you will first have to subtract c from 1 and then halve the result, so that for $c =$ 90% you use the table entry for 5%. Linear interpolation can be used for inter-

Table 5.1 **Confidence limits for the normal distribution.**

$\Pr[X \geq z]$	z
0.1%	3.09
0.5%	2.58
1%	2.33
5%	1.65
10%	1.28
20%	0.84
40%	0.25

mediate confidence levels. Then write the inequality in the preceding expression as an equality and invert it to find an expression for *p*.

The final step involves solving a quadratic equation. Although not hard to do, it leads to an unpleasantly formidable expression for the confidence limits:

$$p=\left(f+\frac{z^2}{2N}\pm z\sqrt{\frac{f}{N}-\frac{f^2}{N}+\frac{z^2}{4N^2}}\right)\Bigg/\left(1+\frac{z^2}{N}\right).$$

The ± in this expression gives two values for p that represent the upper and lower confidence boundaries. Although the formula looks complicated, it is not hard to work out in particular cases.

This result can be used to obtain the values in the preceding numeric example. Setting $f = 75\%$, $N = 1000$, and $c = 80\%$ (so that $z = 1.28$) leads to the interval [0.732,0.767] for p, and $N = 100$ leads to [0.691,0.801] for the same level of confidence. Note that the normal distribution assumption is only valid for large N (say, $N > 100$). Thus $f = 75\%$ and $N = 10$ leads to confidence limits [0.549,0.881]—but these should be taken with a grain of salt.

5.3 Cross-validation

Now consider what to do when the amount of data for training and testing is limited. The holdout method reserves a certain amount for testing and uses the remainder for training (and sets part of that aside for validation, if required). In practical terms, it is common to hold out one-third of the data for testing and use the remaining two-thirds for training.

Of course, you may be unlucky: the sample used for training (or testing) might not be representative. In general, you cannot tell whether a sample is representative or not. But there is one simple check that might be worthwhile: each class in the full dataset should be represented in about the right proportion in the training and testing sets. If, by bad luck, all examples with a certain class were missing from the training set, you could hardly expect a classifier learned from that data to perform well on the examples of that class—and the situation would be exacerbated by the fact that the class would necessarily be overrepresented in the test set because none of its instances made it into the training set! Instead, you should ensure that the random sampling is done in such a way as to guarantee that each class is properly represented in both training and test sets. This procedure is called *stratification,* and we might speak of *stratified holdout.* Although it is generally well worth doing, stratification provides only a primitive safeguard against uneven representation in training and test sets.

A more general way to mitigate any bias caused by the particular sample chosen for holdout is to repeat the whole process, training and testing, several times with different random samples. In each iteration a certain proportion—

say two-thirds—of the data is randomly selected for training, possibly with stratification, and the remainder used for testing. The error rates on the different iterations are averaged to yield an overall error rate. This is the *repeated holdout* method of error rate estimation.

In a single holdout procedure, you might consider swapping the roles of the testing and training data—that is, train the system on the test data and test it on the training data—and average the two results, thus reducing the effect of uneven representation in training and test sets. Unfortunately, this is only really plausible with a 50 : 50 split between training and test data, which is generally not ideal—it is better to use more than half the data for training even at the expense of test data. However, a simple variant forms the basis of an important statistical technique called *cross-validation.* In cross-validation, you decide on a fixed number of *folds,* or partitions of the data. Suppose we use three. Then the data is split into three approximately equal partitions and each in turn is used for testing and the remainder is used for training. That is, use two-thirds for training and one-third for testing and repeat the procedure three times so that, in the end, every instance has been used exactly once for testing. This is called *threefold cross-validation,* and if stratification is adopted as well—which it often is—it is *stratified threefold cross-validation.*

The standard way of predicting the error rate of a learning technique given a single, fixed sample of data is to use stratified 10-fold cross-validation. The data is divided randomly into 10 parts in which the class is represented in approximately the same proportions as in the full dataset. Each part is held out in turn and the learning scheme trained on the remaining nine-tenths; then its error rate is calculated on the holdout set. Thus the learning procedure is executed a total of 10 times on different training sets (each of which have a lot in common). Finally, the 10 error estimates are averaged to yield an overall error estimate.

Why 10? Extensive tests on numerous datasets, with different learning techniques, have shown that 10 is about the right number of folds to get the best estimate of error, and there is also some theoretical evidence that backs this up. Although these arguments are by no means conclusive, and debate continues to rage in machine learning and data mining circles about what is the best scheme for evaluation, 10-fold cross-validation has become the standard method in practical terms. Tests have also shown that the use of stratification improves results slightly. Thus the standard evaluation technique in situations where only limited data is available is stratified 10-fold cross-validation. Note that neither the stratification nor the division into 10 folds has to be exact: it is enough to divide the data into 10 approximately equal sets in which the various class values are represented in approximately the right proportion. Statistical evaluation is not an exact science. Moreover, there is nothing magic about the exact number 10: 5-fold or 20-fold cross-validation is likely to be almost as good.

A single 10-fold cross-validation might not be enough to get a reliable error estimate. Different 10-fold cross-validation experiments with the same learning method and dataset often produce different results, because of the effect of random variation in choosing the folds themselves. Stratification reduces the variation, but it certainly does not eliminate it entirely. When seeking an accurate error estimate, it is standard procedure to repeat the cross-validation process 10 times—that is, 10 times 10-fold cross-validation—and average the results. This involves invoking the learning algorithm 100 times on datasets that are all nine-tenths the size of the original. Obtaining a good measure of performance is a computation-intensive undertaking.

5.4 Other estimates

Tenfold cross-validation is the standard way of measuring the error rate of a learning scheme on a particular dataset; for reliable results, 10 times 10-fold cross-validation. But many other methods are used instead. Two that are particularly prevalent are *leave-one-out* cross-validation and the *bootstrap*.

Leave-one-out

Leave-one-out cross-validation is simply n-fold cross-validation, where n is the number of instances in the dataset. Each instance in turn is left out, and the learning method is trained on all the remaining instances. It is judged by its correctness on the remaining instance—one or zero for success or failure, respectively. The results of all n judgments, one for each member of the dataset, are averaged, and that average represents the final error estimate.

This procedure is an attractive one for two reasons. First, the greatest possible amount of data is used for training in each case, which presumably increases the chance that the classifier is an accurate one. Second, the procedure is deterministic: no random sampling is involved. There is no point in repeating it 10 times, or repeating it at all: the same result will be obtained each time. Set against this is the high computational cost, because the entire learning procedure must be executed n times and this is usually quite infeasible for large datasets. Nevertheless, leave-one-out seems to offer a chance of squeezing the maximum out of a small dataset and obtaining as accurate an estimate as possible.

But there is a disadvantage to leave-one-out cross-validation, apart from the computational expense. By its very nature, it cannot be stratified—worse than that, it *guarantees* a nonstratified sample. Stratification involves getting the correct proportion of examples in each class into the test set, and this is impossible when the test set contains only a single example. A dramatic, although highly artificial, illustration of the problems this might cause is to imagine a completely random dataset that contains the same number of each of two classes. The best that an inducer can do with random data is to predict the majority class, giving a true error rate of 50%. But in each fold of leave-one-out, the opposite class to the test instance is in the majority—and therefore the predictions will always be incorrect, leading to an estimated error rate of 100%!

The bootstrap

The second estimation method we describe, the bootstrap, is based on the statistical procedure of sampling *with replacement.* Previously, whenever a sample was taken from the dataset to form a training or test set, it was drawn without replacement. That is, the same instance, once selected, could not be selected again. It is like picking teams for football: you cannot choose the same person twice. But dataset instances are not like people. Most learning methods can use the same instance twice, and it makes a difference in the result of learning if it is present in the training set twice. (Mathematical sticklers will notice that we should not really be talking about "sets" at all if the same object can appear more than once.)

The idea of the bootstrap is to sample the dataset with replacement to form a training set. We will describe a particular variant, mysteriously (but for a reason that will soon become apparent) called the *0.632 bootstrap.* For this, a dataset of n instances is sampled n times, with replacement, to give another dataset of n instances. Because some elements in this second dataset will (almost certainly) be repeated, there must be some instances in the original dataset that have not been picked: we will use these as test instances.

What is the chance that a particular instance will not be picked for the training set? It has a $1/n$ probability of being picked each time and therefore a $1 - 1/n$ probability of not being picked. Multiply these probabilities together according to the number of picking opportunities, which is n, and the result is a figure of

$$\left(1-\frac{1}{n}\right)^n \approx e^{-1} = 0.368$$

(where e is the base of natural logarithms, 2.7183, not the error rate!). This gives the chance of a particular instance not being picked at all. Thus for a reasonably large dataset, the test set will contain about 36.8% of the instances and the training set will contain about 63.2% of them (now you can see why it's called the *0.632 bootstrap*). Some instances will be repeated in the training set, bringing it up to a total size of n, the same as in the original dataset.

The figure obtained by training a learning system on the training set and calculating its error over the test set will be a pessimistic estimate of the true error rate, because the training set, although its size is n, nevertheless contains only 63% of the instances, which is not a great deal compared, for example, with the

90% used in 10-fold cross-validation. To compensate for this, we combine the test-set error rate with the resubstitution error on the instances in the training set. The resubstitution figure, as we warned earlier, gives a very optimistic estimate of the true error and should certainly not be used as an error figure on its own. But the bootstrap procedure combines it with the test error rate to give a final estimate *e* as follows:

$$e = 0.632 \times e_{\text{test instances}} + 0.368 \times e_{\text{training instances}}.$$

Then, the whole bootstrap procedure is repeated several times, with different replacement samples for the training set, and the results averaged.

The bootstrap procedure may be the best way of estimating error for very small datasets. However, like leave-one-out cross-validation, it has disadvantages that can be illustrated by considering a special, artificial situation. In fact, the very dataset we considered previously will do: a completely random dataset with two classes. The true error rate is 50% for any prediction rule. But a scheme that memorized the training set would give a perfect resubstitution score of 100% so that $e_{\text{training instances}} = 0$, and the 0.632 bootstrap will mix this in with a weight of 0.368 to give an overall error rate of only 31.6% ($0.632 \times 50\% + 0.368 \times 0\%$), which is misleadingly optimistic.

