CHAPTER

Implementations: Real Machine Learning Schemes

6

We have seen the basic ideas of several machine learning methods and studied in detail how to assess their performance on practical data mining problems. Now we are well prepared to look at real, industrial-strength, machine learning algorithms. Our aim is to explain these algorithms both at a conceptual level and with a fair amount of technical detail so that you can understand them fully and appreciate the key implementation issues that arise.

In truth, there is a world of difference between the simplistic methods described in Chapter 4 and the actual algorithms that are widely used in practice. The principles are the same. So are the inputs and outputs—methods of knowledge representation. But the algorithms are far more complex, principally because they have to deal robustly and sensibly with real-world problems such as numeric attributes, missing values, and—most challenging of all—noisy data. To understand how the various schemes cope with noise, we will have to draw on some of the statistical knowledge that we learned in Chapter 5.

Chapter 4 opened with an explanation of how to infer rudimentary rules and then examined statistical modeling and decision trees. Then we returned to rule induction and continued with association rules, linear models, the nearest-neighbor method of instance-based learning, and clustering. This chapter develops all these topics.

We begin with decision tree induction and work up to a full description of the C4.5 system, a landmark decision tree program that is probably the machine learning workhorse most widely used in practice to date. Then we describe decision rule induction. Despite the simplicity of the idea, inducing decision rules that perform comparably with state-of-the-art decision trees turns out to be quite difficult in practice. Most high-performance rule inducers find an initial rule set and then refine it using a rather complex optimization stage that discards or adjusts individual rules to make them work better together. We describe the ideas that underlie rule learning in the presence of noise and then go on to cover a scheme that operates by forming partial decision trees, an approach that has been demonstrated to perform well while avoiding complex and ad hoc heuristics. Following this, we take a brief look at how to generate rules with exceptions, which were described in Section 3.4, and examine fast data structures for learning association rules.

There has been a resurgence of interest in linear models with the introduction of *support vector machines*, a blend of linear modeling and instance-based learning.

Support vector machines select a small number of critical boundary instances called *support vectors* from each class and build a linear discriminant function that separates them as widely as possible. This instance-based approach transcends the limitations of linear boundaries by making it practical to include extra nonlinear terms in the function, making it possible to form quadratic, cubic, and higher-order decision boundaries. The same techniques can be applied to the perceptron described in Section 4.6 to implement complex decision boundaries, and also to least squares regression. An older technique for extending the perceptron is to connect units together into multilayer "neural networks." All of these ideas are described in Section 6.4.

Section 6.5 describes classic instance-based learners, developing the simple nearest-neighbor method introduced in Section 4.7 and showing some more powerful alternatives that perform explicit generalization. Following that we extend linear regression for numeric prediction to a more sophisticated procedure that comes up with the tree representation introduced in Section 3.3 and go on to describe locally weighted regression, an instance-based strategy for numeric prediction. Then we examine Bayesian networks, a potentially very powerful way of extending the Naïve Bayes method to make it less "naïve" by dealing with datasets that have internal dependencies. Next we return to clustering and review some methods that are more sophisticated than simple *k*-means, methods that produce hierarchical clusters and probabilistic clusters. We also look at semi-supervised learning, which can be viewed as combining clustering and classification. Finally, we discuss more advanced schemes for multi-instance learning than those covered in Section 4.9.

Because of the nature of the material it contains, this chapter differs from the others in the book. Sections can be read independently, and each is self-contained, including the references to further reading, which are gathered together in Discussion sections.

6.1 DECISION TREES

The first machine learning scheme that we will develop in detail, the C4.5 algorithm, derives from the simple divide-and-conquer algorithm for producing decision trees that was described in Section 4.3. It needs to be extended in several ways before it is ready for use on real-world problems. First, we consider how to deal with numeric attributes and, after that, missing values. Then we look at the all-important problem of pruning decision trees, because although trees constructed by the divide-and-conquer algorithm as described perform well on the training set, they are usually overfitted to the training data and do not generalize well to independent test sets. We then briefly consider how to convert decision trees to classification rules and examine the options provided by the C4.5 algorithm itself. Finally, we look at an alternative pruning strategy that is implemented in the famous CART system for learning classification and regression trees.

Numeric Attributes

The method we described in Section 4.3 only works when all the attributes are nominal, whereas, as we have seen, most real datasets contain some numeric attributes. It is not too difficult to extend the algorithm to deal with these. For a numeric attribute we will restrict the possibilities to a two-way, or binary, split. Suppose we use the version of the weather data that has some numeric features (see Table 1.3). Then, when temperature is being considered for the first split, the temperature values involved are

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

Repeated values have been collapsed together, and there are only 11 possible positions for the breakpoint—8 if the breakpoint is not allowed to separate items of the same class. The information gain for each can be calculated in the usual way. For example, the test *temperature* < 71.5 produces four *yes*'s and two *no*'s, whereas *temperature* > 71.5 produces five *yes*'s and three *no*'s, and so the information value of this test is

$$\text{info}([4, 2], [5, 3]) = (6/14) \times \text{info}([4, 2]) + (8/14) \times \text{info}([5, 3]) = 0.939 \text{ bits}$$

It is common to place numeric thresholds halfway between the values that delimit the boundaries of a concept, although something might be gained by adopting a more sophisticated policy. For example, we will see in the following that although the simplest form of instance-based learning puts the dividing line between concepts in the middle of the space between them, other methods that involve more than just the two nearest examples have been suggested.

When creating decision trees using the divide-and-conquer method, once the first attribute to split on has been selected, a top-level tree node is created that splits on that attribute, and the algorithm proceeds recursively on each of the child nodes. For each numeric attribute, it appears that the subset of instances at each child node must be re-sorted according to that attribute's values—and, indeed, this is how programs for inducing decision trees are usually written. However, it is not actually necessary to re-sort because the sort order at a parent node can be used to derive the sort order for each child, leading to a speedier implementation. Consider the temperature attribute in the weather data, whose sort order (this time including duplicates) is

64	65	68	69	70	71	72	72	75	75	80	81	83	85
7	*6*	*5*	*9*	*4*	*14*	*8*	*12*	*10*	*11*	*2*	*13*	*3*	*1*

The italicized numbers below each temperature value give the number of the instance that has that value. Thus, instance number 7 has temperature value 64, instance 6 has temperature 65, and so on. Suppose we decide to split at the top level

on the attribute *outlook*. Consider the child node for which *outlook* = *sunny*—in fact, the examples with this value of *outlook* are numbers 1, 2, 8, 9, and 11. If the italicized sequence is stored with the example set (and a different sequence must be stored for each numeric attribute)—that is, instance 7 contains a pointer to instance 6, instance 6 points to instance 5, instance 5 points to instance 9, and so on—then it is a simple matter to read off the examples for which *outlook* = *sunny* in order. All that is necessary is to scan through the instances in the indicated order, checking the *outlook* attribute for each and writing down the ones with the appropriate value:

```
9     8     11     2     1
```

Thus, repeated sorting can be avoided by storing with each subset of instances the sort order for that subset according to each numeric attribute. The sort order must be determined for each numeric attribute at the beginning; no further sorting is necessary thereafter.

When a decision tree tests a nominal attribute as described in Section 4.3, a branch is made for each possible value of the attribute. However, we have restricted splits on numeric attributes to be binary. This creates an important difference between numeric attributes and nominal ones: Once you have branched on a nominal attribute, you have used all the information that it offers; however, successive splits on a numeric attribute may continue to yield new information. Whereas a nominal attribute can only be tested once on any path from the root of a tree to the leaf, a numeric one can be tested many times. This can yield trees that are messy and difficult to understand because the tests on any single numeric attribute are not located together but can be scattered along the path. An alternative, which is harder to accomplish but produces a more readable tree, is to allow a multiway test on a numeric attribute, testing against several different constants at a single node of the tree. A simpler but less powerful solution is to prediscretize the attribute as described in Section 7.2.

Missing Values

The next enhancement to the decision tree–building algorithm deals with the problems of missing values. Missing values are endemic in real-world datasets. As explained in Chapter 2 (page 58), one way of handling them is to treat them as just another possible value of the attribute; this is appropriate if the fact that the attribute is missing is significant in some way. In that case, no further action need be taken. But if there is no particular significance in the fact that a certain instance has a missing attribute value, a more subtle solution is needed. It is tempting to simply ignore all instances in which some of the values are missing, but this solution is often too draconian to be viable. Instances with missing values often provide a good deal of information. Sometimes the attributes with values that are missing play no part in the decision, in which case these instances are as good as any other.

One question is how to apply a given decision tree to an instance in which some of the attributes to be tested have missing values. We outlined a solution in Section

3.3 that involves notionally splitting the instance into pieces, using a numeric weighting scheme, and sending part of it down each branch in proportion to the number of training instances going down that branch. Eventually, the various parts of the instance will each reach a leaf node, and the decisions at these leaf nodes must be recombined using the weights that have percolated to the leaves. The information gain and gain ratio calculations described in Section 4.3 can also be applied to partial instances. Instead of having integer counts, the weights are used when computing both gain figures.

Another question is how to partition the training set once a splitting attribute has been chosen, to allow recursive application of the decision tree formation procedure on each of the daughter nodes. The same weighting procedure is used. Instances for which the relevant attribute value is missing are notionally split into pieces, one piece for each branch, in the same proportion as the known instances go down the various branches. Pieces of the instance contribute to decisions at lower nodes in the usual way through the information gain calculation, except that they are weighted accordingly. They may be further split at lower nodes, of course, if the values of other attributes are unknown as well.

Pruning

Fully expanded decision trees often contain unnecessary structure, and it is generally advisable to simplify them before they are deployed. Now it is time to learn how to prune decision trees.

By building the complete tree and pruning it afterward we are adopting a strategy of *postpruning* (sometimes called *backward pruning*) rather than *prepruning* (or *forward pruning*). Prepruning would involve trying to decide during the tree-building process when to stop developing subtrees—quite an attractive prospect because that would avoid all the work of developing subtrees only to throw them away afterward. However, postpruning does seem to offer some advantages. For example, situations occur in which two attributes individually seem to have nothing to contribute but are powerful predictors when combined—a sort of combination-lock effect in which the correct combination of the two attribute values is very informative but the attributes taken individually are not. Most decision tree builders postprune; however, prepruning can be a viable alternative when runtime is of particular concern.

Two rather different operations have been considered for postpruning: *subtree replacement* and *subtree raising*. At each node, a learning scheme might decide whether it should perform subtree replacement, subtree raising, or leave the subtree as it is, unpruned. Subtree replacement is the primary pruning operation, and we look at it first. The idea is to select some subtrees and replace them with single leaves. For example, the whole subtree in Figure 1.3(a), involving two internal nodes and four leaf nodes, has been replaced by the single leaf *bad*. This will certainly cause the accuracy on the training set to decrease if the original tree was produced by the decision tree algorithm described previously, because that continued to build

the tree until all leaf nodes were pure (or until all attributes had been tested). However, it may increase the accuracy on an independently chosen test set.

When subtree replacement is implemented, it proceeds from the leaves and works back up toward the root. In the Figure 1.3 example, the whole subtree in (a) would not be replaced at once. First, consideration would be given to replacing the three daughter nodes in the *health plan contribution* subtree with a single leaf node. Assume that a decision is made to perform this replacement—we will explain how this decision is made shortly. Then, continuing to work back from the leaves, consideration would be given to replacing the *working hours per week* subtree, which now has just two daughter nodes, by a single leaf node. In the Figure 1.3 example, this replacement was indeed made, which accounts for the entire subtree in (a) being replaced by a single leaf marked *bad*. Finally, consideration would be given to replacing the two daughter nodes in the *wage increase 1st year* subtree with a single leaf node. In this case, that decision was not made, so the tree remains as shown in Figure 1.3(a). Again, we will examine how these decisions are actually made shortly.

The second pruning operation, subtree raising, is more complex, and it is not clear that it is necessarily always worthwhile. However, because it is used in the influential decision tree–building system C4.5, we describe it here. Subtree raising does not occur in the Figure 1.3 example, so we use the artificial example of Figure 6.1 for illustration. Here, consideration is given to pruning the tree in Figure 6.1(a), and the result is shown in Figure 6.1(b). The entire subtree from C downward has been "raised" to replace the B subtree. Note that although the daughters of B and C are shown as leaves, they can be entire subtrees. Of course, if we perform this raising operation, it is necessary to reclassify the examples at the nodes marked 4 and 5 into the new subtree headed by C. This is why the daughters of that node are marked with primes—1′, 2′, and 3′—to indicate that they are not the same as the original

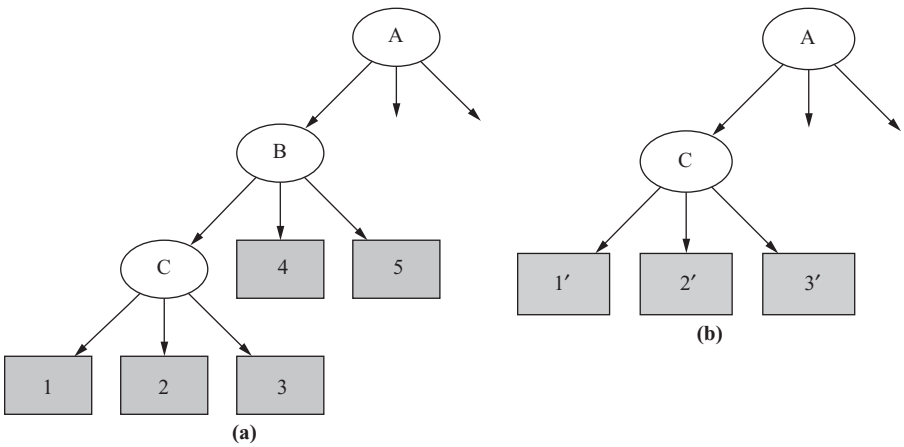


FIGURE 6.1

Example of subtree raising, where (a) node C is "raised" to subsume node B (b).

daughters 1, 2, and 3 but differ by the inclusion of the examples originally covered by 4 and 5.

Subtree raising is a potentially time-consuming operation. In actual implementations it is generally restricted to raising the subtree of the most popular branch. That is, we consider doing the raising illustrated in Figure 6.1 provided that the branch from B to C has more training examples than the branches from B to node 4 or from B to node 5. Otherwise, if (for example) node 4 were the majority daughter of B, we would consider raising node 4 to replace B and reclassifying all examples under C, as well as the examples from node 5, into the new node.

Estimating Error Rates

So much for the two pruning operations. Now we must address the question of how to decide whether to replace an internal node by a leaf (for subtree replacement) or whether to replace an internal node by one of the nodes below it (for subtree raising). To make this decision rationally, it is necessary to estimate the error rate that would be expected at a particular node given an independently chosen test set. We need to estimate the error at internal nodes as well as at leaf nodes. If we had such an estimate, it would be clear whether to replace, or raise, a particular subtree simply by comparing the estimated error of the subtree with that of its proposed replacement. Before estimating the error for a subtree proposed for raising, examples that lie under siblings of the current node—the examples at 4 and 5 of Figure 6.1—would have to be temporarily reclassified into the raised tree.

It is no use taking the training set error as the error estimate: That would not lead to any pruning because the tree has been constructed expressly for that particular training set. One way of coming up with an error estimate is the standard verification technique: Hold back some of the data originally given and use it as an independent test set to estimate the error at each node. This is called *reduced-error* pruning. It suffers from the disadvantage that the actual tree is based on less data.

The alternative is to try to make some estimate of error based on the training data itself. That is what C4.5 does, and we will describe its method here. It is a heuristic based on some statistical reasoning, but the statistical underpinning is rather weak. However, it seems to work well in practice. The idea is to consider the set of instances that reach each node and imagine that the majority class is chosen to represent that node. That gives us a certain number of "errors," E, out of the total number of instances, N. Now imagine that the true probability of error at the node is q, and that the N instances are generated by a Bernoulli process with parameter q, of which E turn out to be errors.

This is almost the same situation as we considered when looking at the holdout method in Section 5.2, where we calculated confidence intervals on the true success probability p given a certain observed success rate. There are two differences. One is trivial: Here we are looking at the error rate q rather than the success rate p; these are simply related by $p + q = 1$. The second is more serious: Here the figures E and N are measured from the training data, whereas in Section 5.2 we were considering

independent test data. Because of this difference we make a pessimistic estimate of the error rate by using the upper confidence limit rather than stating the estimate as a confidence range.

The mathematics involved is just the same as before. Given a particular confidence c (the default figure used by C4.5 is $c = 25\%$), we find confidence limits z such that

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

where N is the number of samples, $f = E/N$ is the observed error rate, and q is the true error rate. As before, this leads to an upper confidence limit for q. Now we use that upper confidence limit as a (pessimistic) estimate for the error rate e at the node:

$$e = \frac{f + \frac{z^2}{2N} + z\sqrt{\frac{f}{N} - \frac{f^2}{N} + \frac{z^2}{4N^2}}}{1 + \frac{z^2}{N}}$$

Note the use of the + sign before the square root in the numerator to obtain the upper confidence limit. Here, z is the number of standard deviations corresponding to the confidence c, which for $c = 25\%$ is $z = 0.69$.

To see how all this works in practice, let's look again at the labor negotiations decision tree of Figure 1.3, salient parts of which are reproduced in Figure 6.2 with the number of training examples that reach the leaves added. We use the previous formula with a 25% confidence figure—that is, with $z = 0.69$. Consider the lower left leaf, for which $E = 2$, $N = 6$, and so $f = 0.33$. Plugging these figures into the formula, the upper confidence limit is calculated as $e = 0.47$. That means that instead of using the training set error rate for this leaf, which is 33%, we will use the pessimistic estimate of 47%. This is pessimistic indeed, considering that it would be a bad mistake to let the error rate exceed 50% for a two-class problem. But things are worse for the neighboring leaf, where $E = 1$ and $N = 2$, because the upper confidence limit becomes $e = 0.72$. The third leaf has the same value of e as the first. The next step is to combine the error estimates for these three leaves in the ratio of the number of examples they cover, 6:2:6, which leads to a combined error estimate of 0.51. Now we consider the error estimate for the parent node, *health plan contribution*. This covers nine bad examples and five good ones, so the training set error rate is $f = 5/14$. For these values, the previous formula yields a pessimistic error estimate of $e = 0.46$. Because this is less than the combined error estimate of the three children, they are pruned away.

The next step is to consider the *working hours per week* node, which now has two children that are both leaves. The error estimate for the first, with $E = 1$ and $N = 2$, is $e = 0.72$, while for the second it is $e = 0.46$, as we have just seen. Combining these in the appropriate ratio of 2:14 leads to a value that is higher than the error estimate for the *working hours* node, so the subtree is pruned away and replaced by a leaf node.

The estimated error figures obtained in these examples should be taken with a grain of salt because the estimate is only a heuristic one and is based on a number of shaky assumptions: the use of the upper confidence limit; the assumption of a normal distribution; and the fact that statistics from the training set are used. However, the qualitative behavior of the error formula is correct and the method seems to work reasonably well in practice. If necessary, the underlying confidence level, which we have taken to be 25%, can be tweaked to produce more satisfactory results.

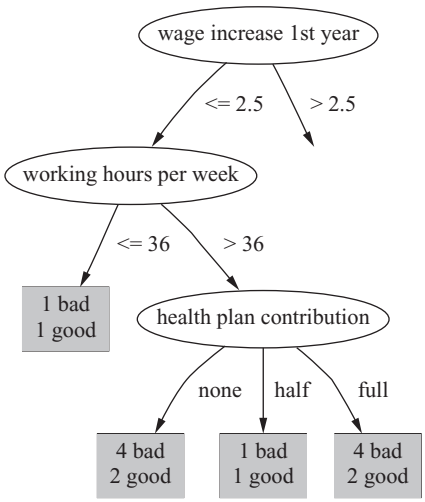


FIGURE 6.2

Pruning the labor negotiations decision tree.

Complexity of Decision Tree Induction

Now that we have learned how to accomplish the pruning operations, we have finally covered all the central aspects of decision tree induction. Let's take stock and examine the computational complexity of inducing decision trees. We will use the standard order notation: O(n) stands for a quantity that grows at most linearly with n, O(n^2) grows at most quadratically with n, and so on.

Suppose the training data contains n instances and m attributes. We need to make some assumption about the size of the tree, and we will assume that its depth is on the order of log n, that is O(log n). This is the standard rate of growth of a tree with n leaves, provided that it remains "bushy" and doesn't degenerate into a few very long, stringy branches. Note that we are tacitly assuming that most of the instances are different from each other and—this is almost the same thing—that the m attributes provide enough tests to allow the instances to be differentiated. For example, if there were only a few binary attributes, they would allow only so many instances to be differentiated and the tree could not grow past a certain point, rendering an "in the limit" analysis meaningless.

The computational cost of building the tree in the first place is O(mnlog n). Consider the amount of work done for one attribute over all nodes of the tree. Not all the examples need to be considered at each node, of course. But at each possible tree depth, the entire set of n instances must be considered in the worst case. And because there are log n different depths in the tree, the amount of work for this one attribute is O(n log n). At each node all attributes are considered, so the total amount of work is O(mn log n).

This reasoning makes some assumptions. If some attributes are numeric, they must be sorted, but once the initial sort has been done there is no need to re-sort at each tree depth if the appropriate algorithm is used (described previously—see page 193). The initial sort takes $O(n \log n)$ operations for each of up to m attributes; thus, the above complexity figure is unchanged. If the attributes are nominal, all attributes do *not* have to be considered at each tree node because attributes that are used further up the tree cannot be reused. However, if attributes are numeric, they can be reused and so they have to be considered at every tree level.

Next, consider pruning by subtree replacement. First an error estimate must be made for every tree node. Provided that counts are maintained appropriately, this is linear in the number of nodes in the tree. Then each node needs to be considered for replacement. The tree has at most n leaves, one for each instance. If it were a binary tree, each attribute being numeric or two-valued, that would give it $2n - 1$ nodes; multiway branches would only serve to decrease the number of internal nodes. Thus, the complexity of subtree replacement is $O(n)$.

Finally, subtree lifting has a basic complexity equal to subtree replacement. But there is an added cost because instances need to be reclassified during the lifting operation. During the whole process, each instance may have to be reclassified at every node between its leaf and the root—that is, as many as $O(\log n)$ times. That makes the total number of reclassifications $O(n \log n)$. And reclassification is not a single operation: One that occurs near the root will take $O(\log n)$ operations, and one of average depth will take half of this. Thus, the total complexity of subtree lifting is as follows: $O(n(\log n)^2)$.

Taking into account all these operations, the full complexity of decision tree induction is

$$O(mn \log n) + O(n(\log n)^2)$$

From Trees to Rules

It is possible to read a set of rules directly off a decision tree, as noted in Section 3.4, by generating a rule for each leaf and making a conjunction of all the tests encountered on the path from the root to that leaf. This produces rules that are unambiguous in that it doesn't matter in what order they are executed. However, the rules are more complex than necessary.

The estimated error rate described previously provides exactly the mechanism necessary to prune the rules. Given a particular rule, each condition in it is considered for deletion by tentatively removing it, working out which of the training examples are now covered by the rule, calculating from this a pessimistic estimate of the error rate of the new rule, and comparing this with the pessimistic estimate for the original rule. If the new rule is better, delete that condition and carry on, looking for other conditions to delete. Leave the rule when there are no remaining conditions that will improve it if they are removed. Once all rules have been pruned in this way, it is necessary to see if there are any duplicates and remove them from the rule set.

This is a greedy approach to detecting redundant conditions in a rule, and there is no guarantee that the best set of conditions will be removed. An improvement would be to consider all subsets of conditions, but this is usually prohibitively expensive. Another solution might be to use an optimization technique such as simulated annealing or a genetic algorithm to select the best version of this rule. However, the simple greedy solution seems to produce quite good rule sets.

The problem, even with the greedy method, is computational cost. For every condition that is a candidate for deletion, the effect of the rule must be reevaluated on all the training instances. This means that rule generation from trees tends to be very slow. The next section describes much faster methods that generate classification rules directly without forming a decision tree first.

C4.5: Choices and Options

The decision tree program C4.5 and its successor C5.0 were devised by Ross Quinlan over a 20-year period beginning in the late 1970s. A complete description of C4.5, the early 1990s version, appears as an excellent and readable book (Quinlan, 1993), along with the full source code. The more recent version, C5.0, is available commercially. Its decision tree induction seems to be essentially the same as that used by C4.5, and tests show some differences but negligible improvements. However, its rule generation is greatly sped up and clearly uses a different technique, although this has not been described in the open literature.

C4.5 works essentially as described in the previous sections. The default confidence value is set at 25% and works reasonably well in most cases; possibly it should be altered to a lower value, which causes more drastic pruning, if the actual error rate of pruned trees on test sets is found to be much higher than the estimated error rate. There is one other important parameter whose effect it is to eliminate tests for which almost all of the training examples have the same outcome. Such tests are often of little use. Consequently, tests are not incorporated into the decision tree unless they have at least two outcomes that have at least a minimum number of instances. The default value for this minimum is 2, but it is controllable and should perhaps be increased for tasks that have a lot of noisy data.

Another heuristic in C4.5 is that candidate splits on numeric attributes are only considered if they cut off a certain minimum number of instances: at least 10% of the average number of instances per class at the current node, or 25 instances—whichever value is smaller (but the minimum just mentioned, 2 by default, is also enforced).

C4.5 Release 8, the last noncommercial version of C4.5, includes an MDL-based adjustment to the information gain for splits on numeric attributes. More specifically, if there are S candidate splits on a certain numeric attribute at the node currently considered for splitting, $\log_2(S)/N$ is subtracted from the information gain, where N is the number of instances at the node. This heuristic, described by Quinlan (1986), is designed to prevent overfitting. The information gain may be negative after subtraction, and tree growing will stop if there are no attributes with positive information gain—a form of prepruning. We mention this here because it can be surprising

to obtain a pruned tree even if postpruning has been turned off! This heuristic is also implemented in the software described in Part 3 of this book.

Cost-Complexity Pruning

As mentioned, the postpruning method in C4.5 is based on shaky statistical assumptions, and it turns out that it often does not prune enough. On the other hand, it is very fast and thus popular in practice. However, in many applications it is worthwhile expending more computational effort to obtain a more compact decision tree. Experiments have shown that C4.5's pruning method can yield unnecessary additional structure in the final tree: Tree size continues to grow when more instances are added to the training data even when this does not further increase performance on independent test data. In that case, the more conservative *cost-complexity pruning* method from the Classification and Regression Trees (CART) learning system may be more appropriate.

Cost-complexity pruning is based on the idea of first pruning those subtrees that, relative to their size, lead to the smallest increase in error on the training data. The increase in error is measured by a quantity α that is defined to be the average error increase per leaf of the subtree concerned. By monitoring this quantity as pruning progresses, the algorithm generates a sequence of successively smaller pruned trees. In each iteration it prunes all subtrees that exhibit the smallest value of α among the remaining subtrees in the current version of the tree.

Each candidate tree in the resulting sequence of pruned trees corresponds to one particular threshold value, α_i. The question becomes, which tree should be chosen as the final classification model? To determine the most predictive tree, cost-complexity pruning either uses a holdout set to estimate the error rate of each tree, or, if data is limited, employs cross-validation.

Using a holdout set is straightforward. However, cross-validation poses the problem of relating the α values observed in the sequence of pruned trees for training fold k of the cross-validation to the α values from the sequence of trees for the full dataset: These values are usually different. This problem is solved by first computing the geometric average of α_i and α_{i+1} for tree i from the full dataset. Then, for each fold k of the cross-validation, the tree that exhibits the largest α value smaller than this average is picked. The average of the error estimates for these trees from the k folds, estimated from the corresponding test datasets, is the cross-validation error for tree i from the full dataset.

Discussion

Top-down induction of decision trees is probably the most extensively researched method of machine learning used in data mining. Researchers have investigated a panoply of variations for almost every conceivable aspect of the learning process—for example, different criteria for attribute selection or modified pruning methods. However, they are rarely rewarded by substantial improvements in accuracy over a spectrum of diverse datasets. As discussed, the pruning method used by the CART system for learning decision trees (Breiman et al., 1984) can often produce smaller

trees than C4.5's pruning method. This has been investigated empirically by Oates and Jensen (1997).

In our description of decision trees, we have assumed that only one attribute is used to split the data into subsets at each node of the tree. However, it is possible to allow tests that involve several attributes at a time. For example, with numeric attributes each test can be on a linear combination of attribute values. Then the final tree consists of a hierarchy of linear models of the kind we described in Section 4.6, and the splits are no longer restricted to being axis-parallel. Trees with tests involving more than one attribute are called *multivariate* decision trees, in contrast to the simple *univariate* trees that we normally use. The CART system has the option of generating multivariate tests. They are often more accurate and smaller than univariate trees but take much longer to generate and are also more difficult to interpret. We briefly mention one way of generating them in the Principal Components Analysis section in Section 7.3.

6.2 CLASSIFICATION RULES

We call the basic covering algorithm for generating rules that was described in Section 4.4 a separate-and-conquer technique because it identifies a rule that covers instances in a class (and excludes ones not in the class), separates them out, and continues on those that are left. Such algorithms have been used as the basis of many systems that generate rules. There, we described a simple correctness-based measure for choosing what test to add to the rule at each stage. However, there are many other possibilities, and the particular criterion that is used has a significant effect on the rules produced. We examine different criteria for choosing tests in this section. We also look at how the basic rule-generation algorithm can be extended to more practical situations by accommodating missing values and numeric attributes.

But the real problem with all these rule-generation schemes is that they tend to overfit the training data and do not generalize well to independent test sets, particularly on noisy data. To be able to generate good rule sets for noisy data, it is necessary to have some way of measuring the real worth of individual rules. The standard approach to assessing the worth of rules is to evaluate their error rate on an independent set of instances, held back from the training set, and we explain this next. After that, we describe two industrial-strength rule learners: one that combines the simple separate-and-conquer technique with a global optimization step, and another that works by repeatedly building partial decision trees and extracting rules from them. Finally, we consider how to generate rules with exceptions, and exceptions to the exceptions.

Criteria for Choosing Tests

When we introduced the basic rule learner in Section 4.4, we had to figure out a way of deciding which of many possible tests to add to a rule to prevent it from covering any negative examples. For this we used the test that maximizes the ratio p/t, where t is the total number of instances that the new rule will cover, and p is

the number of these that are positive—that is, belong to the class in question. This attempts to maximize the "correctness" of the rule on the basis that the higher the proportion of positive examples it covers, the more correct a rule is. One alternative is to calculate an information gain:

$$p\left[\log\frac{p}{t}-\log\frac{P}{T}\right]$$

where p and t are the number of positive instances and the total number of instances covered by the new rule, as before, and P and T are the corresponding number of instances that satisfied the rule *before* the new test was added. The rationale for this is that it represents the total information gained regarding the current positive examples, which is given by the number of them that satisfy the new test, multiplied by the information gained regarding each one.

The basic criterion for choosing a test to add to a rule is to find one that covers as many positive examples as possible while covering as few negative examples as possible. The original correctness-based heuristic, which is just the percentage of positive examples among all examples covered by the rule, attains a maximum when no negative examples are covered regardless of the number of positive examples covered by the rule. Thus, a test that makes the rule exact will be preferred to one that makes it inexact, no matter how few positive examples the former rule covers nor how many positive examples the latter covers. For example, if we consider a test that covers one example that is positive, this criterion will prefer it over a test that covers 1000 positive examples along with one negative one.

The information-based heuristic, on the other hand, places far more emphasis on covering a large number of positive examples regardless of whether the rule so created is exact. Of course, both algorithms continue adding tests until the final rule produced is exact, which means that the rule will be finished earlier using the correctness measure whereas more terms will have to be added if the information-based measure is used. Thus, the correctness-based measure might find special cases and eliminate them completely, saving the larger picture for later (when the more general rule might be simpler because awkward special cases have already been dealt with), whereas the information-based one will try to generate high-coverage rules first and leave the special cases until later. It is by no means obvious that either strategy is superior to the other at producing an exact rule set. Moreover, the whole situation is complicated by the fact that, as described in the following, rules may be pruned and inexact ones tolerated.

Missing Values, Numeric Attributes

As with divide-and-conquer decision tree algorithms, the nasty practical considerations of missing values and numeric attributes need to be addressed. In fact, there is not much more to say. Now that we know how these problems can be solved for decision tree induction, appropriate solutions for rule induction are easily given.

When producing rules using covering algorithms, missing values can be best treated as though they don't match any of the tests. This is particularly suitable when a decision list is being produced, because it encourages the learning algorithm to separate out positive instances using tests that are known to succeed. It has the effect either that instances with missing values are dealt with by rules involving other attributes that are not missing, or that any decisions about them are deferred until most of the other instances have been taken care of, at which time tests will probably emerge that involve other attributes. Covering algorithms for decision lists have a decided advantage over decision tree algorithms in this respect: Tricky examples can be left until late in the process, at which time they will appear less tricky because most of the other examples have already been classified and removed from the instance set.

Numeric attributes can be dealt with in exactly the same way as they are dealt with for trees. For each numeric attribute, instances are sorted according to the attribute's value and, for each possible threshold, a binary less-than/greater-than test is considered and evaluated in exactly the same way that a binary attribute would be.

Generating Good Rules

Suppose you don't want to generate perfect rules that guarantee to give the correct classification on all instances in the training set, but would rather generate "sensible" ones that avoid overfitting the training set and thereby stand a better chance of performing well on new test instances. How do you decide which rules are worthwhile? How do you tell when it becomes counterproductive to continue adding terms to a rule to exclude a few pesky instances of the wrong type, all the while excluding more and more instances of the correct type?

Let's look at a few examples of possible rules—some good and some bad—for the contact lens problem in Table 1.1. Consider first the rule

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

This gives a correct result for four out of the six cases that it covers; thus, its success fraction is 4/6. Suppose we add a further term to make the rule a "perfect" one:

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

This improves accuracy to 2/2. Which rule is better? The second one is more accurate on the training data but covers only two cases, whereas the first one covers six. It may be that the second version is just overfitting the training data. For a practical rule learner we need a principled way of choosing the appropriate version of a rule, preferably one that maximizes accuracy on future test data.

Suppose we split the training data into two parts that we will call a *growing set* and a *pruning set*. The growing set is used to form a rule using the basic covering

algorithm. Then a test is deleted from the rule, and the effect is evaluated by trying out the truncated rule on the pruning set and seeing whether it performs better than the original rule. This pruning process repeats until the rule cannot be improved by deleting any further tests. The whole procedure is repeated for each class, obtaining one best rule for each class, and the overall best rule is established by evaluating the rules on the pruning set. This rule is then added to the rule set, the instances it covers are removed from the training data—from both growing and pruning sets—and the process is repeated.

