Chapter 6

Decision Trees

6.1 Definitions

A *decision tree* (generally defined) is a tree whose internal nodes are tests (on input patterns) and whose leaf nodes are categories (of patterns). We show an example in Fig. 6.1. A decision tree assigns a class number (or output) to an input pattern by filtering the pattern down through the tests in the tree. Each test has mutually exclusive and exhaustive outcomes. For example, test T_2 in the tree of Fig. 6.1 has three outcomes; the left-most one assigns the input pattern to class 3, the middle one sends the input pattern down to test T_4, and the right-most one assigns the pattern to class 1. We follow the usual convention of depicting the leaf nodes by the class number.[1] Note that in discussing decision trees we are not limited to implementing Boolean functions—they are useful for general, categorically valued functions.

There are several dimensions along which decision trees might differ:

1. The tests might be *multivariate* (testing on several features of the input at once) or *univariate* (testing on only one of the features).

2. The tests might have two outcomes or more than two. (If all of the tests have two outcomes, we have a *binary decision tree*.)

[1] One of the researchers who has done a lot of work on learning decision trees is Ross Quinlan. Quinlan distinguishes between classes and categories. He calls the subsets of patterns that filter down to each tip *categories* and subsets of patterns having the same label *classes*. In Quinlan's terminology, our example tree has nine categories and three classes. We will not make this distinction, however, but will use the words "category" and "class" interchangeably to refer to what Quinlan calls "class."

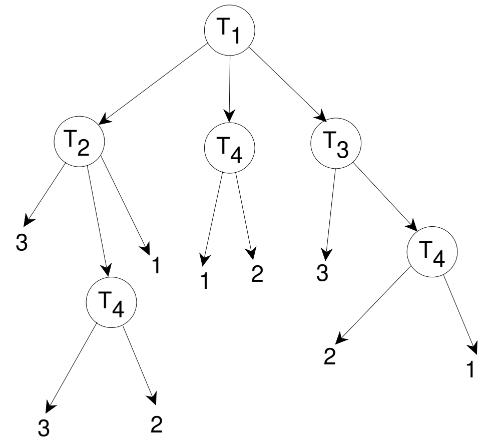


Figure 6.1: A Decision Tree

3. The features or attributes might be categorical or numeric. (Binary-valued ones can be regarded as either.)

4. We might have two classes or more than two. If we have two classes and binary inputs, the tree implements a Boolean function, and is called a Boolean decision tree.

It is straightforward to represent the function implemented by a univariate Boolean decision tree in DNF form. The DNF form implemented by such a tree can be obtained by tracing down each path leading to a tip node corresponding to an output value of 1, forming the conjunction of the tests along this path, and then taking the disjunction of these conjunctions. We show an example in Fig. 6.2. In drawing univariate decision trees, each non-leaf node is depicted by a single attribute. If the attribute has value 0 in the input pattern, we branch left; if it has value 1, we branch right.

The k-DL class of Boolean functions can be implemented by a multivariate decision tree having the (highly unbalanced) form shown in Fig. 6.3. Each test, c_i, is a term of size k or less. The v_i all have values of 0 or 1.

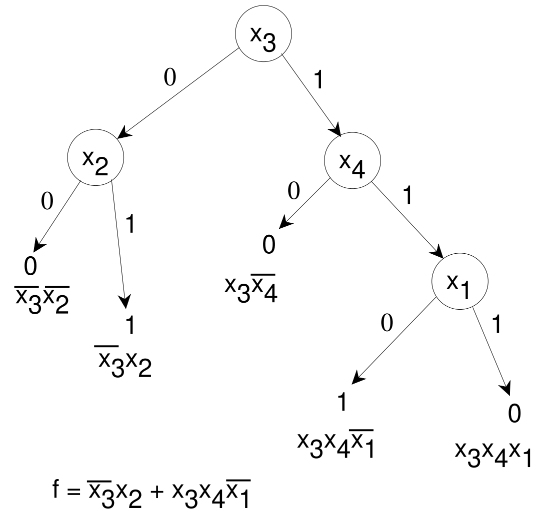


Figure 6.2: A Decision Tree Implementing a DNF Function

6.2 Supervised Learning of Univariate Decision Trees

Several systems for learning decision trees have been proposed. Prominent among these are ID3 and its new version, C4.5 [Quinlan, 1986, Quinlan, 1993], and CART [Breiman, *et al.*, 1984] We discuss here only batch methods, although incremental ones have also been proposed [Utgoff, 1989].

6.2.1 Selecting the Type of Test

As usual, we have n features or attributes. If the attributes are binary, the tests are simply whether the attribute's value is 0 or 1. If the attributes are categorical, but non-binary, the tests might be formed by dividing the attribute values into mutually exclusive and exhaustive subsets. A decision tree with such tests is shown in Fig. 6.4. If the attributes are numeric, the tests might involve "interval tests," for example $7 \leq x_i \leq 13.2$.