5.5 Comparing data mining methods

We often need to compare two different learning methods on the same problem to see which is the better one to use. It seems simple: estimate the error using cross-validation (or any other suitable estimation procedure), perhaps repeated several times, and choose the scheme whose estimate is smaller. This is quite sufficient in many practical applications: if one method has a lower estimated error than another on a particular dataset, the best we can do is to use the former method's model. However, it may be that the difference is simply caused by estimation error, and in some circumstances it is important to determine whether one scheme is really better than another on a particular problem. This is a standard challenge for machine learning researchers. If a new learning algorithm is proposed, its proponents must show that it improves on the state of the art for the problem at hand and demonstrate that the observed improvement is not just a chance effect in the estimation process.

This is a job for a statistical test that gives confidence bounds, the kind we met previously when trying to predict true performance from a given test-set error rate. If there were unlimited data, we could use a large amount for training and evaluate performance on a large independent test set, obtaining confidence bounds just as before. However, if the difference turns out to be significant we must ensure that this is not just because of the particular dataset we happened to base the experiment on. What we want to determine is whether one scheme is better or worse than another on average, across all possible training and test datasets that can be drawn from the domain. Because the amount of training data naturally affects performance, all datasets should be the same size: indeed, the experiment might be repeated with different sizes to obtain a learning curve.

For the moment, assume that the supply of data is unlimited. For definiteness, suppose that cross-validation is being used to obtain the error estimates (other estimators, such as repeated cross-validation, are equally viable). For each learning method we can draw several datasets of the same size, obtain an accuracy estimate for each dataset using cross-validation, and compute the mean of the estimates. Each cross-validation experiment yields a different, independent error estimate. What we are interested in is the mean accuracy across all possible datasets of the same size, and whether this mean is greater for one scheme or the other.

From this point of view, we are trying to determine whether the mean of a set of samples—cross-validation estimates for the various datasets that we sampled from the domain—is significantly greater than, or significantly less than, the mean of another. This is a job for a statistical device known as the *t-test*, or *Student's t-test.* Because the same cross-validation experiment can be used for both learning methods to obtain a matched pair of results for each dataset, a more sensitive version of the *t*-test known as a *paired t-test* can be used.

We need some notation. There is a set of samples $x_1, x_2, \ldots, x_k$ obtained by successive 10-fold cross-validations using one learning scheme, and a second set of samples $y_1, y_2, \ldots, y_k$ obtained by successive 10-fold cross-validations using the other. Each cross-validation estimate is generated using a different dataset (but all datasets are of the same size and from the same domain). We will get the best results if exactly the same cross-validation partitions are used for both schemes so that x_1 and y_1 are obtained using the same cross-validation split, as are x_2 and y_2, and so on. Denote the mean of the first set of samples by $\overline{x}$ and the mean of the second set by $\overline{y}$. We are trying to determine whether $\overline{x}$ is significantly different from $\overline{y}$.

If there are enough samples, the mean ($\overline{x}$) of a set of independent samples ($x_1, x_2, \ldots, x_k$) has a normal (i.e., Gaussian) distribution, regardless of the distribution underlying the samples themselves. We will call the true value of the mean μ. If we knew the variance of that normal distribution, so that it could be reduced to have zero mean and unit variance, we could obtain confidence limits on μ given the mean of the samples ($\overline{x}$). However, the variance is unknown, and the only way we can obtain it is to estimate it from the set of samples.

That is not hard to do. The variance of $\overline{x}$ can be estimated by dividing the variance calculated from the samples $x_1, x_2, \ldots, x_k$—call it σ_x^2—by k. But the fact that we have to *estimate* the variance changes things somewhat. We can reduce the distribution of $\bar{x}$ to have zero mean and unit variance by using

$$\frac{\bar{x}-\mu}{\sqrt{\sigma_x^2/k}}.$$

Because the variance is only an estimate, this does *not* have a normal distribution (although it does become normal for large values of k). Instead, it has what is called a *Student's distribution with k − 1 degrees of freedom.* What this means in practice is that we have to use a table of confidence intervals for Student's distribution rather than the confidence table for the normal distribution given earlier. For 9 degrees of freedom (which is the correct number if we are using the average of 10 cross-validations) the appropriate confidence limits are shown in Table 5.2. If you compare them with Table 5.1 you will see that the Student's figures are slightly more conservative—for a given degree of confidence, the interval is slightly wider—and this reflects the additional uncertainty caused by having to estimate the variance. Different tables are needed for different numbers of degrees of freedom, and if there are more than 100 degrees of freedom the confidence limits are very close to those for the normal distribution. Like Table 5.1, the figures in Table 5.2 are for a "one-sided" confidence interval.

To decide whether the means $\bar{x}$ and $\bar{y}$, each an average of the same number k of samples, are the same or not, we consider the differences d_i between corresponding observations, $d_i = x_i - y_i$. This is legitimate because the observations are paired. The mean of this difference is just the difference between the two means, $\bar{d} = \bar{x} - \bar{y}$, and, like the means themselves, it has a Student's distribution with $k - 1$ degrees of freedom. If the means are the same, the difference is zero (this is called the *null hypothesis*); if they're significantly different, the difference will be significantly different from zero. So for a given confidence level, we will check whether the actual difference exceeds the confidence limit.

Table 5.2 Confidence limits for Student's distribution with 9 degrees of freedom.

$\Pr[X \geq z]$	z
0.1%	4.30
0.5%	3.25
1%	2.82
5%	1.83
10%	1.38
20%	0.88

First, reduce the difference to a zero-mean, unit-variance variable called the *t*-statistic:

$$t = \frac{\bar{d}}{\sqrt{\sigma_d^2/k}}$$

where σ_d^2 is the variance of the difference samples. Then, decide on a confidence level—generally, 5% or 1% is used in practice. From this the confidence limit *z* is determined using Table 5.2 if *k* is 10; if it is not, a confidence table of the Student's distribution for the *k* value in question is used. A two-tailed test is appropriate because we do not know in advance whether the mean of the *x*'s is likely to be greater than that of the *y*'s or vice versa: thus for a 1% test we use the value corresponding to 0.5% in Table 5.2. If the value of *t* according to the preceding formula is greater than *z*, or less than –*z*, we reject the null hypothesis that the means are the same and conclude that there really is a significant difference between the two learning methods on that domain for that dataset size.

Two observations are worth making on this procedure. The first is technical: what if the observations were not paired? That is, what if we were unable, for some reason, to assess the error of each learning scheme on the same datasets? What if the number of datasets for each scheme was not even the same? These conditions could arise if someone else had evaluated one of the methods and published several different estimates for a particular domain and dataset size—or perhaps just their mean and variance—and we wished to compare this with a different learning method. Then it is necessary to use a regular, nonpaired *t*-test. If the means are normally distributed, as we are assuming, the difference between the means is also normally distributed. Instead of taking the mean of the difference, $\bar{d}$, we use the difference of the means, $\bar{x} - \bar{y}$. Of course, that's the same thing: the mean of the difference *is* the difference of the means. But the variance of the difference $\bar{d}$ is *not* the same. If the variance of the samples $x_1, x_2, \ldots, x_k$ is σ_x^2 and the variance of the samples $y_1, y_2, \ldots, y_1$ is σ_y^2, the best estimate of the variance of the difference of the means is

$$\frac{\sigma_x^2}{k} + \frac{\sigma_y^2}{1}.$$

It is this variance (or rather, its square root) that should be used as the denominator of the *t*-statistic given previously. The degrees of freedom, necessary for consulting Student's confidence tables, should be taken conservatively to be the minimum of the degrees of freedom of the two samples. Essentially, knowing that the observations are paired allows the use of a better estimate for the variance, which will produce tighter confidence bounds.

The second observation concerns the assumption that there is essentially unlimited data so that several independent datasets of the right size can be used.

In practice there is usually only a single dataset of limited size. What can be done? We could split the data into (perhaps 10) subsets and perform a cross-validation on each. However, the overall result will only tell us whether a learning scheme is preferable for that particular size—perhaps one-tenth of the original dataset. Alternatively, the original dataset could be reused—for example, with different randomizations of the dataset for each cross-validation.[2] However, the resulting cross-validation estimates will not be independent because they are not based on independent datasets. In practice, this means that a difference may be judged to be significant when in fact it is not. In fact, just increasing the number of samples k, that is, the number of cross-validation runs, will eventually yield an apparently significant difference because the value of the t-statistic increases without bound.

Various modifications of the standard t-test have been proposed to circumvent this problem, all of them heuristic and lacking sound theoretical justification. One that appears to work well in practice is the *corrected resampled t-test.* Assume for the moment that the repeated holdout method is used instead of cross-validation, repeated k times on different random splits of the same dataset to obtain accuracy estimates for two learning methods. Each time, n_1 instances are used for training and n_2 for testing, and differences d_i are computed from performance on the test data. The corrected resampled t-test uses the modified statistic

$$t = \frac{\bar{d}}{\sqrt{\left(\frac{1}{k} + \frac{n_2}{n_1}\right)\sigma_d^2}}$$

in exactly the same way as the standard t-statistic. A closer look at the formula shows that its value cannot be increased simply by increasing k. The same modified statistic can be used with repeated cross-validation, which is just a special case of repeated holdout in which the individual test sets for *one* cross-validation do not overlap. For 10-fold cross-validation repeated 10 times, $k = 100$, $n_2/n_1 = 0.1/0.9$, and σ_d^2 is based on 100 differences.

5.6 Predicting probabilities

Throughout this section we have tacitly assumed that the goal is to maximize the success rate of the predictions. The outcome for each test instance is either *correct*, if the prediction agrees with the actual value for that instance, or *incorrect,* if it does not. There are no grays: everything is black or white, correct or

[2] The method was advocated in the first edition of this book.

incorrect. In many situations, this is the most appropriate perspective. If the learning scheme, when it is actually applied, results in either a correct or an incorrect prediction, success is the right measure to use. This is sometimes called a *0 – 1 loss function:* the "loss" is either zero if the prediction is correct or one if it is not. The use of *loss* is conventional, although a more optimistic terminology might couch the outcome in terms of profit instead.