Why not do the pruning as we build up the rule, rather than building up the whole thing and then throwing parts away? That is, why not preprune rather than post-prune? Just as when pruning decision trees it is often best to grow the tree to its maximum size and then prune back, so with rules it is often best to make a perfect rule and then prune it. Who knows?—adding that last term may make a really good rule, a situation that we might never have noticed had we adopted an aggressive prepruning strategy.

It is essential that the growing and pruning sets are separate because it is misleading to evaluate a rule on the very data that was used to form it: That would lead to serious errors by preferring rules that were overfitted. Usually the training set is split so that two-thirds of instances are used for growing and one-third for pruning. A disadvantage, of course, is that learning occurs from instances in the growing set only, so the algorithm might miss important rules because some key instances had been assigned to the pruning set. Moreover, the wrong rule might be preferred because the pruning set contains only one-third of the data and may not be completely representative. These effects can be ameliorated by resplitting the training data into growing and pruning sets at each cycle of the algorithm—that is, after each rule is finally chosen.

The idea of using a separate pruning set for pruning—which is applicable to decision trees as well as rule sets—is called *reduced-error pruning*. The variant previously described prunes a rule immediately after it has been grown; it is called *incremental reduced-error pruning*. Another possibility is to build a full, unpruned, rule set first, pruning it afterwards by discarding individual tests. However, this method is much slower.

Of course, there are many different ways to assess the worth of a rule based on the pruning set. A simple measure is to consider how well the rule would do at discriminating the predicted class from other classes if it were the only rule in the theory, operating under the closed-world assumption. Suppose it gets p instances right out of the t instances that it covers, and there are P instances of this class out a total of T instances altogether. The instances that it does not cover include $N - n$ negative ones, where $n = t - p$ is the number of negative instances that the rule covers and $N = T - P$ is the total number of negative instances. Thus, in total the rule makes correct decisions on $p + (N - n)$ instances, and so has an overall success ratio of

$$[p + (N - n)]/T$$

This quantity, evaluated on the test set, has been used to evaluate the success of a rule when using reduced-error pruning.

This measure is open to criticism because it treats noncoverage of negative examples as being as important as coverage of positive ones, which is unrealistic in a situation where what is being evaluated is one rule that will eventually serve alongside many others. For example, a rule that gets $p = 2000$ instances right out of a total coverage of 3000 (i.e., it gets $n = 1000$ wrong) is judged as more successful than one that gets $p = 1000$ out of a total coverage of 1001 (i.e., $n = 1$ wrong), because $[p + (N - n)]/T$ is $[1000 + N]/T$ in the first case but only $[999 + N]/T$ in the second. This is counterintuitive: The first rule is clearly less predictive than the second because it has a 33.3% as opposed to only a 0.1% chance of being incorrect.

Using the success rate p/t as a measure, as was done in the original formulation of the covering algorithm (Figure 4.8), is not the perfect solution either because it would prefer a rule that got a single instance right ($p = 1$) out of a total coverage of 1 (so $n = 0$) to the far more useful rule that got 1000 right out of 1001. Another heuristic that has been used is $(p - n)/t$, but that suffers from exactly the same problem because $(p - n)/t = 2p/t - 1$ and so the result, when comparing one rule with another, is just the same as with the success rate. It seems hard to find a simple measure of the worth of a rule that corresponds with intuition in all cases.

Whatever heuristic is used to measure the worth of a rule, the incremental reduced-error pruning algorithm is the same. A possible rule-learning algorithm based on this idea is given in Figure 6.3. It generates a decision list, creating rules for each class in turn and choosing at each stage the best version of the rule according to its worth on the pruning data. The basic covering algorithm for rule generation (Figure 4.8) is used to come up with good rules for each class, choosing conditions to add to the rule using the accuracy measure p/t that we described earlier.

```
Initialize E to the instance set
Split E into Grow and Prune in the ratio 2:1
  For each class C for which Grow and Prune both contain an instance
    Use the basic covering algorithm to create the best perfect rule
      for class C
    Calculate the worth w(R) for the rule on Prune, and for the rule
      with the final condition omitted w(R-)
    While w(R-) > w(R), remove the final condition from the rule and
      repeat the previous step
  From the rules generated, select the one with the largest w(R)
  Print the rule
  Remove the instances covered by the rule from E
Continue
```

FIGURE 6.3

Algorithm for forming rules by incremental reduced-error pruning.

This method has been used to produce rule-induction schemes that can process vast amounts of data and operate very quickly. It can be accelerated by generating rules for the classes in order rather than generating a rule for each class at every stage and choosing the best. A suitable ordering is the increasing order in which they occur in the training set so that the rarest class is processed first and the most common ones are processed later. Another significant speedup is obtained by stopping the whole process when a rule of sufficiently low accuracy is generated, so as not to spend time generating a lot of rules at the end with very small coverage. However, very simple terminating conditions (such as stopping when the accuracy for a rule is lower than the default accuracy for the class it predicts) do not give the best performance. One criterion that seems to work well is a rather complicated one based on the MDL principle, described later.

Using Global Optimization

In general, rules generated using incremental reduced-error pruning in this manner seem to perform quite well, particularly on large datasets. However, it has been found that a worthwhile performance advantage can be obtained by performing a global optimization step on the set of rules induced. The motivation is to increase the accuracy of the rule set by revising or replacing individual rules. Experiments show that both the size and the performance of rule sets are significantly improved by postinduction optimization. On the other hand, the process itself is rather complex.

To give an idea of how elaborate—and heuristic—industrial-strength rule learners become, Figure 6.4 shows an algorithm called RIPPER, an acronym for *repeated incremental pruning to produce error reduction*. Classes are examined in increasing size and an initial set of rules for a class is generated using incremental reduced-error pruning. An extra stopping condition is introduced that depends on the description length of the examples and rule set. The description-length *DL* is a complex formula that takes into account the number of bits needed to send a set of examples with respect to a set of rules, the number of bits required to send a rule with k conditions, and the number of bits needed to send the integer k—times an arbitrary factor of 50% to compensate for possible redundancy in the attributes.

Having produced a rule set for the class, each rule is reconsidered and two variants produced, again using reduced-error pruning—but at this stage, instances covered by other rules for the class are removed from the pruning set, and success rate on the remaining instances is used as the pruning criterion. If one of the two variants yields a better description length, it replaces the rule. Next we reactivate the original building phase to mop up any newly uncovered instances of the class. A final check is made, to ensure that each rule contributes to the reduction of description length, before proceeding to generate rules for the next class.

Obtaining Rules from Partial Decision Trees

There is an alternative approach to rule induction that avoids global optimization but nevertheless produces accurate, compact rule sets. The method combines the divide-and-conquer strategy for decision tree learning with the separate-and-conquer

```
Initialize E to the instance set
For each class C, from smallest to largest
    BUILD:
        Split E into Growing and Pruning sets in the ratio 2:1
        Repeat until (a) there are no more uncovered examples of C; or
             (b) the description length (DL) of ruleset and examples is
             64 bits greater than the smallest DL found so far, or (c)
             the error rate exceeds 50%:

        GROW phase: Grow a rule by greedily adding conditions until the
             rule is 100% accurate by testing every possible value of
             each attribute and selecting the condition with greatest
             information gain G
        PRUNE phase: Prune conditions in last-to-first order. Continue
             as long as the worth W of the rule increases
    OPTIMIZE:
        GENERATE VARIANTS:
        For each rule R for class C,
           Split E afresh into Growing and Pruning sets
           Remove all instances from the Pruning set that are covered
             by other rules for C
           Use GROW and PRUNE to generate and prune two competing rules
             from the newly split data:
               R1 is a new rule, rebuilt from scratch;
               R2 is generated by greedily adding antecedents to R.
           Prune using the metric A (instead of W) on this reduced data
        SELECT REPRESENTATIVE:
        Replace R by whichever of R, R1 and R2 has the smallest DL.
    MOP UP:
        If there are residual uncovered instances of class C, return to
             the BUILD stage to generate more rules based on these
             instances.
    CLEAN UP:
        Calculate DL for the whole ruleset and for the ruleset with each
             rule in turn omitted; delete any rule that increases the DL
        Remove instances covered by the rules just generated
Continue
```

(a)

$G = p[\log(p/t) - \log(P/T)]$

$W = \frac{p+1}{t+2}$

$A = \frac{p+n'}{T}$; accuracy for this rule

p = number of positive examples covered by this rule (true positives)

n = number of negative examples covered by this rule (false negatives)

$t = p + n$; total number of examples covered by this rule

$n' = N - n$; number of negative examples not covered by this rule (true negatives)

P = number of positive examples of this class

N = number of negative examples of this class

$T = P + N$; total number of examples of this class

(b)

FIGURE 6.4

RIPPER: (a) algorithm for rule learning and (b) meaning of symbols.

one for rule learning. It adopts the separate-and-conquer strategy in that it builds a rule, removes the instances it covers, and continues creating rules recursively for the remaining instances until none are left. However, it differs from the standard approach in the way that each rule is created. In essence, to make a single rule a pruned decision tree is built for the current set of instances, the leaf with the largest coverage is made into a rule, and the tree is discarded.

The prospect of repeatedly building decision trees only to discard most of them is not as bizarre as it first seems. Using a pruned tree to obtain a rule instead of pruning a rule incrementally by adding conjunctions one at a time avoids a tendency to overprune, which is a characteristic problem of the basic separate-and-conquer rule learner. Using the separate-and-conquer methodology in conjunction with decision trees adds flexibility and speed. It is indeed wasteful to build a full decision tree just to obtain a single rule, but the process can be accelerated significantly without sacrificing the advantages.

The key idea is to build a partial decision tree instead of a fully explored one. A partial decision tree is an ordinary decision tree that contains branches to undefined subtrees. To generate such a tree, the construction and pruning operations are integrated in order to find a "stable" subtree that can be simplified no further. Once this subtree has been found, tree building ceases and a single rule is read off.

The tree-building algorithm is summarized in Figure 6.5: It splits a set of instances recursively into a partial tree. The first step chooses a test and divides the instances into subsets accordingly. The choice is made using the same information-gain heuristic that is normally used for building decision trees (Section 4.3). Then the subsets are expanded in increasing order of their average entropy. The reason for this is that the later subsets will most likely not end up being expanded, and a subset with low-average entropy is more likely to result in a small subtree and therefore produce a more general rule. This proceeds recursively until a subset is expanded into a leaf, and then continues further by backtracking. But as soon as an internal node appears that has all its children expanded into leaves, the algorithm checks whether that node is better replaced by a single leaf. This is just the standard subtree replacement

```
Expand-subset (S):
  Choose a test T and use it to split the set of examples into subsets
  Sort subsets into increasing order of average entropy
  while (there is a subset X that has not yet been expanded
         AND all subsets expanded so far are leaves)
    expand-subset(X)
  if (all the subsets expanded are leaves
      AND estimated error for subtree ≥ estimated error for node)
    undo expansion into subsets and make node a leaf
```

FIGURE 6.5

Algorithm for expanding examples into a partial tree.

operation of decision tree pruning (see Section 6.1). If replacement is performed the algorithm backtracks in the standard way, exploring siblings of the newly replaced node. However, if during backtracking a node is encountered all of whose children expanded so far are not leaves—and this will happen as soon as a potential subtree replacement is *not* performed—then the remaining subsets are left unexplored and the corresponding subtrees are left undefined. Due to the recursive structure of the algorithm, this event automatically terminates tree generation.

Figure 6.6 shows a step-by-step example. During the stages in Figure 6.6(a–c), tree building continues recursively in the normal way—except that at each point the lowest-entropy sibling is chosen for expansion: node 3 between stages (a) and (b). Gray elliptical nodes are as yet unexpanded; rectangular ones are leaves. Between stages (b) and (c), the rectangular node will have lower entropy than its sibling, node 5, but cannot be expanded further because it is a leaf. Backtracking occurs and node 5 is chosen for expansion. Once stage of Figure 6.6(c) is reached, there is a node—node 5—that has all its children expanded into leaves, and this triggers pruning. Subtree replacement for node 5 is considered and accepted, leading to stage (d). Next node 3 is considered for subtree replacement, and this operation is again accepted. Backtracking continues, and node 4, having lower entropy than node 2, is expanded into two leaves. Now subtree replacement is considered for node 4, but suppose that node 4 is not replaced. At this point, the process terminates with the three-leaf partial tree of stage (e).

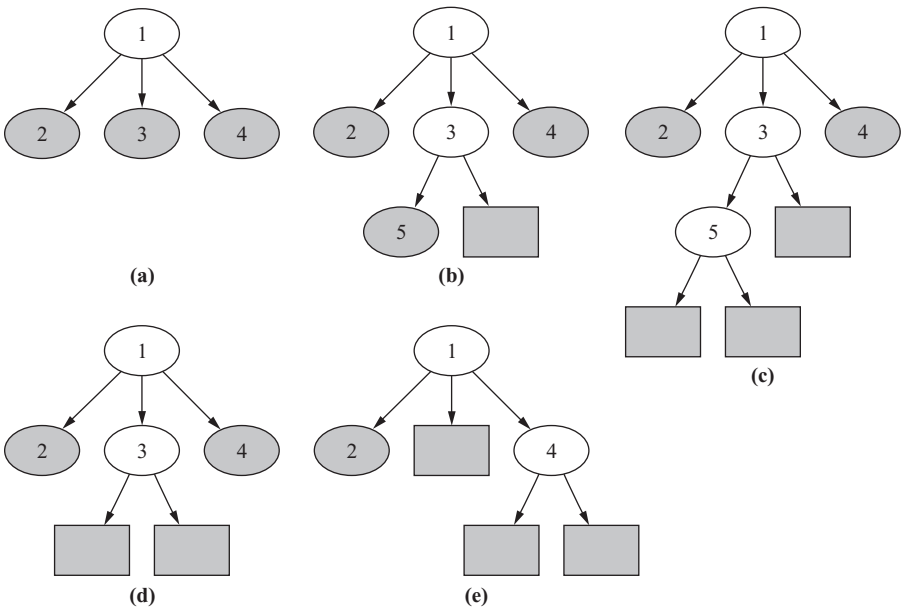


FIGURE 6.6

Example of building a partial tree.

If the data is noise-free and contains enough instances to prevent the algorithm from doing any pruning, just one path of the full decision tree has to be explored. This achieves the greatest possible performance gain over the naïve method that builds a full decision tree each time. The gain decreases as more pruning takes place. For datasets with numeric attributes, the asymptotic time complexity of the algorithm is the same as building the full decision tree because in this case the complexity is dominated by the time required to sort the attribute values in the first place.

Once a partial tree has been built, a single rule is extracted from it. Each leaf corresponds to a possible rule, and we seek the "best" leaf of those subtrees (typically a small minority) that have been expanded into leaves. Experiments show that it is best to aim at the most general rule by choosing the leaf that covers the greatest number of instances.

When a dataset contains missing values, they can be dealt with exactly as they are when building decision trees. If an instance cannot be assigned to any given branch because of a missing attribute value, it is assigned to each of the branches with a weight proportional to the number of training instances going down that branch, normalized by the total number of training instances with known values at the node. During testing, the same procedure is applied separately to each rule, thus associating a weight with the application of each rule to the test instance. That weight is deducted from the instance's total weight before it is passed to the next rule in the list. Once the weight has reduced to 0, the predicted class probabilities are combined into a final classification according to the weights.

This yields a simple but surprisingly effective method for learning decision lists for noisy data. Its main advantage over other comprehensive rule-generation schemes is simplicity, because other methods appear to require a complex global optimization stage to achieve the same level of performance.

Rules with Exceptions

In Section 3.4 (page 73) we learned that a natural extension of rules is to allow them to have exceptions, and exceptions to the exceptions, and so on—indeed, the whole rule set can be considered as exceptions to a default classification rule that is used when no other rules apply. The method of generating a "good" rule, using one of the measures described previously, provides exactly the mechanism needed to generate rules with exceptions.

First, a default class is selected for the top-level rule: It is natural to use the class that occurs most frequently in the training data. Then, a rule is found pertaining to any class other than the default one. Of all such rules it is natural to seek the one with the most discriminatory power—for example, the one with the best evaluation on a test set. Suppose this rule has the form

```
if ⟨condition⟩ then class = ⟨new class⟩
```

It is used to split the training data into two subsets: one containing instances for which the rule's condition is *true* and the other containing those for which it is *false*.

If either subset contains instances of more than one class, the algorithm is invoked recursively on that subset. For the subset for which the condition is *true*, the "default class" is the new class as specified by the rule; for the subset where the condition is *false*, the default class remains as it was before.

Let's examine how this algorithm would work for the rules with exceptions that were given in Section 3.4 for the iris data of Table 1.4. We will represent the rules in the graphical form shown in Figure 6.7, which is in fact equivalent to the textual rules noted in Figure 3.8. The default of *Iris setosa* is the entry node at the top left. Horizontal, dotted paths show exceptions, so the next box, which contains a rule that concludes *Iris versicolor*, is an exception to the default. Below this is an alternative, a second exception—alternatives are shown by vertical, solid lines—leading to the conclusion *Iris virginica*. Following the upper path horizontally leads to an exception to the *Iris versicolor* rule that overrides it whenever the condition in the top right box holds, with the conclusion *Iris virginica*. Below this is an alternative, leading (as it happens) to the same conclusion. Returning to the box at bottom center, this has its own exception, the lower right box, which gives the conclusion *Iris versicolor*. The numbers at the lower right of each box give the "coverage" of the rule, expressed as the number of examples that satisfy it divided by the number that satisfy its condition but not its conclusion. For example, the condition in the top center box applies to 52 of the examples, and 49 of them are *Iris versicolor*. The strength of this representation is that you can get a very good feeling for the effect of the rules from the boxes toward the left side; the boxes at the right cover just a few exceptional cases.

To create these rules, the default is first set to *Iris setosa* by taking the most frequently occurring class in the dataset. This is an arbitrary choice because, for this

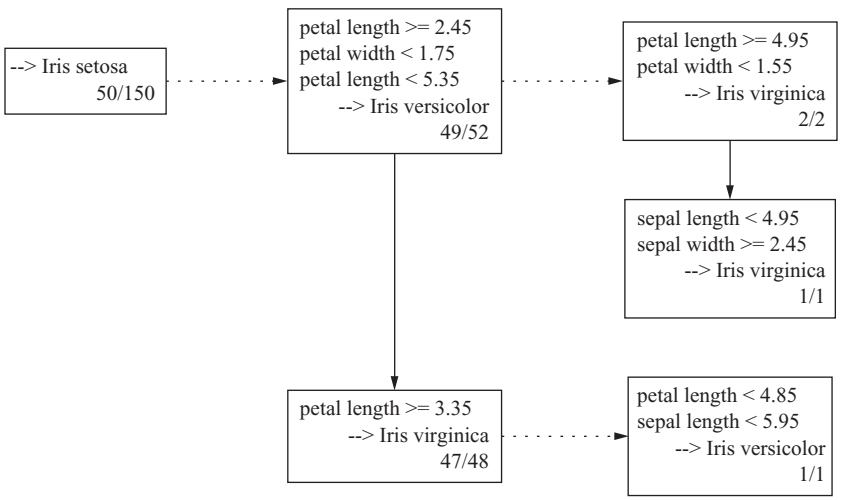


FIGURE 6.7

Rules with exceptions for the iris data.

dataset, all classes occur exactly 50 times; as shown in Figure 6.7 this default "rule" is correct in 50 out of 150 cases. Then the best rule that predicts another class is sought. In this case it is

```
if petal-length ≥ 2.45 and petal-length < 5.355
   and petal-width < 1.75 then Iris-versicolor
```

This rule covers 52 instances, of which 49 are *Iris versicolor*. It divides the dataset into two subsets: the 52 instances that satisfy the condition of the rule and the remaining 98 that do not.

We work on the former subset first. The default class for these instances is *Iris versicolor*: There are only three exceptions, all of which happen to be *Iris virginica*. The best rule for this subset that does not predict *Iris versicolor* is

```
if petal-length ≥ 4.95 and petal-width < 1.55 then Iris-virginica
```

It covers two of the three *Iris virginicas* and nothing else. Again, it divides the subset into two: those instances that satisfy its condition and those that do not. Fortunately, in this case, all those instances that satisfy the condition do indeed have class *Iris virginica*, so there is no need for a further exception. However, the remaining instances still include the third *Iris virginica*, along with 49 *Iris versicolors*, which are the default at this point. Again the best rule is sought:

```
if sepal-length < 4.95 and sepal-width ≥ 2.45 then Iris-virginica
```

This rule covers the remaining *Iris virginica* and nothing else, so it also has no exceptions. Furthermore, all remaining instances in the subset that do not satisfy its condition have the class *Iris versicolor*, which is the default, so no more needs to be done.

Return now to the second subset created by the initial rule, the instances that do not satisfy the condition

```
petal-length ≥ 2.45 and petal-length < 5.355 and petal-width < 1.75
```

Of the rules for these instances that do not predict the default class *Iris setosa*, the best is

```
if petal-length ≥ 3.35 then Iris-virginica
```

It covers all 47 *Iris virginicas* that are in the example set (3 were removed by the first rule, as explained previously). It also covers 1 *Iris versicolor*. This needs to be taken care of as an exception, by the final rule:

```
if petal-length < 4.85 and sepal-length < 5.95 then Iris-versicolor
```

Fortunately, the set of instances that do *not* satisfy its condition are all the default, *Iris setosa*. Thus, the procedure is finished.

The rules that are produced have the property that most of the examples are covered by the high-level rules and the lower-level ones really do represent exceptions. For example, the last exception clause and the deeply nested *else* clause both

cover a solitary example, and removing them would have little effect. Even the remaining nested exception rule covers only two examples. Thus, one can get an excellent feeling for what the rules do by ignoring all the deeper structure and looking only at the first level or two. That is the attraction of rules with exceptions.

Discussion

All algorithms for producing classification rules that we have described use the basic covering or separate-and-conquer approach. For the simple, noise-free case this produces PRISM (Cendrowska, 1987), an algorithm that is simple and easy to understand. When applied to two-class problems with the closed-world assumption, it is only necessary to produce rules for one class: Then the rules are in disjunctive normal form and can be executed on test instances without any ambiguity arising. When applied to multiclass problems, a separate rule set is produced for each class; thus, a test instance may be assigned to more than one class, or to no class, and further heuristics are necessary if a unique prediction is sought.

To reduce overfitting in noisy situations, it is necessary to produce rules that are not "perfect" even on the training set. To do this it is necessary to have a measure for the "goodness," or worth, of a rule. With such a measure it is then possible to abandon the class-by-class approach of the basic covering algorithm and start by generating the very best rule, regardless of which class it predicts, and then remove all examples covered by this rule and continue the process. This yields a method for producing a decision list rather than a set of independent classification rules, and decision lists have the important advantage that they do not generate ambiguities when interpreted.

The idea of incremental reduced-error pruning is from Fürnkranz and Widmer (1994) and forms the basis for fast and effective rule induction. The RIPPER rule learner is from Cohen (1995), although the published description appears to differ from the implementation in precisely how the description length (DL) affects the stopping condition. What we have presented here is the basic idea of the algorithm; there are many more details in the implementation.

The whole question of measuring the value of a rule has not yet been satisfactorily resolved. Many different measures have been proposed, some blatantly heuristic and others based on information-theoretical or probabilistic grounds. However, there seems to be no consensus on the best measure to use. An extensive theoretical study of various criteria has been performed by Fürnkranz and Flach (2005).

The rule-learning scheme based on partial decision trees was developed by Frank and Witten (1998). On standard benchmark datasets it produces rule sets that are as accurate as rules generated by C4.5 and more accurate than those of RIPPER; however, it produces larger rule sets than RIPPER. Its main advantage over other schemes is not performance but simplicity: By combining top-down decision tree induction with separate-and-conquer rule learning, it produces good rule sets without any need for global optimization.

The procedure for generating rules with exceptions was developed as an option in the Induct system by Gaines and Compton (1995), who called them *ripple-down* rules. In an experiment with a large medical dataset (22,000 instances, 32 attributes, and 60 classes), they found that people can understand large systems of rules with exceptions more readily than equivalent systems of regular rules because that is the way they think about the complex medical diagnoses that are involved. Richards and Compton (1998) describe their role as an alternative to classic knowledge engineering.

6.3 ASSOCIATION RULES

In Section 4.5 we studied the Apriori algorithm for generating association rules that meet minimum support and confidence thresholds. Apriori follows a generate-and-test methodology for finding frequent item sets, generating successively longer candidate item sets from shorter ones that are known to be frequent. Each different size of candidate item set requires a scan through the dataset to determine whether its frequency exceeds the minimum support threshold. Although some improvements to the algorithm have been suggested to reduce the number of scans of the dataset, the combinatorial nature of this generation process can prove costly, particularly if there are many item sets or item sets are large. Both conditions readily occur even for modest datasets when low support thresholds are used. Moreover, no matter how high the threshold, if the data is too large to fit in main memory, it is undesirable to have to scan it repeatedly—and many association rule applications involve truly massive datasets.

These effects can be ameliorated by using appropriate data structures. We describe a method called FP-growth that uses an extended prefix tree—a frequent-pattern tree, or FP-tree—to store a compressed version of the dataset in main memory. Only two passes are needed to map a dataset into an FP-tree. The algorithm then processes the tree in a recursive fashion to grow large item sets directly, instead of generating candidate item sets and then having to test them against the entire database.

Building a Frequent-Pattern Tree

Like Apriori, the FP-growth algorithm begins by counting the number of times individual items (i.e., attribute–value pairs) occur in the dataset. After this initial pass, a tree structure is created in a second pass. Initially, the tree is empty and the structure emerges as each instance in the dataset is inserted into it.

The key to obtaining a compact tree structure that can be quickly processed to find large item sets is to sort the items in each instance in descending order of their frequency of occurrence in the dataset, which has already been recorded in the first pass, before inserting them into the tree. Individual items in each instance that do not meet the minimum support threshold are not inserted into the tree, effectively removing them from the dataset. The hope is that many instances will share those

items that occur most frequently individually, resulting in a high degree of compression close to the tree's root.

We illustrate the process with the weather data, reproduced in Table 6.1(a), using a minimum support threshold of 6. The algorithm is complex, and its complexity far exceeds what would be reasonable for such a trivial example, but a small illustration is the best way of explaining it. Table 6.1(b) shows the individual items, with their frequencies, that are collected in the first pass. They are sorted into descending order and ones whose frequency exceeds the minimum threshold are bolded. Table 6.1(c) shows the original instances, numbered as in Table 6.1(a), with the items in each instance sorted into descending frequency order. Finally, to give

Table 6.1 Preparing Weather Data for Insertion into an FP-Tree

(a)	Outlook	Temperature	Humidity	Windy	Play
1	sunny	hot	high	false	no
2	sunny	hot	high	true	no
3	overcast	hot	high	false	yes
4	rainy	mild	high	false	yes
5	rainy	cool	normal	false	yes
6	rainy	cool	normal	true	no
7	overcast	cool	normal	true	yes
8	sunny	mild	high	false	no
9	sunny	cool	normal	false	yes
10	rainy	mild	normal	false	yes
11	sunny	mild	normal	true	yes
12	overcast	mild	high	true	yes
13	overcast	hot	normal	false	yes
14	rainy	mild	high	true	no

(b)	
play = yes	**9**
windy = false	**8**
humidity = normal	**7**
humidity = high	**7**
windy = true	**6**
temperature = mild	**6**
play = no	5
outlook = sunny	5
outlook = rainy	5
temperature = hot	4
temperature = cool	4
outlook = overcast	4

Continued

Table 6.1 Preparing Weather Data for Insertion into an FP-Tree *Continued*

(c)	
1	**windy = false, humidity = high,** play = no, outlook = sunny, temperature = hot
2	**humidity = high, windy = true,** play = no, outlook = sunny, temperature = hot
3	**play = yes, windy = false, humidity = high,** temperature = hot, outlook = overcast
4	**play = yes, windy = false, humidity = high, temperature = mild,** outlook = rainy
5	**play = yes, windy = false, humidity = normal,** outlook = rainy, temperature = cool
6	**humidity = normal, windy = true,** play = no, outlook = rainy, temperature = cool
7	**play = yes, humidity = normal, windy = true,** temperature = cool, outlook = overcast
8	**windy = false, humidity = high, temperature = mild,** play = no, outlook = sunny
9	**play = yes, windy = false, humidity = normal,** outlook = sunny, temperature = cool
10	**play = yes, windy = false, humidity = normal, temperature = mild,** outlook = rainy
11	**play = yes, humidity = normal, windy = true, temperature = mild,** outlook = sunny
12	**play = yes, humidity = high, windy = true, temperature = mild,** outlook = overcast
13	**play = yes, windy = false, humidity = normal,** temperature = hot, outlook = overcast
14	**humidity = high, windy = true, temperature = mild,** play = no, outlook = rainy

(d)	
play = yes and windy = false	6
play = yes and humidity = normal	6

(a) The original data, (b) frequency ordering of items with frequent item sets in bold, (c) the data with each instance sorted into frequency order, and (d) the two multiple-item frequent item sets.

an advance peek at the final outcome, Table 6.1(d) shows the only two multiple-item sets whose frequency satisfies the minimum support threshold. Along with the six single-item sets shown in bold in Table 6.1(b), these form the final answer: a total of eight item sets. We are going to have to do a lot of work to find the two multiple-item sets in Table 6.1(d) using the FP-tree method.

Figure 6.8(a) shows the FP-tree structure that results from this data with a minimum support threshold of 6. The tree itself is shown with solid arrows. The numbers at each node show how many times the sorted prefix of items, up to and including the item at that node, occur in the dataset. For example, following the third branch from the left in the tree we can see that, after sorting, two instances begin with the prefix *humidity = high*—that is, the second and last instances of Table 6.1(c). Continuing down that branch, the next node records that the same two instances also have *windy = true* as their next most frequent item. The lowest node in the branch shows that one of these two instances—that is, the last in Table 6.1(c)—contains *temperature = mild* as well. The other instance—that is, the second in Table 6.1(c)—drops out at this stage because its next most frequent item does not meet the minimum support constraint and is therefore omitted from the tree.

On the left side of the diagram a "header table" shows the frequencies of the individual items in the dataset (Table 6.1(b)). These items appear in descending frequency order, and only those with at least minimum support are included. Each item in the header table points to its first occurrence in the tree, and subsequent items in the tree with the same name are linked together to form a list. These lists, emanating from the header table, are shown in Figure 6.8(a) by dashed arrows.

It is apparent from the tree that only two nodes have counts that satisfy the minimum support threshold, corresponding to the item sets *play = yes* (count of 9) and *play = yes* and *windy = false* (count of 6) in the leftmost branch. Each entry in the header table is itself a single-item set that also satisfies the threshold. This identifies as part of the final answer all the bold items in Table 6.1(b) and the first item set in Table 6.1(d). Since we know the outcome in advance we can see that there is only one more item set to go—the second in Table 6.1(d). But there is no hint of it in the data structure of Figure 6.8(a), and we will have to do a lot of work to discover it!

Finding Large Item Sets

The purpose of the links from the header table into the tree structure is to facilitate traversal of the tree to find other large item sets, apart from the two that are already in the tree. This is accomplished by a divide-and-conquer approach that recursively processes the tree to grow large item sets. Each header table list is followed in turn, starting from the bottom of the table and working upward. Actually, the header table can be processed in any order, but it is easier to think about processing the longest paths in the tree first, and these correspond to the lower-frequency items.

Starting from the bottom of the header table, we can immediately add *temperature = mild* to the list of large item sets. Figure 6.8(b) shows the result of the next stage, which is an FP-tree for just those instances in the dataset that include

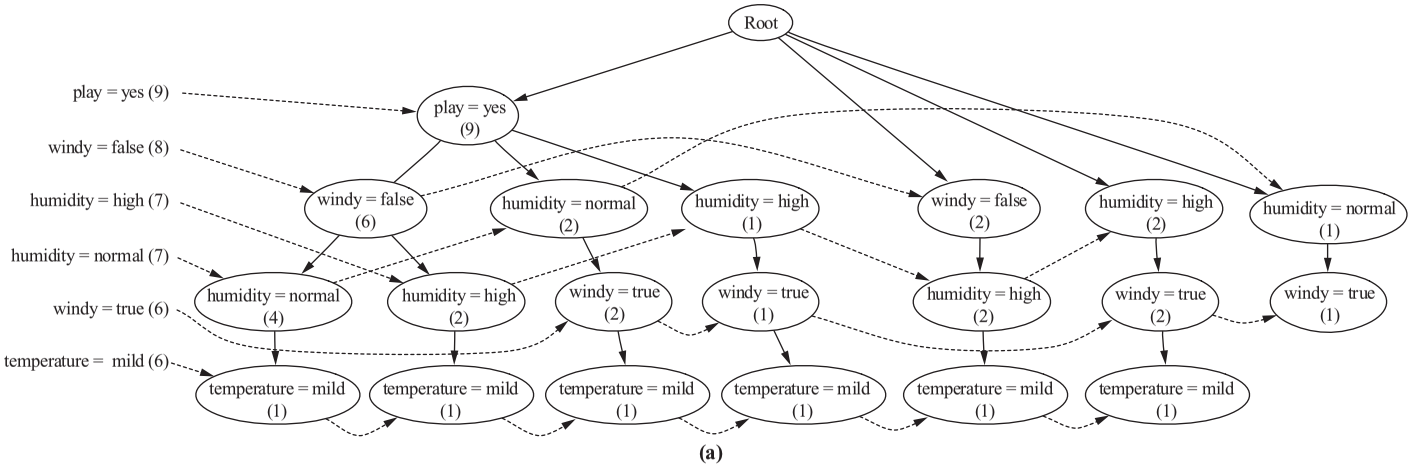


FIGURE 6.8

Extended prefix trees for the weather data: (a) the full data, (b) the data conditional on *temperature = mild*, and (c) the data conditional on *humidity = normal*.