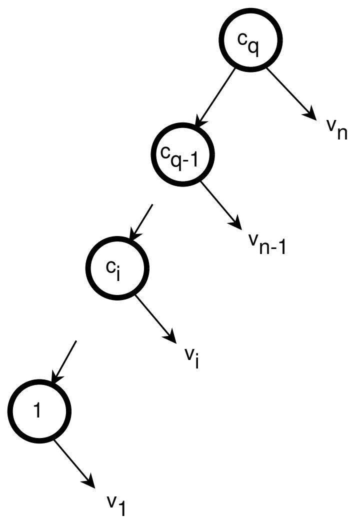


Figure 6.3: A Decision Tree Implementing a Decision List

6.2.2 Using Uncertainty Reduction to Select Tests

The main problem in learning decision trees for the binary-attribute case is selecting the order of the tests. For categorical and numeric attributes, we must also decide what the tests should be (besides selecting the order). Several techniques have been tried; the most popular one is at each stage to select that test that maximally reduces an entropy-like measure.

We show how this technique works for the simple case of tests with binary outcomes. Extension to multiple-outcome tests is straightforward computationally but gives poor results because entropy is always decreased by having more outcomes.

The *entropy* or uncertainty still remaining about the class of a pattern—knowing that it is in some set, Ξ, of patterns is defined as:

$$H(\Xi) = -\sum_i p(i|\Xi) \log_2 p(i|\Xi)$$

where $p(i|\Xi)$ is the probability that a pattern drawn at random from Ξ belongs to class i, and the summation is over all of the classes. We want to select tests at each node such that as we travel down the decision tree, the uncertainty about the class of a pattern becomes less and less.

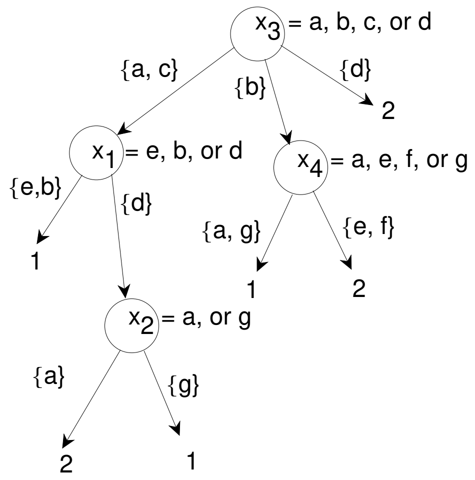


Figure 6.4: A Decision Tree with Categorical Attributes

Since we do not in general have the probabilities $p(i|\Xi)$, we estimate them by sample statistics. Although these estimates might be errorful, they are nevertheless useful in estimating uncertainties. Let $\hat{p}(i|\Xi)$ be the number of patterns in Ξ belonging to class i divided by the total number of patterns in Ξ. Then an estimate of the uncertainty is:

$$\hat{H}(\Xi) = -\sum_i \hat{p}(i|\Xi) \log_2 \hat{p}(i|\Xi)$$

For simplicity, from now on we'll drop the "hats" and use sample statistics as if they were real probabilities.

If we perform a test, T, having k possible outcomes on the patterns in Ξ, we will create k subsets, $\Xi_1, \Xi_2, \ldots, \Xi_k$. Suppose that n_i of the patterns in Ξ are in Ξ_i for $i = 1, ..., k$. (Some n_i may be 0.) If we knew that T applied to a pattern in Ξ resulted in the j-th outcome (that is, we knew that the pattern was in Ξ_j), the uncertainty about its class would be:

$$H(\Xi_j) = -\sum_i p(i|\Xi_j) \log_2 p(i|\Xi_j)$$

and the *reduction* in uncertainty (beyond knowing only that the pattern was in Ξ) would be:

$$H(\Xi) - H(\Xi_j)$$

Of course we cannot say that the test T is guaranteed always to produce that amount of reduction in uncertainty because we don't know that the result of the test will be the j-th outcome. But we can estimate the *average* uncertainty over all the Ξ_j, by:

$$E[H_T(\Xi)] = \sum_j p(\Xi_j)H(\Xi_j)$$

where by $H_T(\Xi)$ we mean the average uncertainty after performing test T on the patterns in Ξ, $p(\Xi_j)$ is the probability that the test has outcome j, and the sum is taken from 1 to k. Again, we don't know the probabilities $p(\Xi_j)$, but we can use sample values. The estimate $\hat{p}(\Xi_j)$ of $p(\Xi_j)$ is just the number of those patterns in Ξ that have outcome j divided by the total number of patterns in Ξ. The *average* reduction in uncertainty achieved by test T (applied to patterns in Ξ) is then:

$$R_T(\Xi) = H(\Xi) - E[H_T(\Xi)]$$

An important family of decision tree learning algorithms selects for the root of the tree that test that gives maximum reduction of uncertainty, and then applies this criterion recursively until some termination condition is met (which we shall discuss in more detail later). The uncertainty calculations are particularly simple when the tests have binary outcomes and when the attributes have binary values. We'll give a simple example to illustrate how the test selection mechanism works in that case.