Other situations are softer edged. Most learning methods can associate a probability with each prediction (as the Naïve Bayes method does). It might be more natural to take this probability into account when judging correctness. For example, a correct outcome predicted with a probability of 99% should perhaps weigh more heavily than one predicted with a probability of 51%, and, in a two-class situation, perhaps the latter is not all that much better than an *incorrect* outcome predicted with probability 51%. Whether it is appropriate to take prediction probabilities into account depends on the application. If the ultimate application really is just a prediction of the outcome, and no prizes are awarded for a realistic assessment of the likelihood of the prediction, it does not seem appropriate to use probabilities. If the prediction is subject to further processing, however—perhaps involving assessment by a person, or a cost analysis, or maybe even serving as input to a second-level learning process—then it may well be appropriate to take prediction probabilities into account.

Quadratic loss function

Suppose that for a single instance there are k possible outcomes, or classes, and for a given instance the learning scheme comes up with a probability vector p_1, $p_2, \ldots, p_k$ for the classes (where these probabilities sum to 1). The actual outcome for that instance will be one of the possible classes. However, it is convenient to express it as a vector $a_1, a_2, \ldots, a_k$ whose ith component, where i is the actual class, is 1 and all other components are 0. We can express the penalty associated with this situation as a loss function that depends on both the p vector and the a vector.

One criterion that is frequently used to evaluate probabilistic prediction is the *quadratic loss function:*

$$\sum_j (p_j - a_j)^2.$$

Note that this is for a single instance: the summation is over possible outputs not over different instances. Just one of the a's will be 1 and the rest will be 0, so the sum contains contributions of p_j^2 for the incorrect predictions and $(1 - p_i)^2$ for the correct one. Consequently, it can be written

$$1 - 2p_i + \sum_j p_j^2,$$

where i is the correct class. When the test set contains several instances, the loss function is summed over them all.

It is an interesting theoretical fact that if you seek to minimize the value of the quadratic loss function in a situation in which the actual class is generated probabilistically, the best strategy is to choose for the p vector the actual probabilities of the different outcomes, that is, $p_i = \Pr[\text{class} = i]$. If the true probabilities are known, they will be the best values for p. If they are not, a system that strives to minimize the quadratic loss function will be encouraged to use its best estimate of $\Pr[\text{class} = i]$ as the value for p_i.

This is quite easy to see. Denote the true probabilities by $p_1^*, p_2^*, \ldots, p_k^*$ so that $p_i^* = \Pr[\text{class} = i]$. The expected value of the quadratic loss function for a test instance can be rewritten as follows:

$$E\left[\sum_j (p_j - a_j)^2\right] = \sum_j \left(E\left[p_j^2\right] - 2E\left[p_j a_j\right] + E\left[a_j^2\right]\right)$$
$$= \sum_j \left(p_j^2 - 2p_j p_j^* + p_j^*\right) = \sum_j \left((p_j - p_j^*)^2 + p_j^*(1 - p_j^*)\right).$$

The first stage just involves bringing the expectation inside the sum and expanding the square. For the second, p_j is just a constant and the expected value of a_j is simply p_j^*; moreover, because a_j is either 0 or 1, $a_j^2 = a_j$ and its expected value is p_j^* too. The third stage is straightforward algebra. To minimize the resulting sum, it is clear that it is best to choose $p_j = p_j^*$ so that the squared term disappears and all that is left is a term that is just the variance of the true distribution governing the actual class.

Minimizing the squared error has a long history in prediction problems. In the present context, the quadratic loss function forces the predictor to be honest about choosing its best estimate of the probabilities—or, rather, it gives preference to predictors that are able to make the best guess at the true probabilities. Moreover, the quadratic loss function has some useful theoretical properties that we will not go into here. For all these reasons it is frequently used as the criterion of success in probabilistic prediction situations.

Informational loss function

Another popular criterion for the evaluation of probabilistic prediction is the *informational loss function:*

$$-\log_2 p_i$$

where the ith prediction is the correct one. This is in fact identical to the negative of the log-likelihood function that is optimized by logistic regression, described in Section 4.6. It represents the information (in bits) required to express the actual class i with respect to the probability distribution $p_1, p_2, \ldots,$

p_k. In other words, if you were given the probability distribution and someone had to communicate to you which class was the one that actually occurred, this is the number of bits that person would need to encode the information if they did it as effectively as possible. (Of course, it is always possible to use *more* bits.) Because probabilities are always less than one, their logarithms are negative, and the minus sign makes the outcome positive. For example, in a two-class situation—heads or tails—with an equal probability of each class, the occurrence of a head would take 1 bit to transmit, because $-\log_2 1/2$ is 1.

The expected value of the informational loss function, if the true probabilities are $p_1^*, p_2^*, \ldots, p_k^*$, is

$$-p_1^* \log_2 p_1 - p_2^* \log_2 p_2 - \ldots - p_k^* \log_2 p_k.$$

Like the quadratic loss function, this expression is minimized by choosing $p_j = p_j^*$, in which case the expression becomes the entropy of the true distribution:

$$-p_1^* \log_2 p_1^* - p_2^* \log_2 p_2^* - \ldots - p_k^* \log_2 p_k^*.$$

Thus the informational loss function also rewards honesty in predictors that know the true probabilities, and encourages predictors that do not to put forward their best guess.

The informational loss function also has a *gambling* interpretation in which you imagine gambling on the outcome, placing odds on each possible class and winning according to the class that comes up. Successive instances are like successive bets: you carry wins (or losses) over from one to the next. The logarithm of the total amount of money you win over the whole test set is the value of the informational loss function. In gambling, it pays to be able to predict the odds as accurately as possible; in that sense, honesty pays, too.

One problem with the informational loss function is that if you assign a probability of zero to an event that actually occurs, the function's value is minus infinity. This corresponds to losing your shirt when gambling. Prudent punters never bet *everything* on a particular event, no matter how certain it appears. Likewise, prudent predictors operating under the informational loss function do not assign zero probability to any outcome. This leads to a problem when no information is available about that outcome on which to base a prediction: this is called the *zero-frequency problem,* and various plausible solutions have been proposed, such as the Laplace estimator discussed for Naïve Bayes on page 91.

Discussion

If you are in the business of evaluating predictions of probabilities, which of the two loss functions should you use? That's a good question, and there is no universally agreed-upon answer—it's really a matter of taste. Both do the fundamental job expected of a loss function: they give maximum reward to predictors that are capable of predicting the true probabilities accurately. However, there are some objective differences between the two that may help you form an opinion.

The quadratic loss function takes account not only of the probability assigned to the event that actually occurred, but also the other probabilities. For example, in a four-class situation, suppose you assigned 40% to the class that actually came up and distributed the remainder among the other three classes. The quadratic loss will depend on how you distributed it because of the sum of the p_j^2 that occurs in the expression given earlier for the quadratic loss function. The loss will be smallest if the 60% was distributed evenly among the three classes: an uneven distribution will increase the sum of the squares. The informational loss function, on the other hand, depends solely on the probability assigned to the class that actually occurred. If you're gambling on a particular event coming up, and it does, who cares how you distributed the remainder of your money among the other events?

If you assign a very small probability to the class that actually occurs, the information loss function will penalize you massively. The maximum penalty, for a zero probability, is infinite. The gambling world penalizes mistakes like this harshly, too! The quadratic loss function, on the other hand, is milder, being bounded by

$$1+\sum_j p_j^2,$$

which can never exceed 2.

Finally, proponents of the informational loss function point to a general theory of performance assessment in learning called the *minimum description length (MDL) principle.* They argue that the size of the structures that a scheme learns can be measured in bits of information, and if the same units are used to measure the loss, the two can be combined in useful and powerful ways. We return to this in Section 5.9.

5.7 Counting the cost

The evaluations that have been discussed so far do not take into account the cost of making wrong decisions, wrong classifications. Optimizing classification rate without considering the cost of the errors often leads to strange results. In one case, machine learning was being used to determine the exact day that each cow in a dairy herd was in estrus, or "in heat." Cows were identified by electronic ear tags, and various attributes were used such as milk volume and chemical composition (recorded automatically by a high-tech milking machine), and milking order—for cows are regular beasts and generally arrive in the milking shed in the same order, except in unusual circumstances such as estrus. In a modern dairy operation it's important to know when a cow is ready: animals are fertilized by artificial insemination and missing a cycle will delay calving unnecessarily, causing complications down the line. In early experiments, machine learning methods stubbornly predicted that each cow was *never* in estrus. Like humans, cows have a menstrual cycle of approximately 30 days, so this "null" rule is correct about 97% of the time—an impressive degree of accuracy in any agricultural domain! What was wanted, of course, were rules that predicted the "in estrus" situation more accurately than the "not in estrus" one: the costs of the two kinds of error were different. Evaluation by classification accuracy tacitly assumes equal error costs.

Other examples in which errors cost different amounts include loan decisions: the cost of lending to a defaulter is far greater than the lost-business cost of refusing a loan to a nondefaulter. And oil-slick detection: the cost of failing to detect an environment-threatening real slick is far greater than the cost of a false alarm. And load forecasting: the cost of gearing up electricity generators for a storm that doesn't hit is far less than the cost of being caught completely unprepared. And diagnosis: the cost of misidentifying problems with a machine that turns out to be free of faults is less than the cost of overlooking problems with one that is about to fail. And promotional mailing: the cost of sending junk mail to a household that doesn't respond is far less than the lost-business cost of not sending it to a household that would have responded. Why—these are all the examples of Chapter 1! In truth, you'd be hard pressed to find an application in which the costs of different kinds of error were the same.

In the two-class case with classes *yes* and *no,* lend or not lend, mark a suspicious patch as an oil slick or not, and so on, a single prediction has the four different possible outcomes shown in Table 5.3. The *true positives* (TP) and *true negatives* (TN) are correct classifications. A *false positive* (FP) occurs when the outcome is incorrectly predicted as *yes* (or positive) when it is actually *no* (negative). A *false negative* (FN) occurs when the outcome is incorrectly predicted as negative when it is actually positive. The *true positive rate* is TP divided

Table 5.3 **Different outcomes of a two-class prediction.**

		Predicted class	
		yes	no
Actual class	yes	true positive	false negative
	no	false positive	true negative

by the total number of positives, which is TP + FN; the *false positive rate* is FP divided by the total number of negatives, FP + TN. The overall success rate is the number of correct classifications divided by the total number of classifications:

$$\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}.$$

Finally, the error rate is one minus this.

