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 schemes 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 are 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, which 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 structures 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 just sketched leads to a different machine learning scheme that is well suited to discovering the underlying concept. The sections of this chapter look at each of these structures in turn. A final

Copyright © 2011 Elsevier Inc. All rights of reproduction in any form reserved.

section introduces simple ways of dealing with multi-instance problems, where each example comprises several different instances.

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.

To see the 1R method at work, consider the weather data of Table 1.2 on page 10 (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

```
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 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 has been made between two equally likely outcomes.*

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 scheme, 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 for each of *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, page 11) 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 the following 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 an excessively 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 pieces, 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 get any examples outside the training set correct. This phenomenon is known as *overfitting*; we have already described overfitting-avoidance bias in Chapter 1, 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 minimum limit is imposed on the number of examples of the majority class in each partition. Suppose that minimum is set at 3. 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 of

```
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 *yes* had been chosen 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 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 discretization procedure is applied just to the instances for which the attribute's value is defined.

Discussion

In a seminal paper entitled "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. *Cross-validation*, an evaluation technique that we will explain in Chapter 5, was used to ensure that the results were the same as would be obtained on independent test sets. After some experimentation, the minimum number of examples in each partition of a numeric attribute was set at six, not three as used in our illustration.

Surprisingly, despite its simplicity 1R did well in comparison with the state-of-the-art learning schemes, 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 schemes, 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—that is, intervals and subsets—are learned from the training data pertaining to each of the classes. For a numeric attribute, the end points of the interval are the minimum and the 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 individual 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 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 (page 10) 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*. The lower part of the table contains the same information expressed as 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

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

Table 4.2 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	?

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' rule of conditional probability. Bayes' rule says that if you have a hypothesis H and evidence E that bears on that hypothesis, then

$$\Pr[H \mid E] = \frac{\Pr[E \mid H]\Pr[H]}{\Pr[E]}$$

We use the notation that $\Pr[A]$ denotes the probability of an event A and $\Pr[A \mid B]$ denotes the probability of A conditional on another event B. The hypothesis H is that *play* will be, say, *yes*, and $\Pr[H \mid E]$ 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:

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

Don't worry about the denominator: We will ignore it and eliminate it in the final normalizing step when we make the probabilities for *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 in question—and 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

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

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' 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 effectively when tested on actual datasets, particularly when combined with some of the attribute selection procedures, which are introduced in Chapter 7, that eliminate redundant, and hence nonindependent, attributes.

Things go badly awry in Naïve Bayes if a particular attribute value does not occur in the training set in conjunction with *every* class value. Suppose that in the training data the attribute value *outlook* = *sunny* was always associated with the outcome *no*. Then the probability of *outlook* = *sunny* being 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* in the previous example 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

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

The value of μ, which was set to 3 before, 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

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

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 1 instead of 0.

Missing Values and Numeric Attributes

One of the really nice things about Naïve Bayes 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

$$\text{Likelihood of } yes = 3/9 \times 3/9 \times 3/9 \times 9/14 = 0.0238$$

$$\text{Likelihood of } no = 1/5 \times 4/5 \times 3/5 \times 5/14 = 0.0343$$

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 calculate counts as before, while for numeric ones we simply list the values that occur. Then, instead of normalizing counts into probabilities as we do for nominal attributes, we calculate the mean and the standard deviation for each class and each numeric attribute. 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 values—that is, the sum divided by the number of values. The standard deviation is the square root of the sample variance, which we 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," take its square root to yield the standard deviation. This is the standard way of calculating the mean and the standard deviation of a set of numbers. (The "one less than" has to do with the number of degrees of freedom in the sample, a statistical notion that we don't want to get into here.)

Table 4.4 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											

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 of, say, 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

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

And by the same token, the probability density of a *yes* outcome when *humidity* has a value of, say, 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 $\varepsilon \times f(x)$. You might think we ought to factor in the accuracy figure ε when using these density values, 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

$$\text{Likelihood of } yes = 2/9 \times 0.0340 \times 0.0221 \times 3/9 \times 9/14 = 0.000036$$

$$\text{Likelihood of } no = 3/5 \times 0.0279 \times 0.0381 \times 3/5 \times 5/14 = 0.000137$$

which leads to probabilities

$$\text{Probability of } yes = \frac{0.000036}{0.000036 + 0.000137} = 20.8\%$$

Table 4.5 Another New Day

Outlook	Temperature	Humidity	Windy	Play
Sunny	66	90	true	?

$$\text{Probability of } no = \frac{0.000137}{0.000036 + 0.000137} = 79.2\%$$

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.