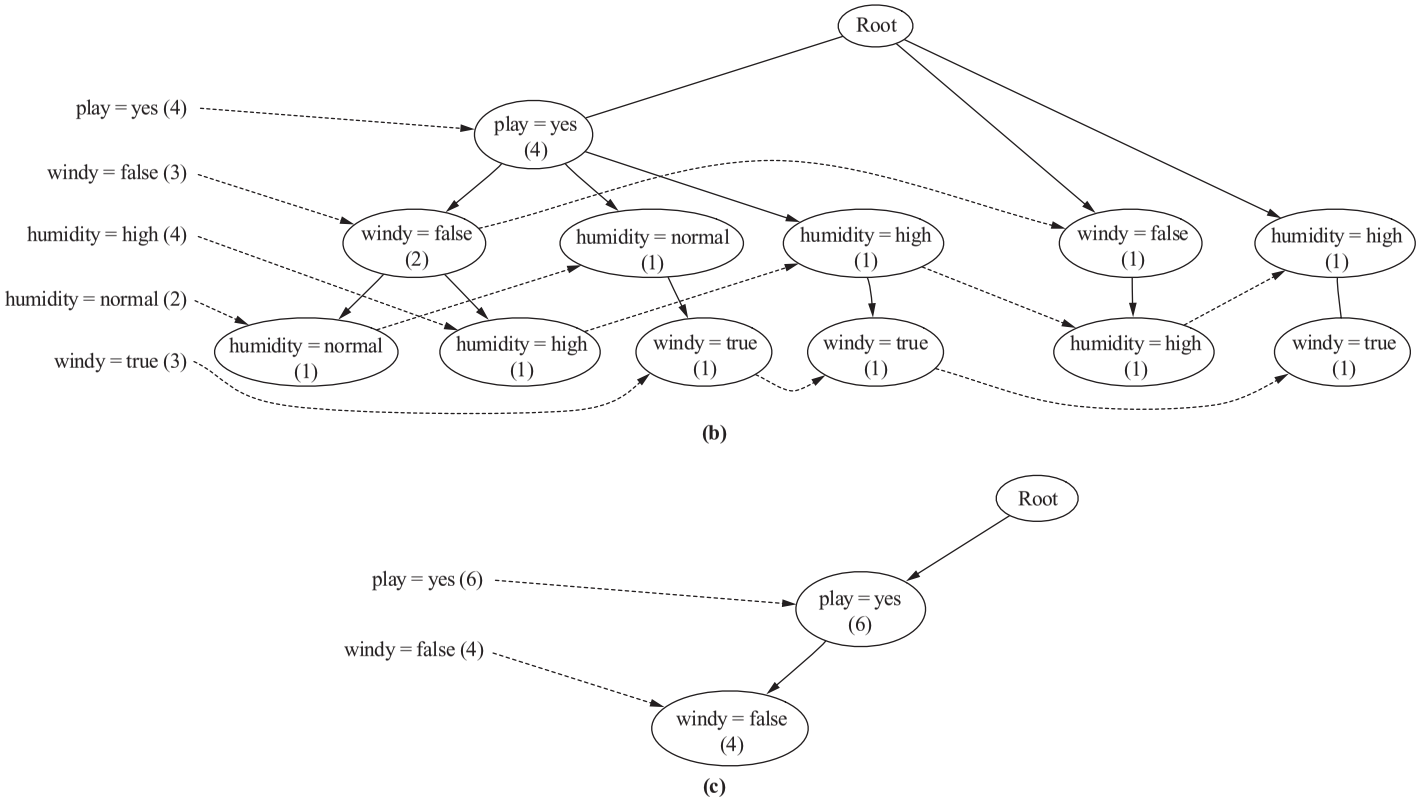


FIGURE 6.8, cont'd

temperature = mild. This tree was not created by rescanning the dataset but by further processing of the tree in Figure 6.8(a), as follows.

To see if a larger item set containing *temperature = mild* can be grown, we follow its link from the header table. This allows us to find all instances that contain *temperature = mild*. From here the new tree in Figure 6.8(b) is created, with counts projected from the original tree corresponding to the set of instances that are conditional on the presence of *temperature = mild*. This is done by propagating the counts from the *temperature = mild* nodes up the tree, each node receiving the sum of its children's counts.

A quick glance at the header table for this new FP-tree shows that the *temperature = mild* pattern cannot be grown any larger because there are no individual items, conditional on *temperature = mild*, that meet the minimum support threshold. Note, however, that it is necessary to create the whole Figure 6.8(b) tree in order to discover this because it is effectively being created bottom up and the counts in the header table to the left are computed from the numbers in the tree. The recursion exits at this point, and processing continues on the remaining header table items in the original FP-tree.

Figure 6.8(c) shows a second example, the FP-tree that results from following the header table link for *humidity = normal*. Here the *windy = false* node has a count of 4, corresponding to the four instances that had *humidity = normal* in the node's left branch in the original tree. Similarly, *play = yes* has a count of 6, corresponding to the four instances from *windy = false* and the two instances that contain *humidity = normal* from the middle branch of the subtree rooted at *play = yes* in Figure 6.8(a).

Processing the header list for this FP-tree shows that the *humidity = normal* item set can be grown to include *play = yes* since these two occur together six times, which meets the minimum support constraint. This corresponds to the second item set in Table 6.1(d), which in fact completes the output. However, in order to be sure that there are no other eligible item sets it is necessary to continue processing the entire header link table in Figure 6.8(a).

Once the recursive tree mining process is complete all large item sets that meet the minimum support threshold have been found. Then association rules are created using the approach explained in Section 4.5. Studies have claimed that the FP-growth algorithm is up to an order of magnitude faster than Apriori at finding large item sets, although this depends on the details of the implementation and the nature of the dataset.

Discussion

The process of recursively creating projected FP-trees can be efficiently implemented within a single prefix tree structure by having a list of frequencies, indexed by recursion depth, at each node in the tree and each element of the header table. The tree structure itself is usually far smaller than the original dataset, and if the dataset is dense it achieves a high level of compression. This outweighs the overhead imposed by the pointers and counters that must be maintained at each node. Only when the support threshold is set very low does the FP-tree's ability to compress the dataset degrade. Under these conditions, the tree becomes bushy, with little node sharing. On massive datasets for which the frequent-pattern tree exceeds main memory,

disk-resident trees can be constructed using indexing techniques that have been developed for relational database systems.

The FP-tree data structure and FP-growth algorithm for finding large item sets without candidate generation were introduced by Han et al. (2000) following pioneering work by Zaki et al. (1997); Han et al. (2004) give a more comprehensive description. It has been extended in various ways. Wang et al. (2003) develop an algorithm called CLOSET+ to mine closed item sets—that is, sets for which there is no proper superset that has the same support. Finding large closed item sets provides essentially the same information as finding the complete set of large item sets, but produces few redundant rules and thus eases the task that users face when examining the output of the mining process. GSP (Generalized Sequential Patterns) is a method based on the Apriori algorithm for mining patterns in databases of event sequences (Srikant and Agrawal, 1996). A similar approach to FP-growth is used for event sequences by algorithms called PrefixSpan (Pei et al., 2004) and CloSpan (Yan et al., 2003), and for graph patterns by algorithms called gSpan (Yan and Han, 2002) and CloseGraph (Yan and Han, 2003).

Ceglar and Roddick (2006) provide a comprehensive survey of association rule mining. Some authors have worked on integrating association rule mining with classification. For example, Liu et al. (1998) mine a kind of association rule that they call a "class association rule," and build a classifier on the rules that are found using a technique they call CBA (Classification Based on Associations). Mutter et al. (2004) use classification to evaluate the output of confidence-based association rule mining, and find that standard learners for classification rules are generally preferable to CBA when runtime and size of the rule sets is of concern.

6.4 EXTENDING LINEAR MODELS

Section 4.6 described how simple linear models can be used for classification in situations where all attributes are numeric. Their biggest disadvantage is that they can only represent linear boundaries between classes, which makes them too simple for many practical applications. Support vector machines use linear models to implement nonlinear class boundaries. (Although it is a widely used term, *support vector machines* is something of a misnomer: These are algorithms, not machines.) How can this be possible? The trick is easy: Transform the input using a nonlinear mapping. In other words, transform the instance space into a new space. With a nonlinear mapping, a straight line in the new space doesn't look straight in the original instance space. A linear model constructed in the new space can represent a nonlinear decision boundary in the original space.

Imagine applying this idea directly to the ordinary linear models in Section 4.6. For example, the original set of attributes could be replaced by one giving all products of n factors that can be constructed from these attributes. An example for two attributes, including all products with three factors, is

$$x = w_1 a_1^3 + w_2 a_1^2 a_2 + w_3 a_1 a_2^2 + w_4 a_2^3$$

Here, x is the outcome, a_1 and a_2 are the two attribute values, and there are four weights w_i to be learned. As described in Section 4.6, the result can be used for classification by training one linear system for each class and assigning an unknown instance to the class that gives the greatest output x—the standard technique of multiresponse linear regression. Then, a_1 and a_2 will be the attribute values for the test instance.

To generate a linear model in the space that is spanned by these products, each training instance is mapped into the new space by computing all possible three-factor products of its two attribute values. The learning algorithm is then applied to the transformed instances. To classify an instance, it is processed by the same transformation prior to classification. There is nothing to stop us from adding in more synthetic attributes. For example, if a constant term were included, the original attributes and all two-factor products of them would yield a total of 10 weights to be learned. (Alternatively, adding an additional attribute with a value that was always a constant would have the same effect.) Indeed, polynomials of sufficiently high degree can approximate arbitrary decision boundaries to any required accuracy.

It seems too good to be true—and it is. As you will probably have guessed, problems arise with this procedure due to the large number of coefficients introduced by the transformation in any realistic setting. The first snag is computational complexity. With 10 attributes in the original dataset, suppose we want to include all products with five factors: then the learning algorithm will have to determine more than 2000 coefficients. If its runtime is cubic in the number of attributes, as it is for linear regression, training will be infeasible. That is a problem of practicality. The second problem is one of principle: overfitting. If the number of coefficients is large relative to the number of training instances, the resulting model will be "too nonlinear"—it will overfit the training data. There are just too many parameters in the model.

Maximum-Margin Hyperplane

Support vector machines address both problems. They are based on an algorithm that finds a special kind of linear model: the *maximum-margin hyperplane*. We already know what a hyperplane is—it's just another term for a linear model. To visualize a maximum-margin hyperplane, imagine a two-class dataset whose classes are linearly separable—that is, there is a hyperplane in instance space that classifies all training instances correctly. The maximum-margin hyperplane is the one that gives the greatest separation between the classes—it comes no closer to either than it has to. An example is shown in Figure 6.9, where the classes are represented by open and filled circles, respectively. Technically, the *convex hull* of a set of points is the tightest enclosing convex polygon: Its outline emerges when you connect every point of the set to every other point. Because we have supposed that the two classes are linearly separable, their convex hulls cannot overlap. Among all hyperplanes that separate the classes, the maximum-margin hyperplane is the one that is as far as

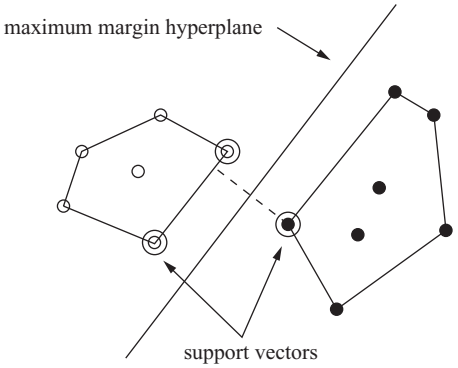


FIGURE 6.9

A maximum-margin hyperplane.

possible from both convex hulls—it is the perpendicular bisector of the shortest line connecting the hulls (shown dashed in the figure).

The instances that are closest to the maximum-margin hyperplane—the ones with the minimum distance to it—are called *support vectors*. There is always at least one support vector for each class, and often there are more. The important thing is that the set of support vectors uniquely defines the maximum-margin hyperplane for the learning problem. Given the support vectors for the two classes, we can easily construct the maximum-margin hyperplane. All other training instances are irrelevant—they can be deleted without changing the position and orientation of the hyperplane.

A hyperplane separating the two classes might be written as

$$x = w_0 + w_1 a_1 + w_2 a_2$$

in the two-attribute case, where a_1 and a_2 are the attribute values and there are three weights w_i to be learned. However, the equation defining the maximum-margin hyperplane can be written in another form, in terms of the support vectors. Write the class value y of a training instance as either 1 (for *yes*, it is in this class) or −1 (for *no*, it is not). Then the maximum-margin hyperplane can be written as

$$x = b + \sum_{i \text{ is support vector}} \alpha_i y_i \mathbf{a(i)} \bullet \mathbf{a}$$

Here, y_i is the class value of training instance **a(i)**, while b and α_i are numeric parameters that have to be determined by the learning algorithm. Note that **a(i)** and **a** are vectors. The vector **a** represents a test instance—just as the vector $[a_1, a_2]$ represented a test instance in the earlier formulation. The vectors **a(i)** are the support vectors, those circled in Figure 6.9; they are selected members of the training set. The term **a(i)** • **a** represents the dot product of the test instance with one of the support vectors: **a(i)** • **a** = $\Sigma_j a(i)_j a_j$. If you are not familiar with dot product notation, you should still be able to understand the gist of what follows: Just think of **a(i)** as the whole set of attribute values for the ith support vector. Finally, b and α_i are parameters that determine the hyperplane, just as the weights w_0, w_1, and w_2 are parameters that determine the hyperplane in the earlier formulation.

It turns out that finding the support vectors for the training instances and determining the parameters b and α_i belongs to a standard class of optimization problems known as *constrained quadratic optimization*. There are off-the-shelf software packages for solving these problems (see Fletcher, 1987, for a comprehensive and practical account of solution methods). However, the computational complexity can be reduced, and learning accelerated, if special-purpose algorithms for training support vector machines are applied—but the details of these algorithms lie beyond the scope of this book (see Platt, 1998).

Nonlinear Class Boundaries

We motivated the introduction of support vector machines by claiming that they can be used to model nonlinear class boundaries. However, so far we have only described the linear case. Consider what happens when an attribute transformation, as described before, is applied to the training data before determining the maximum-margin hyperplane. Recall that there are two problems with the straightforward application of such transformations to linear models: computational complexity on the one hand and overfitting on the other.

With support vectors, overfitting is unlikely to occur. The reason is that it is inevitably associated with instability: With an algorithm that overfits, changing one or two instance vectors will make sweeping changes to large sections of the decision boundary. But the maximum-margin hyperplane is relatively stable: It only moves if training instances are added or deleted that are support vectors—and this is true even in the high-dimensional space spanned by the nonlinear transformation. Overfitting is caused by too much flexibility in the decision boundary. The support vectors are global representatives of the whole set of training points, and there are usually few of them, which gives little flexibility. Thus, overfitting is less likely to occur.

What about computational complexity? This is still a problem. Suppose that the transformed space is a high-dimensional one so that the transformed support vectors and test instance have many components. According to the preceding equation, every time an instance is classified its dot product with all support vectors must be calculated. In the high-dimensional space produced by the nonlinear mapping this is rather expensive. Obtaining the dot product involves one multiplication and one addition for each attribute, and the number of attributes in the new space can be huge. This problem occurs not only during classification but also during training because the optimization algorithms have to calculate the same dot products very frequently. Fortunately, it turns out that it is possible to calculate the dot product *before* the nonlinear mapping is performed, on the original attribute set, using a so-called kernel function based on the dot product.

A high-dimensional version of the preceding equation is simply

$$x = b + \sum \alpha_i y_i (\mathbf{a(i)} \bullet \mathbf{a})^n$$

where n is chosen as the number of factors in the transformation (three in the example we used earlier). If you expand the term $(\mathbf{a(i)} \bullet \mathbf{a})^n$, you will find that it contains all the high-dimensional terms that would have been involved if the test and training vectors were first transformed by including all products of n factors and the dot product of the result was taken. (If you actually do the calculation, you will notice that some constant factors—binomial coefficients—are introduced. However, these do not matter: It is the dimensionality of the space that concerns us; the constants merely scale the axes.)

Because of this mathematical equivalence, the dot products can be computed in the original low-dimensional space, and the problem becomes feasible. In implementation terms, you take a software package for constrained quadratic optimization and every time

a(i) • a is evaluated you evaluate (**a(i) • a**)^n instead. It's as simple as that because in both the optimization and the classification algorithms these vectors are only used in this dot product form. The training vectors, including the support vectors, and the test instance all remain in the original low-dimensional space throughout the calculations.

The function (**x** • **y**)^n, which computes the dot product of two vectors **x** and **y** and raises the result to the power n, is called a *polynomial kernel.* A good way of choosing the value of n is to start with 1 (a linear model) and increment it until the estimated error ceases to improve. Usually, quite small values suffice. To include lower-order terms, we can use the kernel (**x** • **y** + 1)^n.

Other kernel functions can be used instead to implement different nonlinear mappings. Two that are often suggested are the *radial basis function (RBF) kernel* and the *sigmoid kernel.* Which one produces the best results depends on the application, although the differences are rarely large in practice. It is interesting to note that a support vector machine with the RBF kernel is simply a type of neural network called an *RBF network* (which we describe later), and one with the sigmoid kernel implements another type of neural network, a multilayer perceptron with one hidden layer (also described later).

Mathematically, any function K(**x**, **y**) is a kernel function if it can be written as K(**x**, **y**) = Φ(**x**) • Φ(**y**), where Φ is a function that maps an instance into a (potentially high-dimensional) feature space. In other words, the kernel function represents a dot product in the feature space created by Φ. Practitioners sometimes apply functions that are not proper kernel functions (the sigmoid kernel with certain parameter settings is an example). Despite the lack of theoretical guarantees, this can nevertheless produce accurate classifiers.

Throughout this section, we have assumed that the training data is linearly separable—either in the instance space or in the new space spanned by the nonlinear mapping. It turns out that support vector machines can be generalized to the case where the training data is not separable. This is accomplished by placing an upper bound on the coefficients α_i. Unfortunately, this parameter must be chosen by the user, and the best setting can only be determined by experimentation. Also, except in trivial cases it is not possible to determine a priori whether the data is linearly separable or not.

Finally, we should mention that compared with other methods such as decision tree learners, even the fastest training algorithms for support vector machines are slow when applied in the nonlinear setting. However, they often produce very accurate classifiers because subtle and complex decision boundaries can be obtained.

Support Vector Regression

The maximum-margin hyperplane concept only applies to classification. However, support vector machine algorithms have been developed for numeric prediction that share many of the properties encountered in the classification case: They

produce a model that can usually be expressed in terms of a few support vectors and can be applied to nonlinear problems using kernel functions. As with regular support vector machines, we will describe the concepts involved, but will not attempt to describe the algorithms that actually perform the work.

As with linear regression, covered in Section 4.6, the basic idea is to find a function that approximates the training points well by minimizing the prediction error. The crucial difference is that all deviations up to a user-specified parameter ε are simply discarded. Also, when minimizing the error, the risk of overfitting is reduced by simultaneously trying to maximize the flatness of the function. Another difference is that what is minimized is normally the predictions' absolute error instead of the squared error used in linear regression. (There are, however, versions of the algorithm that use the squared error instead.)

A user-specified parameter ε defines a tube around the regression function in which errors are ignored: For linear support vector regression, the tube is a cylinder. If all training points can fit within a tube of width 2ε, the algorithm outputs the function in the middle of the flattest tube that encloses them. In this case the total perceived error is 0. Figure 6.10(a) shows a regression problem with one attribute,

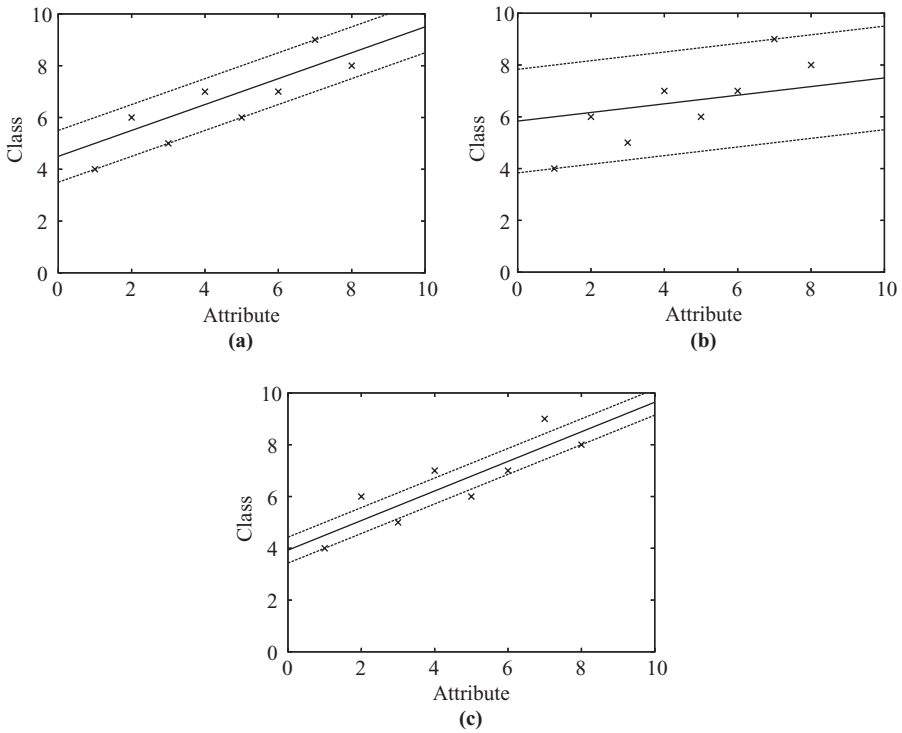


FIGURE 6.10

Support vector regression: (a) $\varepsilon = 1$, (b) $\varepsilon = 2$, and (c) $\varepsilon = 0.5$.

a numeric class, and eight instances. In this case ε was set to 1, so the width of the tube around the regression function (indicated by dotted lines) is 2. Figure 6.10(b) shows the outcome of the learning process when ε is set to 2. As you can see, the wider tube makes it possible to learn a flatter function.

The value of ε controls how closely the function will fit the training data. Too large a value will produce a meaningless predictor—in the extreme case, when 2ε exceeds the range of class values in the training data, the regression line is horizontal and the algorithm just predicts the mean class value. On the other hand, for small values of ε there may be no tube that encloses all the data. In that case, some training points will have nonzero error, and there will be a tradeoff between the prediction error and the tube's flatness. In Figure 6.10(c), ε was set to 0.5 and there is no tube of width 1 that encloses all the data.

For the linear case, the support vector regression function can be written as

$$x = b + \sum_{i \text{ is support vector}} \alpha_i \mathbf{a(i)} \bullet \mathbf{a}$$

As with classification, the dot product can be replaced by a kernel function for nonlinear problems. The support vectors are all those points that do not fall strictly within the tube—that is, the points outside the tube and on its border. As with classification, all other points have coefficient 0 and can be deleted from the training data without changing the outcome of the learning process. In contrast to the classification case, the α_i may be negative.

We have mentioned that as well as minimizing the error, the algorithm simultaneously tries to maximize the flatness of the regression function. In Figures 6.10(a) and (b), where there is a tube that encloses all the training data, the algorithm simply outputs the flattest tube that does so. However, in Figure 6.10(c), there is no tube with error 0, and a tradeoff is struck between the prediction error and the tube's flatness. This tradeoff is controlled by enforcing an upper limit C on the absolute value of the coefficients α_i. The upper limit restricts the influence of the support vectors on the shape of the regression function and is a parameter that the user must specify in addition to ε. The larger C is, the more closely the function can fit the data. In the degenerate case $\varepsilon = 0$, the algorithm simply performs least-absolute-error regression under the coefficient size constraint and all training instances become support vectors. Conversely, if ε is large enough that the tube can enclose all the data, the error becomes 0, there is no tradeoff to make, and the algorithm outputs the flattest tube that encloses the data irrespective of the value of C.

Kernel Ridge Regression

Chapter 4 introduced classic least-squares linear regression as a technique for predicting numeric quantities. In the previous section we saw how the powerful idea of support vector machines can be applied to regression and, furthermore, how nonlinear problems can be tackled by replacing the dot product in the support vector formulation by a kernel function—this is often known as the "kernel trick." For

classic linear regression using squared loss, only simple matrix operations are needed to find the model, but this is not the case for support vector regression with the user-specified loss parameter ε. It would be nice to combine the power of the kernel trick with the simplicity of standard least-squares regression. Kernel ridge regression does just that. In contrast to support vector regression, it does not ignore errors smaller than ε, and the squared error is used instead of the absolute error.

Instead of expressing the linear regression model's predicted class value for a given test instance $\mathbf{a}$ as a weighted sum of the attribute values, as in Chapter 4, it can be expressed as a weighted sum over the dot products of each training instance $\mathbf{a}_j$ and the test instance in question:

$$\sum_{j=0}^{n} \alpha_j \mathbf{a}_j \bullet \mathbf{a}$$

where we assume that the function goes through the origin and an intercept is not required. This involves a coefficient α_j for each training instance, which resembles the situation with support vector machines—except that here j ranges over *all* instances in the training data, not just the support vectors. Again, the dot product can be replaced by a kernel function to yield a nonlinear model.

The sum of the squared errors of the model's predictions on the training data is given by

$$\sum_{i=1}^{n} \left(y_i - \sum_{j=0}^{n} \alpha_j \mathbf{a}_j \bullet \mathbf{a}_i \right)^2$$

This is the squared loss, just as in Chapter 4, and again we seek to minimize it by choosing appropriate α_j's. But now there is a coefficient for each training instance, not just for each attribute, and most data sets have far more instances than attributes. This means that there is a serious risk of overfitting the training data when a kernel function is used instead of the dot product to obtain a nonlinear model.

That is where the *ridge* part of kernel ridge regression comes in. Instead of minimizing the squared loss, we trade closeness of fit against model complexity by introducing a penalty term:

$$\sum_{i=1}^{n} \left(y_i - \sum_{j=0}^{n} \alpha_j \mathbf{a}_j \cdot \mathbf{a}_i \right)^2 + \lambda \sum_{i,j=1}^{n} \alpha_i \alpha_j \mathbf{a}_j \bullet \mathbf{a}_i$$

The second sum penalizes large coefficients. This prevents the model from placing too much emphasis on individual training instances by giving them large coefficients, unless this yields a correspondingly large drop in error. The parameter λ controls the tradeoff between closeness of fit and model complexity. When matrix operations are used to solve for the coefficients of the model, the ridge penalty also has the added benefit of stabilizing degenerate cases. For this reason, it is often applied in standard least-squares linear regression as well.

Although kernel ridge regression has the advantage over support vector machines of computational simplicity, one disadvantage is that there is no sparseness in the vector of coefficients—in other words, no concept of "support vectors." This makes a difference at prediction time because support vector machines have to sum only over the set of support vectors, not the entire training set.

In a typical situation with more instances than attributes, kernel ridge regression is more computationally expensive than standard linear regression, even when using the dot product rather than a kernel. This is because of the complexity of the matrix inversion operation used to find the model's coefficient vector. Standard linear regression requires inverting an $m \times m$ matrix, which has complexity $O(m^3)$, where m is the number of attributes in the data. Kernel ridge regression, on the other hand, involves an $n \times n$ matrix, with complexity $O(n^3)$ where n is the number of instances in the training data. Nevertheless, it is advantageous to use kernel ridge regression in cases where a nonlinear fit is desired, or where there are more attributes than training instances.

Kernel Perceptron

In Section 4.6 we introduced the perceptron algorithm for learning a linear classifier. It turns out that the kernel trick can also be used to upgrade this algorithm to learn nonlinear decision boundaries.

To see this, we first revisit the linear case. The perceptron algorithm repeatedly iterates through the training data instance by instance and updates the weight vector every time one of these instances is misclassified based on the weights learned so far. The weight vector is updated simply by adding or subtracting the instance's attribute values to or from it. This means that the final weight vector is just the sum of the instances that have been misclassified. The perceptron makes its predictions based on whether

$$\sum_i w_i a_i$$

is greater or less than 0, where w_i is the weight for the ith attribute and a_i the corresponding attribute value of the instance that we wish to classify. Instead, we could use

$$\sum_i \sum_j y(j)\, a'(j)_i\, a_i$$

Here, $\mathbf{a'(j)}$ is the jth misclassified training instance, $a'(j)_i$ its ith attribute value, and $y(j)$ its class value (either +1 or −1). To implement this we no longer keep track of an explicit weight vector: We simply store the instances that have been misclassified so far and use the previous expression to make a prediction.

It looks like we've gained nothing—in fact, the algorithm is much slower because it iterates through all misclassified training instances every time a prediction is made. However, closer inspection of this formula reveals that it can be expressed in terms of dot products between instances. First, swap the summation signs to yield

$$\sum_j y(j) \sum_i a'(j)_i\, a_i$$

The second sum is just a dot product between two instances and can be written as

$$\sum_j y(j)\, \mathbf{a'(j) \bullet a}$$

This rings a bell! A similar expression for support vector machines enabled the use of kernels. Indeed, we can apply exactly the same trick here and use a kernel function instead of the dot product. Writing this function as $K(\ldots)$ gives

$$\sum_j y(j) K(\mathbf{a'(j)}, \mathbf{a})$$

In this way the perceptron algorithm can learn a nonlinear classifier simply by keeping track of the instances that have been misclassified during the training process and using this expression to form each prediction.

If a separating hyperplane exists in the high-dimensional space implicitly created by the kernel function, this algorithm will learn one. However, it won't learn the maximum-margin hyperplane found by a support vector machine classifier. This means that classification performance is usually worse. On the plus side, the algorithm is easy to implement and supports incremental learning.

This classifier is called the *kernel perceptron*. It turns out that all sorts of algorithms for learning linear models can be upgraded by applying the kernel trick in a similar fashion. For example, logistic regression can be turned into *kernel logistic regression*. As we saw before, the same applies to regression problems: Linear regression can also be upgraded using kernels. Again, a drawback of these advanced methods for linear and logistic regression (if they are done in a straightforward manner) is that the solution is not "sparse": Every training instance contributes to the solution vector. In support vector machines and the kernel perceptron, only some of the training instances affect the solution, and this can make a big difference in computational efficiency.

The solution vector found by the perceptron algorithm depends greatly on the order in which the instances are encountered. One way to make the algorithm more stable is to use all the weight vectors encountered during learning, not just the final one, letting them vote on a prediction. Each weight vector contributes a certain number of votes. Intuitively, the "correctness" of a weight vector can be measured roughly as the number of successive trials after its inception in which it correctly classified subsequent instances and thus didn't have to be changed. This measure can be used as the number of votes given to the weight vector, giving an algorithm known as the *voted perceptron* that performs almost as well as a support vector machine. (Note that, as mentioned earlier, the various weight vectors in the voted perceptron don't need to be stored explicitly, and the kernel trick can be applied here too.)

Multilayer Perceptrons

Using a kernel is not the only way to create a nonlinear classifier based on the perceptron. In fact, kernel functions are a recent development in machine learning. Previously, neural network proponents used a different approach for nonlinear classification: They connected many simple perceptron-like models in a hierarchical structure. This can represent nonlinear decision boundaries.

Section 4.6 explained that a perceptron represents a hyperplane in instance space. We mentioned there that it is sometimes described as an artificial "neuron." Of course, human and animal brains successfully undertake very complex classification tasks—for example, image recognition. The functionality of each individual neuron that is in a brain is certainly not sufficient to perform these feats. How can they be solved by brainlike structures? The answer must lie in the fact that the neurons in the brain are massively interconnected, allowing a problem to be decomposed into subproblems that can be solved at the neuron level.

This observation inspired the development of artificial networks of neurons—neural nets.

Consider the simple dataset in Figure 6.11. Part (a) shows a two-dimensional instance space with four instances having classes 0 and 1, represented by white and black dots, respectively. No matter how you draw a straight line through this space, you will not be able to find one that separates all the black points from all the white ones. In other words, the problem is not linearly separable, and the simple perceptron algorithm will fail to generate a separating hyperplane (in this two-dimensional instance space a hyperplane is just a straight line). The situation is different in Figure 6.11(b) and Figure 6.11(c): Both these problems are linearly separable. The same holds for Figure 6.11(d), which shows two points in a one-dimensional instance space (in the case of one dimension the separating hyperplane degenerates to a separating point).

If you are familiar with propositional logic, you may have noticed that the four situations in Figure 6.11 correspond to four types of logical connectives. Figure 6.11(a) represents a logical XOR (exclusive-OR), where the class is 1 if and only if exactly one of the attributes has value 1. Figure 6.11(b) represents logical AND, where the class is 1 if and only if both attributes have value 1. Figure 6.11(c) represents OR, where the class is 0 only if both attributes have value 0. Figure 6.11(d) represents NOT, where the class is 0 if and only if the attribute has value 1. Because the last three are linearly separable, a perceptron can represent AND, OR, and NOT. Indeed, perceptrons for the corresponding datasets are shown in Figures 6.11(f–h), respectively. However, a simple perceptron cannot represent XOR because that is not linearly separable. To build a classifier for this type of problem a single perceptron is not sufficient—we need several of them.

Figure 6.11(e) shows a network with three perceptrons, or *units*, labeled A, B, and C. The first two are connected to what is sometimes called the *input layer* of the network, representing the attributes in the data. As in a simple perceptron, the input layer has an additional constant input called the *bias*. However, the third unit does not have any connections to the input layer. Its input consists of the output of units A and B (either 0 or 1) and another constant bias unit. These three units make up the *hidden layer* of the multilayer perceptron. They are called "hidden" because the units have no direct connection to the environment. This layer is what enables the system to represent XOR. You can verify this by trying all four possible combinations of input signals. For example, if attribute a_1 has value 1 and a_2 has value 1, then unit A will output 1 (because $1 \times 1 + 1 \times 1 + -0.5 \times 1 > 0$), unit B will output 0 (because $-1 \times 1 + -1 \times 1 + -1.5 \times 1 < 0$), and unit C will output 0 (because $1 \times 1 + 1 \times 0 + -1.5 \times 1 < 0$). This is the correct answer. Closer inspection of the behavior of the three units reveals that the first one represents OR, the second represents NAND (NOT combined with AND), and the third represents AND. Together they represent the expression (a_1 OR a_2) AND (a_1 NAND a_2), which is precisely the definition of XOR.