In a multiclass prediction, the result on a test set is often displayed as a two-dimensional *confusion matrix* with a row and column for each class. Each matrix element shows the number of test examples for which the actual class is the row and the predicted class is the column. Good results correspond to large numbers down the main diagonal and small, ideally zero, off-diagonal elements. Table 5.4(a) shows a numeric example with three classes. In this case the test set has 200 instances (the sum of the nine numbers in the matrix), and 88 + 40 + 12 = 140 of them are predicted correctly, so the success rate is 70%.

But is this a fair measure of overall success? How many agreements would you expect *by chance*? This predictor predicts a total of 120 *a*'s, 60 *b*'s, and 20 *c*'s; what if you had a random predictor that predicted the same total numbers of the three classes? The answer is shown in Table 5.4(b). Its first row divides the 100 *a*'s in the test set into these overall proportions, and the second and third rows do the same thing for the other two classes. Of course, the row and column totals for this matrix are the same as before—the number of instances hasn't changed, and we have ensured that the random predictor predicts the same number of *a*'s, *b*'s, and *c*'s as the actual predictor.

This random predictor gets 60 + 18 + 4 = 82 instances correct. A measure called the *Kappa statistic* takes this expected figure into account by deducting it from the predictor's successes and expressing the result as a proportion of the total for a perfect predictor, to yield 140 − 82 = 58 extra successes out

Table 5.4 Different outcomes of a three-class prediction: (a) actual and (b) expected.

(a)

		Predicted class			
		a	b	c	Total
Actual class	a	88	10	2	100
	b	14	40	6	60
	c	18	10	12	40
	Total	120	60	20	

(b)

		Predicted class			
		a	b	c	Total
Actual class	a	60	30	10	100
	b	36	18	6	60
	c	24	12	4	40
	Total	120	60	20	

of a possible total of 200 − 82 = 118, or 49.2%. The maximum value of Kappa is 100%, and the expected value for a random predictor with the same column totals is zero. In summary, the Kappa statistic is used to measure the agreement between predicted and observed categorizations of a dataset, while correcting for agreement that occurs by chance. However, like the plain success rate, it does not take costs into account.

Cost-sensitive classification

If the costs are known, they can be incorporated into a financial analysis of the decision-making process. In the two-class case, in which the confusion matrix is like that of Table 5.3, the two kinds of error—false positives and false negatives—will have different costs; likewise, the two types of correct classification may have different benefits. In the two-class case, costs can be summarized in the form of a 2×2 matrix in which the diagonal elements represent the two types of correct classification and the off-diagonal elements represent the two types of error. In the multiclass case this generalizes to a square matrix whose size is the number of classes, and again the diagonal elements represent the cost of correct classification. Table 5.5(a) and (b) shows default cost matrixes for the two- and three-class cases whose values simply give the number of errors: misclassification costs are all 1.

Taking the cost matrix into account replaces the success rate by the average cost (or, thinking more positively, profit) per decision. Although we will not do so here, a complete financial analysis of the decision-making process might also take into account the cost of using the machine learning tool—including the cost of gathering the training data—and the cost of using the model, or decision structure, that it produces—that is, the cost of determining the attributes for the test instances. If all costs are known, and the projected number of the

Table 5.5 Default cost matrixes: (a) a two-class case and (b) a three-class case.

		Predicted class	
		yes	no
Actual class	yes	0	1
	no	1	0

(a)

		Predicted class		
		a	b	c
Actual class	a	0	1	1
	b	1	0	1
	c	1	1	0

(b)

different outcomes in the cost matrix can be estimated—say, using cross-validation—it is straightforward to perform this kind of financial analysis.

Given a cost matrix, you can calculate the cost of a particular learned model on a given test set just by summing the relevant elements of the cost matrix for the model's prediction for each test instance. Here, the costs are ignored when making predictions, but taken into account when evaluating them.

If the model outputs the probability associated with each prediction, it can be adjusted to minimize the expected cost of the predictions. Given a set of predicted probabilities for each outcome on a certain test instance, one normally selects the most likely outcome. Instead, the model could predict the class with the smallest expected misclassification cost. For example, suppose in a three-class situation the model assigns the classes *a, b,* and *c* to a test instance with probabilities p_a, p_b, and p_c, and the cost matrix is that in Table 5.5(b). If it predicts *a*, the expected cost of the prediction is obtained by multiplying the first column of the matrix, [0,1,1], by the probability vector, [p_a, p_b, p_c], yielding $p_b + p_c$ or $1 - p_a$ because the three probabilities sum to 1. Similarly, the costs for predicting the other two classes are $1 - p_b$ and $1 - p_c$. For this cost matrix, choosing the prediction with the lowest expected cost is the same as choosing the one with the greatest probability. For a different cost matrix it might be different.

We have assumed that the learning method outputs probabilities, as Naïve Bayes does. Even if they do not normally output probabilities, most classifiers can easily be adapted to compute them. In a decision tree, for example, the probability distribution for a test instance is just the distribution of classes at the corresponding leaf.

Cost-sensitive learning

We have seen how a classifier, built without taking costs into consideration, can be used to make predictions that are sensitive to the cost matrix. In this case, costs are ignored at training time but used at prediction time. An alternative is to do just the opposite: take the cost matrix into account during the training process and ignore costs at prediction time. In principle, better performance might be obtained if the classifier were tailored by the learning algorithm to the cost matrix.

In the two-class situation, there is a simple and general way to make any learning method cost sensitive. The idea is to generate training data with a different proportion of *yes* and *no* instances. Suppose that you artificially increase the number of *no* instances by a factor of 10 and use the resulting dataset for training. If the learning scheme is striving to minimize the number of errors, it will come up with a decision structure that is biased toward avoiding errors on the *no* instances, because such errors are effectively penalized 10-fold. If data with the original proportion of *no* instances is used for testing, fewer errors will be made on these than on *yes* instances—that is, there will be fewer false positives than false negatives—because false positives have been weighted 10 times more heavily than false negatives. Varying the proportion of instances in the training set is a general technique for building cost-sensitive classifiers.

One way to vary the proportion of training instances is to duplicate instances in the dataset. However, many learning schemes allow instances to be weighted. (As we mentioned in Section 3.2, this is a common technique for handling missing values.) Instance weights are normally initialized to one. To build cost-sensitive trees the weights can be initialized to the relative cost of the two kinds of error, false positives and false negatives.

Lift charts

In practice, costs are rarely known with any degree of accuracy, and people will want to ponder various scenarios. Imagine you're in the direct mailing business and are contemplating a mass mailout of a promotional offer to 1,000,000 households—most of whom won't respond, of course. Let us say that, based on previous experience, the proportion who normally respond is known to be 0.1% (1000 respondents). Suppose a data mining tool is available that, based on known information about the households, identifies a subset of 100,000 for which the response rate is 0.4% (400 respondents). It may well pay off to restrict the mailout to these 100,000 households—that depends on the mailing cost compared with the return gained for each response to the offer. In marketing terminology, the increase in response rate, a factor of four in this case, is known as the *lift* factor yielded by the learning tool. If you knew the costs, you could determine the payoff implied by a particular lift factor.

But you probably want to evaluate other possibilities, too. The same data mining scheme, with different parameter settings, may be able to identify 400,000 households for which the response rate will be 0.2% (800 respondents), corresponding to a lift factor of two. Again, whether this would be a more profitable target for the mailout can be calculated from the costs involved. It may be necessary to factor in the cost of creating and using the model—including collecting the information that is required to come up with the attribute values. After all, if developing the model is very expensive, a mass mailing may be more cost effective than a targeted one.

Given a learning method that outputs probabilities for the predicted class of each member of the set of test instances (as Naïve Bayes does), your job is to find subsets of test instances that have a high proportion of positive instances, higher than in the test set as a whole. To do this, the instances should be sorted in descending order of predicted probability of *yes.* Then, to find a sample of a given size with the greatest possible proportion of positive instances, just read

Table 5.6 Data for a lift chart.

Rank	Predicted probability	Actual class	Rank	Predicted probability	Actual class
1	0.95	*yes*	11	0.77	*no*
2	0.93	*yes*	12	0.76	*yes*
3	0.93	*no*	13	0.73	*yes*
4	0.88	*yes*	14	0.65	*no*
5	0.86	*yes*	15	0.63	*yes*
6	0.85	*yes*	16	0.58	*no*
7	0.82	*yes*	17	0.56	*yes*
8	0.80	*yes*	18	0.49	*no*
9	0.80	*no*	19	0.48	*yes*
10	0.79	*yes*	. . .	. . .	. . .

the requisite number of instances off the list, starting at the top. If each test instance's class is known, you can calculate the lift factor by simply counting the number of positive instances that the sample includes, dividing by the sample size to obtain a success proportion and dividing by the success proportion for the complete test set to determine the lift factor.

Table 5.6 shows an example for a small dataset with 150 instances, of which 50 are *yes* responses—an overall success proportion of 33%. The instances have been sorted in descending probability order according to the predicted probability of a *yes* response. The first instance is the one that the learning scheme thinks is most likely to be positive, the second is the next most likely, and so on. The numeric values of the probabilities are unimportant: rank is the only thing that matters. With each rank is given the actual class of the instance. Thus the learning method was right about items 1 and 2—they are indeed positives—but wrong about item 3, which turned out to be a negative. Now, if you were seeking the most promising sample of size 10 but only knew the predicted probabilities and not the actual classes, your best bet would be the top ten ranking instances. Eight of these are positive, so the success proportion for this sample is 80%, corresponding to a lift factor of four.