Naïve Bayes for Document Classification

An 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 sports. 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 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 called *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 $\Pr[E \mid H]$ in Bayes' rule—is

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

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 could 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, 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 two words, *yellow* and *blue*, in the vocabulary, and a particular document class H has Pr[*yellow* | H] = 75% and Pr[*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

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

The other three, with their probabilities, are

$$\Pr[\{\textit{blue blue blue}\} \mid H] = \frac{1}{64}$$
$$\Pr[\{\textit{yellow yellow blue}\} \mid H] = \frac{27}{64}$$
$$\Pr[\{\textit{yellow blue blue}\} \mid H] = \frac{9}{64}$$

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* | H'] = 10% and Pr[*blue* | 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' 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 14 to 24% disparity and tip the balance in favor of the *yellowish green* class.

The factorials in the 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. It can achieve impressive results. People often find that Naïve Bayes rivals, and indeed outperforms, more sophisticated classifiers on many datasets. The moral is, always try the simple things first. Over and over again people have eventually, after an extended struggle, managed to obtain good results using sophisticated learning schemes, only to discover 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 well, however, and it is easy to see why. Because attributes are treated as though they were independent given the class, 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, 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—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 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 the trees in Figure 4.2.

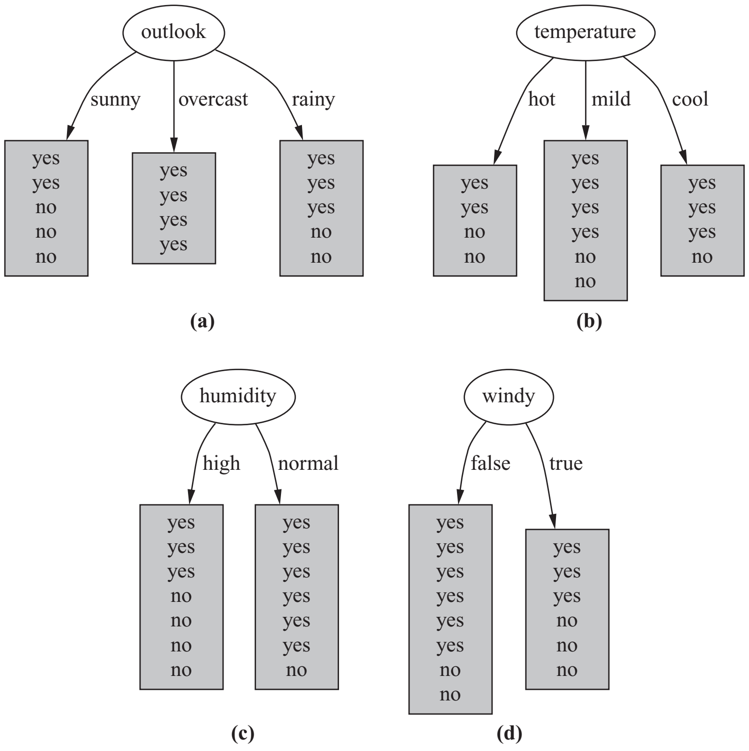


FIGURE 4.2

Tree stumps for the weather data: (a) outlook, (b) temperature, (c) humidity, and (d) windy.

Which is the best choice? The number of *yes* and *no* classes is 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 each 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 1! It is calculated 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 number 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

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

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

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

We 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:

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

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 any of the nascent tree structures in Figure 4.2 were created, 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

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

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 split on the one that gains the most information. In the situation that is shown in Figure 4.2:

- gain(*outlook*) = 0.247 bits
- gain(*temperature*) = 0.029 bits
- gain(*humidity*) = 0.152 bits
- gain(*windy*) = 0.048 bits