As this example illustrates, any expression from propositional calculus can be converted into a multilayer perceptron, because the three connectives AND, OR, and

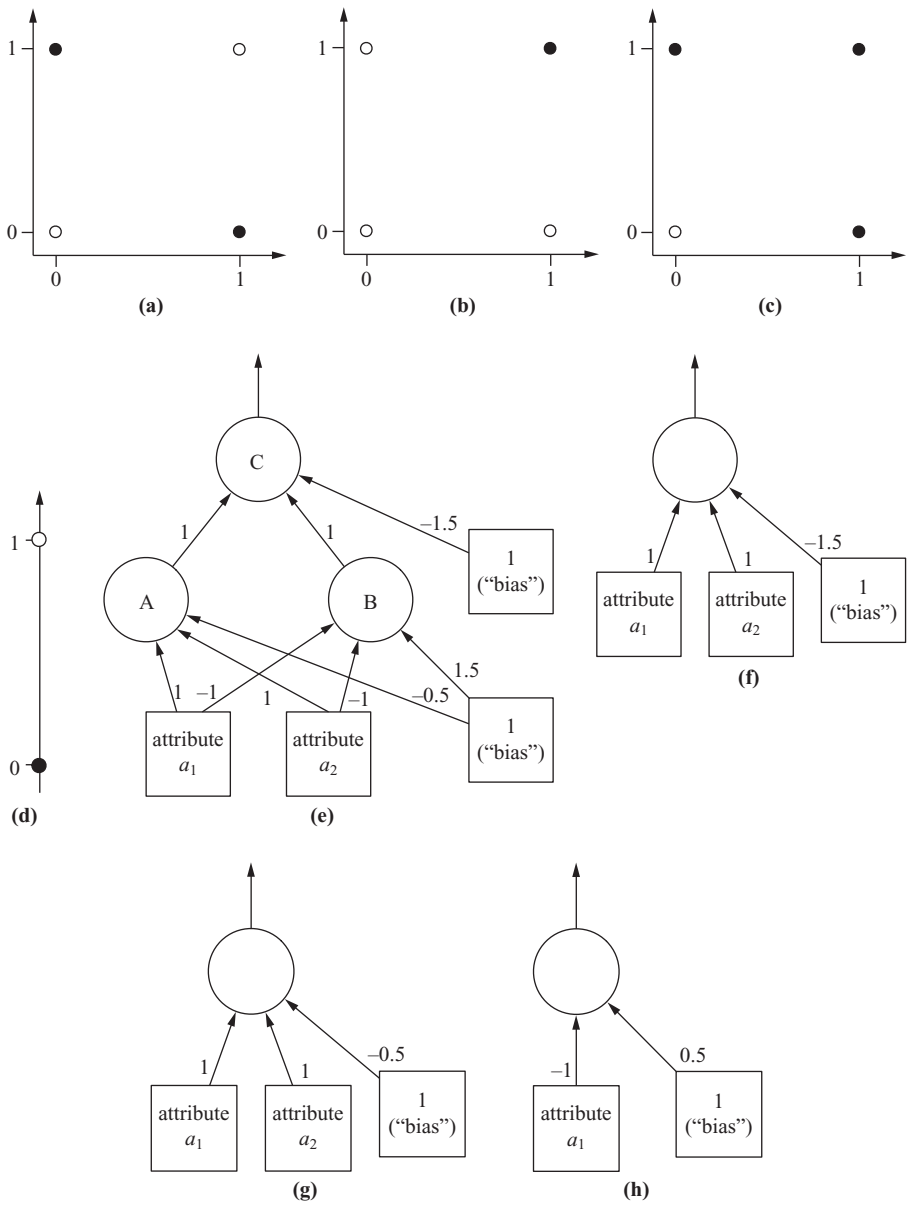


FIGURE 6.11

Example datasets and corresponding perceptrons.

NOT are sufficient for this and we have seen how each can be represented using a perceptron. Individual units can be connected together to form arbitrarily complex expressions. Hence, a multilayer perceptron has the same expressive power as, say, a decision tree. In fact, it turns out that a two-layer perceptron (not counting the input layer) is sufficient. In this case, each unit in the hidden layer corresponds to a variant of AND—because we assume that it may negate some of the inputs before forming the conjunction—joined by an OR that is represented by a single unit in the output layer. In other words, each node in the hidden layer has the same role as a leaf in a decision tree or a single rule in a set of decision rules.

The big question is how to learn a multilayer perceptron. There are two aspects to the problem: learning the structure of the network and learning the connection weights. It turns out that there is a relatively simple algorithm for determining the weights given a fixed network structure. This algorithm is called *backpropagation* and is described in the next section. However, although there are many algorithms that attempt to identify network structure, this aspect of the problem is commonly solved by experimentation—perhaps combined with a healthy dose of expert knowledge. Sometimes the network can be separated into distinct modules that represent identifiable subtasks (e.g., recognizing different components of an object in an image recognition problem), which opens up a way of incorporating domain knowledge into the learning process. Often a single hidden layer is all that is necessary, and an appropriate number of units for that layer is determined by maximizing the estimated accuracy.

Backpropagation

Suppose we have some data and seek a multilayer perceptron that is an accurate predictor for the underlying classification problem. Given a fixed network structure, we must determine appropriate weights for the connections in the network. In the absence of hidden layers, the perceptron learning rule from Section 4.6 can be used to find suitable values. But suppose there are hidden units. We know what the output unit should predict and could adjust the weights of the connections leading to that unit based on the perceptron rule. But the correct outputs for the hidden units are unknown, so the rule cannot be applied there.

It turns out that, roughly speaking, the solution is to modify the weights of the connections leading to the hidden units based on the strength of each unit's contribution to the final prediction. There is a standard mathematical optimization algorithm, called *gradient descent*, which achieves exactly that. Unfortunately, it requires taking derivatives, and the step function that the simple perceptron uses to convert the weighted sum of the inputs into a 0/1 prediction is not differentiable. We need to see whether the step function can be replaced by something else.

Figure 6.12(a) shows the step function: If the input is smaller than 0, it outputs 0; otherwise, it outputs 1. We want a function that is similar in shape but differentiable. A commonly used replacement is shown in Figure 6.12(b). In neural networks terminology it is called the *sigmoid* function, and the version we consider here is defined by

$$f(x) = \frac{1}{1+e^{-x}}$$

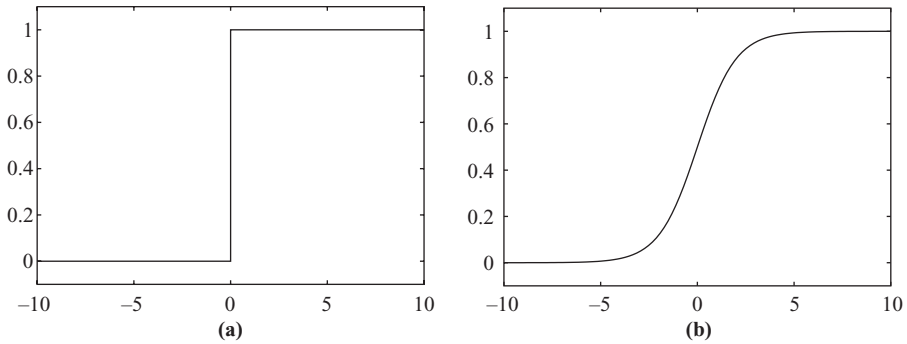


FIGURE 6.12

Step versus sigmoid: (a) step function and (b) sigmoid function.

We encountered it in Section 4.6 when we described the logit transform used in logistic regression. In fact, learning a multilayer perceptron is closely related to logistic regression.

To apply the gradient descent procedure, the error function—the thing that is to be minimized by adjusting the weights—must also be differentiable. The number of misclassifications—measured by the discrete 0 – 1 loss mentioned in Section 5.6—does not fulfill this criterion. Instead, multilayer perceptrons are usually trained by minimizing the squared error of the network's output, essentially treating it as an estimate of the class probability. (Other loss functions are also applicable. For example, if the negative log-likelihood is used instead of the squared error, learning a sigmoid-based perceptron is identical to logistic regression.)

We work with the squared-error loss function because it is most widely used. For a single training instance, it is

$$E = \frac{1}{2}(y - f(x))^2$$

where $f(x)$ is the network's prediction obtained from the output unit and y is the instance's class label (in this case, it is assumed to be either 0 or 1). The factor $\frac{1}{2}$ is included just for convenience and will drop out when we start taking derivatives.

Gradient descent exploits information given by the derivative of the function that is to be minimized—in this case, the error function. As an example, consider a hypothetical error function that happens to be identical to $w^2 + 1$, shown in Figure 6.13. The x-axis represents a hypothetical parameter w that is to be optimized. The derivative of $w^2 + 1$ is simply $2w$. The crucial observation is that, based on the derivative, we can figure out the slope of the function at any particular point. If the derivative is negative, the function slopes downward to the right; if it is positive,

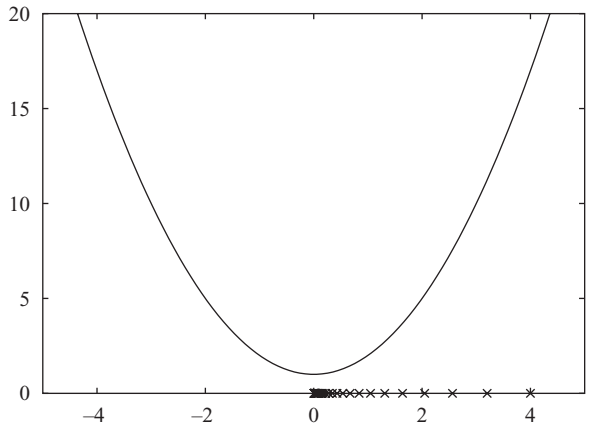


FIGURE 6.13

Gradient descent using the error function $w^2 + 1$.

it slopes downward to the left; and the size of the derivative determines how steep the decline is. Gradient descent is an iterative optimization procedure that uses this information to adjust a function's parameters. It takes the value of the derivative, multiplies it by a small constant called the *learning rate*, and subtracts the result from the current parameter value. This is repeated for the new parameter value, and so on, until a minimum is reached.

Returning to the example, assume that the learning rate is set to 0.1 and the current parameter value w is 4. The derivative is double this—8 at this point. Multiplying by the learning rate yields 0.8, and subtracting this from 4 gives 3.2, which becomes the new parameter value. Repeating the process for 3.2, we get 2.56, then 2.048, and so on. The little crosses in Figure 6.13 show the values encountered in this process. The process stops once the change in parameter value becomes too small. In the example this happens when the value approaches 0, the value corresponding to the location on the x-axis where the minimum of the hypothetical error function is located.

The learning rate determines the step size and hence how quickly the search converges. If it is too large and the error function has several minima, the search may overshoot and miss a minimum entirely, or it may oscillate wildly. If it is too small, progress toward the minimum may be slow. Note that gradient descent can only find a *local* minimum. If the function has several minima—and error functions for multilayer perceptrons usually have many—it may not find the best one. This is a significant drawback of standard multilayer perceptrons compared with, for example, support vector machines.

To use gradient descent to find the weights of a multilayer perceptron, the derivative of the squared error must be determined with respect to each parameter—that is, each weight in the network. Let's start with a simple perceptron without a hidden layer. Differentiating the error function with respect to a particular weight w_i yields

$$\frac{dE}{dw_i} = (f(x) - y)\frac{f(x)}{dw_i}$$

Here, $f(x)$ is the perceptron's output and x is the weighted sum of the inputs.

To compute the second factor on the right side, the derivative of the sigmoid function $f(x)$ is needed. It turns out that this has a particularly simple form that can be written in terms of $f(x)$ itself:

$$\frac{df(x)}{dx} = f(x)(1 - f(x))$$

We use $f'(x)$ to denote this derivative. But we seek the derivative with respect to w_i, not x. Because

$$x = \sum_i w_i a_i$$

the derivative of $f(x)$ with respect to w_i is

$$\frac{df(x)}{dw_i} = f'(x)\, a_i$$

Plugging this back into the derivative of the error function yields

$$\frac{dE}{dw_i} = (f(x) - y)f'(x)\, a_i$$

This expression gives all that is needed to calculate the change of weight w_i caused by a particular example vector **a** (extended by 1 to represent the bias, as explained previously). Having repeated this computation for each training instance, we add up the changes associated with a particular weight w_i, multiply by the learning rate, and subtract the result from w_i's current value.

So far so good. But all this assumes that there is no hidden layer. With a hidden layer, things get a little trickier. Suppose $f(x_i)$ is the output of the ith hidden unit, w_{ij} is the weight of the connection from input j to the ith hidden unit, and w_i is the weight of the ith hidden unit to the output unit. The situation is depicted in Figure 6.14. As before, $f(x)$ is the output of the single unit in the output layer. The update rule for the weights w_i is essentially the same as above, except that a_i is replaced by the output of the ith hidden unit:

$$\frac{dE}{dw_i} = (f(x) - y)f'(x)f(x_i)$$

However, to update the weights w_{ij} the corresponding derivatives must be calculated. Applying the chain rule gives

$$\frac{dE}{dw_{ij}} = \frac{dE}{dx}\frac{dx}{dw_{ij}} = (f(x) - y)f'(x)\frac{dx}{dw_{ij}}$$

The first two factors are the same as in the previous equation. To compute the third factor, differentiate further. Because

$$x = \sum_i w_i f(x_i)$$

then

$$\frac{dx}{dw_{ij}} = w_i \frac{df(x_i)}{dw_{ij}}$$

Furthermore,

$$x_i = \sum_j w_{ij} a_j$$

so

$$\frac{df(x_i)}{dw_{ij}} = f'(x_i)\frac{dx_i}{dw_{ij}} = f'(x_i)a_j$$

This means that we are finished. Putting everything together yields an equation for the derivative of the error function with respect to the weights w_{ij}:

$$\frac{dE}{dw_{ij}} = (f(x) - y)f'(x)w_i f'(x_i)\, a_j$$

As before, we calculate this value for every training instance, add up the changes associated with a particular weight w_{ij}, multiply by the learning rate, and subtract the outcome from the current value of w_{ij}.

This derivation applies to a perceptron with one hidden layer. If there are two hidden layers, the same strategy can be applied a second time to update the weights pertaining to the input connections of the first hidden layer, propagating the error from the output unit through the second hidden layer to the first one. Because of this error propagation mechanism, this version of the generic gradient descent strategy is called *backpropagation*.

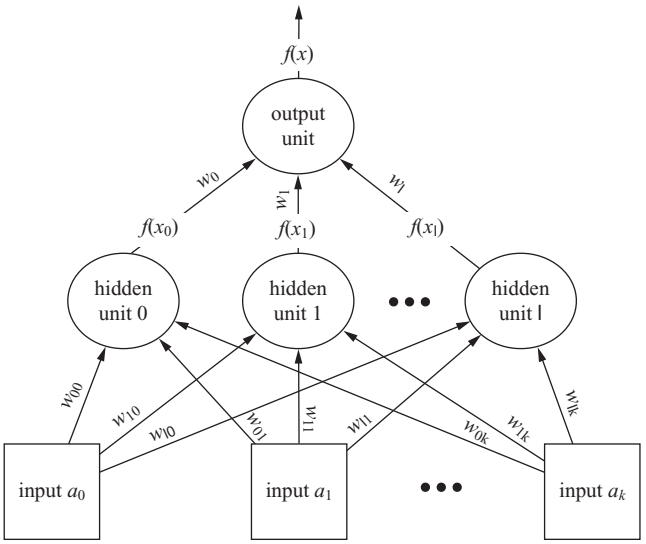


FIGURE 6.14

Multilayer perceptron with a hidden layer.

We have tacitly assumed that the network's output layer has just one unit, which is appropriate for two-class problems. For more than two classes, a separate network could be learned for each class that distinguishes it from the remaining classes. A more compact classifier can be obtained from a single network by creating an output unit for each class, connecting every unit in the hidden layer to every output unit.

The squared error for a particular training instance is the sum of squared errors taken over all output units. The same technique can be applied to predict several targets, or attribute values, simultaneously by creating a separate output unit for each one. Intuitively, this may give better predictive accuracy than building a separate classifier for each class attribute if the underlying learning tasks are in some way related.

We have assumed that weights are only updated after all training instances have been fed through the network and all the corresponding weight changes have been accumulated. This is *batch* learning because all the training data is processed together. But exactly the same formulas can be used to update the weights incrementally after each training instance has been processed. This is called *stochastic backpropagation* because the overall error does not necessarily decrease after every update. It can be used for online learning, in which new data arrives in a continuous stream and every training instance is processed just once. In both variants of backpropagation, it is often helpful to standardize the attributes to have zero mean and unit standard deviation. Before learning starts, each weight is initialized to a small, randomly chosen value based on a normal distribution with zero mean.

Like any other learning scheme, multilayer perceptrons trained with backpropagation may suffer from overfitting, especially if the network is much larger than what is actually necessary to represent the structure of the underlying learning problem. Many modifications have been proposed to alleviate this. A very simple one, called *early stopping*, works like reduced-error pruning in rule learners: A holdout set is used to decide when to stop performing further iterations of the backpropagation algorithm. The error on the holdout set is measured and the algorithm is terminated once the error begins to increase because that indicates overfitting to the training data. Another method, called *weight decay*, adds to the error function a penalty term that consists of the squared sum of all weights in the network, as in ridge regression. This attempts to limit the influence of irrelevant connections on the network's predictions by penalizing large weights that do not contribute a correspondingly large reduction in the error.

Although standard gradient descent is the simplest technique for learning the weights in a multilayer perceptron, it is by no means the most efficient one. In practice, it tends to be rather slow. A trick that often improves performance is to include a *momentum* term when updating weights: Add to the new weight change a small proportion of the update value from the previous iteration. This smoothes the search process by making changes in direction less abrupt. More sophisticated methods make use of information obtained from the second derivative of the error function as well; they can converge much more quickly. However, even those algorithms can be very slow compared with other methods of classification learning.

A serious disadvantage of multilayer perceptrons that contain hidden units is that they are essentially opaque. There are several techniques that attempt to extract rules from trained neural networks. However, it is unclear whether they offer any advantages over standard rule learners that induce rule sets directly from data, especially considering that this can generally be done much more quickly than learning a multilayer perceptron in the first place.

Although multilayer perceptrons are the most prominent type of neural network, many others have been proposed. Multilayer perceptrons belong to a class of networks called *feed-forward networks* because they do not contain any cycles and the network's output depends only on the current input instance. *Recurrent* neural networks do have cycles. Computations derived from earlier input are fed back into the network, which gives them a kind of memory.

Radial Basis Function Networks

Another popular type of feed-forward network is the *radial basis function* (RBF) network. It has two layers, not counting the input layer, and differs from a multilayer perceptron in the way that the hidden units perform computations. Each hidden unit essentially represents a particular point in input space, and its output, or *activation*, for a given instance depends on the distance between its point and the instance, which is just another point. Intuitively, the closer these two points, the stronger the activation. This is achieved by using a nonlinear transformation function to convert the distance into a similarity measure. A bell-shaped Gaussian *activation function*, of which the width may be different for each hidden unit, is commonly used for this purpose. The hidden units are called RBFs because the points in instance space for which a given hidden unit produces the same activation form a hypersphere or hyperellipsoid. (In a multilayer perceptron, this is a hyperplane.)

The output layer of an RBF network is the same as that of a multilayer perceptron: It takes a linear combination of the outputs of the hidden units and—in classification problems—pipes it through the sigmoid function (or something with a similar shape).

The parameters that such a network learns are (a) the centers and widths of the RBFs and (b) the weights used to form the linear combination of the outputs obtained from the hidden layer. A significant advantage over multilayer perceptrons is that the first set of parameters can be determined independently of the second set and still produce accurate classifiers.

One way to determine the first set of parameters is to use clustering. The simple *k*-means clustering algorithm described in Section 4.8 can be applied, clustering each class independently to obtain *k*-basis functions for each class. Intuitively, the resulting RBFs represent prototype instances. The second set of parameters is then learned by keeping the first parameters fixed. This involves learning a simple linear classifier using one of the techniques we have discussed (e.g., linear or logistic regression). If there are far fewer hidden units than training instances, this can be done very quickly.

A disadvantage of RBF networks is that they give every attribute the same weight because all are treated equally in the distance computation, unless attribute weight parameters are included in the overall optimization process. Thus, they cannot deal effectively with irrelevant attributes, in contrast to multilayer perceptrons. Support vector machines share the same problem. In fact, support vector machines with Gaussian kernels (i.e., "RBF kernels") are a particular type of RBF network, in which one basis function is centered on every training instance, all basis functions

have the same width, and the outputs are combined linearly by computing the maximum-margin hyperplane. This has the effect that only some of the RBFs have a nonzero weight—the ones that represent the support vectors.

Stochastic Gradient Descent

We have introduced gradient descent and stochastic backpropagation as optimization methods for learning the weights in a neural network. Gradient descent is, in fact, a general-purpose optimization technique that can be applied whenever the objective function is differentiable. Actually, it turns out that it can even be applied in cases where the objective function is not completely differentiable through use of a device called *subgradients*.

One application is the use of gradient descent to learn linear models such as linear support vector machines or logistic regression. Learning such models using gradient descent is easier than optimizing nonlinear neural networks because the objective function has a global minimum rather than many local minima, which is usually the case for nonlinear networks. For linear problems, a stochastic gradient descent procedure can be designed that is computationally simple and converges very rapidly, allowing models such as linear support vector machines and logistic regression to be learned from large datasets. Moreover, stochastic gradient descent allows models to be learned incrementally, in an online setting.

For support vector machines, the error function—the thing that is to be minimized—is called the *hinge loss*. Illustrated in Figure 6.15, this is so named because it comprises a downwards sloping linear segment joined to a horizontal part at $z = 1$—more formally, $E(z) = \max\{0, 1 - z\}$. For comparison, the figure also shows the 0 – 1 loss, which is discontinuous, and the squared loss, which is both continuous

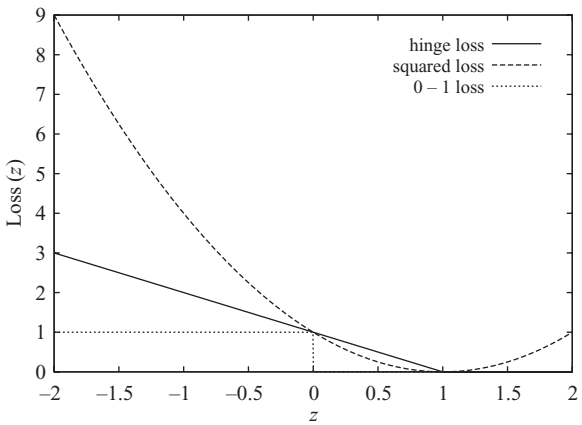


FIGURE 6.15

Hinge, squared, and 0 – 1 loss functions.

and differentiable. These functions are plotted as a function of the margin $z = y f(x)$, where the class y is either -1 or $+1$ and $f(x)$ is the output of the linear model. Misclassification occurs when $z < 0$, so all loss functions incur their most serious penalties in the negative region. In the linearly separable case, the hinge loss is 0 for a function that successfully separates the data. The maximum-margin hyperplane is given by the smallest weight vector that achieves a zero hinge loss.

The hinge loss is continuous, unlike the 0 – 1 loss, but is not differentiable at $z = 1$, unlike the squared loss, which is differentiable everywhere. This lack of differentiability presents a problem if gradient descent is used to update the model's weights after a training example has been processed, because the loss function's derivative is needed for this. That is where subgradients come in. The basic idea is that even though the gradient cannot be computed, the minimum will still be found if something resembling a gradient can be substituted. In the case of the hinge loss, the gradient is taken to be 0 at the point of nondifferentiability. In fact, since the hinge loss is 0 for $z \geq 1$, we can focus on that part of the function that is differentiable ($z < 1$) and proceed as usual.

Ignoring the weight decay necessary to find the smallest weight vector, the weight update for a linear support vector machine using the hinge loss is $\Delta w_i = \eta a_i y$, where η is the learning rate. For stochastic gradient descent, all that is needed to compute z for each training instance is to take the dot product between the current weight vector and the instance, multiply the result by the instance's class value, and check to see if the resulting value is less than 1. If so, the weights are updated accordingly. As with perceptrons, a bias term can be included by extending the weight vector by one element and including an additional attribute with each training instance that always has the value 1.

Discussion

Support vector machines originated from research in statistical learning theory (Vapnik, 1999), and a good starting point for exploration is a tutorial by Burges (1998). A general description, including generalization to the case in which the data is not linearly separable, has been published by Cortes and Vapnik (1995). We have introduced the standard version of support vector regression; Schölkopf et al. (1999) present a different version that has one parameter instead of two. Smola and Schölkopf (2004) provide an extensive tutorial on support vector regression.

Ridge regression was introduced in statistics by Hoerl and Kennard (1970) and can now be found in standard statistics texts. Hastie et al. (2009) give a good description of kernel ridge regression. Kernel ridge regression is equivalent to a technique called Gaussian process regression in terms of point estimates produced, but a discussion of Gaussian processes is beyond the scope of this book. The complexity of the most efficient general matrix inversion algorithm is in fact $O(n^{2.807})$ rather than $O(n^3)$.

The (voted) kernel perceptron is due to Freund and Schapire (1999). Cristianini and Shawe-Taylor (2000) provide a nice introduction to support vector machines

and other kernel-based methods, including the optimization theory underlying the support vector learning algorithms. We have barely skimmed the surface of these learning schemes, mainly because advanced mathematics lies just beneath. The idea of using kernels to solve nonlinear problems has been applied to many algorithms, for example, principal components analysis (described in Section 7.3). A kernel is essentially a similarity function with certain mathematical properties, and it is possible to define kernel functions over all sorts of structures—for example, sets, strings, trees, and probability distributions. Shawe-Taylor and Cristianini (2004) and Schölkopf and Smola (2002) cover kernel-based learning in detail.

There is extensive literature on neural networks, and Bishop (1995) provides an excellent introduction to both multilayer perceptrons and RBF networks. Interest in neural networks appears to have declined since the arrival of support vector machines, perhaps because the latter generally require fewer parameters to be tuned to achieve the same (or greater) accuracy. However, multilayer perceptrons have the advantage that they can learn to ignore irrelevant attributes, and RBF networks trained using *k*-means can be viewed as a quick-and-dirty method for finding a nonlinear classifier. Recent studies have shown that multilayer perceptrons achieve performance competitive with more modern learning techniques on many practical datasets.

Recently there has been renewed interest in gradient methods for learning classifiers. In particular, stochastic gradient methods have been explored because they are applicable to large data sets and online learning scenarios. Kivinen et al. (2002), Zhang (2004), and Shalev-Shwartz et al. (2007) explore such methods when applied to learning support vector machines. Kivinen et al. and Shalev-Shwartz et al. provide heuristics for setting the learning rate for gradient descent based on the current iteration, which only require the user to provide a value for a single parameter that determines the closeness of fit to the training data (a so-called regularization parameter). In the vanilla approach, regularization is performed by limiting the number of updates that can be performed.

6.5 INSTANCE-BASED LEARNING

In Section 4.7 we saw how the nearest-neighbor rule can be used to implement a basic form of instance-based learning. There are several practical problems with this simple scheme. First, it tends to be slow for large training sets because the entire set must be searched for each test instance—unless sophisticated data structures such as *k*D-trees or ball trees are used. Second, it performs badly with noisy data because the class of a test instance is determined by its single nearest neighbor without any "averaging" to help eliminate noise. Third, it performs badly when different attributes affect the outcome to different extents—in the extreme case, when some attributes are completely irrelevant—because all attributes contribute equally to the distance formula. Fourth, it does not perform explicit generalization, although we

intimated in Section 3.5 (and illustrated in Figure 3.10) that some instance-based learning systems do indeed perform explicit generalization.

Reducing the Number of Exemplars

The plain nearest-neighbor rule stores a lot of redundant exemplars. Yet it is almost always completely unnecessary to save all the examples seen so far. A simple variant is to classify each example with respect to the examples already seen and to save only ones that are misclassified. We use the term *exemplars* to refer to the already-seen instances that are used for classification. Discarding correctly classified instances reduces the number of exemplars and proves to be an effective way to prune the exemplar database. Ideally, only a single exemplar is stored for each important region of the instance space. However, early in the learning process examples may be discarded that later turn out to be important, possibly leading to some decrease in predictive accuracy. As the number of stored instances increases, the accuracy of the model improves, and so the system makes fewer mistakes.

Unfortunately, the strategy of only storing misclassified instances does not work well in the face of noise. Noisy examples are very likely to be misclassified, and so the set of stored exemplars tends to accumulate those that are least useful. This effect is easily observed experimentally. Thus, this strategy is only a stepping-stone on the way toward more effective instance-based learners.

Pruning Noisy Exemplars

Noisy exemplars inevitably lower the performance of any nearest-neighbor scheme that does not suppress them, because they have the effect of repeatedly misclassifying new instances. There are two ways of dealing with this. One is to locate, instead of the single nearest neighbor, the k-nearest neighbors for some predetermined constant k, and assign the majority class to the unknown instance. The only problem here is determining a suitable value of k. Plain nearest-neighbor learning corresponds to $k = 1$. The more noise, the greater the optimal value of k. One way to proceed is to perform cross-validation tests with several different values and choose the best. Although this is expensive in computation time, it often yields excellent predictive performance.

A second solution is to monitor the performance of each exemplar that is stored and discard ones that do not perform well. This can be done by keeping a record of the number of correct and incorrect classification decisions that each exemplar makes. Two predetermined thresholds are set on the success ratio. When an exemplar's performance drops below the lower one, it is deleted from the exemplar set. If its performance exceeds the upper threshold, it is used for predicting the class of new instances. If its performance lies between the two, it is not used for prediction but, whenever it is the closest exemplar to the new instance (and thus would have

been used for prediction if its performance record had been good enough), its success statistics are updated as though it had been used to classify that new instance.

To accomplish this, we use the confidence limits on the success probability of a Bernoulli process that we derived in Section 5.2. Recall that we took a certain number of successes S out of a total number of trials N as evidence on which to base confidence limits on the true underlying success rate p. Given a certain confidence level of, say, 5%, we can calculate upper and lower bounds and be 95% sure that p lies between them.

To apply this to the problem of deciding when to accept a particular exemplar, suppose that it has been used n times to classify other instances and that s of these have been successes. That allows us to estimate bounds, at a particular confidence level, on the true success rate of this exemplar. Now suppose that the exemplar's class has occurred c times out of a total number N of training instances. This allows us to estimate bounds on the default success rate—that is, the probability of successfully classifying an instance of this class without any information about other instances. We insist that the *lower* confidence bound on an exemplar's success rate exceeds the *upper* confidence bound on the default success rate. We use the same method to devise a criterion for rejecting a poorly performing exemplar, requiring that the *upper* confidence bound on its success rate lies below the *lower* confidence bound on the default success rate.

With suitable choices of thresholds, this scheme works well. In a particular implementation, called *IB3* for *Instance-Based Learner version 3*, a confidence level of 5% is used to determine acceptance whereas a level of 12.5% is used for rejection. The lower percentage figure produces a wider confidence interval, which makes for a more stringent criterion because it is harder for the lower bound of one interval to lie above the upper bound of the other. The criterion for acceptance is more stringent than for rejection, making it more difficult for an instance to be accepted. The reason for a less stringent rejection criterion is that there is little to be lost by dropping instances with only moderately poor classification accuracies: They will probably be replaced by similar instances later. Using these thresholds has been found to improve the performance of instance-based learning and, at the same time, dramatically reduce the number of exemplars—particularly noisy exemplars—that are stored.

Weighting Attributes

The Euclidean distance function, modified to scale all attribute values to between 0 and 1, works well in domains in which the attributes are equally relevant to the outcome. Such domains, however, are the exception rather than the rule. In most domains some attributes are irrelevant and some relevant ones are less important than others. The next improvement in instance-based learning is to learn the relevance of each attribute incrementally by dynamically updating feature weights.

In some schemes, the weights are class specific in that an attribute may be more important to one class than to another. To cater for this, a description is produced

for each class that distinguishes its members from members of all other classes. This leads to the problem that an unknown test instance may be assigned to several different classes, or no classes at all—a problem that is all too familiar from our description of rule induction. Heuristic solutions are applied to resolve these situations.

The distance metric incorporates the feature weights w_1, w_2, …, w_m on each dimension:

$$\sqrt{w_1^2(x_1 - y_1)^2 + w_2^2(x_2 - y_2)^2 + \ldots + w_m^2(x_m - y_m)^2}$$

In the case of class-specific feature weights, there will be a separate set of weights for each class.

All attribute weights are updated after each training instance is classified, and the most similar exemplar (or the most similar exemplar of each class) is used as the basis for updating. Call the training instance x and the most similar exemplar y. For each attribute i, the difference $|x_i - y_i|$ is a measure of the contribution of that attribute to the decision. If this difference is small then the attribute contributes positively, whereas if it is large it may contribute negatively. The basic idea is to update the ith weight on the basis of the size of this difference and whether the classification was indeed correct. If the classification is correct the associated weight is increased, and if it is incorrect it is decreased, the amount of increase or decrease being governed by the size of the difference: large if the difference is small and vice versa. The weight change is generally followed by a renormalization step. A simpler strategy, which may be equally effective, is to leave the weights alone if the decision is correct, and if it is incorrect to increase the weights for those attributes that differ most greatly, accentuating the difference. Details of these weight adaptation algorithms are described by Aha (1992).

A good test of whether an attribute weighting scheme works is to add irrelevant attributes to all examples in a dataset. Ideally, the introduction of irrelevant attributes should not affect either the quality of predictions or the number of exemplars stored.

Generalizing Exemplars

Generalized exemplars are rectangular regions of instance space, called *hyperrectangles* because they are high-dimensional. When classifying new instances it is necessary to modify the distance function as described below to allow the distance to a hyperrectangle to be computed. When a new exemplar is classified correctly, it is generalized by simply merging it with the nearest exemplar of the same class. The nearest exemplar may be either a single instance or a hyperrectangle. In the former case, a new hyperrectangle is created that covers the old and the new instance. In the latter, the hyperrectangle is enlarged to encompass the new instance. Finally, if the prediction is incorrect and it was a hyperrectangle that was responsible for the incorrect prediction, the hyperrectangle's boundaries are altered so that it shrinks away from the new instance.

It is necessary to decide at the outset whether overgeneralization caused by nesting or overlapping hyperrectangles is to be permitted or not. If it is to be avoided, a check is made before generalizing a new example to see whether any regions of feature space conflict with the proposed new hyperrectangle. If they do, the generalization is aborted and the example is stored verbatim. Note that overlapping hyperrectangles are precisely analogous to situations in which the same example is covered by two or more rules in a rule set.

In some schemes, generalized exemplars can be nested in that they may be completely contained within one another, in the same way that in some representations rules may have exceptions. To do this, whenever an example is incorrectly classified, a fallback heuristic is tried using the second nearest neighbor if it produces a correct prediction in a further attempt to perform generalization. This second-chance mechanism promotes nesting of hyperrectangles. If an example falls within a rectangle of the wrong class that already contains an exemplar of the same class, the two are generalized into a new "exception" hyperrectangle nested within the original one. For nested generalized exemplars, the learning process frequently begins with a small number of seed instances to prevent all examples of the same class from being generalized into a single rectangle that covers most of the problem space.