If you knew the different costs involved, you could work them out for each sample size and choose the most profitable. But a graphical depiction of the various possibilities will often be far more revealing than presenting a single "optimal" decision. Repeating the preceding operation for different-sized samples allows you to plot a lift chart like that of Figure 5.1. The horizontal axis shows the sample size as a proportion of the total possible mailout. The vertical axis shows the number of responses obtained. The lower left and upper right points correspond to no mailout at all, with a response of 0, and a full mailout, with a response of 1000. The diagonal line gives the expected result for different-

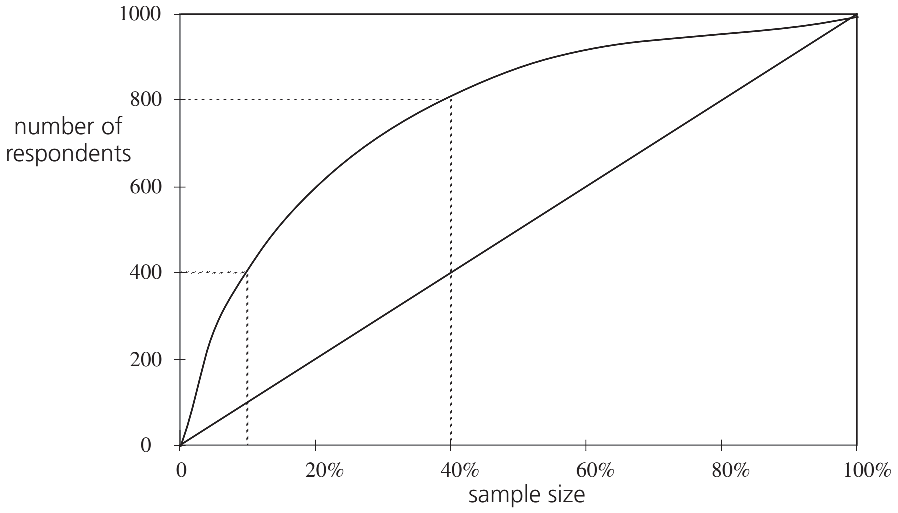


Figure 5.1 A hypothetical lift chart.

sized random samples. But we do not choose random samples; we choose those instances which, according to the data mining tool, are most likely to generate a positive response. These correspond to the upper line, which is derived by summing the actual responses over the corresponding percentage of the instance list sorted in probability order. The two particular scenarios described previously are marked: a 10% mailout that yields 400 respondents and a 40% one that yields 800.

Where you'd like to be in a lift chart is near the upper left-hand corner: at the very best, 1000 responses from a mailout of just 1000, where you send only to those households that will respond and are rewarded with a 100% success rate. Any selection procedure worthy of the name will keep you above the diagonal—otherwise, you'd be seeing a response that was worse than for random sampling. So the operating part of the diagram is the upper triangle, and the farther to the northwest the better.

ROC curves

Lift charts are a valuable tool, widely used in marketing. They are closely related to a graphical technique for evaluating data mining schemes known as *ROC curves,* which are used in just the same situation as the preceding one, in which the learner is trying to select samples of test instances that have a high proportion of positives. The acronym stands for *receiver operating characteristic,* a term used in signal detection to characterize the tradeoff between hit rate and false alarm rate over a noisy channel. ROC curves depict the performance of a classifier without regard to class distribution or error costs. They plot the number

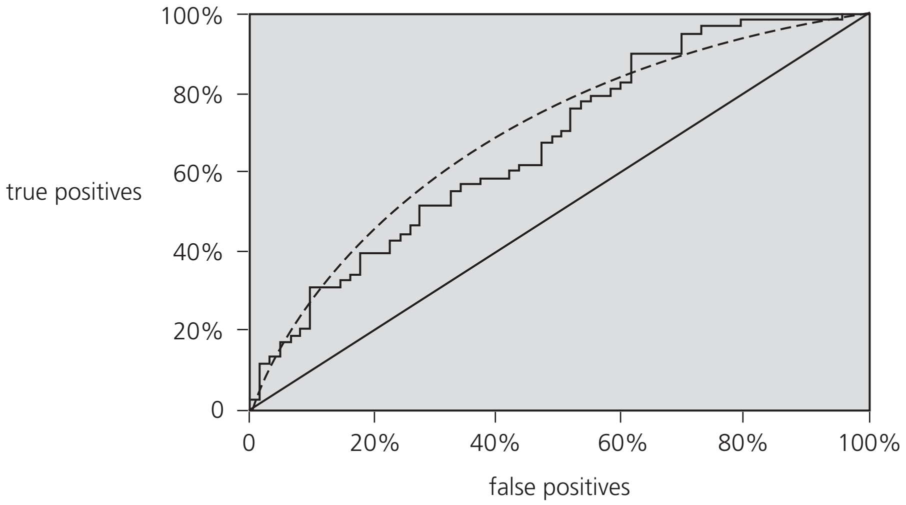


Figure 5.2 A sample ROC curve.

of positives included in the sample on the vertical axis, expressed as a percentage of the total number of positives, against the number of negatives included in the sample, expressed as a percentage of the total number of negatives, on the horizontal axis. The vertical axis is the same as that of the lift chart except that it is expressed as a percentage. The horizontal axis is slightly different—number of negatives rather than sample size. However, in direct marketing situations in which the proportion of positives is very small anyway (like 0.1%), there is negligible difference between the size of a sample and the number of negatives it contains, so the ROC curve and lift chart look very similar. As with lift charts, the northwest corner is the place to be.

Figure 5.2 shows an example ROC curve—the jagged line—for the sample of test data in Table 5.6. You can follow it along with the table. From the origin, go up two (two positives), along one (one negative), up five (five positives), along one (one negative), up one, along one, up two, and so on. Each point corresponds to drawing a line at a certain position on the ranked list, counting the *yes*'s and *no*'s above it, and plotting them vertically and horizontally, respectively. As you go farther down the list, corresponding to a larger sample, the number of positives and negatives both increase.

The jagged ROC line in Figure 5.2 depends intimately on the details of the particular sample of test data. This sample dependence can be reduced by applying cross-validation. For each different number of *no*'s—that is, each position along the horizontal axis—take just enough of the highest-ranked instances to include that number of *no*'s, and count the number of *yes*'s they contain. Finally, average that number over different folds of the cross-validation. The result is a smooth curve like that in Figure 5.2—although in reality such curves do not generally look quite so smooth.

This is just one way of using cross-validation to generate ROC curves. A simpler approach is to collect the predicted probabilities for all the various test sets (of which there are 10 in a 10-fold cross-validation), along with the true class labels of the corresponding instances, and generate a single ranked list based on this data. This assumes that the probability estimates from the classifiers built from the different training sets are all based on equally sized random samples of the data. It is not clear which method is preferable. However, the latter method is easier to implement.

If the learning scheme does not allow the instances to be ordered, you can first make it cost sensitive as described earlier. For each fold of a 10-fold cross-validation, weight the instances for a selection of different cost ratios, train the scheme on each weighted set, count the true positives and false positives in the test set, and plot the resulting point on the ROC axes. (It doesn't matter whether the test set is weighted or not because the axes in the ROC diagram are expressed as the percentage of true and false positives.) However, for inherently cost-sensitive probabilistic classifiers such as Naïve Bayes it is far more costly than the method described previously because it involves a separate learning problem for every point on the curve.

It is instructive to look at cross-validated ROC curves obtained using different learning methods. For example, in Figure 5.3, method A excels if a small, focused sample is sought; that is, if you are working toward the left-hand side of the graph. Clearly, if you aim to cover just 40% of the true positives you

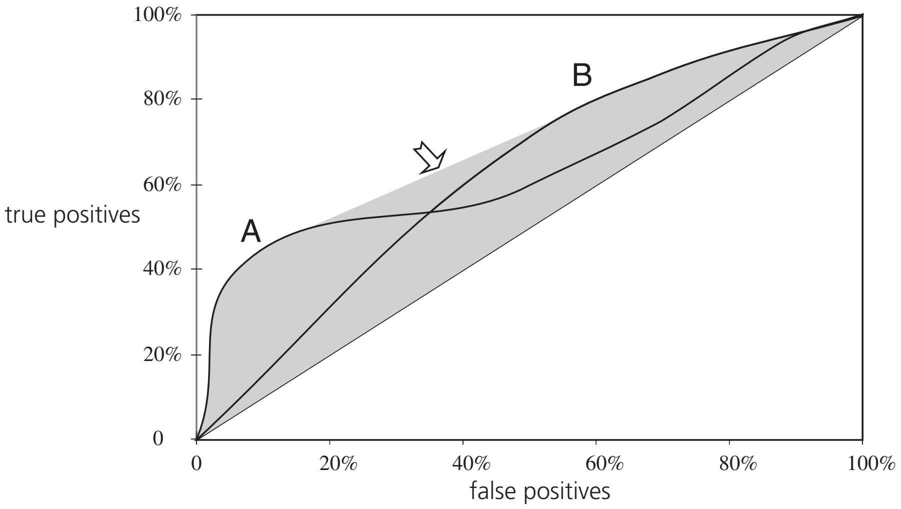


Figure 5.3 ROC curves for two learning methods.

should choose method A, which gives a false positive rate of around 5%, rather than method B, which gives more than 20% false positives. But method B excels if you are planning a large sample: if you are covering 80% of the true positives, method B will give a false positive rate of 60% as compared with method A's 80%. The shaded area is called the *convex hull* of the two curves, and you should always operate at a point that lies on the upper boundary of the convex hull.

What about the region in the middle where neither method A nor method B lies on the convex hull? It is a remarkable fact that you can get anywhere in the shaded region by combining methods A and B and using them at random with appropriate probabilities. To see this, choose a particular probability cutoff for method A that gives true and false positive rates of t_A and f_A, respectively, and another cutoff for method B that gives t_B and f_B. If you use these two schemes at random with probability p and q, where $p + q = 1$, then you will get true and false positive rates of $p.t_A + q.t_B$ and $p.f_A + q.f_B$. This represents a point lying on the straight line joining the points (t_A,f_A) and (t_B,f_B), and by varying p and q you can trace out the entire line between these two points. Using this device, the entire shaded region can be reached. Only if a particular scheme generates a point that lies on the convex hull should it be used alone: otherwise, it would always be better to use a combination of classifiers corresponding to a point that lies on the convex hull.

Recall–precision curves

People have grappled with the fundamental tradeoff illustrated by lift charts and ROC curves in a wide variety of domains. Information retrieval is a good example. Given a query, a Web search engine produces a list of hits that represent documents supposedly relevant to the query. Compare one system that locates 100 documents, 40 of which are relevant, with another that locates 400 documents, 80 of which are relevant. Which is better? The answer should now be obvious: it depends on the relative cost of false positives, documents that are returned that aren't relevant, and false negatives, documents that are relevant that aren't returned. Information retrieval researchers define parameters called *recall* and *precision:*

$$\text{recall} = \frac{\text{number of documents retrieved that are relevant}}{\text{total number of documents that are relevant}}$$

$$\text{precision} = \frac{\text{number of documents retrieved that are relevant}}{\text{total number of documents that are retrieved}}.$$

For example, if the list of *yes*'s and *no*'s in Table 5.6 represented a ranked list of retrieved documents and whether they were relevant or not, and the entire collection contained a total of 40 relevant documents, then "recall at 10" would

Table 5.7 Different measures used to evaluate the false positive versus the false negative tradeoff.