Therefore, 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

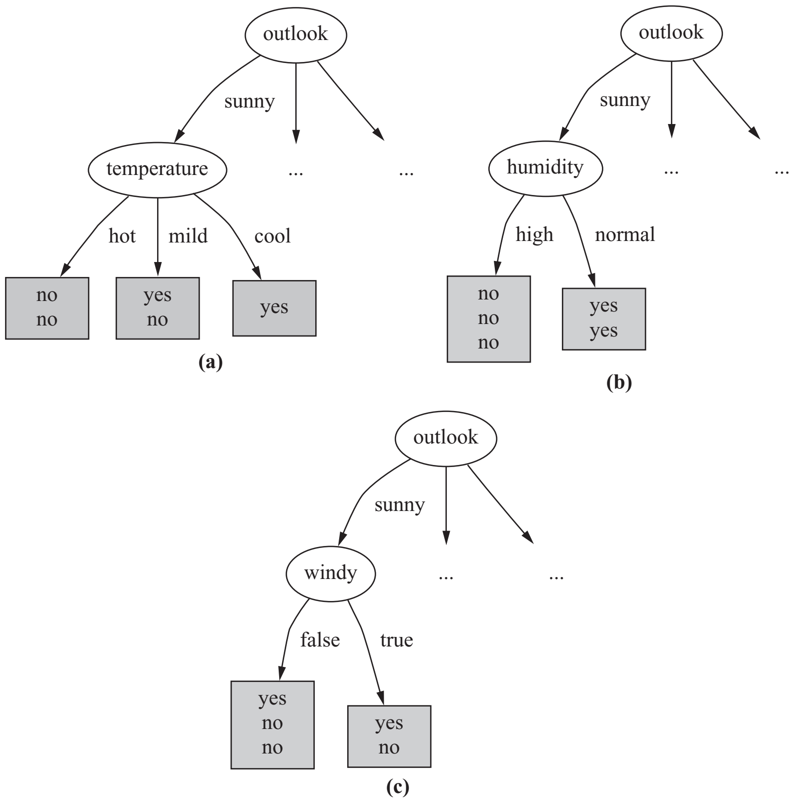


FIGURE 4.3

Expanded tree stumps for the weather data: (a) temperature, (b) humidity, and (c) windy.

outlook will produce nothing new, so we only consider the other three attributes. The information gain for each turns out to be

- gain(*temperature*) = 0.571 bits
- gain(*humidity*) = 0.971 bits
- gain(*windy*) = 0.020 bits

Therefore, 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. Alternatively, one could stop if the information gain is zero. This is slightly more conservative because

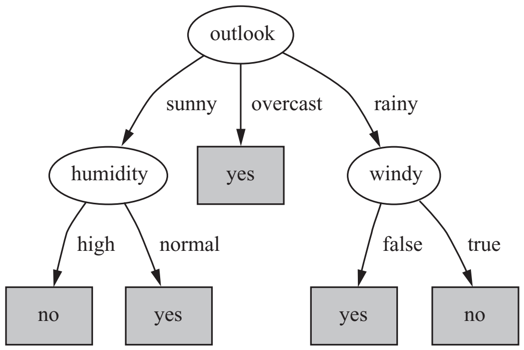


FIGURE 4.4

Decision tree for the weather data.

it is possible to encounter cases where the data can be split into subsets exhibiting identical class distributions, which would make the information gain zero.

Calculating Information

Now it is time to explain how to calculate the information measure that is used as the 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

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 should hold regardless of the actual values. Thus, we could add a further criterion to the list above:

3. The information should obey the multistage property that we have illustrated.

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*:

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

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

The arguments p_1, p_2, … of the entropy formula are expressed as fractions that add up to 1, 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

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

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 node of Figure 4.2(a) is

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

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 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 0 because each of the 14 terms is 0. 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

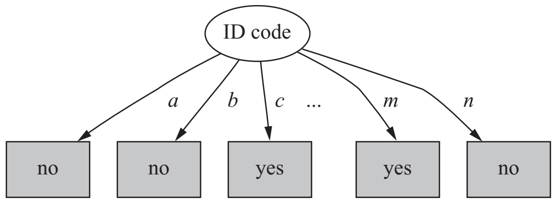


FIGURE 4.5

Tree stump for the *ID code* attribute.

Table 4.6 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

the number of bits needed to determine to which branch each instance is assigned, and the more branches there are, the greater this value. 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 greater 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 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 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 at the University of Sydney in Australia. Although others have

Table 4.7 Gain Ratio Calculations for Figure 4.2 Tree Stumps

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])	1.362	split info: info([7,7])	1.000	split info: info([8,6])	0.985
gain ratio: 0.247/1.577	0.156	gain ratio: 0.029/1.557	0.019	gain ratio: 0.152/1	0.152	gain ratio: 0.048/0.985	0.049