Distance Functions for Generalized Exemplars

With generalized exemplars it is necessary to generalize the distance function to compute the distance from an instance to a generalized exemplar, as well as to another instance. The distance from an instance to a hyperrectangle is defined to be zero if the point lies within the hyperrectangle. The simplest way to generalize the distance function to compute the distance from an exterior point to a hyperrectangle is to choose the closest instance within it and to measure the distance to that. However, this reduces the benefit of generalization because it reintroduces dependence on a particular single example. More precisely, whereas new instances that happen to lie within a hyperrectangle continue to benefit from generalizations, ones that lie outside do not. It might be better to use the distance from the nearest part of the hyperrectangle instead.

Figure 6.16 shows the implicit boundaries that are formed between two rectangular classes if the distance metric is adjusted to measure distance to the nearest point of a rectangle. Even in two dimensions the boundary contains a total of nine regions (they are numbered for easy identification); the situation will be more complex for higher-dimensional hyperrectangles.

Proceeding from the lower left, the first region, in which the boundary is linear, lies outside the extent of both rectangles—to the left of both borders of the larger one and below both borders of the smaller one. The second is within the extent of one rectangle—to the right of the leftmost border of the larger rectangle—but outside that of the other—below both borders of the smaller one. In this region the boundary is parabolic because the locus of a point that is the same distance from a given line

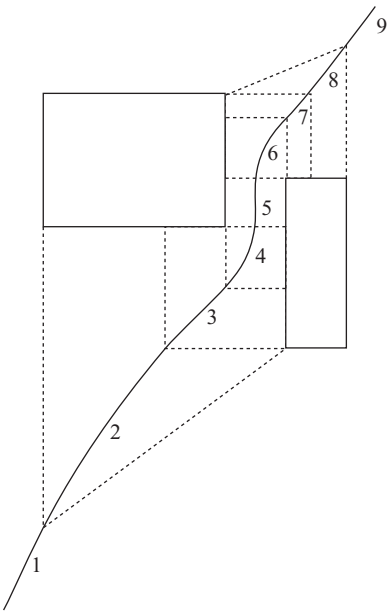


FIGURE 6.16

A boundary between two rectangular classes.

as from a given point is a *parabola*. The third region is where the boundary meets the lower border of the larger rectangle when projected upward and the left border of the smaller one when projected to the right. The boundary is linear in this region because it is equidistant from these two borders. The fourth is where the boundary lies to the right of the larger rectangle but below the bottom of that rectangle. In this case the boundary is parabolic because it is the locus of points equidistant from the lower right corner of the larger rectangle and the left side of the smaller one. The fifth region lies between the two rectangles: Here the boundary is vertical. The pattern is repeated in the upper right part of the diagram: first parabolic, then linear, then parabolic (although this particular parabola is almost indistinguishable from a straight line), and finally linear as the boundary escapes from the scope of both rectangles.

This simple situation certainly defines a complex boundary! Of course, it is not necessary to represent the boundary explicitly; it is generated implicitly by the nearest-neighbor calculation. Nevertheless, the solution is still not a very good one. Whereas taking the distance from the nearest instance within a hyperrectangle is overly dependent on the position of that particular instance, taking the distance to the nearest point of the hyperrectangle is overly dependent on that corner of the rectangle—the nearest example might be far from the corner.

A final problem concerns measuring the distance to hyperrectangles that overlap or are nested. This complicates the situation because an instance may fall within more than one hyperrectangle. A suitable heuristic for use in this case is to choose the class of the most specific hyperrectangle containing the instance—that is, the one covering the smallest area of instance space.

Whether or not overlap or nesting is permitted, the distance function should be modified to take account of both the observed prediction accuracy of exemplars and the relative importance of different features, as described in the sections above on pruning noisy exemplars and attribute weighting.

Generalized Distance Functions

There are many different ways of defining a distance function, and it is hard to find rational grounds for any particular choice. An elegant solution is to consider one

instance being transformed into another through a sequence of predefined elementary operations and to calculate the probability of such a sequence occurring if operations are chosen randomly. Robustness is improved if all possible transformation paths are considered, weighted by their probabilities, and the scheme generalizes naturally to the problem of calculating the distance between an instance and a set of other instances by considering transformations to all instances in the set. Through such a technique it is possible to consider each instance as exerting a "sphere of influence," but a sphere with soft boundaries rather than the hard-edged cutoff implied by the *k*-nearest-neighbor rule, in which any particular example is either "in" or "out" of the decision.

With such a measure, given a test instance that has a class that is unknown, its distance to the set of all training instances in each class in turn is calculated and the closest class is chosen. It turns out that nominal and numeric attributes can be treated in a uniform manner within this transformation-based approach by defining different transformation sets, and it is even possible to take account of unusual attribute types—such as degrees of arc or days of the week, which are measured on a circular scale.

Discussion

Nearest-neighbor methods gained popularity in machine learning through the work of Aha (1992), who showed that, when combined with noisy exemplar pruning and attribute weighting, instance-based learning performs well in comparison with other methods. It is worth noting that although we have described it solely in the context of classification rather than numeric prediction problems, it applies to these equally well: Predictions can be obtained by combining the predicted values of the *k*-nearest neighbors and weighting them by distance.

Viewed in instance space, the standard rule- and tree-based representations are only capable of representing class boundaries that are parallel to the axes defined by the attributes. This is not a handicap for nominal attributes, but it is for numeric ones. Non-axis-parallel class boundaries can only be approximated by covering the region above or below the boundary with several axis-parallel rectangles, the number of rectangles determining the degree of approximation. In contrast, the instance-based method can easily represent arbitrary linear boundaries. Even with just one example of each of two classes, the boundary implied by the nearest-neighbor rule is a straight line of arbitrary orientation, namely the perpendicular bisector of the line joining the examples.

Plain instance-based learning does not produce explicit knowledge representations except by selecting representative exemplars. However, when combined with exemplar generalization, a set of rules can be obtained that may be compared with those produced by other machine learning schemes. The rules tend to be more conservative because the distance metric, modified to incorporate generalized exemplars, can be used to process examples that do not fall within the rules. This reduces the pressure to produce rules that cover the whole example space or even all of the

training examples. On the other hand, the incremental nature of most instance-based learning schemes means that rules are formed eagerly, after only part of the training set has been seen; this inevitably reduces their quality.

We have not given precise algorithms for variants of instance-based learning that involve generalization, because it is not clear what the best way to do generalization is. Salzberg (1991) suggested that generalization with nested exemplars can achieve a high degree of classification of accuracy on a variety of different problems, a conclusion disputed by Wettschereck and Dietterich (1995), who argued that these results were fortuitous and did not hold in other domains. Martin (1995) explored the idea that it is not generalization but the overgeneralization that occurs when hyperrectangles nest or overlap that is responsible for poor performance, and demonstrated that if nesting and overlapping are avoided, excellent results are achieved in a large number of domains. The generalized distance function based on transformations is described by Cleary and Trigg (1995).

Exemplar generalization is a rare example of a learning strategy in which the search proceeds from specific to general rather than from general to specific as in the case of tree or rule induction. There is no particular reason why specific-to-general searching should necessarily be handicapped by forcing the examples to be considered in a strictly incremental fashion, and batch-oriented methods exist that generate rules using a basic instance-based approach. Moreover, it seems that the idea of producing conservative generalizations and coping with instances that are not covered by choosing the "closest" generalization may be generally useful for tree and rule inducers.

6.6 NUMERIC PREDICTION WITH LOCAL LINEAR MODELS

Trees that are used for numeric prediction are just like ordinary decision trees, except that at each leaf they store either a class value that represents the average value of instances that reach the leaf, in which case the tree is called a *regression tree*, or a linear regression model that predicts the class value of instances that reach the leaf, in which case it is called a *model tree*. In what follows we will talk about model trees because regression trees are really a special case.

Regression and model trees are constructed by first using a decision tree induction algorithm to build an initial tree. However, whereas most decision tree algorithms choose the splitting attribute to maximize the information gain, it is appropriate for numeric prediction to instead minimize the intrasubset variation in the class values down each branch. Once the basic tree has been formed, consideration is given to pruning the tree back from each leaf, just as with ordinary decision trees. The only difference between regression tree and model tree induction is that, for the latter, each node is replaced by a regression plane instead of a constant value. The attributes that serve to define that plane are generally those that participate in decisions in the subtree that will be pruned—that is, in nodes beneath the current one and perhaps those that occur on the path to the root node.

Following an extensive description of model trees, we briefly explain how to generate rules from model trees, and then describe another approach to numeric prediction based on generating local linear models: locally weighted linear regression. Whereas model trees derive from the basic divide-and-conquer decision tree methodology, locally weighted regression is inspired by the instance-based methods for classification that we described in the previous section. Like instance-based learning, it performs all "learning" at prediction time. Although locally weighted regression resembles model trees in that it uses linear regression to fit models locally to particular areas of instance space, it does so in quite a different way.

Model Trees

When a model tree is used to predict the value for a test instance, the tree is followed down to a leaf in the normal way, using the instance's attribute values to make routing decisions at each node. The leaf will contain a linear model based on some of the attribute values, and this is evaluated for the test instance to yield a raw predicted value.

Instead of using this raw value directly, however, it turns out to be beneficial to use a smoothing process to reduce the sharp discontinuities that will inevitably occur between adjacent linear models at the leaves of the pruned tree. This is a particular problem for models constructed from a small number of training instances. Smoothing can be accomplished by producing linear models for each internal node, as well as for the leaves, at the time the tree is built. Then, once the leaf model has been used to obtain the raw predicted value for a test instance, that value is filtered along the path back to the root, smoothing it at each node by combining it with the value predicted by the linear model for that node.

An appropriate smoothing calculation is

$$p' = \frac{np + kq}{n + k}$$

where p' is the prediction passed up to the next higher node, p is the prediction passed to this node from below, q is the value predicted by the model at this node, n is the number of training instances that reach the node below, and k is a smoothing constant. Experiments show that smoothing substantially increases the accuracy of predictions.

However, discontinuities remain and the resulting function is not smooth. In fact, exactly the same smoothing process can be accomplished by incorporating the interior models into each leaf model after the tree has been built. Then, during the classification process, only the leaf models are used. The disadvantage is that the leaf models tend to be larger and more difficult to comprehend because many coefficients that were previously zero become nonzero when the interior nodes' models are incorporated.

Building the Tree

The splitting criterion is used to determine which attribute is the best to split that portion T of the training data that reaches a particular node. It is based on treating the standard deviation of the class values in T as a measure of the error at that node, and calculating the expected reduction in error as a result of testing each attribute at that node. The attribute that maximizes the expected error reduction is chosen for splitting at the node.

The expected error reduction, which we call SDR for *standard deviation reduction*, is calculated by

$$\text{SDR} = sd(T) - \sum_i \frac{|T_i|}{|T|} \times sd(T_i)$$

where T_1, T_2, … are the sets that result from splitting the node according to the chosen attribute, and $sd(T)$ is the standard deviation of the class values.

The splitting process terminates when the class values of the instances that reach a node vary just slightly—that is, when their standard deviation is only a small fraction (say less than 5%) of the standard deviation of the original instance set. Splitting also terminates when just a few instances remain (say four or fewer). Experiments show that the results obtained are not very sensitive to the exact choice of these parameters.

Pruning the Tree

As noted earlier, a linear model is needed for each interior node of the tree, not just at the leaves, for use in the smoothing process. Before pruning, a model is calculated for each node of the unpruned tree. The model takes the form

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

where a_1, a_2, …, a_k are attribute values. The weights w_1, w_2, …, w_k are calculated using standard regression. However, only a subset of the attributes are generally used here—for example, those that are tested in the subtree below this node and perhaps those occurring along the path to the root node. Note that we have tacitly assumed that attributes are numeric; we describe the handling of nominal attributes in the next section.

The pruning procedure makes use of an estimate, at each node, of the expected error for test data. First, the absolute difference between the predicted value and the actual class value is averaged over each of the training instances that reach that node. Because the tree has been built expressly for this dataset, this average will underestimate the expected error for unseen cases. To compensate, it is multiplied by the factor $(n + \nu)/(n - \nu)$, where n is the number of training instances that reach the node and ν is the number of parameters in the linear model that gives the class value at that node.

The expected error for test data at a node is calculated as described previously, using the linear model for prediction. Because of the compensation factor $(n + v)/(n - v)$, it may be that the linear model can be further simplified by dropping terms to minimize the estimated error. Dropping a term decreases the multiplication factor, which may be enough to offset the inevitable increase in average error over the training instances. Terms are dropped one by one, greedily, as long as the error estimate decreases.

Finally, once a linear model is in place for each interior node, the tree is pruned back from the leaves as long as the expected estimated error decreases. The expected error for the linear model at that node is compared with the expected error from the subtree below. To calculate the latter, the error from each branch is combined into a single, overall value for the node by weighting the branch by the proportion of the training instances that go down it and combining the error estimates linearly using those weights. Alternatively, one can calculate the training error of the subtree and multiply it by the above modification factor based on an ad hoc estimate of the number of parameters in the tree—perhaps adding one for each split point.

Nominal Attributes

Before constructing a model tree, all nominal attributes are transformed into binary variables that are then treated as numeric. For each nominal attribute, the average class value corresponding to each possible value in the set is calculated from the training instances, and the values are sorted according to these averages. Then, if the nominal attribute has k possible values, it is replaced by $k - 1$ synthetic binary attributes, the ith being 0 if the value is one of the first i in the ordering and 1 otherwise. Thus, all splits are binary: They involve either a numeric attribute or a synthetic binary attribute that is treated as numeric.

It is possible to prove analytically that the best split at a node for a nominal variable with k values is one of the $k - 1$ positions obtained by ordering the average class values for each value of the attribute. This sorting operation should really be repeated at each node; however, there is an inevitable increase in noise due to small numbers of instances at lower nodes in the tree (and in some cases nodes may not represent all values for some attributes), and not much is lost by performing the sorting just once before starting to build a model tree.

Missing Values

To take account of missing values, a modification is made to the SDR formula. The final formula, including the missing value compensation, is

$$\mathrm{SDR} = \frac{m}{|\,\mathrm{T}\,|} \times \left[sd(T) - \sum_{j \in \{L,R\}} \frac{|\,T_j\,|}{|\,T\,|} \times sd(T_j) \right]$$

where m is the number of instances without missing values for that attribute, and T is the set of instances that reach this node. T_L, T_R are sets that result from splitting on this attribute because all tests on attributes are now binary.

When processing both training and test instances, once an attribute is selected for splitting it is necessary to divide the instances into subsets according to their value for this attribute. An obvious problem arises when the value is missing. An interesting technique called *surrogate splitting* has been developed to handle this situation. It involves finding another attribute to split on in place of the original one and using it instead. The attribute is chosen as the one most highly correlated with the original attribute. However, this technique is both complex to implement and time consuming to execute.

A simpler heuristic is to use the class value as the surrogate attribute, in the belief that, a priori, this is the attribute most likely to be correlated with the one being used for splitting. Of course, this is only possible when processing the training set because for test examples the class is not known. A simple solution for test examples is simply to replace the unknown attribute value by the average value of that attribute for the training examples that reach the node, which has the effect, for a binary attribute, of choosing the most populous subnode. This simple approach seems to work well in practice.

Let's consider in more detail how to use the class value as a surrogate attribute during the training process. We first deal with all instances for which the value of the splitting attribute is known. We determine a threshold for splitting in the usual way, by sorting the instances according to the splitting attribute's value and, for each possible split point, calculating the SDR according to the preceding formula, choosing the split point that yields the greatest reduction in error. Only the instances for which the value of the splitting attribute is known are used to determine the split point.

Next we divide these instances into the two sets L and R according to the test. We determine whether the instances in L or R have the greater average class value, and we calculate the average of these two averages. Then an instance for which this attribute value is unknown is placed into L or R according to whether its class value exceeds this overall average or not. If it does, it goes into whichever of L and R has the greater average class value; otherwise, it goes into the one with the smaller average class value. When the splitting stops, all the missing values will be replaced by the average values of the corresponding attributes of the training instances reaching the leaves.

Pseudocode for Model Tree Induction

Figure 6.17 gives pseudocode for the model tree algorithm we have described. The two main parts are creating a tree by successively splitting nodes, performed by *split*, and pruning it from the leaves upward, performed by *prune*. The *node* data structure contains a type flag indicating whether it is an internal node or a leaf, pointers to the left and right child, the set of instances that reach that node, the attribute that is used for splitting at that node, and a structure representing the linear model for the node.

The `sd` function called at the beginning of the main program and again at the beginning of *split* calculates the standard deviation of the class values of a set of instances. This is followed by the procedure for obtaining synthetic binary attributes that was described previously. Standard procedures for creating new nodes and printing the final tree are not shown. In *split*, `sizeof` returns the number of elements in a

```
MakeModelTree (instances)
{
  SD = sd(instances)
  for each k-valued nominal attribute
    convert into k-1 synthetic binary attributes
  root = newNode
  root.instances = instances
  split(root)
  prune(root)
  printTree(root)
}
split(node)
{
  if sizeof(node.instances) < 4 or sd(node.instances) < 0.05*SD
    node.type = LEAF
  else
    node.type = INTERIOR
    for each attribute
      for all possible split positions of the attribute
        calculate the attribute's SDR
    node.attribute = attribute with maximum SDR
    split(node.left)
    split(node.right)
}
prune(node)
{
  if node = INTERIOR then
    prune(node.leftChild)
    prune(node.rightChild)
    node.model = linearRegression(node)
    if subtreeError(node) > error(node) then
      node.type = LEAF
}
subtreeError(node)
{
  l = node.left; r = node.right
  if node = INTERIOR then
    return (sizeof(l.instances)*subtreeError(l)
          + sizeof(r.instances)*subtreeError(r))/sizeof(node.
          instances)
  else return error(node)
}
```

FIGURE 6.17

Pseudocode for model tree induction.

set. Missing attribute values are dealt with as described earlier. The SDR is calculated according to the equation at the beginning of the previous section. Although not shown in the code, it is set to infinity if splitting on the attribute would create a leaf with less than two instances. In *prune*, the `linearRegression` routine recursively descends the subtree collecting attributes, performs a linear regression on the instances

at that node as a function of those attributes, and then greedily drops terms if doing so improves the error estimate, as described earlier. Finally, the *error* function returns

$$\frac{n+\nu}{n-\nu} \times \frac{\sum_{instances} |\text{deviation from predicted class value}|}{n}$$

where n is the number of instances at the node and ν is the number of parameters in the node's linear model.

Figure 6.18 gives an example of a model tree formed by this algorithm for a problem with two numeric and two nominal attributes. What is to be predicted is the rise time of a simulated servo system involving a servo amplifier, motor, lead screw, and sliding carriage. The nominal attributes play important roles. Four synthetic binary attributes have been created for each of the five-valued nominal attributes *motor* and *screw* and are shown in Table 6.2 in terms of the two sets of values to which they correspond. The ordering of these values—D, E, C, B, A for *motor* and coincidentally D, E, C, B, A for *screw* also—is determined from the training data: the rise time averaged over all examples for which *motor* = D is less than that averaged over examples for which *motor* = E, which is less than when *motor* = C, and so on. It is apparent from the magnitude of the coefficients in Table 6.2 that *motor* = D versus E, C, B, A and *screw* = D, E, C, B versus A play leading roles in the LM2, LM3, and LM4 models (among others). Both *motor* and *screw* also play a minor role in several of the models.

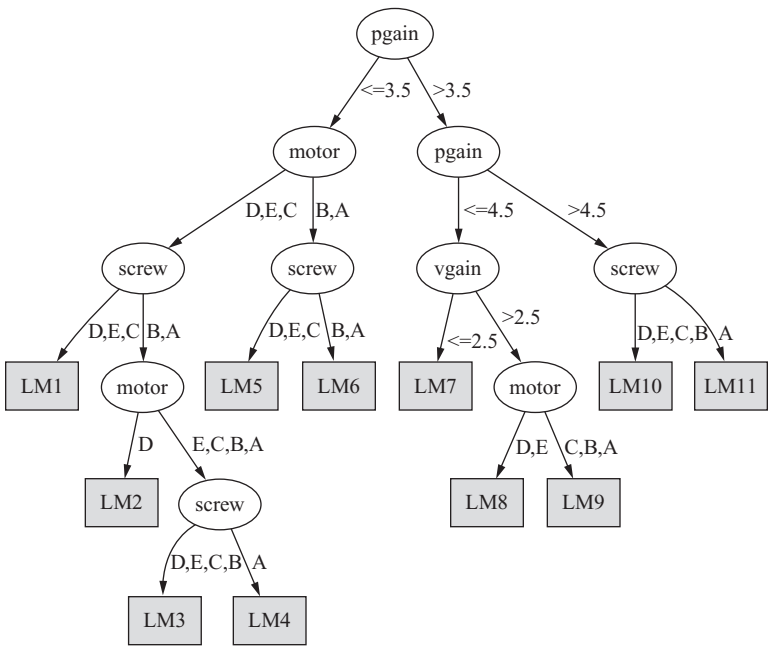


FIGURE 6.18

Model tree for a dataset with nominal attributes.

Table 6.2 Linear Models in the Model Tree

Model	LM1	LM2	LM3	LM4	LM5	LM6	LM7	LM8	LM9	LM10	LM 11
constant term	0.96	1.14	1.43	1.52	2.69	2.91	0.88	0.98	1.11	1.06	0.97
pgain	–0.38	–0.38	–0.38	–0.38	–0.38	–0.38	–0.24	–0.24	–0.24	–0.25	–0.25
vgain	0.71	0.49	0.49	0.49	0.56	0.45	0.13	0.15	0.15	0.10	0.14
motor = D vs. E, C, B, A	0.66	1.14	1.06	1.06	0.50	0.50	0.30	0.40	0.30	0.14	0.14
motor = D, E vs. C, B, A	0.97	0.61	0.65	0.59	0.42	0.42	–0.02	0.06	0.06	0.17	0.22
motor = D, E, C vs. B, A	0.32	0.32	0.32	0.32	0.41	0.41	0.05				
motor = D, E, C, B vs. A					0.08	0.05					
screw = D vs. E, C, B, A											
screw = D, E vs. C, B, A	0.13										
screw = D, E, C vs. B, A	0.49	0.54	0.54	0.54	0.39	0.40	0.30	0.20	0.16	0.08	0.08
screw = D, E, C, B vs. A		1.73	1.79	1.79	0.96	1.13	0.22	0.15	0.15	0.16	0.19

Rules from Model Trees

Model trees are essentially decision trees with linear models at the leaves. Like decision trees, they may suffer from the replicated subtree problem explained in Section 3.4, and sometimes the structure can be expressed much more concisely using a set of rules instead of a tree. Can we generate *rules* for numeric prediction? Recall the rule learner described in Section 6.2 that uses separate-and-conquer in conjunction with partial decision trees to extract decision rules from trees. The same strategy can be applied to model trees to generate decision lists for numeric prediction.

First build a partial model tree from all the data. Pick one of the leaves and make it into a rule. Remove the data covered by that leaf, then repeat the process with the remaining data. The question is, how do we build the partial model tree—that is, a tree with unexpanded nodes? This boils down to the question of how to pick which node to expand next. The algorithm of Figure 6.5 (Section 6.2) picks the node of which the entropy for the class attribute is smallest. For model trees, the predictions of which are numeric, simply use the standard deviation instead. This is based on the same rationale: The lower the standard deviation, the shallower the subtree and the shorter the rule. The rest of the algorithm stays the same, with the model tree learner's split selection method and pruning strategy replacing the decision tree learner's. Because the model tree's leaves are linear models, the corresponding rules will have linear models on the right side.

There is one caveat when using model trees in this fashion to generate rule sets. It turns out that using smoothed model trees does not reduce the error in the final rule set's predictions. This may be because smoothing works best for contiguous data, but the separate-and-conquer scheme removes data covered by previous rules, leaving holes in the distribution. Smoothing, if it is done at all, must be performed after the rule set has been generated.

Locally Weighted Linear Regression

An alternative approach to numeric prediction is the method of locally weighted linear regression. With model trees, the tree structure divides the instance space into regions, and a linear model is found for each of them. In effect, the training data determines how the instance space is partitioned. Locally weighted regression, on the other hand, generates local models at prediction time by giving higher weight to instances in the neighborhood of the particular test instance. More specifically, it weights the training instances according to their distance to the test instance and performs a linear regression on the weighted data. Training instances close to the test instance receive a high weight; those far away, a low one. In other words, a linear model is tailor-made for the particular test instance at hand and used to predict the instance's class value.

To use locally weighted regression, you need to decide on a distance-based weighting scheme for the training instances. A common choice is to weight the

instances according to the inverse of their Euclidean distance from the test instance. Another possibility is to use the Euclidean distance in conjunction with a Gaussian kernel function. However, there is no clear evidence that the choice of weighting function is critical. More important is the selection of a "smoothing parameter" that is used to scale the distance function—the distance is multiplied by the inverse of this parameter. If it is set to a small value, only instances very close to the test instance will receive significant weight; if it is large, more distant instances will also have a significant impact on the model.

One way of choosing the smoothing parameter is to set it to the distance of the kth-nearest training instance so that its value becomes smaller as the volume of training data increases. If the weighting function is linear, say 1 – *distance*, the weight is 0 for all instances further than the kth-nearest one. Then the weighting function has bounded support and only the $(k – 1)$th-nearest neighbors need to be considered for building the linear model. The best choice of k depends on the amount of noise in the data. The more noise there is, the more neighbors should be included in the linear model. Generally, an appropriate smoothing parameter is found using cross-validation.

Like model trees, locally weighted linear regression is able to approximate nonlinear functions. One of its main advantages is that it is ideally suited for incremental learning: All training is done at prediction time, so new instances can be added to the training data at any time. However, like other instance-based methods, it is slow at deriving a prediction for a test instance. First, the training instances must be scanned to compute their weights; then a weighted linear regression is performed on these instances. Also, like other instance-based methods, locally weighted regression provides little information about the global structure of the training dataset. Note that if the smoothing parameter is based on the kth-nearest neighbor and the weighting function gives zero weight to more distant instances, the kD-trees (page 132) and ball trees described in Section 4.7 can be used to accelerate the process of finding the relevant neighbors.

Locally weighted learning is not restricted to linear regression: It can be applied with any learning technique that can handle weighted instances. In particular, you can use it for classification. Most algorithms can be easily adapted to deal with weights. The trick is to realize that (integer) weights can be simulated by creating several copies of the same instance. Whenever the learning algorithm uses an instance when computing a model, just pretend that it is accompanied by the appropriate number of identical shadow instances. This also works if the weight is not an integer. For example, in the Naïve Bayes algorithm described in Section 4.2, multiply the counts derived from an instance by the instance's weight, and—Voilà!—you have a version of Naïve Bayes that can be used for locally weighted learning.

It turns out that locally weighted Naïve Bayes works quite well in practice, outperforming both Naïve Bayes itself and the k-nearest-neighbor technique. It also compares favorably with more sophisticated ways of enhancing Naïve Bayes by relaxing its intrinsic independence assumption. Locally weighted learning only assumes independence within a neighborhood, not globally in the whole instance space as standard Naïve Bayes does.

In principle, locally weighted learning can also be applied to decision trees and other models that are more complex than linear regression and Naïve Bayes. However, it is less beneficial here because it is primarily a way of allowing simple models to become more flexible by allowing them to approximate arbitrary targets. If the underlying learning algorithm can already do that, there is little point in applying locally weighted learning. Nevertheless, it may improve other simple models—for example, linear support vector machines and logistic regression.

Discussion

Regression trees were introduced in the CART system of Breiman et al. (1984). CART, for *classification and regression trees*, incorporated a decision tree inducer for discrete classes like that of C4.5, as well as a scheme for inducing regression trees. Many of the techniques described in this section, such as the method of handling nominal attributes and the surrogate device for dealing with missing values, were included in CART. However, model trees did not appear until much more recently, being first described by Quinlan (1992). Using model trees for generating rule sets (although not partial trees) has been explored by Hall et al. (1999).

A comprehensive description (and implementation) of model tree induction is given by Wang and Witten (1997). Neural nets are also commonly used for predicting numeric quantities, although they suffer from the disadvantage that the structures they produce are opaque and cannot be used to help understand the nature of the solution. There are techniques for producing understandable insights from the structure of neural networks, but the arbitrary nature of the internal representation means that there may be dramatic variations between networks of identical architecture trained on the same data. By dividing the function being induced into linear patches, model trees provide a representation that is reproducible and at least somewhat comprehensible.

There are many variations of locally weighted learning. For example, statisticians have considered using locally quadratic models instead of linear ones and have applied locally weighted logistic regression to classification problems. Also, many different potential weighting and distance functions can be found in the literature. Atkeson et al. (1997) have written an excellent survey on locally weighted learning, primarily in the context of regression problems. Frank et al. (2003) evaluated the use of locally weighted learning in conjunction with Naïve Bayes.

6.7 BAYESIAN NETWORKS

The Naïve Bayes classifier of Section 4.2 and the logistic regression models of Section 4.6 both produce probability estimates rather than hard classifications. For each class value, they estimate the probability that a given instance belongs to that class. Most other types of classifiers can be coerced into yielding this kind of information if necessary. For example, probabilities can be obtained from a decision tree

by computing the relative frequency of each class in a leaf and from a decision list by examining the instances that a particular rule covers.

Probability estimates are often more useful than plain predictions. They allow predictions to be ranked and their expected cost to be minimized (see Section 5.7). In fact, there is a strong argument for treating classification learning as the task of learning class probability estimates from data. What is being estimated is the conditional probability distribution of the values of the class attribute given the values of the other attributes. Ideally, the classification model represents this conditional distribution in a concise and easily comprehensible form.

Viewed in this way, Naïve Bayes classifiers, logistic regression models, decision trees, and so on, are just alternative ways of representing a conditional probability distribution. Of course, they differ in representational power. Naïve Bayes classifiers and logistic regression models can only represent simple distributions, whereas decision trees can represent—or at least approximate—arbitrary distributions. However, decision trees have their drawbacks: They fragment the training set into smaller and smaller pieces, which inevitably yields less reliable probability estimates, and they suffer from the replicated subtree problem described in Section 3.4. Rule sets go some way toward addressing these shortcomings, but the design of a good rule learner is guided by heuristics with scant theoretical justification.

Does this mean that we have to accept our fate and live with these shortcomings? No! There is a statistically based alternative: a theoretically well-founded way of representing probability distributions concisely and comprehensibly in a graphical manner; the structures are called *Bayesian networks*. They are drawn as a network of nodes, one for each attribute, connected by directed edges in such a way that there are no cycles—a *directed acyclic graph*.

In our explanation of how to interpret Bayesian networks and how to learn them from data, we will make some simplifying assumptions. We assume that all attributes are nominal and that there are no missing values. Some advanced learning algorithms can create new attributes in addition to the ones present in the data—so-called hidden attributes with values that cannot be observed. These can support better models if they represent salient features of the underlying problem, and Bayesian networks provide a good way of using them at prediction time. However, they make both learning and prediction far more complex and time consuming, so we will not consider them here.

Making Predictions

Figure 6.19 shows a simple Bayesian network for the weather data. It has a node for each of the four attributes *outlook*, *temperature*, *humidity*, and *windy* and one for the class attribute *play*. An edge leads from the *play* node to each of the other nodes. However, in Bayesian networks the structure of the graph is only half the story. Figure 6.19 shows a table inside each node. The information in the tables defines a probability distribution that is used to predict the class probabilities for any given instance.

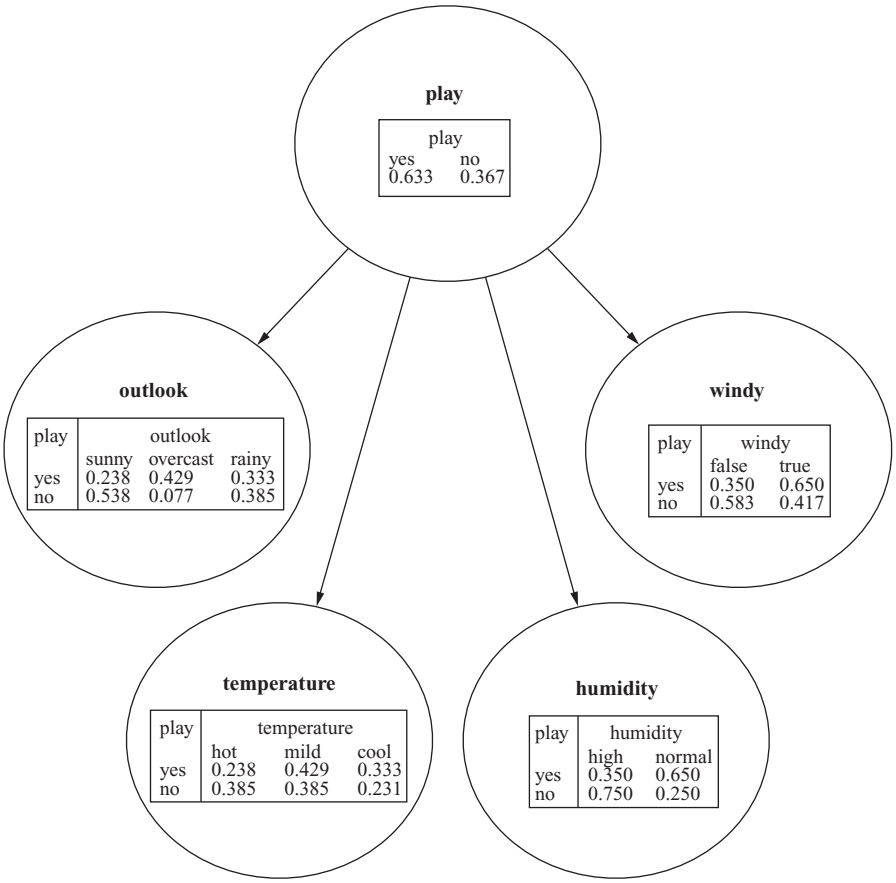


FIGURE 6.19

A simple Bayesian network for the weather data.

Before looking at how to compute this probability distribution, consider the information in the tables. The lower four tables (for *outlook*, *temperature*, *humidity*, and *windy*) have two parts separated by a vertical line. On the left are the values of *play*, and on the right are the corresponding probabilities for each value of the attribute represented by the node. In general, the left side contains a column for every edge pointing to the node, in this case just the *play* attribute. That is why the table associated with *play* itself does not have a left side: It has no parents. Each row of probabilities corresponds to one combination of values of the parent attributes, and the entries in the row show the probability of each value of the node's attribute given this combination. In effect, each row defines a probability distribution over the values of the node's attribute. The entries in a row always sum to 1.