	Domain	Plot	Axes	Explanation of axes
lift chart	marketing	TP vs. subset size	TP	number of true positives
			subset size	$\frac{TP+FP}{TP+FP+TN+FN} \times 100\%$
ROC curve	communications	TP rate vs. FP rate	TP rate	$tp = \frac{TP}{TP+FN} \times 100\%$
			FP rate	$fp = \frac{FP}{FP+TN} \times 100\%$
recall–precision curve	information retrieval	recall vs. precision	recall	same as TP rate *tp*
			precision	$\frac{TP}{TP+FP} \times 100\%$

refer to recall for the top ten documents, that is, 8/40 = 5%; while "precision at 10" would be 8/10 = 80%. Information retrieval experts use *recall–precision curves* that plot one against the other, for different numbers of retrieved documents, in just the same way as ROC curves and lift charts—except that because the axes are different, the curves are hyperbolic in shape and the desired operating point is toward the upper right.

Discussion

Table 5.7 summarizes the three different ways we have met of evaluating the same basic tradeoff; TP, FP, TN, and FN are the number of true positives, false positives, true negatives, and false negatives, respectively. You want to choose a set of instances with a high proportion of *yes* instances and a high coverage of the *yes* instances: you can increase the proportion by (conservatively) using a smaller coverage, or (liberally) increase the coverage at the expense of the proportion. Different techniques give different tradeoffs, and can be plotted as different lines on any of these graphical charts.

People also seek single measures that characterize performance. Two that are used in information retrieval are *3-point average recall,* which gives the average precision obtained at recall values of 20%, 50%, and 80%, and *11-point average recall,* which gives the average precision obtained at recall values of 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100%. Also used in information retrieval is the *F-measure,* which is:

$$\frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}} = \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FP} + \text{FN}}$$

Different terms are used in different domains. Medics, for example, talk about the *sensitivity* and *specificity* of diagnostic tests. Sensitivity refers to the proportion of people with disease who have a positive test result, that is, *tp*. Specificity refers to the proportion of people without disease who have a negative test result, which is 1 − *fp*. Sometimes the product of these is used as an overall measure:

$$\text{sensitivity} \times \text{specificity} = tp(1 - fp) = \frac{\text{TP} \cdot \text{TN}}{(\text{TP} + \text{FN}) \cdot (\text{FP} + \text{TN})}$$

Finally, of course, there is our old friend the success rate:

$$\frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}.$$

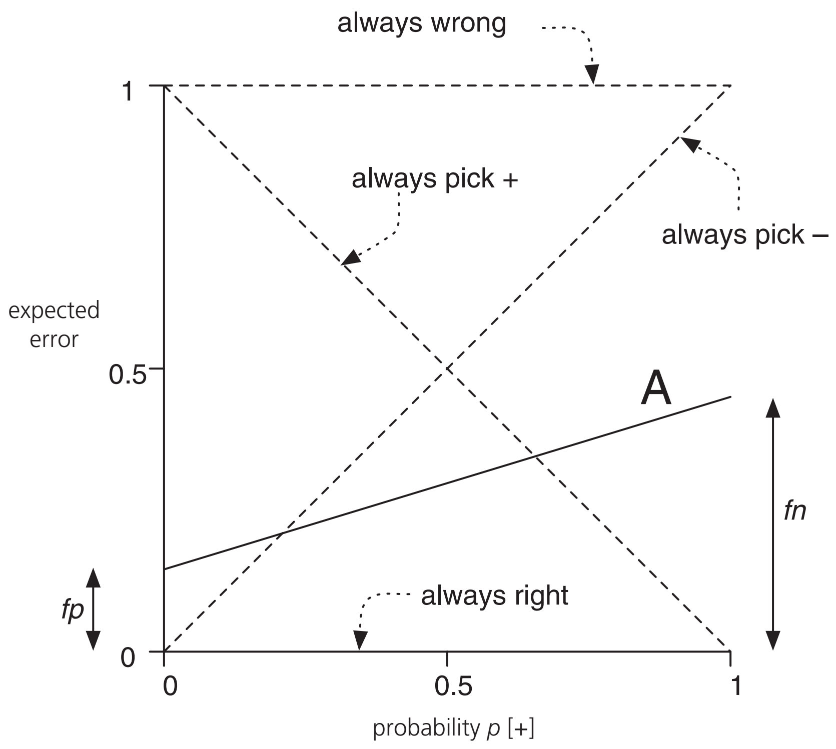
To summarize ROC curves in a single quantity, people sometimes use the area under the curve (AUC) because, roughly speaking the larger the area the better the model. The area also has a nice interpretation as the probability that the classifier ranks a randomly chosen positive instance above a randomly chosen negative one. Although such measures may be useful if costs and class distributions are unknown and one method must be chosen to handle all situations, no single number is able to capture the tradeoff. That can only be done by two-dimensional depictions such as lift charts, ROC curves, and recall–precision diagrams.

Cost curves

ROC curves and their relatives are very useful for exploring the tradeoffs among different classifiers over a range of costs. However, they are not ideal for evaluating machine learning models in situations with known error costs. For example, it is not easy to read off the expected cost of a classifier for a fixed cost matrix and class distribution. Neither can you easily determine the ranges of applicability of different classifiers. For example, from the crossover point between the two ROC curves in Figure 5.3 it is hard to tell for what cost and class distributions classifier A outperforms classifier B.

Cost curves are a different kind of display on which a single classifier corresponds to a straight line that shows how the performance varies as the class distribution changes. Again, they work best in the two-class case, although you can always make a multiclass problem into a two-class one by singling out one class and evaluating it against the remaining ones.

Figure 5.4(a) plots the expected error against the probability of one of the classes. You could imagine adjusting this probability by resampling the test set in a nonuniform way. We denote the two classes using + and −. The diagonals show the performance of two extreme classifiers: one always predicts +, giving

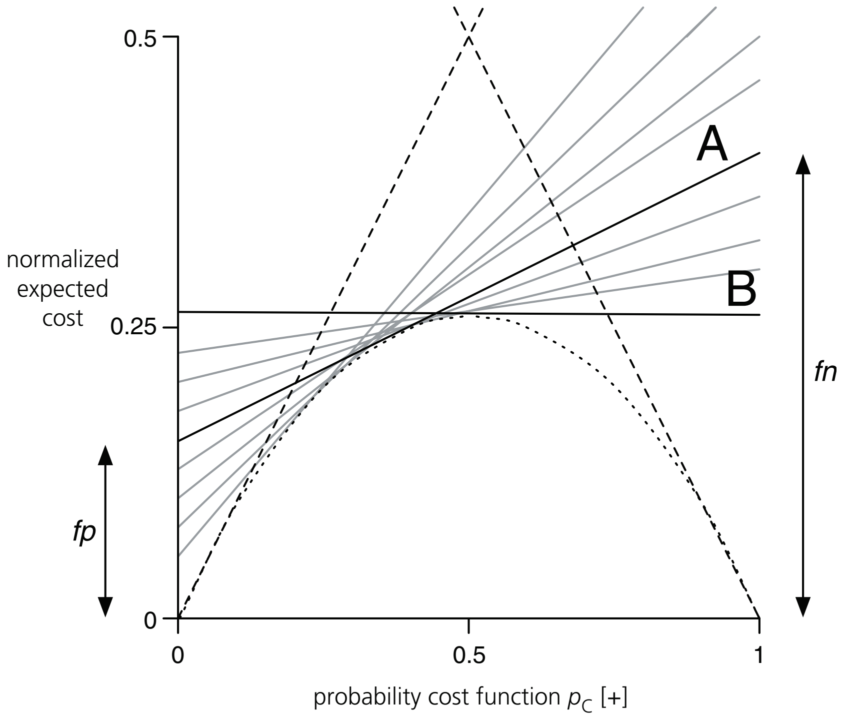


(a)

Figure 5.4 Effect of varying the probability threshold: (a) the error curve and (b) the cost curve.

an expected error of one if the dataset contains no + instances and zero if all its instances are +; the other always predicts –, giving the opposite performance. The dashed horizontal line shows the performance of the classifier that is always wrong, and the X-axis itself represents the classifier that is always correct. In practice, of course, neither of these is realizable. Good classifiers have low error rates, so where you want to be is as close to the bottom of the diagram as possible.

The line marked A represents the error rate of a particular classifier. If you calculate its performance on a certain test set, its false positive rate *fp* is its expected error on a subsample of the test set that contains only negative examples ($p[+] = 0$), and its false negative rate *fn* is the error on a subsample that contains only positive examples ($p[+] = 1$). These are the values of the intercepts at the left and right, respectively. You can see immediately from the plot that if $p[+]$ is smaller than about 0.2, predictor A is outperformed by the extreme classifier that always predicts –, and if it is larger than about 0.65, the other extreme classifier is better.



(b)
Figure 5.4 (continued)

So far we have not taken costs into account, or rather we have used the default cost matrix in which all errors cost the same. Cost curves, which do take cost into account, look very similar—very similar indeed—but the axes are different. Figure 5.4(b) shows a cost curve for the same classifier A (note that the vertical scale has been enlarged, for convenience, and ignore the gray lines for now). It plots the expected cost of using A against the *probability cost function,* which is a distorted version of $p[+]$ that retains the same extremes: zero when $p[+] = 0$ and one when $p[+] = 1$. Denote by $C[+|-]$ the cost of predicting + when the instance is actually –, and the reverse by $C[-|+]$. Then the axes of Figure 5.4(b) are

$$\text{Normalized expected cost} = fn \times p_C[+] + fp \times (1 - p_C[+])$$

$$\text{Probability cost function } p_C[+] = \frac{p[+]C[+|-]}{p[+]C[+|-] + p[-]C[-|+]}.$$

We are assuming here that correct predictions have no cost: $C[+|+] = C[-|-] = 0$. If that is not the case the formulas are a little more complex.

The maximum value that the normalized expected cost can have is 1—that is why it is "normalized." One nice thing about cost curves is that the extreme cost values at the left and right sides of the graph are *fp* and *fn*, just as they are for the error curve, so you can draw the cost curve for any classifier very easily.