worked on similar methods, Quinlan's research has always been at the very forefront of decision tree induction. The scheme 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 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 top-down, seeking at each stage an attribute to split on that best separates the classes, and 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 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 it 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 needed, 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
```

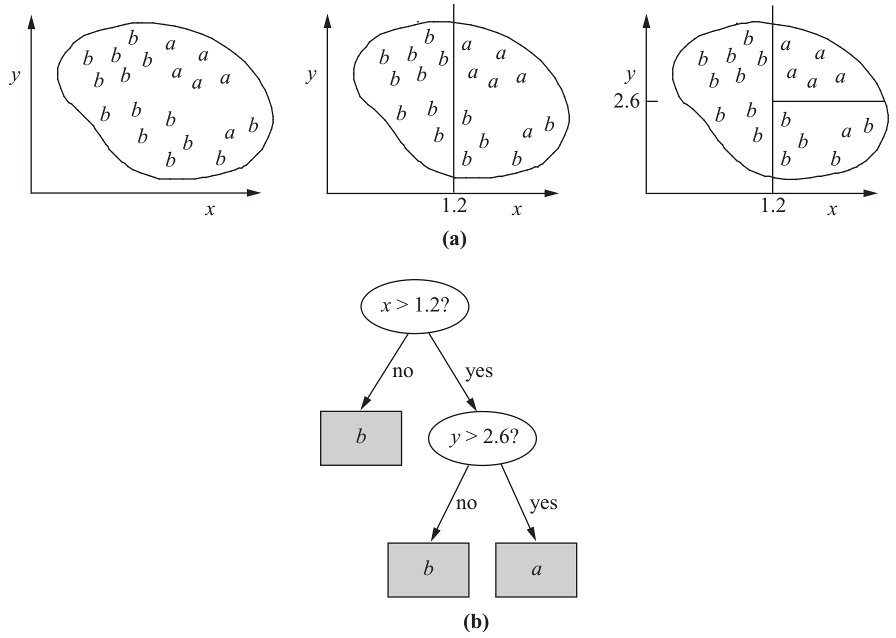


FIGURE 4.6

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

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 be needed 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 3.4, 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 in 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 between 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 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 (page 6). 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
```

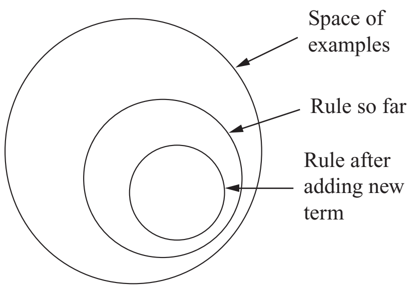


FIGURE 4.7

The instance space during operation of a covering algorithm.

```
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 8 instances, 2 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 (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 list, and create the rule:

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

This rule is quite inaccurate, 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
```

Considering the possibilities for the unknown term, `?` yields the following 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 4 instances correct out of the 6 that it covers, and it 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
```

It is necessary for us 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 selects

Table 4.8 Part of 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

Table 4.9 Part of 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

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 out 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 one out of 7 the reason for the 7 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 two of the original set of instances, one of which is 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

```
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.

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 around. 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, we 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 we choose the most promising test—that is, the one that maximizes the accuracy of the rule. Finally, we 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 last line 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 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 in any order.

As described in Section 3.4, order-independent rules seem to provide more modularity by 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, it may satisfy rules that apply to different classes. Other test examples may receive no classification at all. A simple strategy to force a decision in 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 will 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 with the process on those that remain. This contrasts with the divide-and-conquer approach of decision trees. The "separate" step results in an efficient 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 side of the rule. However, not only might any attribute occur on the right side with any possible value, but 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 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 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 in Table 1.2 (page 10), 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 the one-item sets. 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 six 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 rows of the table, for example, show that there are five days when *outlook* = *sunny*, two of which have *temperature* = *hot*, and, in fact, on both of those days *humidity* = *high* and *play* = *no* as well.

Table 4.10 Item Sets for 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		

Continued

Table 4.10 Item Sets for Weather Data with Coverage or Greater (*Continued*)

	One-Item Sets		Two-Item Sets		Three-Item Sets		Four-Item Sets	
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		
…			…		…			
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				

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 a 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:

```
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    4/14
```