Figure 6.20 shows a more complex network for the same problem, where three nodes (*windy*, *temperature*, and *humidity*) have two parents. Again, there is one column on the left for each parent and as many columns on the right as the attribute has values. Consider the first row of the table associated with the *temperature* node. The left side gives a value for each parent attribute, *play* and *outlook*; the right gives a probability for each value of *temperature*. For example, the first number (0.143) is the probability of *temperature* taking on the value *hot*, given that *play* and *outlook* have values *yes* and *sunny*, respectively.

How are the tables used to predict the probability of each class value for a given instance? This turns out to be very easy because we are assuming that there are no missing values. The instance specifies a value for each attribute. For each node in the network, look up the probability of the node's attribute value based on the row determined by its parents' attribute values. Then just multiply all these probabilities together.

For example, consider an instance with values *outlook* = *rainy*, *temperature* = *cool*, *humidity* = *high*, and *windy* = *true*. To calculate the probability for *play* = *no*, observe that the network in Figure 6.20 gives probability 0.367 from node *play,* 0.385 from *outlook*, 0.429 from *temperature*, 0.250 from *humidity*, and 0.167 from *windy*. The product is 0.0025. The same calculation for *play* = *yes* yields 0.0077. However, these are clearly not the final answers: The final probabilities must sum to 1, whereas 0.0025 and 0.0077 don't. They are actually the joint probabilities Pr[*play* = *no*, *E*] and Pr[*play* = *yes*, *E*], where *E* denotes all the evidence given by the instance's attribute values. Joint probabilities measure the likelihood of observing an instance that exhibits the attribute values in *E* as well as the respective class value. They only sum to 1 if they exhaust the space of all possible attribute–value combinations, including the class attribute. This is certainly not the case in our example.

The solution is quite simple (we already encountered it in Section 4.2). To obtain the conditional probabilities Pr[*play* = *no* | *E*] and Pr[*play* = *yes* | *E*], normalize the joint probabilities by dividing them by their sum. This gives probability 0.245 for *play* = *no* and 0.755 for *play* = *yes*.

Just one mystery remains: Why multiply all those probabilities together? It turns out that the validity of the multiplication step hinges on a single assumption—namely that, given values for each of a node's parents, knowing the values for any other set of nondescendants does not change the probability associated with each of the node's possible values. In other words, other sets of nondescendants do not provide any information about the likelihood of the node's values over and above the information provided by the parents. This can be written as

$$\Pr[\text{node} \mid \text{parents plus any other nondescendants}] = \Pr[\text{node} \mid \text{parents}]$$

which must hold for all values of the nodes and attributes involved. In statistics this property is called *conditional independence*. Multiplication is valid provided that each node is conditionally independent of its grandparents, great-grandparents, and

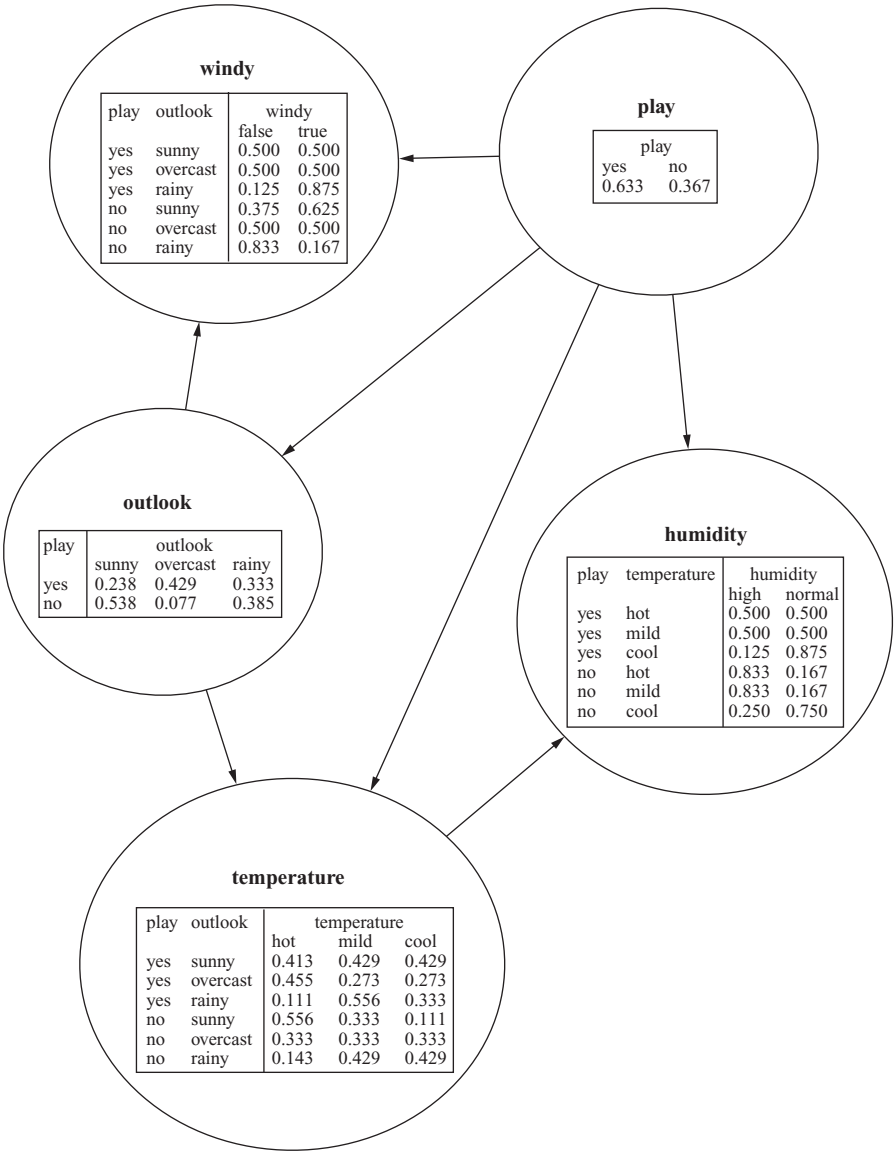


FIGURE 6.20

Another Bayesian network for the weather data.

indeed any other set of nondescendants, given its parents. The multiplication step follows as a direct result of the chain rule in probability theory, which states that the joint probability of m attributes a_i can be decomposed into this product:

$$\Pr[a_1, a_2, \ldots, a_n] = \prod_{i=1}^{m} \Pr[a_i | a_{i-1}, \ldots, a_1]$$

The decomposition holds for any order of the attributes. Because our Bayesian network is an acyclic graph, its nodes can be ordered to give all ancestors of a node a_i indices smaller than i. Then, because of the conditional independence assumption,

$$\Pr[a_1, a_2, \ldots, a_m] = \prod_{i=1}^{m} \Pr[a_i | a_{i-1}, \ldots, a_1] = \prod_{i=1}^{m} \Pr[a_i | a_i\text{'s parents}]$$

which is exactly the multiplication rule that we applied earlier.

The two Bayesian networks in Figures 6.19 and 6.20 are fundamentally different. The first (Figure 6.19) makes stronger independence assumptions because for each of its nodes the set of parents is a subset of the corresponding set of parents in the second (Figure 6.20). In fact, Figure 6.19 is almost identical to the simple Naïve Bayes classifier of Section 4.2. (The probabilities are slightly different but only because each count has been initialized to 0.5 to avoid the zero-frequency problem.) The network in Figure 6.20 has more rows in the conditional probability tables and hence more parameters; it may be a more accurate representation of the underlying domain.

It is tempting to assume that the directed edges in a Bayesian network represent causal effects. But be careful! In our case, a particular value of *play* may enhance the prospects of a particular value of *outlook*, but it certainly doesn't cause it—it is more likely to be the other way around. Different Bayesian networks can be constructed for the same problem, representing exactly the same probability distribution. This is done by altering the way in which the joint probability distribution is factorized to exploit conditional independencies. The network that has directed edges model causal effects is often the simplest one with the fewest parameters. Thus, human experts who construct Bayesian networks for a particular domain often benefit by representing causal effects by directed edges. However, when machine learning techniques are applied to induce models from data whose causal structure is unknown, all they can do is construct a network based on the correlations that are observed in the data. Inferring causality from correlation is always a dangerous business.

Learning Bayesian Networks

The main way to construct a learning algorithm for Bayesian networks is to define two components: a function for evaluating a given network based on the data and a

method for searching through the space of possible networks. The quality of a given network is measured by the probability of the data given the network. We calculate the probability that the network accords to each instance and multiply these probabilities together over all instances. In practice, this quickly yields numbers too small to be represented properly (called *arithmetic underflow*), so we use the sum of the logarithms of the probabilities rather than their product. The resulting quantity is the log-likelihood of the network given the data.

Assume that the structure of the network—the set of edges—is given. It's easy to estimate the numbers in the conditional probability tables: Just compute the relative frequencies of the associated combinations of attribute values in the training data. To avoid the zero-frequency problem each count is initialized with a constant as described in Section 4.2. For example, to find the probability that *humidity = normal* given that *play = yes* and *temperature = cool* (the last number of the third row of the *humidity* node's table in Figure 6.20), observe from Table 1.2 (page 10) that there are three instances with this combination of attribute values in the weather data and no instances with *humidity = high* and the same values for *play* and *temperature*. Initializing the counts for the two values of *humidity* to 0.5 yields the probability $(3 + 0.5)/(3 + 0 + 1) = 0.875$ for *humidity = normal*.

The nodes in the network are predetermined, one for each attribute (including the class). Learning the network structure amounts to searching through the space of possible sets of edges, estimating the conditional probability tables for each set, and computing the log-likelihood of the resulting network based on the data as a measure of the network's quality. Bayesian network learning algorithms differ mainly in the way in which they search through the space of network structures. Some algorithms are introduced below.

There is one caveat. If the log-likelihood is maximized based on the training data, it will always be better to add more edges: The resulting network will simply overfit. Various methods can be employed to combat this problem. One possibility is to use cross-validation to estimate the goodness of fit. A second is to add a penalty for the complexity of the network based on the number of parameters—that is, the total number of independent estimates in all the probability tables. For each table, the number of independent probabilities is the total number of entries minus the number of entries in the last column, which can be determined from the other columns because all rows must sum to 1. Let K be the number of parameters, LL the log-likelihood, and N the number of instances in the data. Two popular measures for evaluating the quality of a network are the *Akaike Information Criterion* (AIC):

$$\text{AIC score} = -LL + K$$

and the following *MDL metric* based on the MDL principle:

$$\text{MDL score} = -LL + \frac{K}{2}\log N$$

In both cases the log-likelihood is negated, so the aim is to minimize these scores.

A third possibility is to assign a prior distribution over network structures and find the most likely network by combining its prior probability with the probability accorded to the network by the data. This is the "Bayesian" approach to network scoring. Depending on the prior distribution used, it can take various forms. However, true Bayesians would average over all possible network structures rather than singling out one particular network for prediction. Unfortunately, this generally requires a great deal of computation. A simplified approach is to average over all network structures that are substructures of a given network. It turns out that this can be implemented very efficiently by changing the method for calculating the conditional probability tables so that the resulting probability estimates implicitly contain information from all subnetworks. The details of this approach are rather complex and will not be described here.

The task of searching for a good network structure can be greatly simplified if the right metric is used for scoring. Recall that the probability of a single instance based on a network is the product of all the individual probabilities from the various conditional probability tables. The overall probability of the dataset is the product of these products for all instances. Because terms in a product are interchangable, the product can be rewritten to group together all factors relating to the same table. The same holds for the log-likelihood, using sums instead of products. This means that the likelihood can be optimized separately for each node of the network. This can be done by adding, or removing, edges from other nodes to the node that is being optimized—the only constraint is that cycles must not be introduced. The same trick also works if a local scoring metric such as AIC or MDL is used instead of plain log-likelihood, because the penalty term splits into several components, one for each node, and each node can be optimized independently.

Specific Algorithms

Now we move on to actual algorithms for learning Bayesian networks. One simple and very fast learning algorithm, called *K2*, starts with a given ordering of the attributes (i.e., nodes). Then it processes each node in turn and greedily considers adding edges from previously processed nodes to the current one. In each step it adds the edge that maximizes the network's score. When there is no further improvement, attention turns to the next node. As an additional mechanism for overfitting avoidance, the number of parents for each node can be restricted to a predefined maximum. Because only edges from previously processed nodes are considered and there is a fixed ordering, this procedure cannot introduce cycles. However, the result depends on the initial ordering, so it makes sense to run the algorithm several times with different random orderings.

The Naïve Bayes classifier is a network with an edge leading from the class attribute to each of the other attributes. When building networks for classification, it sometimes helps to use this network as a starting point for the search. This can be done in K2 by forcing the class variable to be the first one in the ordering and initializing the set of edges appropriately.

Another potentially helpful trick is to ensure that every attribute in the data is in the *Markov blanket* of the node that represents the class attribute. A node's Markov blanket includes all its parents, children, and children's parents. It can be shown that a node is conditionally independent of all other nodes given values for the nodes in its Markov blanket. Thus, if a node is absent from the class attribute's Markov blanket, its value is completely irrelevant to the classification. Conversely, if K2 finds a network that does not include a relevant attribute in the class node's Markov blanket, it might help to add an edge that rectifies this shortcoming. A simple way of doing this is to add an edge from the attribute's node to the class node or from the class node to the attribute's node, depending on which option avoids a cycle.

A more sophisticated but slower version of K2 is not to order the nodes but to greedily consider adding or deleting edges between arbitrary pairs of nodes (all the while ensuring acyclicity, of course). A further step is to consider inverting the direction of existing edges as well. As with any greedy algorithm, the resulting network only represents a *local* maximum of the scoring function: It is always advisable to run such algorithms several times with different random initial configurations. More sophisticated optimization strategies such as simulated annealing, tabu search, or genetic algorithms can also be used.

Another good learning algorithm for Bayesian network classifiers is called *tree-augmented Naïve Bayes* (TAN). As the name implies, it takes the Naïve Bayes classifier and adds edges to it. The class attribute is the single parent of each node of a Naïve Bayes network. TAN considers adding a second parent to each node. If the class node and all corresponding edges are excluded from consideration, and assuming that there is exactly one node to which a second parent is not added, the resulting classifier has a tree structure rooted at the parentless node—this is where the name comes from. For this restricted type of network there is an efficient algorithm for finding the set of edges that maximizes the network's likelihood based on computing the network's maximum weighted spanning tree. This algorithm is linear in the number of instances and quadratic in the number of attributes.

The type of network learned by the TAN algorithm is called a *one-dependence estimator*. An even simpler type of network is the *superparent one-dependence estimator*. Here, exactly one other node, apart from the class node, is elevated to parent status and becomes the parent of every other nonclass node. It turns out that a simple ensemble of these one-dependence estimators yields very accurate classifiers: In each of these estimators, a different attribute becomes the extra parent node. Then, at prediction time, class probability estimates from the different one-dependence estimators are simply averaged. This scheme is known as *AODE*, for *averaged one-dependence estimator*. Normally, only estimators with certain supports in the data are used in the ensemble, but more sophisticated selection schemes are possible. Because no structure learning is involved for each superparent one-dependence estimator, AODE is a very efficient classifier.

All the scoring metrics that we have described so far are likelihood-based in the sense that they are designed to maximize the joint probability $\Pr[a_1, a_2, \ldots, a_n]$ for each instance. However, in classification, what we really want to maximize is the

conditional probability of the class given the values of the other attributes—in other words, the conditional likelihood. Unfortunately, there is no closed-form solution for the maximum conditional-likelihood probability estimates that are needed for the tables in a Bayesian network. On the other hand, computing the conditional likelihood for a given network and dataset is straightforward—after all, this is what logistic regression does. Thus, it has been proposed to use standard maximum-likelihood probability estimates in the network, but to use the conditional likelihood to evaluate a particular network structure.

Another way of using Bayesian networks for classification is to build a separate network for each class value, based on the data pertaining to that class, and combine their predictions using Bayes' rule. The set of networks is called a *Bayesian multinet*. To obtain a prediction for a particular class value, take the corresponding network's probability and multiply it by the class's prior probability. Do this for each class and normalize the result as we did previously. In this case we would not use the conditional likelihood to learn the network for each class value.

All the network learning algorithms we have introduced are score-based. A different strategy, which we will not explain here, is to piece a network together by testing individual conditional independence assertions based on subsets of the attributes. This is known as *structure learning by conditional independence tests*.

Data Structures for Fast Learning

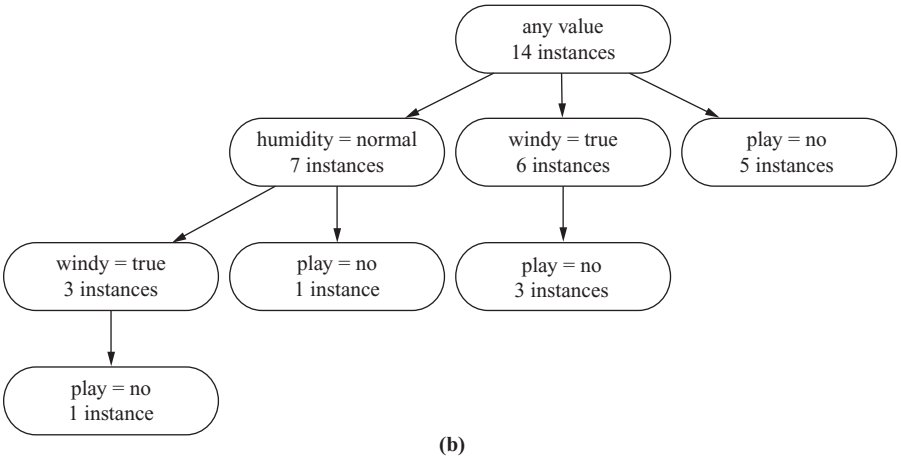
Learning Bayesian networks involves a lot of counting. For each network structure considered in the search, the data must be scanned afresh to obtain the counts needed to fill out the conditional probability tables. Instead, could they be stored in a data structure that eliminated the need for scanning the data over and over again? An obvious way is to precompute the counts and store the nonzero ones in a table—say, the hash table mentioned in Section 4.5. Even so, any nontrivial dataset will have a huge number of nonzero counts.

Again, consider the weather data from Table 1.2. There are five attributes, two with three values and three with two values. This gives $4 \times 4 \times 3 \times 3 \times 3 = 432$ possible counts. Each component of the product corresponds to an attribute, and its contribution to the product is one more than the number of its values because the attribute may be missing from the count. All these counts can be calculated by treating them as item sets, as explained in Section 4.5, and setting the minimum coverage to 1. But even without storing counts that are 0, this simple scheme runs into memory problems very quickly. The FP-growth data structure described in Section 6.3 was designed for efficient representation of data in the case of item set mining. In the following, we describe a structure that has been used for Bayesian networks.

It turns out that the counts can be stored effectively in a structure called an *all-dimensions (AD) tree*, which is analogous to the *k*D-trees used for the nearest-neighbor search described in Section 4.7. For simplicity, we illustrate this using a reduced version of the weather data that only has the attributes *humidity*, *windy*, and *play*. Figure 6.21(a) summarizes the data. The number of possible counts is $3 \times 3 \times$

Humidity	Windy	Play	Count
high	true	yes	1
high	true	no	2
high	false	yes	2
high	false	no	2
normal	true	yes	2
normal	true	no	1
normal	false	yes	4
normal	false	no	0

(a)



(b)

FIGURE 6.21

The weather data: (a) reduced version and (b) corresponding AD-tree.

3 = 27, although only eight of them are shown. For example, the count for *play = no* is 5 (count them!).

Figure 6.21(b) shows an AD-tree for this data. Each node says how many instances exhibit the attribute values that are tested along the path from the root to that node. For example, the leftmost leaf says that there is one instance with values *humidity = normal, windy = true,* and *play = no*, and the rightmost leaf says that there are five instances with *play = no*.

It would be trivial to construct a tree that enumerates all 27 counts explicitly. However, that would gain nothing over a plain table and is obviously not what the tree in Figure 6.21(b) does because it contains only 8 counts. There is, for example, no branch that tests *humidity = high*. How was the tree constructed, and how can all counts be obtained from it?

Assume that each attribute in the data has been assigned an index. In the reduced version of the weather data we give *humidity* index 1, *windy* index 2, and *play* index 3. An AD-tree is generated by expanding each node corresponding to an attribute i with the values of all attributes that have indices $j > i$, with two important

restrictions: The most populous expansion for each attribute is omitted (breaking ties arbitrarily) as are expansions with counts that are zero. The root node is given index 0, so for this node all attributes are expanded, subject to the same restrictions.

For example, Figure 6.21(b) contains no expansion for *windy = false* from the root node because with eight instances it is the most populous expansion: The value *false* occurs more often in the data than the value *true*. Similarly, from the node labeled *humidity = normal* there is no expansion for *windy = false* because *false* is the most common value for *windy* among all instances with *humidity = normal.* In fact, in our example the second restriction—namely that expansions with zero counts are omitted—never kicks in because the first restriction precludes any path that starts with the tests *humidity = normal* and *windy = false*, which is the only way to reach the solitary 0 in Figure 6.21(a).

Each node of the tree represents the occurrence of a particular combination of attribute values. It is straightforward to retrieve the count for a combination that occurs in the tree. However, the tree does not explicitly represent many nonzero counts because the most populous expansion for each attribute is omitted. For example, the combination *humidity = high* and *play = yes* occurs three times in the data but has no node in the tree. Nevertheless, it turns out that any count can be calculated from those that the tree stores explicitly.

Here's a simple example. Figure 6.21(b) contains no node for *humidity = normal*, *windy = true*, and *play = yes*. However, it shows three instances with *humidity = normal* and *windy = true*, and one of them has a value for *play* that is different from *yes*. It follows that there must be two instances for *play = yes*. Now for a trickier case: How many times does *humidity = high*, *windy = true*, and *play = no* occur? At first glance it seems impossible to tell because there is no branch for *humidity = high*. However, we can deduce the number by calculating the count for *windy = true* and *play = no* (3) and subtracting the count for *humidity = normal*, *windy = true*, and *play = no* (1). This gives 2, the correct value.

This idea works for any subset of attributes and any combination of attribute values, but it may have to be applied recursively. For example, to obtain the count for *humidity = high*, *windy = false*, and *play = no*, we need the count for *windy = false* and *play = no* and the count for *humidity = normal*, *windy = false*, and *play = no*. We obtain the former by subtracting the count for *windy = true* and *play = no* (3) from the count for *play = no* (5), giving 2, and we obtain the latter by subtracting the count for *humidity = normal*, *windy = true*, and *play = no* (1) from the count for *humidity = normal* and *play = no* (1), giving 0. Thus, there must be 2 – 0 = 2 instances with *humidity = high*, *windy = false*, and *play = no*, which is correct.

AD-trees only pay off if the data contains many thousands of instances. It is pretty obvious that they do not help on the weather data. The fact that they yield no benefit on small datasets means that, in practice, it makes little sense to expand the tree all the way down to the leaf nodes. Usually, a cutoff parameter *k* is employed, and nodes covering fewer than *k* instances hold a list of pointers to these instances rather than a list of pointers to other nodes. This makes the trees smaller and more efficient to use.

Discussion

The K2 algorithm for learning Bayesian networks was introduced by Cooper and Herskovits (1992). Bayesian scoring metrics are covered by Heckerman et al. (1995). The TAN algorithm was introduced by Friedman et al. (1997), who also describe multinets. Grossman and Domingos (2004) show how to use the conditional likelihood for scoring networks. Guo and Greiner (2004) present an extensive comparison of scoring metrics for Bayesian network classifiers. Bouckaert (1995) describes averaging over subnetworks. Averaged one-dependence estimators are described by Webb et al. (2005). AD-trees were introduced and analyzed by Moore and Lee (1998)—the same Andrew Moore whose work on *k*D-trees and ball trees was mentioned in Section 4.7. In a more recent paper, Komarek and Moore (2000) introduce AD-trees for incremental learning that are also more efficient for datasets with many attributes.

We have only skimmed the surface of the subject of learning Bayesian networks. We left open questions of missing values, numeric attributes, and hidden attributes. We did not describe how to use Bayesian networks for regression tasks. Bayesian networks are a special case of a wider class of statistical models called *graphical models*, which include networks with undirected edges called *Markov networks*. Graphical models are attracting great attention in the machine learning community today.

6.8 CLUSTERING

In Section 4.8 we examined the *k*-means clustering algorithm in which *k* initial points are chosen to represent initial cluster centers, all data points are assigned to the nearest one, the mean value of the points in each cluster is computed to form its new cluster center, and iteration continues until there are no changes in the clusters. This procedure only works when the number of clusters is known in advance, and this section begins by describing what you can do if it is not.

Next we take a look at techniques for creating a hierarchical clustering structure by "agglomeration"—that is, starting with individual instances and successively joining them up into clusters. Then we look at a method that works incrementally; that is, process each new instance as it appears. This method was developed in the late 1980s and embodied in a pair of systems called Cobweb (for nominal attributes) and Classit (for numeric attributes). Both come up with a hierarchical grouping of instances and use a measure of cluster "quality" called *category utility*. Finally, we examine a statistical clustering method based on a mixture model of different probability distributions, one for each cluster. It does not partition instances into disjoint clusters as *k*-means does but instead assigns instances to classes probabilistically, not deterministically. We explain the basic technique and sketch the working of a comprehensive clustering scheme called AutoClass.

Choosing the Number of Clusters

Suppose you are using *k*-means but do not know the number of clusters in advance. One solution is to try out different possibilities and see which is best. A simple strategy is to start from a given minimum, perhaps $k = 1$, and work up to a small fixed maximum. Note that on the training data the "best" clustering according to the total squared distance criterion will always be to choose as many clusters as there are data points! To penalize solutions with many clusters you will have to apply something like the MDL criterion of Section 5.9.

Another possibility is to begin by finding a few clusters and determining whether it is worth splitting them. You could choose $k = 2$, perform *k*-means clustering until it terminates, and then consider splitting each cluster. Computation time will be reduced considerably if the initial two-way clustering is considered irrevocable and splitting is investigated for each component independently. One way to split a cluster is to make a new seed one standard deviation away from the cluster's center in the direction of its greatest variation, and to make a second seed the same distance in the opposite direction. (Alternatively, if this is too slow, choose a distance proportional to the cluster's bounding box and a random direction.) Then apply *k*-means to the points in the cluster with these two new seeds.

Having tentatively split a cluster, is it worthwhile retaining the split or is the original cluster equally plausible by itself? It's no good looking at the total squared distance of all points to their cluster center—this is bound to be smaller for two subclusters. A penalty should be incurred for inventing an extra cluster, and this is a job for the MDL criterion. That principle can be applied to see whether the information required to specify the two new cluster centers, along with the information required to specify each point with respect to them, exceeds the information required to specify the original center and all the points with respect to it. If so, the new clustering is unproductive and should be abandoned. If the split is retained, try splitting each new cluster further. Continue the process until no worthwhile splits remain.

Additional implementation efficiency can be achieved by combining this iterative clustering process with the *k*D-tree or ball tree data structure advocated in Section 4.8. Then the data points are reached by working down the tree from the root. When considering splitting a cluster, there is no need to consider the whole tree; just look at those parts of it that are needed to cover the cluster. For example, when deciding whether to split the lower left cluster in Figure 4.16(a) (below the thick line), it is only necessary to consider nodes A and B of the tree in Figure 4.16(b) because node C is irrelevant to that cluster.

Hierarchical Clustering

Forming an initial pair of clusters and then recursively considering whether it is worth splitting each one further produces a hierarchy that can be represented as a binary tree called a *dendrogram*. In fact, we illustrated a dendrogram in Figure

3.11(d) (there some of the branches were three-way). The same information could be represented as a Venn diagram of sets and subsets: The constraint that the structure is hierarchical corresponds to the fact that, although subsets can include one another, they cannot intersect. In some cases there exists a measure of the degree of dissimilarity between the clusters in each set; then the height of each node in the dendrogram can be made proportional to the dissimilarity between its children. This provides an easily interpretable diagram of a hierarchical clustering.

An alternative to the top-down method for forming a hierarchical structure of clusters is to use a bottom-up approach, which is called *agglomerative* clustering. This idea was proposed many years ago and has recently enjoyed a resurgence in popularity. The basic algorithm is simple. All you need is a measure of distance (or a similarity measure) between any two clusters. (If you have a similarity measure instead, it is easy to convert that into a distance.) You begin by regarding each instance as a cluster in its own right; then find the two closest clusters, merge them, and keep on doing this until only one cluster is left. The record of mergings forms a hierarchical clustering structure—a binary dendrogram.

There are numerous possibilities for the distance measure. One is the minimum distance between the clusters—the distance between their two closest members. This yields what is called the *single-linkage* clustering algorithm. Since this measure takes into account only the two closest members of a pair of clusters, the procedure is sensitive to outliers: The addition of just a single new instance can radically alter the entire clustering structure. Also, if we define the diameter of a cluster to be the greatest distance between its members, single-linkage clustering can produce clusters with very large diameters. Another measure is the maximum distance between the clusters, instead of the minimum. Two clusters are considered close only if all instances in their union are relatively similar—sometimes called the *complete-linkage* method. This measure, which is also sensitive to outliers, seeks compact clusters with small diameters. However, some instances may end up much closer to other clusters than they are to the rest of their own cluster.

There are other measures that represent a compromise between the extremes of minimum and maximum distance between cluster members. One is to represent clusters by the centroid of their members, as the *k*-means algorithm does, and use the distance between centroids—the *centroid-linkage* method. This works well when the instances are positioned in multidimensional Euclidean space and the notion of centroid is clear, but not when all we have is a pairwise similarity measure between instances, because centroids are not instances and the similarity between them may be impossible to define.

Another measure, which avoids this problem, is to calculate the average distance between each pair of members of the two clusters—the *average-linkage* method. Although this seems like a lot of work, you would have to calculate all pairwise distances in order to find the maximum or minimum anyway, and averaging them isn't much additional burden. Both these measures have a technical deficiency: Their results depend on the numerical scale on which distances are measured. The minimum and maximum distance measures produce a result that depends only on the *ordering*

between the distances involved. In contrast, the result of both centroid-based and average-distance clustering can be altered by a monotonic transformation of all distances, even though it preserves their relative ordering.

Another method, called *group-average* clustering, uses the average distance between all members of the merged cluster. This differs from the "average" method just described because it includes in the average pairs from the same original cluster. Finally, *Ward's* clustering method calculates the increase in the sum of squares of the distances of the instances from the centroid before and after fusing two clusters. The idea is to minimize the increase in this squared distance at each clustering step.

All these measures will produce the same hierarchical clustering result if the clusters are compact and well separated. However, in other cases they can yield quite different structures.

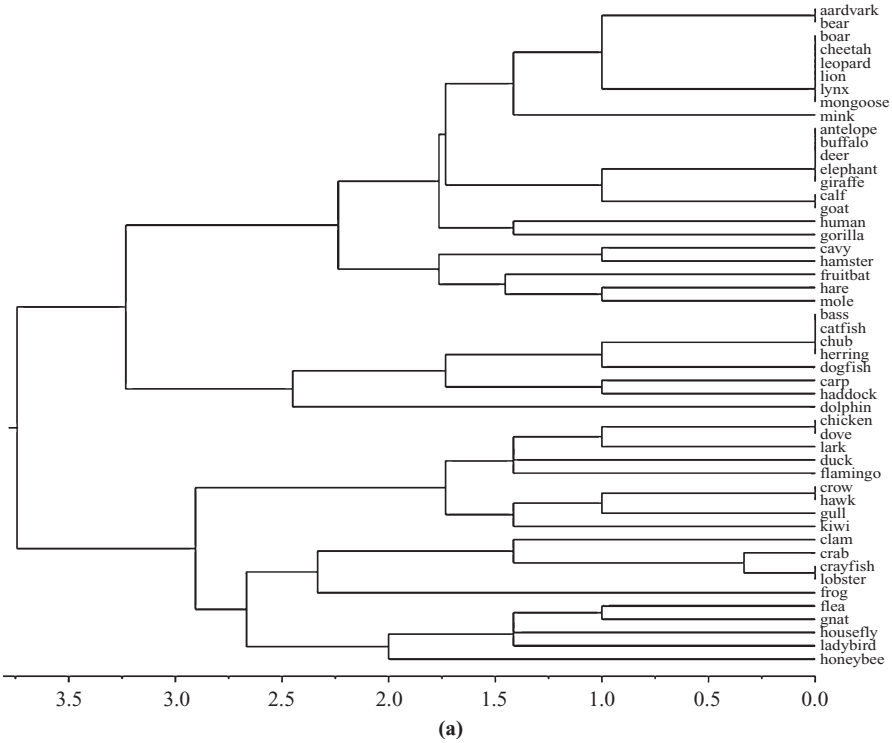
Example of Hierarchical Clustering

Figure 6.22 shows the result of agglomerative hierarchical clustering. (These visualizations have been generated using the FigTree program.[1]) In this case the dataset contained 50 examples of different kinds of creatures, from dolphin to mongoose, from giraffe to lobster. There was 1 numeric attribute (number of legs, ranging from 0 to 6, but scaled to the range [0, 1]) and 15 Boolean attributes such as *has feathers*, *lays eggs*, and *venomous*, which are treated as binary attributes with values 0 and 1 in the distance calculation.