Figure 5.4(b) also shows classifier B, whose expected cost remains the same across the range—that is, its false positive and false negative rates are equal. As you can see, it outperforms classifier A if the probability cost function exceeds about 0.45, and knowing the costs we could easily work out what this corresponds to in terms of class distribution. In situations that involve different class distributions, cost curves make it easy to tell when one classifier will outperform another.

In what circumstances might this be useful? To return to the example of predicting when cows will be in estrus, their 30-day cycle, or 1/30 prior probability, is unlikely to vary greatly (barring a genetic cataclysm!). But a particular herd may have different proportions of cows that are likely to reach estrus in any given week, perhaps synchronized with—who knows?—the phase of the moon. Then, different classifiers would be appropriate at different times. In the oil spill example, different batches of data may have different spill probabilities. In these situations cost curves can help to show which classifier to use when.

Each point on a lift chart, ROC curve, or recall–precision curve represents a classifier, typically obtained using different threshold values for a method such as Naïve Bayes. Cost curves represent each classifier using a straight line, and a suite of classifiers will sweep out a curved envelope whose lower limit shows how well that type of classifier can do if the parameter is well chosen. Figure 5.4(b) indicates this with a few gray lines. If the process were continued, it would sweep out the dotted parabolic curve.

The operating region of classifier B ranges from a probability cost value of about 0.25 to a value of about 0.75. Outside this region, classifier B is outperformed by the trivial classifiers represented by dashed lines. Suppose we decide to use classifier B within this range and the appropriate trivial classifier below and above it. All points on the parabola are certainly better than this scheme. But how much better? It is hard to answer such questions from an ROC curve, but the cost curve makes them easy. The performance difference is negligible if the probability cost value is around 0.5, and below a value of about 0.2 and above 0.8 it is barely perceptible. The greatest difference occurs at probability cost values of 0.25 and 0.75 and is about 0.04, or 4% of the maximum possible cost figure.

5.8 Evaluating numeric prediction

All the evaluation measures we have described pertain to classification situations rather than numeric prediction situations. The basic principles—using an independent test set rather than the training set for performance evaluation, the holdout method, and cross-validation—apply equally well to numeric prediction. But the basic quality measure offered by the error rate is no longer appropriate: errors are not simply present or absent; they come in different sizes.

Several alternative measures, summarized in Table 5.8, can be used to evaluate the success of numeric prediction. The predicted values on the test instances are $p_1, p_2, \ldots, p_n$; the actual values are $a_1, a_2, \ldots, a_n$. Notice that p_i means something very different here from what it did in the last section: there it was the probability that a particular prediction was in the ith class; here it refers to the numeric value of the prediction for the ith test instance.

Mean-squared error is the principal and most commonly used measure; sometimes the square root is taken to give it the same dimensions as the predicted value itself. Many mathematical techniques (such as linear regression, explained in Chapter 4) use the mean-squared error because it tends to be the easiest measure to manipulate mathematically: it is, as mathematicians say, "well behaved." However, here we are considering it as a performance measure: all the performance measures are easy to calculate, so mean-squared error has no particular advantage. The question is, is it an appropriate measure for the task at hand?

Mean absolute error is an alternative: just average the magnitude of the individual errors without taking account of their sign. Mean-squared error tends to exaggerate the effect of outliers—instances whose prediction error is larger than the others—but absolute error does not have this effect: all sizes of error are treated evenly according to their magnitude.

Sometimes it is the *relative* rather than *absolute* error values that are of importance. For example, if a 10% error is equally important whether it is an error of 50 in a prediction of 500 or an error of 0.2 in a prediction of 2, then averages of absolute error will be meaningless: relative errors are appropriate. This effect would be taken into account by using the relative errors in the mean-squared error calculation or the mean absolute error calculation.

Relative squared error in Table 5.8 refers to something quite different. The error is made relative to what it would have been if a simple predictor had been used. The simple predictor in question is just the average of the actual values from the training data. Thus relative squared error takes the total squared error and normalizes it by dividing by the total squared error of the default predictor.

The next error measure goes by the glorious name of *relative absolute error* and is just the total absolute error, with the same kind of normalization. In these three relative error measures, the errors are normalized by the error of the simple predictor that predicts average values.

The final measure in Table 5.8 is the *correlation coefficient,* which measures the statistical correlation between the a's and the p's. The correlation coefficient ranges from 1 for perfectly correlated results, through 0 when there is no cor-

Table 5.8 Performance measures for numeric prediction*.

Performance measure	Formula
mean-squared error	$\frac{(p_1-a_1)^2+\ldots+(p_n-a_n)^2}{n}$
root mean-squared error	$\sqrt{\frac{(p_1-a_1)^2+\ldots+(p_n-a_n)^2}{n}}$
mean absolute error	$\frac{\lvert p_1-a_1\rvert+\ldots+\lvert p_n-a_n\rvert}{n}$
relative squared error	$\frac{(p_1-a_1)^2+\ldots+(p_n-a_n)^2}{(a_1-\bar{a})^2+\ldots+(a_n-\bar{a})^2}$, where $\bar{a}=\frac{1}{n}\sum_i a_i$
root relative squared error	$\sqrt{\frac{(p_1-a_1)^2+\ldots+(p_n-a_n)^2}{(a_1-\bar{a})^2+\ldots+(a_n-\bar{a})^2}}$
relative absolute error	$\frac{\lvert p_1-a_1\rvert+\ldots+\lvert p_n-a_n\rvert}{\lvert a_1-\bar{a}\rvert+\ldots+\lvert a_n-\bar{a}\rvert}$
correlation coefficient	$\frac{S_{PA}}{\sqrt{S_P S_A}}$, where $S_{PA}=\frac{\sum_i(p_i-\bar{p})(a_i-\bar{a})}{n-1}$, $S_p=\frac{\sum_i(p_i-\bar{p})^2}{n-1}$, and $S_A=\frac{\sum_i(a_i-\bar{a})^2}{n-1}$

* *p* are predicted values and *a* are actual values.

relation, to −1 when the results are perfectly correlated negatively. Of course, negative values should not occur for reasonable prediction methods. Correlation is slightly different from the other measures because it is scale independent in that, if you take a particular set of predictions, the error is unchanged if all the predictions are multiplied by a constant factor and the actual values are left unchanged. This factor appears in every term of S_{PA} in the numerator and in every term of S_P in the denominator, thus canceling out. (This is not true for the relative error figures, despite normalization: if you multiply all the predictions by a large constant, then the difference between the predicted and the actual values will change dramatically, as will the percentage errors.) It is also different in that good performance leads to a large value of the correlation coefficient, whereas because the other methods measure error, good performance is indicated by small values.

Which of these measures is appropriate in any given situation is a matter that can only be determined by studying the application itself. What are we trying to minimize? What is the cost of different kinds of error? Often it is not easy to decide. The squared error measures and root squared error measures weigh large

Table 5.9 Performance measures for four numeric prediction models.

	A	B	C	D
root mean-squared error	67.8	91.7	63.3	57.4
mean absolute error	41.3	38.5	33.4	29.2
root relative squared error	42.2%	57.2%	39.4%	35.8%
relative absolute error	43.1%	40.1%	34.8%	30.4%
correlation coefficient	0.88	0.88	0.89	0.91

discrepancies much more heavily than small ones, whereas the absolute error measures do not. Taking the square root (root mean-squared error) just reduces the figure to have the same dimensionality as the quantity being predicted. The relative error figures try to compensate for the basic predictability or unpredictability of the output variable: if it tends to lie fairly close to its average value, then you expect prediction to be good and the relative figure compensate for this. Otherwise, if the error figure in one situation is far greater than that in another situation, it may be because the quantity in the first situation is inherently more variable and therefore harder to predict, not because the predictor is any worse.

Fortunately, it turns out that in most practical situations the best numeric prediction method is still the best no matter which error measure is used. For example, Table 5.9 shows the result of four different numeric prediction techniques on a given dataset, measured using cross-validation. Method D is the best according to all five metrics: it has the smallest value for each error measure and the largest correlation coefficient. Method C is the second best by all five metrics. The performance of methods A and B is open to dispute: they have the same correlation coefficient, method A is better than method B according to both mean-squared and relative squared errors, and the reverse is true for both absolute and relative absolute error. It is likely that the extra emphasis that the squaring operation gives to outliers accounts for the differences in this case.

When comparing two different learning schemes that involve numeric prediction, the methodology developed in Section 5.5 still applies. The only difference is that success rate is replaced by the appropriate performance measure (e.g., root mean-squared error) when performing the significance test.

5.9 The minimum description length principle

What is learned by a machine learning method is a kind of "theory" of the domain from which the examples are drawn, a theory that is predictive in that it is capable of generating new facts about the domain—in other words, the class of unseen instances. Theory is a rather grandiose term: we are using it here only in the sense of a predictive model. Thus theories might comprise decision trees or sets of rules—they don't have to be any more "theoretical" than that.

There is a long-standing tradition in science that, other things being equal, simple theories are preferable to complex ones. This is known as *Occam's razor* after the medieval philosopher William of Occam (or Ockham). Occam's razor shaves philosophical hairs off a theory. The idea is that the best scientific theory is the smallest one that explains all the facts. As Albert Einstein is reputed to have said, "Everything should be made as simple as possible, but no simpler." Of course, quite a lot is hidden in the phrase "other things being equal," and it can be hard to assess objectively whether a particular theory really does "explain" all the facts on which it is based—that's what controversy in science is all about.

In our case, in machine learning, most theories make errors. If what is learned is a theory, then the errors it makes are like *exceptions* to the theory. One way to ensure that other things *are* equal is to insist that the information embodied in the exceptions is included as part of the theory when its "simplicity" is judged.

Imagine an imperfect theory for which there are a few exceptions. Not all the data is explained by the theory, but most is. What we do is simply adjoin the exceptions to the theory, specifying them explicitly as exceptions. This new theory is larger: that is a price that, quite justifiably, has to be paid for its inability to explain all the data. However, it may be that the simplicity—is it too much to call it *elegance?*—of the original theory is sufficient to outweigh the fact that it does not quite explain everything compared with a large, baroque theory that is more comprehensive and accurate.