The figures at the right in this list 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 (although 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
```

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).

Table 4.11 Association Rules for 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%

Table 4.11 *Continued*

	Association Rule	Coverage	Accuracy

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%

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 the 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 with the same first two members. 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 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 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 we have described 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 at the same time. 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 must be made through the dataset. In many situations we would like 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 of the coverage values and repeating until the desired number of rules has been generated.

The tabular input format that we use throughout this book, and in particular the standard ARFF format based on it, is very inefficient for many association-rule problems. Association rules are often used in situations where 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, namely the linear models introduced in Section 3.2; we examine them in more detail here. They can form components of more complex learning methods, which we will investigate 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:

$$x = w_0 + w_1 a_1 + w_2 a_2 + \ldots + w_k a_k$$

where x is the class; a_1, a_2, …, a_k are the attribute values; and w_0, w_1, …, 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 $a_1^{(1)}$, $a_2^{(1)}$, … , $a_k^{(1)}$, where the superscript denotes that it is the first example. Moreover, it is notationally convenient to assume an extra attribute a_0, with a value that is always 1.

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

$$w_0 a_0^{(1)} + w_1 a_1^{(1)} + w_2 a_2^{(1)} + \ldots + w_k a_k^{(1)} = \sum_{j=0}^{k} w_j a_j^{(1)}$$

This is the predicted, not the actual, value for the class. Of interest is the difference between the predicted and 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

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

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 can be used 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 3.4(a) (page 68). 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 1 for training instances that belong to the class and 0 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 scheme 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 blatently 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

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

which cannot be approximated accurately using a linear function, by

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

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

$$\Pr[1 \mid a_1, a_2, \ldots, a_k] = 1/(1 + \exp(-w_0 - w_1 a_1 - \ldots - w_k a_k))$$

with weights w. Figure 4.9(b) shows an example of this function in one dimension, with two weights $w_0 = -1.25$ and $w_1 = 0.5$.

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

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

where the $x^{(i)}$ are either 0 or 1.

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 above for multiresponse linear regression by performing logistic regression independently for each class. Unfortunately, the resulting probability estimates will not sum to 1. 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.

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:

$$\Pr[1 \mid a_1, a_2, \ldots, a_k] = 1/(1 + \exp(-w_0 - w_1 a_1 - \ldots - w_k a_k)) = 0.5$$

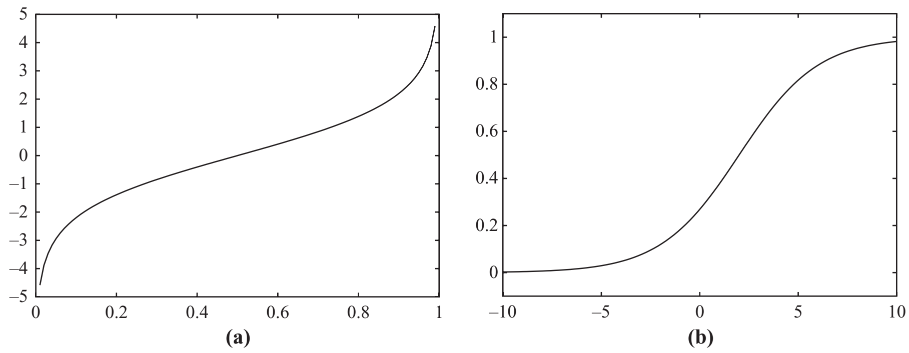


FIGURE 4.9

Logistic regression: (a) the logit transformation and (b) example logistic regression function.

This occurs when

$$-w_0 - w_1 a_1 - \ldots - w_\text{k} a_\text{k} = 0$$

Because this is a linear equality in the attribute values, the boundary is a 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

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

This is a linear inequality in the attribute values, so the boundary between each pair of classes is a hyperplane.

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, $a_1, a_2, \ldots, a_k$ are the attribute values, and $w_0, w_1, \ldots, w_k$ are the weights that define the hyperplane. We will assume that each training instance $a_1, a_2, \ldots$ 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 means that we don't have to include an additional constant element in the sum. If the sum is greater than 0, 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 that the output for a has increased by

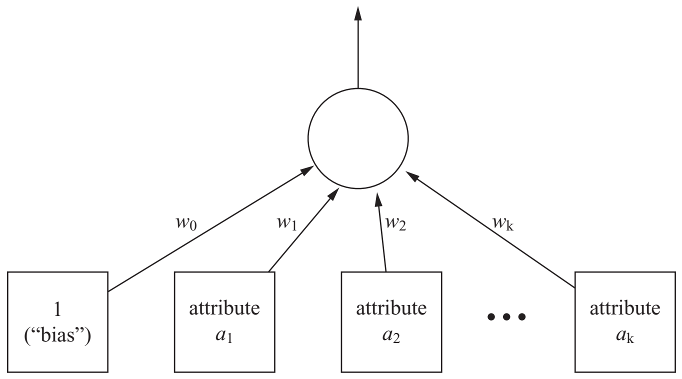
$$a_0 \times a_0 + a_1 \times a_1 + a_2 \times a_2 + \ldots + a_k \times a_k$$

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 in 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.

```
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.

The resulting hyperplane is called a *perceptron*, and it's the grandfather of neural networks (we return to neural networks in Section 6.4). 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 1. 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*, which is illustrated in Figure 4.11(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 $\alpha$
        (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
          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) unbalanced version and (b) balanced version.

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 a user-specified parameter α (or its inverse). The attribute values a_i are either 0 or 1 because we 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 1, and the w_i are set to a constant at the start.

The algorithm we have described doesn't allow for 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

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

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.

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

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

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 often measured on different scales, so if the Euclidean distance formula were used directly, the effect 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 1, whereas if the values are the same the difference is 0. 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 1; the difference is 0 only if they are not missing and both are the same. For numeric attributes, the difference between two missing values is also taken as 1. However, if just one value is missing, the difference is often taken as either the (normalized) size of the other value or 1 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 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, with *k* being the number of attributes.