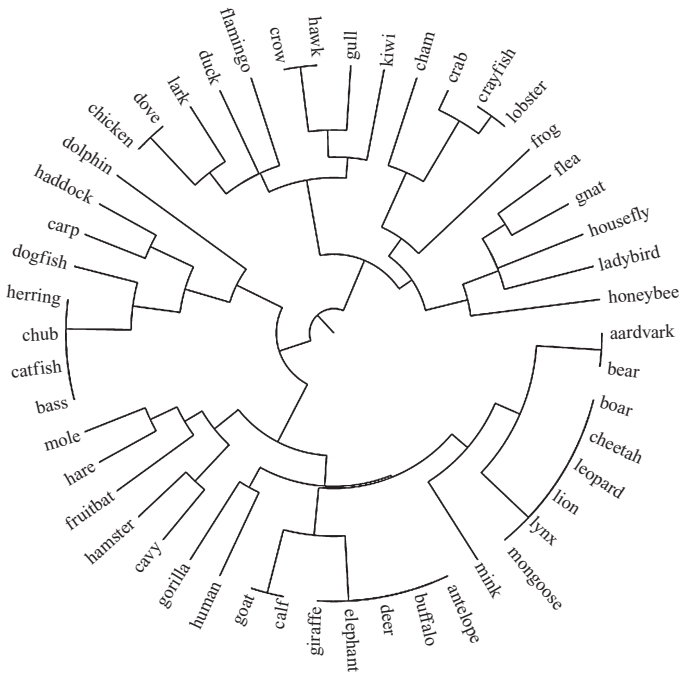
Two kinds of display are shown: a standard dendrogram and a polar plot. Figures 6.22(a) and (b) show the output from an agglomerative clusterer plotted in two different ways, and Figures 6.22(c) and (d) show the result of a different agglomerative clusterer plotted in the same two ways. The difference is that the pair in Figures 6.22(a) and (b) was produced using the complete-linkage measure and the pair in Figures 6.22(c) and (d) was produced using the single-linkage measure. You can see that the complete-linkage method tends to produce compact clusters while the single-linkage method produces clusters with large diameters at fairly low levels of the tree.

In all four visualizations the height of each node in the dendrogram is proportional to the dissimilarity between its children, measured as the Euclidean distance between instances. A numeric scale is provided beneath Figures 6.22(a) and (c). The total dissimilarity from root to leaf is far greater for the complete-linkage method in Figures 6.22(a) and (b) than for the single-linkage method in Figures 6.22(c) and (d) since the former involves the maximum distance and the latter the minimum distance between instances in each cluster. In the first case the total dissimilarity is a little less than 3.75, which is almost the maximum possible distance between instances—the distance between two instances that differ in 14 of the 15 attributes is $\sqrt{14} \approx 3.74$. In the second it is a little greater than 2 (that is, $\sqrt{4}$), which is what a difference in four Boolean attributes would produce.

[1]See *http://tree.bio.ed.ac.uk/software/figtree/* for more information.



(a)

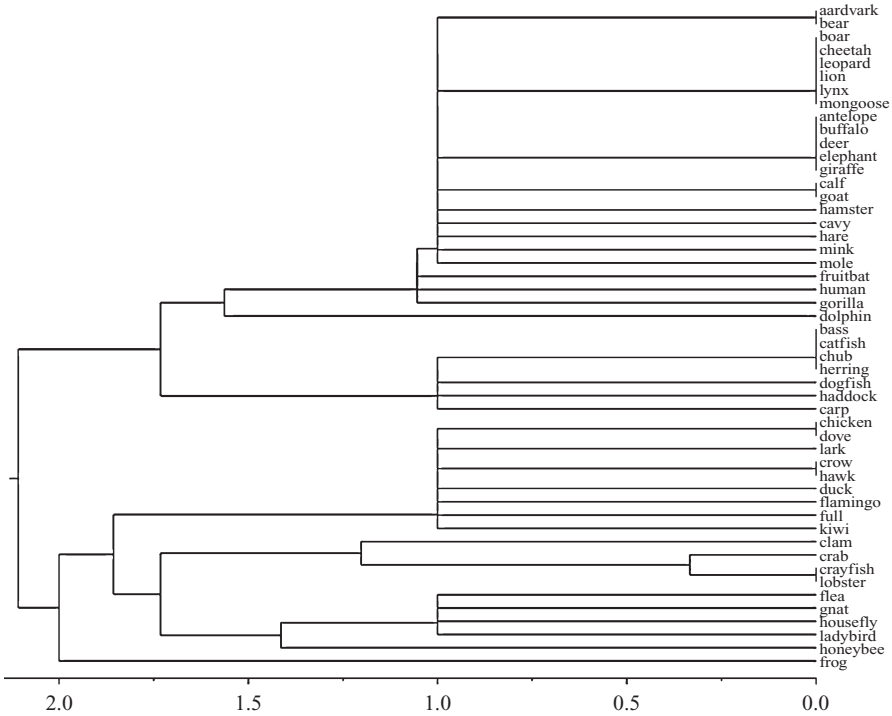


(b)

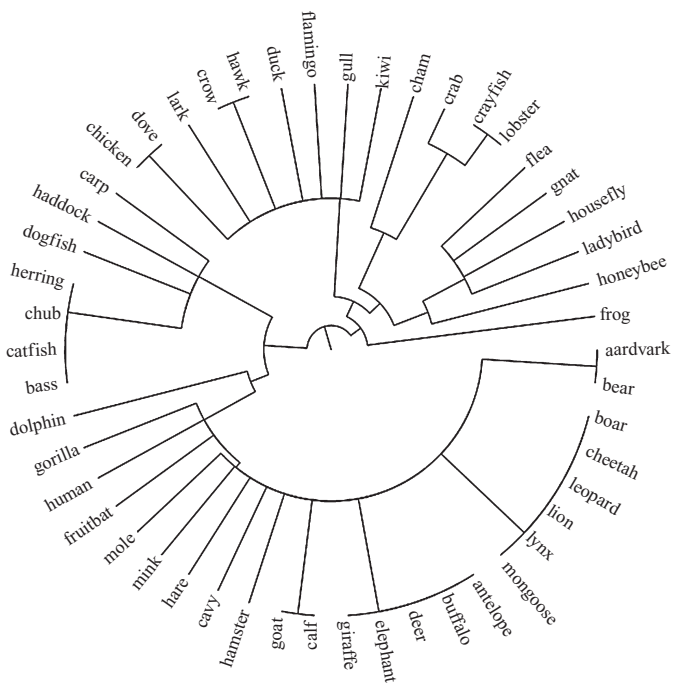
FIGURE 6.22

Hierarchical clustering displays.

Continued



(c)



(d)

FIGURE 6.22, cont'd

For the complete-linkage method (Figure 6.22(a)), many elements join together at a dissimilarity of 1, which corresponds to a difference in a single Boolean attribute. Only one pair has a smaller dissimilarity: *crab* and *crayfish*, which differ only in the number of legs (4/6 and 6/6, respectively, after scaling). Other popular dissimilarities are $\sqrt{2}$, $\sqrt{3}$, $\sqrt{4}$, and so on, corresponding to differences in two, three, and four Boolean attributes. For the single-linkage method (Figure 6.22(c)) that uses the minimum distance between clusters, even more elements join together at a dissimilarity of 1.

Which of the two display methods—the standard dendogram and the polar plot—is more useful is a matter of taste. Although more unfamiliar at first, the polar plot spreads the visualization more evenly over the space available.

Incremental Clustering

Whereas the *k*-means algorithm iterates over the whole dataset until convergence is reached and the hierarchical method examines all the clusters present so far at each stage of merging, the clustering methods we examine next work incrementally, instance by instance. At any stage the clustering forms a tree with instances at the leaves and a root node that represents the entire dataset. In the beginning the tree consists of the root alone. Instances are added one by one, and the tree is updated appropriately at each stage. Updating may be merely a case of finding the right place to put a leaf representing the new instance, or it may involve a radical restructuring of the part of the tree that is affected by the new instance. The key to deciding how and where to update is a quantity called *category utility* that measures the overall quality of a partition of instances into clusters. We defer detailed consideration of how this is defined until the next section and look first at how the clustering algorithm works.

The procedure is best illustrated by an example. We will use the familiar weather data again, but without the *play* attribute. To track progress, the 14 instances are labeled *a*, *b*, *c*, …, *n* (as in Table 4.6), and for interest we include the classes *yes* or *no* in the label—although it should be emphasized that for this artificial dataset there is little reason to suppose that the two classes of instance should fall into separate categories. Figure 6.23 shows the situation at salient points throughout the clustering procedure.

At the beginning, when new instances are absorbed into the structure, they each form their own subcluster under the overall top-level cluster. Each new instance is processed by tentatively placing it in each of the existing leaves and evaluating the category utility of the resulting set of the top-level node's children to see if the leaf is a good "host" for the new instance. For each of the first five instances, there is no such host: It is better, in terms of category utility, to form a new leaf for each instance. With the sixth it finally becomes beneficial to form a cluster, joining the new instance *f* with the old one—the host—*e*. If you look back at Table 4.6 you will see that the fifth and sixth instances are indeed very similar, differing only in the *windy* attribute (and *play*, which is being ignored here). The next example, *g*, is

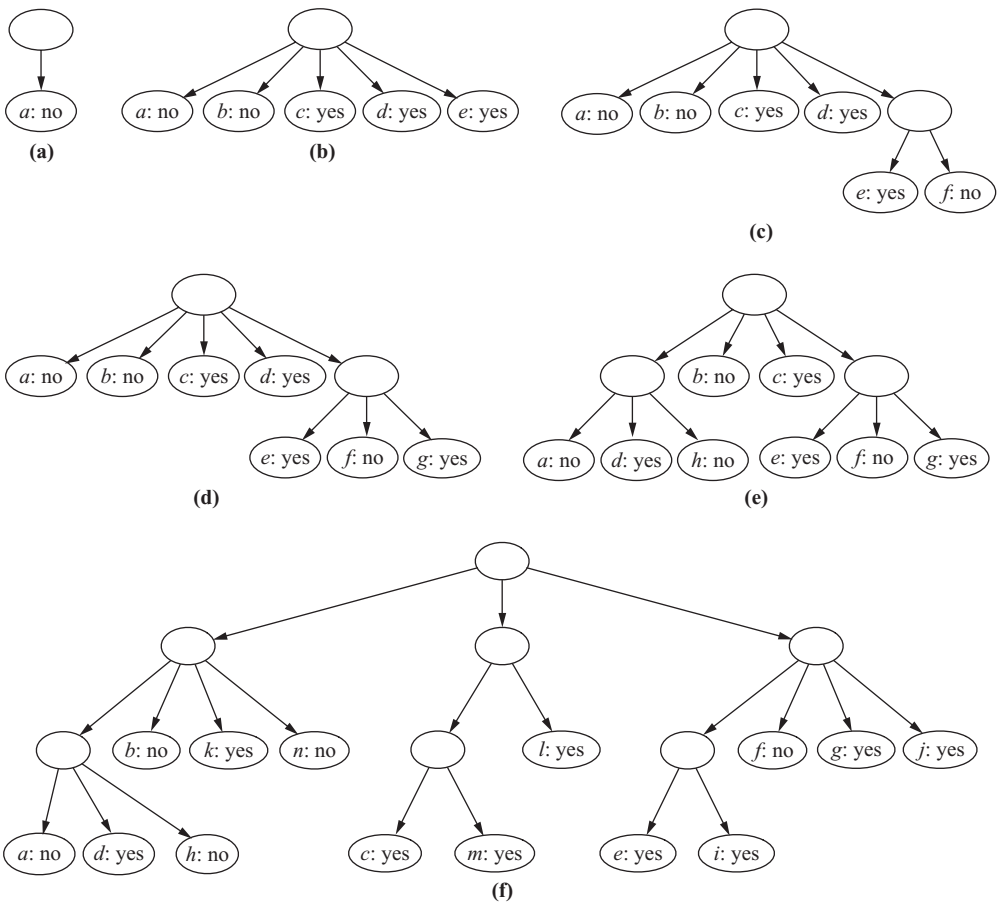


FIGURE 6.23

Clustering the weather data.

placed in the same cluster (it differs from *f* only in *outlook*). This involves another call to the clustering procedure. First, *g* is evaluated to see which of the five children of the root makes the best host; it turns out to be the rightmost, the one that is already a cluster. Then the clustering algorithm is invoked with this as the root, and its two children are evaluated to see which would make the better host. In this case it proves best, according to the category utility measure, to add the new instance as a subcluster in its own right.

If we were to continue in this vein, there would be no possibility of any radical restructuring of the tree, and the final clustering would be excessively dependent on the ordering of examples. To avoid this, there is provision for restructuring, and you can see it come into play when instance *h* is added in the next step shown in Figure 6.23(e). In this case two existing nodes are *merged* into a single cluster: Nodes *a* and

d are merged before the new instance *h* is added. One way of accomplishing this would be to consider all pairs of nodes for merging and evaluate the category utility of each pair. However, that would be computationally expensive, and would involve a lot of repeated work if it were undertaken whenever a new instance was added.

Instead, whenever the nodes at a particular level are scanned for a suitable host, both the best-matching node—the one that produces the greatest category utility for the split at that level—and the runner-up are noted. The best one will form the host for the new instance (unless that new instance is better off in a cluster of its own). However, before setting to work on putting the new instance in with the host, consideration is given to merging the host and the runner-up. In this case, *a* is the preferred host and *d* is the runner-up. When a merge of *a* and *d* is evaluated, it turns out that it would improve the category utility measure. Consequently, these two nodes are merged, yielding a version of the fifth hierarchy before *h* is added. Then consideration is given to the placement of *h* in the new, merged node; it turns out to be best to make it a subcluster in its own right, as shown.

An operation converse to merging is also implemented, called *splitting*. Whenever the best host is identified, and merging has not proved beneficial, consideration is given to splitting the host node. Splitting has exactly the opposite effect of merging, taking a node and replacing it with its children. For example, splitting the rightmost node in Figure 6.23(d) would raise the *e*, *f*, and *g* leaves up a level, making them siblings of *a*, *b*, *c*, and *d*. Merging and splitting provide an incremental way of restructuring the tree to compensate for incorrect choices caused by infelicitous ordering of examples.

The final hierarchy for all 14 examples is shown in Figure 6.23(f). There are three major clusters, each of which subdivides further into its own subclusters. If the *play/don't play* distinction really represented an inherent feature of the data, a single cluster would be expected for each outcome. No such clean structure is observed, although a (very) generous eye might discern a slight tendency at lower levels for *yes* instances to group together, and likewise for *no* instances.

Exactly the same scheme works for numeric attributes. Category utility is defined for these as well, based on an estimate of the mean and standard deviation of the value of that attribute. Details are deferred to the next subsection. However, there is just one problem that we must attend to here: When estimating the standard deviation of an attribute for a particular node, the result will be zero if the node contains only one instance, as it does more often than not. Unfortunately, zero variances produce infinite values in the category utility formula. A simple heuristic solution is to impose a minimum variance on each attribute. It can be argued that because no measurement is completely precise, it is reasonable to impose such a minimum: It represents the measurement error in a single sample. This parameter is called *acuity*.

Figure 6.24(a) shows a hierarchical clustering produced by the incremental algorithm for part of the iris dataset (30 instances, 10 from each class). At the top level there are two clusters (i.e., subclusters of the single node representing the whole dataset). The first contains both *Iris virginicas* and *Iris versicolors*, and the second contains only *Iris setosas*. The *Iris setosas* themselves split into two subclusters, one

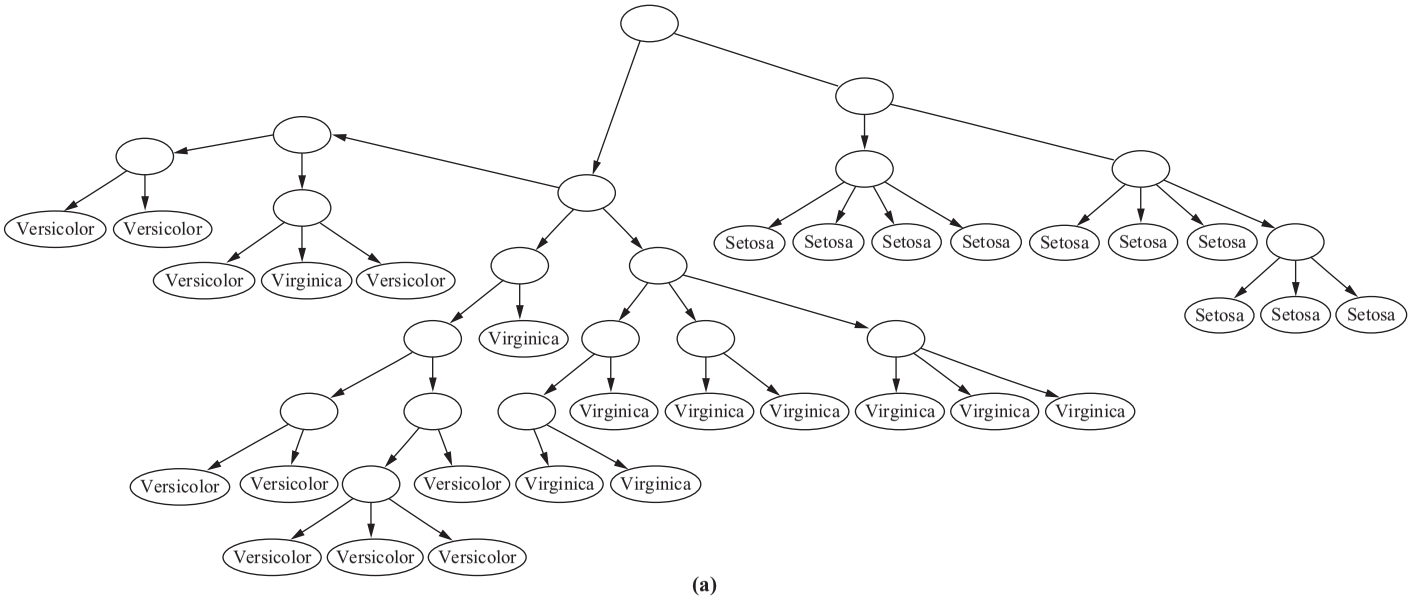
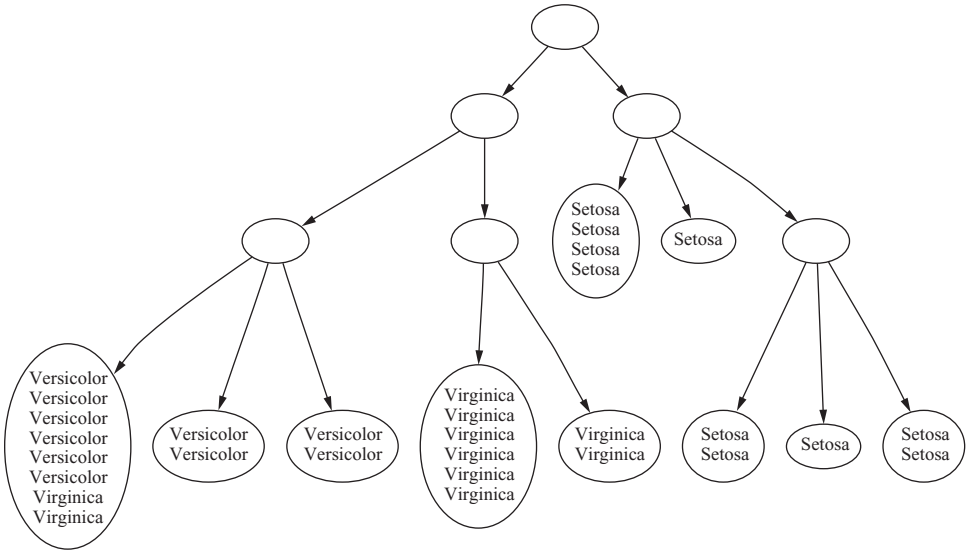


FIGURE 6.24

Hierarchical clusterings of the iris data.



(b)

FIGURE 6.24, cont'd

with four cultivars and the other with six. The other top-level cluster splits into three subclusters, each with a fairly complex structure. Both the first and second contain only *Iris versicolors*, with one exception, a stray *Iris virginica*, in each case; the third contains only *Iris virginicas*. This represents a fairly satisfactory clustering of the iris data: It shows that the three genera are not artificial at all but reflect genuine differences in the data. This is, however, a slightly overoptimistic conclusion because quite a bit of experimentation with the acuity parameter was necessary to obtain such a nice division.

The clusterings produced by this scheme contain one leaf for every instance. This produces an overwhelmingly large hierarchy for datasets of any reasonable size, corresponding, in a sense, to overfitting the particular dataset. Consequently, a second numerical parameter called *cutoff* is used to suppress growth. Some instances are deemed to be sufficiently similar to others not to warrant formation of their own child node, and this parameter governs the similarity threshold. Cutoff is specified in terms of category utility: When the increase in category utility from adding a new node is sufficiently small, that node is cut off.

Figure 6.24(b) shows the same iris data, clustered with cutoff in effect. Many leaf nodes contain several instances: These are children of the parent node that have been cut off. The division into the three types of iris is a little easier to see from this hierarchy because some of the detail is suppressed. Again, however, some experimentation with the cutoff parameter was necessary to get this result, and in fact a sharper cutoff leads to much less satisfactory clusters.

Similar clusterings are obtained if the full iris dataset of 150 instances is used. However, the results depend on the ordering of examples: Figure 6.24 was obtained by alternating the three varieties of iris in the input file. If all *Iris setosas* are presented first, followed by all *Iris versicolors* and then all *Iris virginicas*, the resulting clusters are quite unsatisfactory.

Category Utility

Now we look at how the category utility, which measures the overall quality of a partition of instances into clusters, is calculated. In Section 5.9 we learned how the MDL measure could, in principle, be used to evaluate the quality of clustering. Category utility is not MDL-based but rather resembles a kind of quadratic loss function defined on conditional probabilities.

The definition of category utility is rather formidable:

$$CU(C_1, C_2, \ldots, C_k) = \frac{\sum_\ell \Pr[C_\ell] \sum_i \sum_j (\Pr[a_i = v_{ij} \mid C_\ell]^2 - \Pr[a_i = v_{ij}]^2)}{k}$$

where C_1, C_2, …, C_k are the k clusters; the outer summation is over these clusters; the next inner one sums over the attributes; a_i is the ith attribute, and it takes on values v_{i1}, v_{i2}, …, which are dealt with by the sum over j. Note that the probabilities themselves are obtained by summing over all instances; thus, there is a further implied level of summation.

This expression makes a great deal of sense if you take the time to examine it. The point of having a cluster is that it will give some advantage in predicting the values of attributes of instances in that cluster—that is, $\Pr[a_i = v_{ij} \mid C_\ell]$ is a better estimate of the probability that attribute a_i has value v_{ij}, for an instance in cluster C_ℓ, than $\Pr[a_i = v_{ij}]$ because it takes account of the cluster the instance is in. If that information doesn't help, the clusters aren't doing much good! So what the measure calculates, inside the multiple summation, is the amount by which that information *does* help in terms of the differences between squares of probabilities. This is not quite the standard squared-difference metric because that sums the squares of the differences (which produces a symmetric result) and the present measure sums the difference of the squares (which, appropriately, does not produce a symmetric result). The differences between squares of probabilities are summed over all attributes, and all their possible values, in the inner double summation. Then it is summed over all clusters, weighted by their probabilities, in the outer summation.

The overall division by k is a little hard to justify because the squared differences have already been summed over the categories. It essentially provides a "per cluster" figure for the category utility that discourages overfitting. Otherwise, because the probabilities are derived by summing over the appropriate instances, the very best category utility would be obtained by placing each instance in its own cluster. Then $\Pr[a_i = v_{ij} \mid C_\ell]$ would be 1 for the value that attribute a_i actually has for the single instance in category C_ℓ and 0 for all other values; the numerator of the category utility formula will end up as

$$m - \sum_i \sum_j \Pr[a_i = v_{ij}]^2$$

where m is the total number of attributes. This is the greatest value that the numerator can have; thus, if it were not for the additional division by k in the category utility formula, there would never be any incentive to form clusters containing more than one member. This extra factor is best viewed as a rudimentary overfitting-avoidance heuristic.

This category utility formula applies only to nominal attributes. However, it can be easily extended to numeric attributes by assuming that their distribution is normal with a given (observed) mean μ and standard deviation σ. The probability density function for an attribute a is

$$f(a) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right)$$

The analog of summing the squares of attribute–value probabilities is

$$\sum_j \Pr[a_i = v_{ij}]^2 \Leftrightarrow \int f(a_i)^2 da_i = \frac{1}{2\sqrt{\pi}\sigma_i}$$

where σ_i is the standard deviation of the attribute a_i. Thus, for a numeric attribute we estimate the standard deviation from the data, both within the cluster (σ_{il}) and for the data over all clusters (σ_i), and use these in the category utility formula:

$$CU(C_1, C_2, \ldots, C_k) = \frac{1}{k}\sum_\ell \Pr[C_\ell] \frac{1}{2\sqrt{\pi}} \sum_i \left(\frac{1}{\sigma_{il}} - \frac{1}{\sigma_i}\right)$$

Now the problem mentioned that occurs when the standard deviation estimate is zero becomes apparent: A zero standard deviation produces an infinite value of the category utility formula. Imposing a prespecified minimum variance on each attribute, the acuity, is a rough-and-ready solution to the problem.

Probability-Based Clustering

Some of the shortcomings of the heuristic clustering method have already become apparent: the arbitrary division by k in the category utility formula that is necessary to prevent overfitting, the need to supply an artificial minimum value for the standard deviation of clusters, and the ad hoc cutoff value to prevent every single instance from becoming a cluster in its own right. On top of this is the uncertainty inherent in incremental algorithms. To what extent is the result dependent on the order of examples? Are the local restructuring operations of merging and splitting really enough to reverse the effect of bad initial decisions caused by unlucky ordering? Does the final result represent even a *local* maximum of category utility? Add to this the problem that one never knows how far the final configuration is to a *global* maximum—and, of course, the standard trick of repeating the clustering procedure several times and choosing the best will destroy the incremental nature of the algorithm. Finally, doesn't the hierarchical nature of the result really beg the question of which are the *best* clusters? There are so many clusters in Figure 6.24 that it is difficult to separate the wheat from the chaff.

A more principled statistical approach to the clustering problem can overcome some of these shortcomings. From a probabilistic perspective, the goal of clustering is to find the most likely set of clusters available given the data (and, inevitably, prior expectations). Because no finite amount of evidence is enough to make a completely firm decision on the matter, instances—even training instances—should not be placed categorically in one cluster or the other: Instead, they have a certain

probability of belonging to each cluster. This helps to eliminate the brittleness that is often associated with schemes that make hard and fast judgments.

The foundation for statistical clustering is a statistical model called *finite mixtures*. A *mixture* is a set of k probability distributions, representing k clusters, that govern the attribute values for members of that cluster. In other words, each distribution gives the probability that a particular instance would have a certain set of attribute values if it were *known* to be a member of that cluster. Each cluster has a different distribution. Any particular instance "really" belongs to one and only one of the clusters, but it is not known which one. Finally, the clusters are not equally likely: There is some probability distribution that reflects their relative populations.

The simplest finite-mixture situation is when there is only one numeric attribute, which has a Gaussian or normal distribution for each cluster—but with different means and variances. The clustering problem is to take a set of instances—in this case each instance is just a number—and a prespecified number of clusters, and work out each cluster's mean and variance and the population distribution between the clusters. The mixture model combines several normal distributions, and its probability density function looks like a mountain range with a peak for each component.

Figure 6.25 shows a simple example. There are two clusters, A and B, and each has a normal distribution with means and standard deviations μ_A and σ_A for cluster A and μ_B and σ_B for cluster B. Samples are taken from these distributions, using cluster A with probability p_A and cluster B with probability p_B (where $p_A + p_B = 1$), resulting in a dataset like that shown. Now, imagine being given the dataset without the classes—just the numbers—and being asked to determine the five parameters that characterize the model: μ_A, σ_A, μ_B, σ_B, and p_A (the parameter p_B can be calculated directly from p_A). That is the finite-mixture problem.

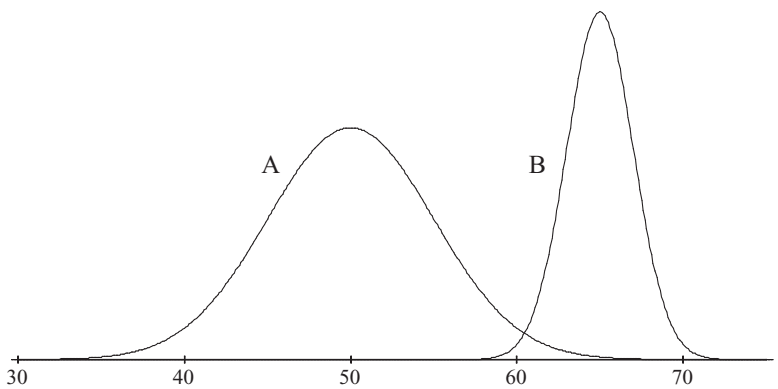


FIGURE 6.25

A two-class mixture model.

If you knew which of the two distributions each instance came from, finding the five parameters would be easy—just estimate the mean and standard deviation for the cluster A samples and the cluster B samples separately, using the formulas

$$\mu = \frac{x_1 + x_2 + \ldots + x_n}{n}$$

and

$$\sigma = \sqrt{\frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \ldots + (x_n - \mu)^2}{n-1}}$$

(The use of $n - 1$ rather than n as the denominator in the second formula is a technicality of sampling: It makes little difference in practice if n is used instead.) Here, x_1, x_2, ..., x_n are the samples from the distribution A or B. To estimate the fifth parameter p_A, just take the proportion of the instances that are in the A cluster.

If you knew the five parameters, finding the probabilities that a given instance comes from each distribution would be easy. Given an instance x, the probability that it belongs to cluster A is

$$\Pr[\text{A} \mid x] = \frac{\Pr[x \mid \text{A}] \times \Pr[\text{A}]}{\Pr[x]} = \frac{f(x; \mu_A, \sigma_A) p_A}{\Pr[x]}$$

where $f(x; \mu_A, \sigma_A)$ is the normal distribution function for cluster A—that is,

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

The denominator $\Pr[x]$ will disappear: We calculate the numerators for both $\Pr[\text{A} \mid x]$ and $\Pr[\text{B} \mid x]$ and normalize them by dividing by their sum. This whole procedure is just the same as the way numeric attributes are treated in the Naïve Bayes learning scheme of Section 4.2. And the caveat explained there applies here too: Strictly speaking, $f(x; \mu_A, \sigma_A)$ is not the probability $\Pr[x \mid \text{A}]$ because the probability of x being any particular real number is zero, but the normalization process makes the final result correct. Note that the final outcome is not a particular cluster but rather the *probabilities* with which x belongs to cluster A and cluster B.

The EM Algorithm

The problem is that we know neither of these things: not the distribution that each training instance came from nor the five mixture model parameters. So we adopt the procedure used for the *k*-means clustering algorithm and iterate. Start with initial guesses for the five parameters, use them to calculate the cluster probabilities for each instance, use these probabilities to reestimate the parameters, and repeat. (If you prefer, you can start with guesses for the classes of the instances instead.) This is called the *EM algorithm*, for *expectation maximization*. The first step—calculation of the cluster probabilities, which are the "expected" class values—is "expectation"; the second, calculation of the distribution parameters, is "maximization" of the likelihood of the distributions given the data available.

A slight adjustment must be made to the parameter estimation equations to account for the fact that it is only cluster probabilities, not the clusters themselves, that are known for each instance. These probabilities just act like weights. If w_i is the probability that instance i belongs to cluster A, the mean and standard deviation for cluster A are

$$\mu_A = \frac{w_1 x_1 + w_2 x_2 + \ldots + w_n x_n}{w_1 + w_2 + \ldots + w_n}$$

and

$$\sigma_A^2 = \frac{w_1 (x_1 - \mu)^2 + w_2 (x_2 - \mu)^2 + \ldots + w_n (x_n - \mu)^2}{w_1 + w_2 + \ldots + w_n}$$

where now the x_i are *all* the instances, not just those belonging to cluster A. (This differs in a small detail from the estimate for the standard deviation given later: If all weights are equal, the denominator is n rather than $n - 1$. Technically speaking, this is a "maximum-likelihood" estimator for the variance whereas the previous formula is for an "unbiased" estimator. The difference is not important in practice.)

Now consider how to terminate the iteration. The *k*-means algorithm stops when the classes of the instances don't change from one iteration to the next—a "fixed point" has been reached. In the EM algorithm things are not quite so easy: The algorithm converges toward a fixed point but never actually gets there. We can see how close it is getting by calculating the overall likelihood that the data came from this dataset, given the values for the five parameters. This overall likelihood is obtained by multiplying the probabilities of the individual instances i:

$$\prod_i (p_A \Pr[x_i \mid A] + p_B \Pr[x_i \mid B])$$

where the probabilities given the clusters A and B are determined from the normal distribution function $f(x; \mu, \sigma)$. This overall likelihood is a measure of the "goodness" of the clustering and increases at each iteration of the EM algorithm.

Again, there is a technical difficulty with equating the probability of a particular value of x with $f(x; \mu, \sigma)$, and in this case the effect does not disappear because no probability normalization operation is applied. The upshot is that the likelihood expression is not a probability and does not necessarily lie between 0 and 1; nevertheless, its magnitude still reflects the quality of the clustering. In practical implementations its logarithm is calculated instead: This is done by summing the logarithms of the individual components, avoiding multiplications. But the overall conclusion still holds: You should iterate until the increase in log-likelihood becomes negligible. For example, a practical implementation might iterate until the difference between successive values of log-likelihood is less than 10^{-10} for 10 successive iterations. Typically, the log-likelihood will increase very sharply over the first few iterations and then converge rather quickly to a point that is virtually stationary.

Although the EM algorithm is guaranteed to converge to a maximum, this is a *local* maximum and may not necessarily be the same as the global maximum. For a better chance of obtaining the global maximum, the whole procedure should be repeated several times, with different initial guesses for the parameter values. The overall log-likelihood figure can be used to compare the different final configurations obtained: Just choose the largest of the local maxima.

Extending the Mixture Model

Now that we have seen the Gaussian mixture model for two distributions, let's consider how to extend it to more realistic situations. The basic method is just the same, but because the mathematical notation becomes formidable we will not develop it in full detail.

Changing the algorithm from two-class problems to multiclass problems is completely straightforward as long as the number k of normal distributions is given in advance. The model can easily be extended from a single numeric attribute per instance to multiple attributes as long as independence between attributes is assumed. The probabilities for each attribute are multiplied together to obtain the joint probability for the instance, just as in the Naïve Bayes method.

When the dataset is known in advance to contain correlated attributes, the independence assumption no longer holds. Instead, two attributes can be modeled jointly by a bivariate normal distribution, in which each has its own mean value but the two standard deviations are replaced by a "covariance matrix" with four numeric parameters. There are standard statistical techniques for estimating the class probabilities of instances and for estimating the means and covariance matrix given the instances and their class probabilities. Several correlated attributes can be handled using a multivariate distribution. The number of parameters increases with the square of the number of jointly varying attributes. With n independent attributes, there are $2n$ parameters, a mean and a standard deviation for each. With n covariant attributes, there are $n + n(n + 1)/2$ parameters, a mean for each, and an $n \times n$ covariance matrix that is symmetric and therefore involves $n(n + 1)/2$ different quantities. This escalation in the number of parameters has serious consequences for overfitting, as we will explain later.