For example, if Kepler's three laws of planetary motion did not at the time account for the known data quite so well as Copernicus's latest refinement of the Ptolemaic theory of epicycles, they had the advantage of being far less complex, and that would have justified any slight apparent inaccuracy. Kepler was well aware of the benefits of having a theory that was compact, despite the fact that his theory violated his own aesthetic sense because it depended on "ovals" rather than pure circular motion. He expressed this in a forceful metaphor: "I have cleared the Augean stables of astronomy of cycles and spirals, and left behind me only a single cartload of dung."

The *minimum description length* or MDL principle takes the stance that the best theory for a body of data is one that minimizes the size of the theory plus the amount of information necessary to specify the exceptions relative to the theory—the smallest cartload of dung. In statistical estimation theory, this has been applied successfully to various parameter-fitting problems. It applies to machine learning as follows: given a set of instances, a learning method infers a theory—be it ever so simple; unworthy, perhaps, to be called a "theory"—from them. Using a metaphor of communication, imagine that the instances are to be transmitted through a noiseless channel. Any similarity that is detected among them can be exploited to give a more compact coding. According to the MDL principle, the best generalization is the one that minimizes the number of bits required to communicate the generalization, along with the examples from which it was made.

Now the connection with the informational loss function introduced in Section 5.6 should be starting to emerge. That function measures the error in terms of the number of bits required to transmit the instances, given the probabilistic predictions made by the theory. According to the MDL principle we need to add to this the "size" of the theory in bits, suitably encoded, to obtain an overall figure for complexity. However, the MDL principle refers to the information required to transmit the examples from which the theory was formed, that is, the *training* instances—not a test set. The overfitting problem is avoided because a complex theory that overfits will be penalized relative to a simple one by virtue of the fact that it takes more bits to encode. At one extreme is a very complex, highly overfitted theory that makes no errors on the training set. At the other is a very simple theory—the null theory—which does not help at all when transmitting the training set. And in between are theories of intermediate complexity, which make probabilistic predictions that are imperfect and need to be corrected by transmitting some information about the training set. The MDL principle provides a means of comparing all these possibilities on an equal footing to see which is the best. We have found the holy grail: an evaluation scheme that works on the training set alone and does not need a separate test set. But the devil is in the details, as we will see.

Suppose a learning method comes up with a theory T, based on a training set E of examples, that requires a certain number of bits $\mathrm{L}[T]$ to encode (L for *length*). Given the theory, the training set itself can be encoded in a certain number of bits, $\mathrm{L}[E|T]$. $\mathrm{L}[E|T]$ is in fact given by the informational loss function summed over all members of the training set. Then the total description length of theory plus training set is

$$\mathrm{L}[T]+\mathrm{L}[E|T]$$

and the MDL principle recommends choosing the theory T that minimizes this sum.

There is a remarkable connection between the MDL principle and basic probability theory. Given a training set E, we seek the "most likely" theory T, that is, the theory for which the a posteriori probability $\Pr[T|E]$—the probability after the examples have been seen—is maximized. Bayes's rule of conditional probability, the same rule that we encountered in Section 4.2, dictates that

$$\Pr[T|E]=\frac{\Pr[E|T]\Pr[T]}{\Pr[E]}.$$

Taking negative logarithms,

$$-\log \Pr[T|E] = -\log \Pr[E|T] - \log \Pr[T] + \log \Pr[E].$$

Maximizing the probability is the same as minimizing its negative logarithm. Now (as we saw in Section 5.6) the number of bits required to code something is just the negative logarithm of its probability. Furthermore, the final term, log Pr[E], depends solely on the training set and not on the learning method. Thus choosing the theory that maximizes the probability Pr[$T|E$] is tantamount to choosing the theory that minimizes

$$\mathrm{L}[E|T] + \mathrm{L}[T]$$

—in other words, the MDL principle!

This astonishing correspondence with the notion of maximizing the a posteriori probability of a theory after the training set has been taken into account gives credence to the MDL principle. But it also points out where the problems will sprout when the MDL principle is applied in practice. The difficulty with applying Bayes's rule directly is in finding a suitable prior probability distribution Pr[T] for the theory. In the MDL formulation, that translates into finding how to code the theory T into bits in the most efficient way. There are many ways of coding things, and they all depend on presuppositions that must be shared by encoder and decoder. If you know in advance that the theory is going to take a certain form, you can use that information to encode it more efficiently. How are you going to actually encode T? The devil is in the details.

Encoding E with respect to T to obtain L[$E|T$] seems a little more straightforward: we have already met the informational loss function. But actually, when you encode one member of the training set after another, you are encoding a *sequence* rather than a *set.* It is not necessary to transmit the training set in any particular order, and it ought to be possible to use that fact to reduce the number of bits required. Often, this is simply approximated by subtracting log n! (where n is the number of elements in E), which is the number of bits needed to specify a particular permutation of the training set (and because this is the same for all theories, it doesn't actually affect the comparison between them). But one can imagine using the frequency of the individual errors to reduce the number of bits needed to code them. Of course, the more sophisticated the method that is used to code the errors, the less the need for a theory in the first place—so whether a theory is justified or not depends to some extent on how the errors are coded. The details, the details.

We will not go into the details of different coding methods here. The whole question of using the MDL principle to evaluate a learning scheme based solely on the training data is an area of active research and vocal disagreement among researchers.

We end this section as we began, on a philosophical note. It is important to appreciate that Occam's razor, the preference of simple theories over complex ones, has the status of a philosophical position or "axiom" rather than something that can be proved from first principles. Although it may seem self-evident to us, this is a function of our education and the times we live in. A preference for simplicity is—or may be—culture specific rather than absolute.

The Greek philosopher Epicurus (who enjoyed good food and wine and supposedly advocated sensual pleasure—in moderation—as the highest good) expressed almost the opposite sentiment. His *principle of multiple explanations* advises "if more than one theory is consistent with the data, keep them all" on the basis that if several explanations are equally in agreement, it may be possible to achieve a higher degree of precision by using them together—and anyway, it would be unscientific to discard some arbitrarily. This brings to mind instance-based learning, in which all the evidence is retained to provide robust predictions, and resonates strongly with decision combination methods such as bagging and boosting (described in Chapter 7) that actually do gain predictive power by using multiple explanations together.

5.10 Applying the MDL principle to clustering

One of the nice things about the MDL principle is that unlike other evaluation criteria, it can be applied under widely different circumstances. Although in some sense equivalent to Bayes's rule in that, as we saw previously, devising a coding scheme for theories is tantamount to assigning them a prior probability distribution, schemes for coding are somehow far more tangible and easier to think about in concrete terms than intuitive prior probabilities. To illustrate this we will briefly describe—without entering into coding details—how you might go about applying the MDL principle to clustering.

Clustering seems intrinsically difficult to evaluate. Whereas classification or association learning has an objective criterion of success—predictions made on test cases are either right or wrong—this is not so with clustering. It seems that the only realistic evaluation is whether the result of learning—the clustering—proves useful in the application context. (It is worth pointing out that really this is the case for all types of learning, not just clustering.)

Despite this, clustering can be evaluated from a description length perspective. Suppose a cluster-learning technique divides the training set E into k clusters. If these clusters are natural ones, it should be possible to use them to encode E more efficiently. The best clustering will support the most efficient encoding.

One way of encoding the instances in E with respect to a given clustering is to start by encoding the cluster centers—the average value of each attribute over all instances in the cluster. Then, for each instance in E, transmit which cluster it belongs to (in $\log_2 k$ bits) followed by its attribute values with respect to the cluster center—perhaps as the numeric difference of each attribute value from the center. Couched as it is in terms of averages and differences, this description presupposes numeric attributes and raises thorny questions about how to code numbers efficiently. Nominal attributes can be handled in a similar manner: for each cluster there is a probability distribution for the attribute values, and the distributions are different for different clusters. The coding issue becomes more straightforward: attribute values are coded with respect to the relevant probability distribution, a standard operation in data compression.

If the data exhibits extremely strong clustering, this technique will result in a smaller description length than simply transmitting the elements of *E* without any clusters. However, if the clustering effect is not so strong, it will likely increase rather than decrease the description length. The overhead of transmitting cluster-specific distributions for attribute values will more than offset the advantage gained by encoding each training instance relative to the cluster it lies in. This is where more sophisticated coding techniques come in. Once the cluster centers have been communicated, it is possible to transmit cluster-specific probability distributions adaptively, in tandem with the relevant instances: the instances themselves help to define the probability distributions, and the probability distributions help to define the instances. We will not venture further into coding techniques here. The point is that the MDL formulation, properly applied, may be flexible enough to support the evaluation of clustering. But actually doing it satisfactorily in practice is not easy.

5.11 Further reading

The statistical basis of confidence tests is well covered in most statistics texts, which also give tables of the normal distribution and Student's distribution. (We use an excellent course text, Wild and Seber 1995, which we recommend very strongly if you can get hold of it.) "Student" is the nom de plume of a statistician called William Gosset, who obtained a post as a chemist in the Guinness brewery in Dublin, Ireland, in 1899 and invented the *t*-test to handle small samples for quality control in brewing. The corrected resampled *t*-test was proposed by Nadeau and Bengio (2003). Cross-validation is a standard statistical technique, and its application in machine learning has been extensively investigated and compared with the bootstrap by Kohavi (1995a). The bootstrap technique itself is thoroughly covered by Efron and Tibshirani (1993).

The Kappa statistic was introduced by Cohen (1960). Ting (2002) has investigated a heuristic way of generalizing to the multiclass case the algorithm given in Section 5.7 to make two-class learning schemes cost sensitive. Lift charts are described by Berry and Linoff (1997). The use of ROC analysis in signal detection theory is covered by Egan (1975); this work has been extended for visualizing and analyzing the behavior of diagnostic systems (Swets 1988) and is also used in medicine (Beck and Schultz 1986). Provost and Fawcett (1997) brought the idea of ROC analysis to the attention of the machine learning and data mining community. Witten et al. (1999b) explain the use of recall and precision in information retrieval systems; the F-measure is described by van Rijsbergen (1979). Drummond and Holte (2000) introduced cost curves and investigated their properties.

The MDL principle was formulated by Rissanen (1985). Kepler's discovery of his economical three laws of planetary motion, and his doubts about them, are recounted by Koestler (1964).

Epicurus's principle of multiple explanations is mentioned by Li and Vityani (1992), quoting from Asmis (1984).