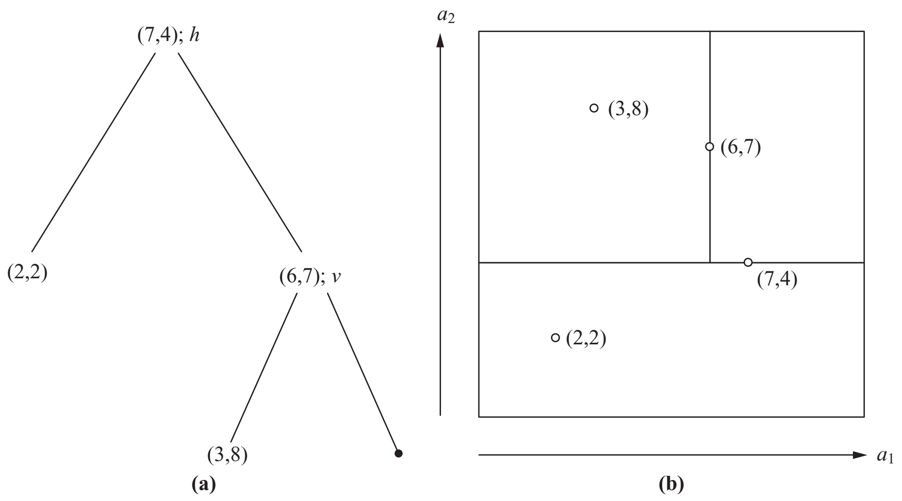


FIGURE 4.12

A *kD*-tree for four training instances: (a) the tree and (b) instances and splits.

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 right child is empty, and its left 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.

How do you build a *kD*-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

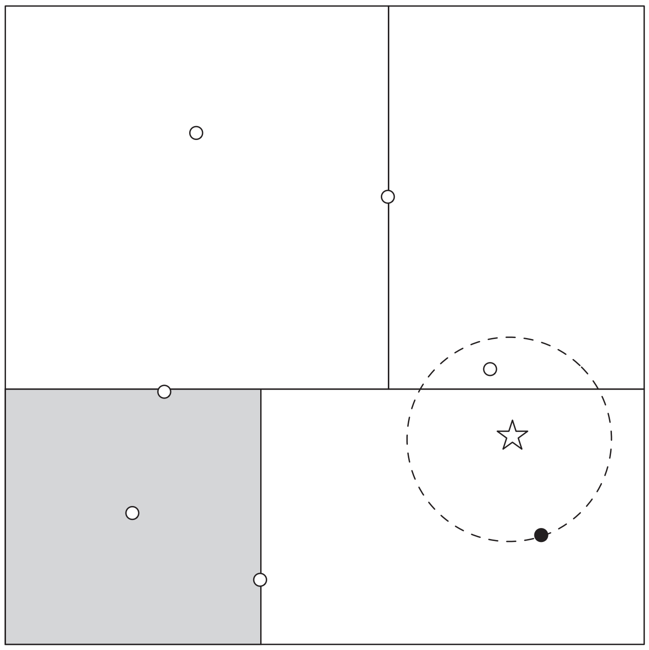


FIGURE 4.13

Using a *kD*-tree to find the nearest neighbor of the star.

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 if 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, $\log_2 n$ if the tree is well balanced. 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 with nodes that are approximately square rather than long skinny rectangles, it can also be shown to be logarithmic in the number of nodes (if the number of attributes in the dataset is not too large).

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 *kD*-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 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 a 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, *kD*-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 important, 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 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 *kD*-trees 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

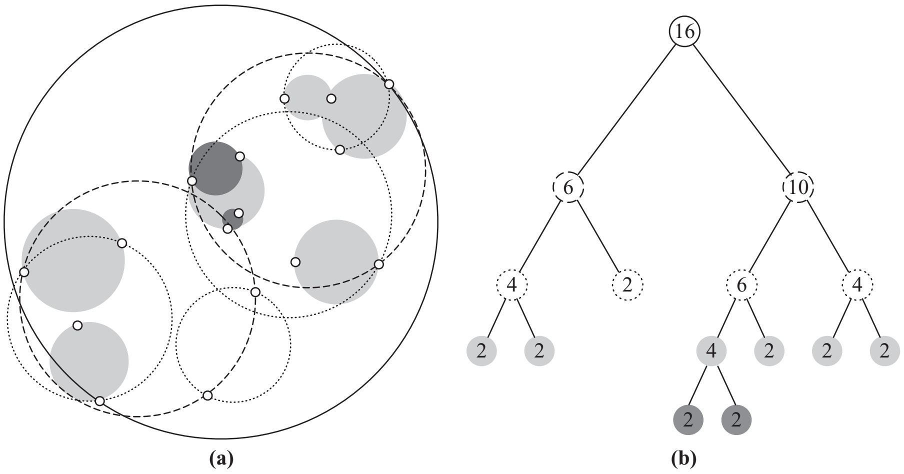


FIGURE 4.14

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

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 *kD*-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 *kD*-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 *kD*-trees. Here is one possible splitting method. Choose the point in the ball that