To cater for nominal attributes, the normal distribution must be abandoned. Instead, a nominal attribute with v possible values is characterized by v numbers representing the probability of each one. A different set of numbers is needed for every class; kv parameters in all. The situation is very similar to the Naïve Bayes method. The two steps of expectation and maximization correspond exactly to operations we have studied before. Expectation—estimating the cluster to which each instance belongs given the distribution parameters—is just like determining the class of an unknown instance. Maximization—estimating the parameters from the classified instances—is just like determining the attribute–value probabilities from the training instances, with the small difference that in the EM algorithm instances are assigned to classes probabilistically rather than categorically. In Section 4.2 we encountered the problem that probability estimates can turn out to be zero, and the same problem occurs here too. Fortunately, the solution is just as simple—use the Laplace estimator.

Naïve Bayes assumes that attributes are independent—that is the reason why it is called "naïve." A pair of correlated nominal attributes with v_1 and v_2 possible values, respectively, can be replaced by a single covariant attribute with v_1v_2 possible values. Again, the number of parameters escalates as the number of dependent

attributes increases, and this has implications for probability estimates and overfitting.

The presence of both numeric and nominal attributes in the data to be clustered presents no particular problem. Covariant numeric and nominal attributes are more difficult to handle, and we will not describe them here.

Missing values can be accommodated in various different ways. In principle, they should be treated as unknown and the EM process adapted to estimate them as well as the cluster means and variances. A simple way is to replace them by means or modes in a preprocessing step.

With all these enhancements, probabilistic clustering becomes quite sophisticated. The EM algorithm is used throughout to do the basic work. The user must specify the number of clusters to be sought, the type of each attribute (numeric or nominal), which attributes are to be modeled as covarying, and what to do about missing values. Moreover, different distributions can be used. Although the normal distribution is usually a good choice for numeric attributes, it is not suitable for attributes (such as weight) that have a predetermined minimum (0 in the case of weight) but no upper bound; in this case a "log-normal" distribution is more appropriate. Numeric attributes that are bounded above and below can be modeled by a "log-odds" distribution. Attributes that are integer counts rather than real values are best modeled by the "Poisson" distribution. A comprehensive system might allow these distributions to be specified individually for each attribute. In each case, the distribution involves numeric parameters—probabilities of all possible values for discrete attributes and mean and standard deviation for continuous ones.

In this section we have been talking about clustering. But you may be thinking that these enhancements could be applied just as well to the Naïve Bayes algorithm too—and you'd be right. A comprehensive probabilistic modeler could accommodate both clustering and classification learning, nominal and numeric attributes with a variety of distributions, various possibilities of covariation, and different ways of dealing with missing values. The user would specify, as part of the domain knowledge, which distributions to use for which attributes.

Bayesian Clustering

However, there is a snag: overfitting. You might say that if we are not sure which attributes are dependent on each other, why not be on the safe side and specify that *all* the attributes are covariant? The answer is that the more parameters there are, the greater the chance that the resulting structure is overfitted to the training data—and covariance increases the number of parameters dramatically. The problem of overfitting occurs throughout machine learning, and probabilistic clustering is no exception. There are two ways that it can occur: through specifying too large a number of clusters and through specifying distributions with too many parameters.

The extreme case of too many clusters occurs when there is one for every data point: Clearly, that will be overfitted to the training data. In fact, in the mixture model, problems will occur whenever any of the normal distributions becomes so

narrow that the cluster is centered on just one data point. Consequently, implementations generally insist that clusters contain at least two different data values.

Whenever there are a large number of parameters, the problem of overfitting arises. If you were unsure of which attributes were covariant, you might try out different possibilities and choose the one that maximized the overall probability of the data given the clustering that was found. Unfortunately, the more parameters there are, the larger the overall data probability will tend to be—not necessarily because of better clustering but because of overfitting. The more parameters there are to play with, the easier it is to find a clustering that seems good.

It would be nice if somehow you could penalize the model for introducing new parameters. One principled way of doing this is to adopt a Bayesian approach in which every parameter has a prior probability distribution. Then, whenever a new parameter is introduced, its prior probability must be incorporated into the overall likelihood figure. Because this will involve multiplying the overall likelihood by a number less than 1—the prior probability—it will automatically penalize the addition of new parameters. To improve the overall likelihood, the new parameters will have to yield a benefit that outweighs the penalty.

In a sense, the Laplace estimator that was introduced in Section 4.2, and whose use we advocated earlier to counter the problem of zero probability estimates for nominal values, is just such a device. Whenever observed probabilities are small, the Laplace estimator exacts a penalty because it makes probabilities that are zero, or close to zero, greater, and this will decrease the overall likelihood of the data. Making two nominal attributes covariant will exacerbate the problem of small probabilities. Instead of $v_1 + v_2$ parameters, where v_1 and v_2 are the number of possible values, there are now $v_1 v_2$, greatly increasing the chance of a large number of small estimated probabilities. In fact, the Laplace estimator is tantamount to using a particular prior distribution for the introduction of new parameters.

The same technique can be used to penalize the introduction of large numbers of clusters, just by using a prespecified prior distribution that decays sharply as the number of clusters increases. AutoClass is a comprehensive Bayesian clustering scheme that uses the finite-mixture model with prior distributions on all the parameters. It allows both numeric and nominal attributes and uses the EM algorithm to estimate the parameters of the probability distributions to best fit the data. Because there is no guarantee that the EM algorithm converges to the global optimum, the procedure is repeated for several different sets of initial values. But that is not all. AutoClass considers different numbers of clusters and can consider different amounts of covariance and different underlying probability distribution types for the numeric attributes. This involves an additional, outer level of search. For example, it initially evaluates the log-likelihood for 2, 3, 5, 7, 10, 15, and 25 clusters: After that, it fits a log-normal distribution to the resulting data and randomly selects from it more values to try. As you might imagine, the overall algorithm is extremely computation intensive. In fact, the actual implementation starts with a prespecified time bound and continues to iterate as long as time allows. Give it longer and the results may be better!

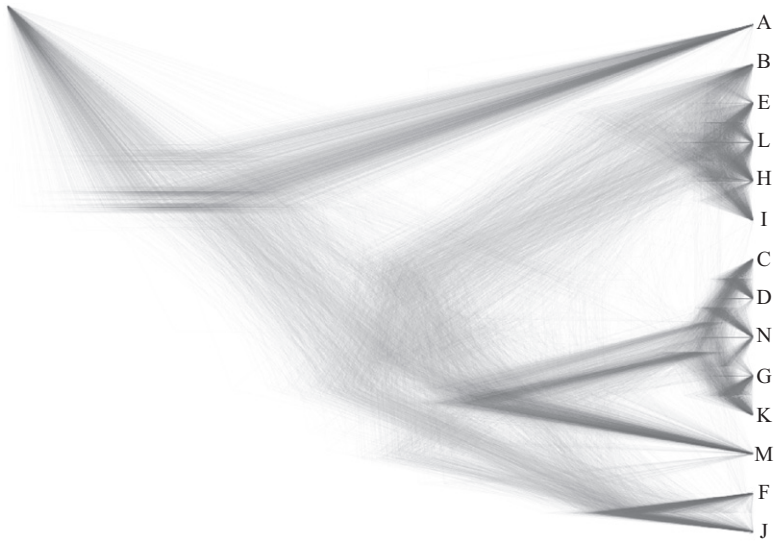


FIGURE 6.26

DensiTree showing possible hierarchical clusterings of a given dataset.

Rather than showing just the most likely clustering to the user, it may be best to present all of them, weighted by probability. Recently, fully Bayesian techniques for *hierarchical* clustering have been developed that produce as output a probability distribution over possible hierarchical structures representing a dataset. Figure 6.26 is a visualization, known as a *DensiTree*, that shows the set of all trees for a particular dataset in a triangular shape. The tree is best described in terms of its "clades," a biological term from the Greek *klados* meaning *branch*, for a group of the same species that includes all ancestors. Here, there are five clearly distinguishable clades. The first and fourth correspond to a single leaf, while the fifth has two leaves that are so distinct they might be considered clades in their own right. The second and third clades each have five leaves, and there is large uncertainty in their topology. Such visualizations make it easy for people to grasp the possible hierarchical clusterings of their data, at least in terms of the big picture.

Discussion

The clustering methods that have been described produce different kinds of output. All are capable of taking new data in the form of a test set and classifying it according to clusters that were discovered by analyzing a training set. However, the hierarchical and incremental clustering methods are the only ones that generate an explicit knowledge structure that describes the clustering in a way that can be visualized and reasoned about. The other algorithms produce clusters that could be visualized in instance space if the dimensionality were not too high.

If a clustering method were used to label the instances of the training set with cluster numbers, that labeled set could then be used to train a rule or decision tree learner. The resulting rules or tree would form an explicit description of the classes. A probabilistic clustering scheme could be used for the same purpose, except that each instance would have multiple weighted labels and the rule or decision tree learner would have to be able to cope with weighted instances—as many can.

Another application of clustering is to fill in any values of the attributes that may be missing. For example, it is possible to make a statistical estimate of the value of unknown attributes of a particular instance, based on the class distribution for the instance itself and the values of the unknown attributes for other examples.

All the clustering methods we have examined make, at some level, a basic assumption of independence among the attributes. AutoClass does allow the user to specify in advance that two or more attributes are dependent and should be modeled with a joint probability distribution. (There are restrictions, however: Nominal attributes may vary jointly, as may numeric attributes, but not both together. Moreover, missing values for jointly varying attributes are not catered for.) It may be advantageous to preprocess a dataset to make the attributes more independent, using statistical techniques such as the principal components transform described in Section 7.3. Note that joint variation that is specific to particular classes will not be removed by such techniques; they only remove overall joint variation that runs across all classes.

Our description of how to modify *k*-means to find a good value of *k* by repeatedly splitting clusters and seeing whether the split is worthwhile follows the *X*-means algorithm of Moore and Pelleg (2000). However, instead of the MDL principle, they use a probabilistic scheme called the Bayes Information Criterion (Kass and Wasserman, 1995). Efficient agglomerative methods for hierarchical clustering were developed by Day and Edelsbrünner (1984), and the ideas are described in recent books (Duda et al., 2001; Hastie et al., 2009). The incremental clustering procedure, based on the merging and splitting operations, was introduced in systems called Cobweb for nominal attributes (Fisher, 1987) and Classit for numeric attributes (Gennari et al., 1990). Both are based on a measure of category utility that had been defined previously (Gluck and Corter, 1985). The AutoClass program is described by Cheeseman and Stutz (1995). Two implementations have been produced: the original research implementation, written in LISP, and a follow-up public implementation in C that is 10 or 20 times faster but somewhat more restricted—for example, only the normal-distribution model is implemented for numeric attributes. DensiTrees were developed by Bouckaert (2010).

A hierarchical clustering method called BIRCH (balanced iterative reducing and clustering using hierarchies) has been developed specifically for large multidimensional datasets, where it is necessary for efficient operation to minimize input–output costs (Zhang et al., 1996). It incrementally and dynamically clusters multidimensional metric data points, seeking the best clustering within given memory and time constraints. It typically finds a good clustering with a single scan of the data, which can then be improved by further scans.

6.9 SEMISUPERVISED LEARNING

When introducing the machine learning process in Chapter 2 (page 40), we drew a sharp distinction between supervised and unsupervised learning—classification and clustering. In this chapter we have studied a lot of techniques for both. Recently, researchers have begun to explore territory between the two, sometimes called *semisupervised learning*, in which the goal is classification but the input contains both unlabeled and labeled data. You can't do classification without labeled data, of course, because only the labels tell what the classes are. But it is sometimes attractive to augment a small amount of labeled data with a large pool of unlabeled data. It turns out that the unlabeled data can help you learn the classes. How can this be?

First, why would you want it? Many situations present huge volumes of raw data, but assigning classes is expensive because it requires human insight. Text mining provides some classic examples. Suppose you want to classify web pages into predefined groups. In an academic setting you might be interested in faculty pages, graduate student pages, course information pages, research group pages, and department pages. You can easily download thousands, or millions, of relevant pages from university web sites. But labeling the training data is a laborious manual process. Or suppose your job is to use machine learning to spot names in text, differentiating between personal names, company names, and place names. You can easily download megabytes, or gigabytes, of text, but making this into training data by picking out the names and categorizing them can only be done manually. Cataloging news articles, sorting electronic mail, learning users' reading interests—the applications are legion. Leaving text aside, suppose you want to learn to recognize certain famous people in television broadcast news. You can easily record hundreds or thousands of hours of newscasts, but again labeling is manual. In any of these scenarios it would be enormously attractive to be able to leverage a large pool of unlabeled data to obtain excellent performance from just a few labeled examples, particularly if you were the graduate student who had to do the labeling!

Clustering for Classification

How can unlabeled data be used to improve classification? Here's a simple idea. Use Naïve Bayes to learn classes from a small labeled dataset and then extend it to a large unlabeled dataset using the EM iterative clustering algorithm from the previous section. The procedure is this. First, train a classifier using the labeled data. Second, apply it to the unlabeled data to label it with class probabilities (the "expectation" step). Third, train a new classifier using the labels for all the data (the "maximization" step). Fourth, iterate until convergence. You could think of this as iterative clustering, where starting points and cluster labels are gleaned from the labeled data. The EM procedure guarantees finding model parameters that have equal or greater likelihood at each iteration. The key question, which can only be answered empirically, is whether these higher likelihood parameter estimates will improve classification accuracy.

Intuitively, this might work well. Consider document classification. Certain phrases are indicative of the classes. Some of them occur in labeled documents, whereas others occur only in unlabeled ones. There are, however, probably some documents that contain both, and the EM procedure uses these to generalize the learned model to use phrases that do not appear in the labeled dataset. For example, both *supervisor* and *Ph.D. topic* might indicate a graduate student's home page. Suppose only the former phrase occurs in the labeled documents. EM iteratively generalizes the model to correctly classify documents that contain just the latter.

This might work with any classifier and any iterative clustering algorithm. But it is basically a bootstrapping procedure, and you must take care to ensure that the feedback loop is a positive one. Using probabilities rather than hard decisions seems beneficial because it allows the procedure to converge slowly instead of jumping to conclusions that may be wrong. Naïve Bayes, together with the basic probabilistic EM procedure, is a particularly apt choice because the two share the same fundamental assumption: independence between attributes or, more precisely, conditional independence between attributes given the class.

Of course, the independence assumption is universally violated. Even our little example used the two-word phrase *Ph.D. topic*, whereas actual implementations would likely use individual words as attributes—and the example would have been far less compelling if we had substituted either of the single terms *Ph.D.* or *topic*. The phrase *Ph.D. students* is probably more indicative of faculty rather than graduate student home pages; the phrase *research topic* is probably less discriminating. It is the very fact that *Ph.D.* and *topic* are *not* conditionally independent given the class that makes the example work: It is their combination that characterizes graduate student pages.

Nevertheless, coupling Naïve Bayes and EM in this manner works well in the domain of document classification. In a particular classification task it attained the performance of a traditional learner using fewer than one-third of the labeled training instances, as well as five times as many unlabeled ones. This is a good tradeoff when labeled instances are expensive but unlabeled ones are virtually free. With a small number of labeled documents, classification accuracy can be improved dramatically by incorporating many unlabeled ones.

Two refinements to the procedure have been shown to improve performance. The first is motivated by experimental evidence showing that when there are many labeled documents the incorporation of unlabeled data may reduce rather than increase accuracy. Hand-labeled data is (or should be) inherently less noisy than automatically labeled data. The solution is to introduce a weighting parameter that reduces the contribution of the unlabeled data. This can be incorporated into the maximization step of EM by maximizing the weighted likelihood of the labeled and unlabeled instances. When the parameter is close to 0, unlabeled documents have little influence on the shape of EM's hill-climbing surface; when it is close to 1, the algorithm reverts to the original version in which the surface is equally affected by both kinds of document.

The second refinement is to allow each class to have several clusters. As explained in the previous section, the EM clustering algorithm assumes that the data is generated randomly from a mixture of different probability distributions, one per cluster. Until now, a one-to-one correspondence between mixture components and classes has been assumed. In many circumstances, including document classification, this is unrealistic because most documents address multiple topics. With several clusters per class, each labeled document is initially assigned randomly to each of its components in a probabilistic fashion. The maximization step of the EM algorithm remains as before, but the expectation step is modified not only to probabilistically label each example with the classes but to probabilistically assign it to the components within the class. The number of clusters per class is a parameter that depends on the domain and can be set by cross-validation.

Co-training

Another situation in which unlabeled data can improve classification performance is when there are two different and independent perspectives on the classification task. The classic example again involves documents, this time web documents, where the two perspectives are the *content* of a web page and the *links* to it from other pages. These two perspectives are well known to be both useful and different: Successful web search engines capitalize on them both using secret recipes. The text that labels a link to another web page gives a revealing clue as to what that page is about—perhaps even more revealing than the page's own content, particularly if the link is an independent one. Intuitively, a link labeled *my advisor* is strong evidence that the target page is a faculty member's home page.

The idea, called *co-training*, is this. Given a few labeled examples, first learn a different model for each perspective—in this case a content-based and a hyperlink-based model. Then use each one separately to label the unlabeled examples. For each model, select the example that it most confidently labels as positive and the one it most confidently labels as negative, and add these to the pool of labeled examples. Better yet, maintain the ratio of positive and negative examples in the labeled pool by choosing more of one kind than the other. In either case, repeat the whole procedure, training both models on the augmented pool of labeled examples, until the unlabeled pool is exhausted.

There is some experimental evidence, using Naïve Bayes throughout as the learner, that this bootstrapping procedure outperforms one that employs all the features from both perspectives to learn a single model from the labeled data. It relies on having two different views of an instance that are redundant but not completely correlated. Various domains have been proposed, from spotting celebrities in televised newscasts using video and audio separately to mobile robots with vision, sonar, and range sensors. The independence of the views reduces the likelihood of both hypotheses agreeing on an erroneous label.

EM and Co-training

On datasets with two feature sets that are truly independent, experiments have shown that co-training gives better results than using EM as described previously. Even better performance, however, can be achieved by combining the two into a modified version of co-training called *co-EM*. Co-training trains two classifiers representing different perspectives, A and B, and uses both to add new examples to the training pool by choosing whichever unlabeled examples they classify most positively or negatively. The new examples are few in number and deterministically labeled. Co-EM, on the other hand, trains classifier A on the labeled data and uses it to *probabilistically* label *all* the unlabeled data. Next it trains classifier B on both the labeled data and the unlabeled data with classifier A's tentative labels, and then it probabilistically relabels all the data for use by classifier A. The process iterates until the classifiers converge. This procedure seems to perform consistently better than co-training because it does not commit to the class labels that are generated by classifiers A and B but rather reestimates their probabilities at each iteration.

The range of applicability of co-EM, like co-training, is still limited by the requirement for multiple independent perspectives. But there is some experimental evidence to suggest that even when there is no natural split of features into independent perspectives, benefits can be achieved by manufacturing such a split and using co-training—or, better yet, co-EM—on the split data. This seems to work even when the split is made randomly; performance could surely be improved by engineering the split so that the feature sets are maximally independent. Why does this work? Researchers have hypothesized that these algorithms succeed in part because the split makes them more robust to the assumptions that their underlying classifiers make.

There is no particular reason to restrict the base classifier to Naïve Bayes. Support vector machines probably represent the most successful technology for text categorization today. However, for the EM iteration to work it is necessary that the classifier labels the data probabilistically; it must also be able to use probabilistically weighted examples for training. Support vector machines can easily be adapted to do both. We explained how to adapt learning algorithms to deal with weighted instances in Section 6.6, under Locally Weighted Linear Regression (page 258). One way of obtaining probability estimates from support vector machines is to fit a one-dimensional logistic model to the output, effectively performing logistic regression as described in Section 4.6 on the output. Excellent results have been reported for text classification using co-EM with the support vector machine (SVM) classifier. It outperforms other variants of SVM and seems quite robust to varying proportions of labeled and unlabeled data.

Discussion

Nigam et al. (2000) thoroughly explored the idea of clustering for classification, showing how the EM clustering algorithm can use unlabeled data to improve an initial classifier built by Naïve Bayes. The idea of co-training is older: Blum

and Mitchell (1998) pioneered it and developed a theoretical model for the use of labeled and unlabeled data from different independent perspectives. Nigam and Ghani (2000) analyzed the effectiveness and applicability of co-training, relating it to the traditional use of standard expectation maximization to fill in missing values; they also introduced the co-EM algorithm. Up to this point, co-training and co-EM have been applied mainly to small two-class problems. Ghani (2002) used error-correcting output codes to address multiclass situations with many classes. Brefeld and Scheffer (2004) extended co-EM to use a support vector machine rather than Naïve Bayes.

6.10 MULTI-INSTANCE LEARNING

All the techniques described in this chapter so far are for the standard machine learning scenario where each example consists of a single instance. Before moving on to methods for transforming the input data in Chapter 7, we revisit the more complex setting of multi-instance learning, in which each example consists of a bag of instances instead. We describe approaches that are more advanced than the simple techniques discussed in Section 4.9. First, we consider how to convert multi-instance learning to single-instance learning by transforming the data. Then we discuss how to upgrade single-instance learning algorithms to the multi-instance case. Finally, we take a look at some methods that have no direct equivalent in single-instance learning.

Converting to Single-Instance Learning

Section 4.9 (page 142) presented some ways of applying standard single-instance learning algorithms to multi-instance data by aggregating the input or the output. Despite their simplicity, these techniques often work surprisingly well in practice. Nevertheless, there are clearly situations in which they will fail. Consider the method of aggregating the input by computing the minimum and maximum values of numeric attributes present in the bag and treating the result as a single instance. This will yield a huge loss of information because attributes are condensed to summary statistics individually and independently. Can a bag be converted to a single instance without discarding quite so much information?

The answer is yes, although the number of attributes that are present in the so-called "condensed" representation may increase substantially. The basic idea is to partition the instance space into regions and create one attribute per region in the single-instance representation. In the simplest case, attributes can be Boolean: If a bag has at least one instance in the region corresponding to a particular attribute the value of the attribute is set to true; otherwise, it is set to false. However, to preserve more information the condensed representation could instead contain numeric attributes, the values of which are counts that indicate how many instances of the bag lie in the corresponding region.

Regardless of the exact types of attributes that are generated, the main problem is to come up with a partitioning of the input space. A simple approach is to partition it into hypercubes of equal size. Unfortunately, this only works when the space has very few dimensions (i.e., attributes): The number of cubes required to achieve a given granularity grows exponentially with the dimension of the space. One way to make this approach more practical is to use unsupervised learning. Simply take all the instances from all the bags in the training data, discard their class labels, and form a big single-instance dataset; then process it with a clustering technique such as *k*-means. This will create regions corresponding to the different clusters (*k* regions, in the case of *k*-means). Then, for each bag, create one attribute per region in the condensed representation and use it as described previously.

Clustering is a rather heavy-handed way to infer a set of regions from the training data because it ignores information about class membership. An alternative approach that often yields better results is to partition the instance space using decision tree learning. Each leaf of a tree corresponds to one region of instance space. But how can a decision tree be learned when the class labels apply to entire bags of instances rather than to individual instances? The approach described under Aggregating the Output in Section 4.9 can be used: Take the bag's class label and attach it to each of its instances. This yields a single-instance dataset, ready for decision tree learning. Many of the class labels will be incorrect—the whole point of multi-instance learning is that it is not clear how bag-level labels relate to instance-level ones. However, these class labels are only being used to obtain a partition of instance space. The next step is to transform the multi-instance dataset into a single-instance one that represents how instances from each bag are distributed throughout the space. Then another single-instance learning method is applied—perhaps, again, decision tree learning—that determines the importance of individual attributes in the condensed representation which correspond to regions in the original space.

Using decision trees and clustering yields "hard" partition boundaries, where an instance either does or does not belong to a region. Such partitions can also be obtained using a distance function, combined with some reference points, by assigning instances to their closest reference point. This implicitly divides the space into regions, each corresponding to one reference point. (In fact, this is exactly what happens in *k*-means clustering: The cluster centers are the reference points.) But there is no fundamental reason to restrict attention to hard boundaries: We can make the region membership function "soft" by using distance—transformed into a similarity score—to compute attribute values in the condensed representation of a bag. All that is needed is some way of aggregating the similarity scores between each bag and reference point into a single value—for example, by taking the maximum similarity between each instance in that bag and the reference point.

In the simplest case, each instance in the training data can be used as a reference point. That creates a large number of attributes in the condensed representation, but it preserves much of the information from a bag of instances in its corresponding single-instance representation. This method has been successfully applied to multi-instance problems.

Regardless of how the approach is implemented, the basic idea is to convert a bag of instances into a single one by describing the distribution of instances from this bag in instance space. Alternatively, ordinary learning methods can be applied to multi-instance data by aggregating the output rather than the input. Section 4.9 described a simple way: Join instances of bags in the training data into a single dataset by attaching bag-level class labels to them, perhaps weighting instances to give each bag the same total weight. A single-instance classification model can then be built. At classification time, predictions for individual instances are combined—for example, by averaging predicted class probabilities.

Although this approach often works well in practice, attaching bag-level class labels to instances is simplistic. Generally, the assumption in multi-instance learning is that only some of the instances—perhaps just one—are responsible for the class label of the associated bag. How can the class labels be corrected to yield a more accurate representation of the true underlying situation? This is obviously a difficult problem; if it were solved, it would make little sense to investigate other approaches to multi-instance learning. One method that has been applied is iterative: Start by assigning each instance its bag's class label and learn a single-instance classification model; then replace the instances' class labels by the predicted labels of this single-instance classification model for these instances. Repeat the whole procedure until the class labels remain unchanged from one iteration to the next.

Some care is needed to obtain sensible results. For example, suppose every instance in a bag were to receive a class label that differs from the bag's label. Such a situation should be prevented by forcing the bag's label on at least one instance—for example, the one with the largest predicted probability for this class.

This iterative approach has been investigated for the original multi-instance scenario with two class values, where a bag is positive if and only if one of its instances is positive. In that case it makes sense to assume that all instances from negative bags are truly negative and modify only the class labels of instances from positive bags. At prediction time, bags are classified as positive if one of their instances is classified as positive.

Upgrading Learning Algorithms

Tackling multi-instance learning by modifying the input or output so that single-instance schemes can be applied is appealing because there is a large battery of such techniques that can then be used directly, without any modification. However, it may not be the most efficient approach. An alternative is to adapt the internals of a single-instance algorithm to the multi-instance setting. This can be done in a particularly elegant fashion if the algorithm in question only considers the data through application of a distance (or similarity) function, as with nearest-neighbor classifiers or support vector machines. These can be adapted by providing a distance (or similarity) function for multi-instance data that computes a score between two bags of instances.

In the case of kernel-based methods such as support vector machines, the similarity must be a proper kernel function that satisfies certain mathematical properties. One that has been used for multi-instance data is the so-called *set kernel*. Given a kernel function for pairs of instances that support vector machines can apply to single-instance data—for example, one of the kernel functions considered in Section 6.4—the set kernel sums it over all pairs of instances from the two bags being compared. This idea is generic and can be applied with any single-instance kernel function.

Nearest-neighbor learning has been adapted to multi-instance data by applying variants of the Hausdorff distance, which is defined for sets of points. Given two bags and a distance function between pairs of instances—for example, the Euclidean distance—the Hausdorff distance between the bags is the largest distance from any instance in one bag to its closest instance in the other bag. It can be made more robust to outliers by using the *n*th-largest distance rather than the maximum.

For learning algorithms that are not based on similarity scores, more work is required to upgrade them to multi-instance data. There are multi-instance algorithms for rule learning and for decision tree learning, but we will not describe them here. Adapting algorithms to the multi-instance case is more straightforward if the algorithm concerned is essentially a numerical optimization strategy that is applied to the parameters of some function by minimizing a loss function on the training data. Logistic regression (Section 4.6) and multilayer perceptrons (Section 6.4) fall into this category; both have been adapted to multi-instance learning by augmenting them with a function that aggregates instance-level predictions. The so-called "soft maximum" is a differentiable function that is suitable for this purpose: It aggregates instance-level predictions by taking their (soft) maximum as the bag-level prediction.

Dedicated Multi-Instance Methods

Some multi-instance learning schemes are not based directly on single-instance algorithms. Here is an early technique that was specifically developed for the drug activity prediction problem mentioned in Section 2.2 (page 49), in which instances are conformations of a molecule and a molecule (i.e., a bag) is considered positive if and only if it has at least one active conformation. The basic idea is to learn a single hyperrectangle that contains at least one instance from each positive bag in the training data and no instances from any negative bags. Such a rectangle encloses an area of instance space where all positive bags overlap, but it contains no negative instances—an area that is common to all active molecules but not represented in any inactive ones. The particular drug activity data originally considered was high-dimensional, with 166 attributes describing each instance. In such a case it is computationally difficult to find a suitable hyperrectangle. Consequently, a heuristic approach was developed that is tuned to this particular problem.

Other geometric shapes can be used instead of hyperrectangles. Indeed, the same basic idea has been applied using hyperspheres (balls). Training instances are treated

as potential ball centers. For each one, a radius is found that yields the smallest number of errors for the bags in the training data. The original multi-instance assumption is used to make predictions: A bag is classified as positive if and only if it has at least one instance inside the ball. A single ball is generally not powerful enough to yield good classification performance. However, this method is not intended as a standalone algorithm. Rather, it is advocated as a "weak" learner to be used in conjunction with boosting algorithms (see Section 8.4) to obtain a powerful ensemble classifier—an ensemble of balls.

The dedicated multi-instance methods discussed so far have hard decision boundaries: An instance either falls inside or outside a ball or hyperrectangle. Other multi-instance algorithms use soft concept descriptions couched in terms of probability theory. The so-called *diverse-density* method is a classic example, again designed with the original multi-instance assumption in mind. Its basic and most commonly used form learns a single reference point in instance space. The probability that an instance is positive is computed from its distance to this point: It is 1 if the instance coincides with the reference point and decreases with increasing distance from this point, usually based on a bell-shaped function.

The probability that a bag is positive is obtained by combining the individual probabilities of the instances it contains, generally using the "noisy-OR" function. This is a probabilistic version of the logical OR. If all instance-level probabilities are 0, the noisy-OR value—and thus the bag-level probability—is 0; if at least one instance-level probability is 1, the value is 1; otherwise, the value falls somewhere in between.

The diverse density is defined as the probability of the class labels of the bags in the training data, computed based on this probabilistic model. It is maximized when the reference point is located in an area where positive bags overlap and no negative bags are present, just as for the two geometric methods discussed previously. A numerical optimization routine such as gradient ascent can be used to find the reference point that maximizes the diverse-density measure. In addition to the location of the reference point, implementations of diverse density also optimize the scale of the distance function in each dimension because generally not all attributes are equally important. This can improve predictive performance significantly.

Discussion

Condensing the input data by aggregating information into simple summary statistics is a well-known technique in multirelational learning, used in the RELAGGS system by Krogel and Wrobel (2002); multi-instance learning can be viewed as a special case of this more general setting (de Raedt, 2008). The idea of replacing simple summary statistics by region-based attributes, derived from partitioning the instance space, was explored by Weidmann et al. (2003) and Zhou and Zhang (2007). Using reference points to condense bags was investigated by Chen et al. (2006) and evaluated in a broader context by Foulds and Frank (2008). Andrews et al. (2003) proposed manipulating the class labels of individual instances using an iterative learning

process for learning support vector machine classifiers based on the original multi-instance assumption.

Nearest-neighbor learning based on variants of the Hausdorff distance was investigated by Wang and Zucker (2000). Gärtner et al. (2002) experimented with the set kernel to learn support vector machine classifiers for multi-instance data. Multi-instance algorithms for rule and decision tree learning, which are not covered here, have been described by Chevaleyre and Zucker (2001) and Blockeel et al. (2005). Logistic regression has been adapted for multi-instance learning by Xu and Frank (2004) and Ray and Craven (2005); multilayer perceptrons have been adapted by Ramon and de Raedt (2000).

Hyperrectangles and spheres were considered as concept descriptions for multi-instance learning by Dietterich et al. (1997) and Auer and Ortner (2004), respectively. The diverse-density method is the subject of Maron's (1998) Ph.D. thesis, and is also described in Maron and Lozano-Peréz (1997).

The multi-instance literature makes many different assumptions regarding the type of concept to be learned, defining, for example, how the bag-level and instance-level class labels are connected, starting with the original assumption that a bag is labeled positive if and only if one of its instances is positive. A review of assumptions in multi-instance learning can be found in Foulds and Frank (2010).

6.11 WEKA IMPLEMENTATIONS

For classifiers, see Section 11.4 and Table 11.5. For clustering methods, see Section 11.6 and Table 11.7.

- Decision trees:
  - *J48* (implementation of C4.5)
  - *SimpleCart* (minimum cost-complexity pruning à la CART)
  - *REPTree* (reduced-error pruning)
- Classification rules:
  - *JRip* (RIPPER rule learner)
  - *Part* (rules from partial decision trees)
  - *Ridor* (ripple-down rule learner)
- Association rules (see Section 11.7 and Table 11.8):
  - *FPGrowth* (frequent-pattern trees)
  - *GeneralizedSequentialPatterns* (find large item trees in sequential data)
- Linear models and extensions:
  - *SMO* and variants for learning support vector machines
  - *LibSVM* (uses third-party *libsvm* library)
  - *MultilayerPerceptron*
  - *RBFNetwork* (radial-basis function network)
  - *SPegasos* (SVM using stochastic gradient descent)

- Instance-based learning:
  - *IBk* (k-nearest neighbour classifier)
  - *KStar* (generalized distance functions)
  - *NNge* (rectangular generalizations)
- Numeric prediction:
  - *M5P* (model trees)
  - *M5Rules* (rules from model trees)
  - *LWL* (locally weighted learning)
- Bayesian networks:
  - *BayesNet*
  - *AODE*, *WAODE* (averaged one-dependence estimator)
- Clustering:
  - *XMeans*
  - *Cobweb* (includes Classit)
  - *EM*
- Multi-instance learning:
  - *MISVM* (iterative method for learning SVM by relabeling instances)
  - *MISMO* (SVM with multi-instance kernel)
  - *CitationKNN* (nearest-neighbor method with Hausdorff distance)
  - *MILR* (logistic regression for multi-instance data)
  - *MIOptimalBall* (learning balls for multi-instance classification)
  - *MIDD* (the diverse-density method using the noisy-OR function)