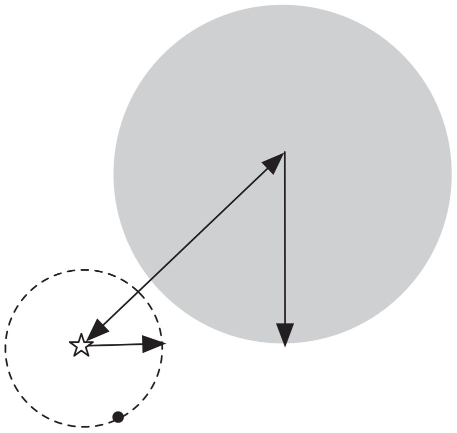


FIGURE 4.15

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

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 scheme we have described, 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 earlier we used *k* to denote the number of attributes; 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, which are 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 scheme in the early 1960s and has been widely used in the field of pattern recognition for almost half a century.

Nearest-neighbor classification was notoriously slow until *kD*-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 they 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 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 the instance 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 the instances are to be divided into natural groups. These clusters presumably reflect some mechanism that is 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 that we have considered so far.

As we saw in Section 3.6, there are different ways in which the result of clustering can be expressed. The groups that are identified may be exclusive: Any instance belongs in only one group. Or they may be overlapping: An instance may fall into several groups. Or they may be probabilistic: An instance belongs to each group with a certain probability. Or they may be hierarchical: A rough division of instances into groups at the top level and each group 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 works 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 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.

k-means clustering can be dramatically improved by careful choice of the initial cluster centers, often called *seeds*. Instead of beginning with an arbitrary set of seeds, here is a better procedure. Choose the initial seed at random from the entire space, with a uniform probability distribution. Then choose the second seed with a probability that is proportional to the square of the distance from the first. Proceed, at each stage choosing the next seed with a probability proportional to the square of the distance from the closest seed that has already been chosen. This procedure, called *k-means*++, improves both speed and accuracy over the original algorithm with random seeds.

Faster Distance Calculations

The *k*-means clustering algorithm usually requires several iterations, each involving finding the distance of the *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—*kD*-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 *kD*-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 in 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 0. 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 not descend any 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 we need not descend any 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.

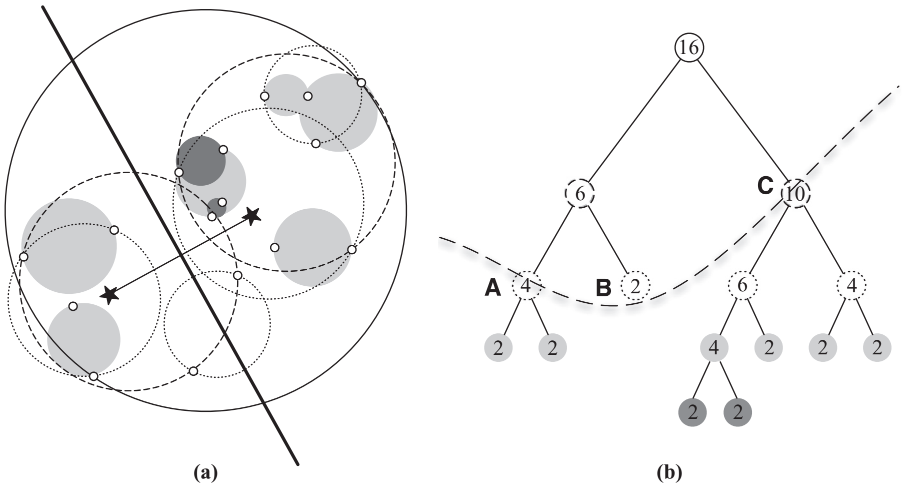


FIGURE 4.16

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

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 6.8.

4.9 MULTI-INSTANCE LEARNING

In Chapter 2 we introduced *multi-instance* learning, where each example in the data comprises several different instances. We call these examples *bags* (we noted the difference between bags and sets in Section 4.2). In supervised multi-instance learning, a class label is associated with each bag, and the goal of learning is to determine how the class can be inferred from the instances that make up the bag. While advanced algorithms have been devised to tackle such problems, it turns out that the simplicity-first methodology can be applied here with surprisingly good results. A simple but effective approach is to manipulate the input data to transform it into a single-instance learning problem and then apply standard learning methods, such as the ones described in this chapter. Two such approaches are described in the following sections.

Aggregating the Input

You can convert a multiple-instance problem to a single-instance one by calculating values such as mean, mode, minimum, and maximum that summarize the instances in the bag and adding these as new attributes. Each "summary" instance retains the class label of the bag it was derived from. To classify a new bag the same process is used: A single aggregated instance is created with attributes that summarize the instances in the bag. Surprisingly, for the original drug activity dataset that spurred the development of multi-instance learning, results comparable with special-purpose multi-instance learners can be obtained using just the minimum and maximum values of each attribute for each bag, combined with a support vector machine classifier (see Chapter 6). One potential drawback of this approach is that the best summary statistics to compute depend on the problem at hand. However, the additional computational cost associated with exploring combinations of different summary statistics is offset by the fact that the summarizing process means that fewer instances are processed by the learning algorithm.

Aggregating the Output

Instead of aggregating the instances in each bag, another approach is to learn a classifier directly from the original instances that comprise the bag. To achieve this, the instances in a given bag are all assigned the bag's class label. At classification time, a prediction is produced for each instance in the bag to be predicted, and the predictions are aggregated in some fashion to form a prediction for the bag as a whole. One approach is to treat the predictions as votes for the various class labels. If the classifier is capable of assigning probabilities to the class labels, these could be averaged to yield an overall probability distribution for the bag's class label. This method treats the instances independently and gives them equal influence on the predicted class label.

One problem is that the bags in the training data can contain different numbers of instances. Ideally, each bag should have the same influence on the final model that is learned. If the learning algorithm can accept instance-level weights, this can be achieved by assigning each instance in a given bag a weight inversely proportional to the bag's size. If a bag contains n instances, giving each one a weight of $1/n$ ensures that the instances contribute equally to the bag's class label and each bag receives a total weight of 1.

Discussion

Both methods described previously for tackling multi-instance problems disregard the original multi-instance assumption that a bag is positive if and only if at least one of its instances is positive. Instead, making each instance in a bag contribute equally to its label is the key element that allows standard learning algorithms to be applied. Otherwise, it is necessary to try to identify the "special" instances that are the key to determining the bag's label.

4.10 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 schemes at the time and that putting high-powered inductive inference schemes 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 scheme that generates one simple rule per class is due to 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.

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' 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 see in Chapter 5. With a particular dataset, prior probabilities for Naïve Bayes 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 efforts 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., 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 useful for text classification, was investigated by McCallum and Nigam (1998).

The classic paper on decision tree induction is 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 Lawson and Hanson (1995). The use of linear models for classification enjoyed a great deal of popularity in the 1960s; Nilsson (1965) is 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 that 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 6.4. The Winnow algorithms were introduced by Nick Littlestone in his Ph.D. thesis (Littlestone, 1988, 1989). Multiresponse linear classifiers have found application in an operation called *stacking* that combines the output of other learning algorithms, described in Chapter 8 (see Wolpert, 1992).

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 $k/n \to 0$. 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 Ph.D. thesis (Moore, 1991). Moore, 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 interval method mentioned in the Discussion section 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., Hartigan, 1975). The *k*-means++ variant, which yields a significant improvement by choosing the initial seeds more carefully, was introduced as recently as 2007 by Arthur and Vassilvitskii (2007). The clever use of *kD*-trees to speed up *k*-means clustering, which we have chosen to illustrate using ball trees instead, was pioneered by Moore and Pelleg (2000) in their *X*-means clustering algorithm. That algorithm contains some other innovations, described in Section 6.8.

The method of dealing with multi-instance learning problems by applying standard single-instance learners to summarized bag-level data was applied in conjunction with support vector machines by Gärtner et al. (2002). The alternative approach of aggregating the output is explained by Frank and Xu (2003).

4.11 WEKA IMPLEMENTATIONS

For classifiers, see Section 11.4 and Table 11.5.

- Inferring rudimentary rules: *OneR, HyperPipes* (learns one rule per class)
- Statistical modeling:
  - *NaïveBayes* and many variants, including *NaiveBayesMultinomial*
- Decision trees: *Id3*
- Decision rules: *Prism*
- Association rules (see Section 11.7 and Table 11.8): *a priori*
- Linear models:
  - SimpleLinearRegression, LinearRegression, Logistic (regression)
  - VotedPerceptron, Winnow
- Instance-based learning: *IB1*, *VFI* (voting feature intervals)
- Clustering (see Section 11.6 and Table 11.7): *SimpleKMeans*
- Multi-instance learning: *SimpleMI*, *MIWrapper*