Suppose we want to use the uncertainty-reduction method to build a decision tree to classify the following patterns:

pattern	class
(0, 0, 0)	0
(0, 0, 1)	0
(0, 1, 0)	0
(0, 1, 1)	0
(1, 0, 0)	0
(1, 0, 1)	1
(1, 1, 0)	0
(1, 1, 1)	1

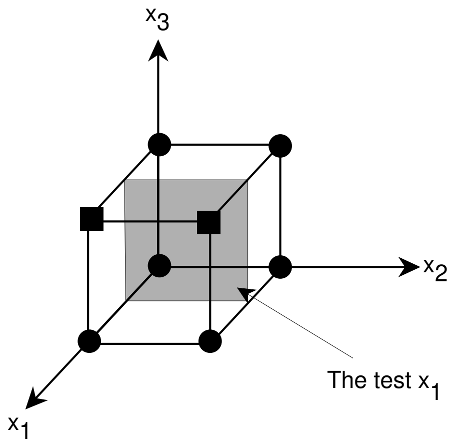


Figure 6.5: Eight Patterns to be Classified by a Decision Tree

What single test, x_1, x_2, or x_3, should be performed first? The illustration in Fig. 6.5 gives geometric intuition about the problem.

The initial uncertainty for the set, Ξ, containing all eight points is:

$$H(\Xi) = -(6/8)\log_2(6/8) - (2/8)\log_2(2/8) = 0.81$$

Next, we calculate the uncertainty reduction if we perform x_1 first. The left-hand branch has only patterns belonging to class 0 (we call them the set Ξ_l), and the right-hand-branch (Ξ_r) has two patterns in each class. So, the uncertainty of the left-hand branch is:

$$H_{x_1}(\Xi_l) = -(4/4)\log_2(4/4) - (0/4)\log_2(0/4) = 0$$

And the uncertainty of the right-hand branch is:

$$H_{x_1}(\Xi_r) = -(2/4)\log_2(2/4) - (2/4)\log_2(2/4) = 1$$

Half of the patterns "go left" and half "go right" on test x_1. Thus, the average uncertainty after performing the x_1 test is:

$$1/2 H_{x_1}(\Xi_l) + 1/2 H_{x_1}(\Xi_r) = 0.5$$

Therefore the uncertainty reduction on Ξ achieved by x_1 is:

$$R_{x_1}(\Xi) = 0.81 - 0.5 = 0.31$$

By similar calculations, we see that the test x_3 achieves exactly the same uncertainty reduction, but x_2 achieves no reduction whatsoever. Thus, our "greedy" algorithm for selecting a first test would select either x_1 or x_3. Suppose x_1 is selected. The uncertainty-reduction procedure would select x_3 as the next test. The decision tree that this procedure creates thus implements the Boolean function: $f = x_1x_3$.

See [Quinlan, 1986, sect. 4] for another example.

6.2.3 Non-Binary Attributes

If the attributes are non-binary, we can still use the uncertainty-reduction technique to select tests. But now, in addition to selecting an attribute, we must select a test on that attribute. Suppose for example that the value of an attribute is a real number and that the test to be performed is to set a threshold and to test to see if the number is greater than or less than that threshold. In principle, given a set of labeled patterns, we can measure the uncertainty reduction for each test that is achieved by every possible threshold (there are only a finite number of thresholds that give different test results if there are only a finite number of training patterns). Similarly, if an attribute is categorical (with a finite number of categories), there are only a finite number of mutually exclusive and exhaustive subsets into which the values of the attribute can be split. We can calculate the uncertainty reduction for each split.

6.3 Networks Equivalent to Decision Trees

Since univariate Boolean decision trees are implementations of DNF functions, they are also equivalent to two-layer, feedforward neural networks. We show an example in Fig. 6.6. The decision tree at the left of the figure implements the same function as the network at the right of the figure. Of course, when implemented as a network, all of the features are evaluated in parallel for any input pattern, whereas when implemented as a decision tree only those features on the branch traveled down by the input pattern need to be evaluated. The decision-tree induction methods discussed in this chapter can thus be thought of as particular ways to establish the structure and the weight values for networks.

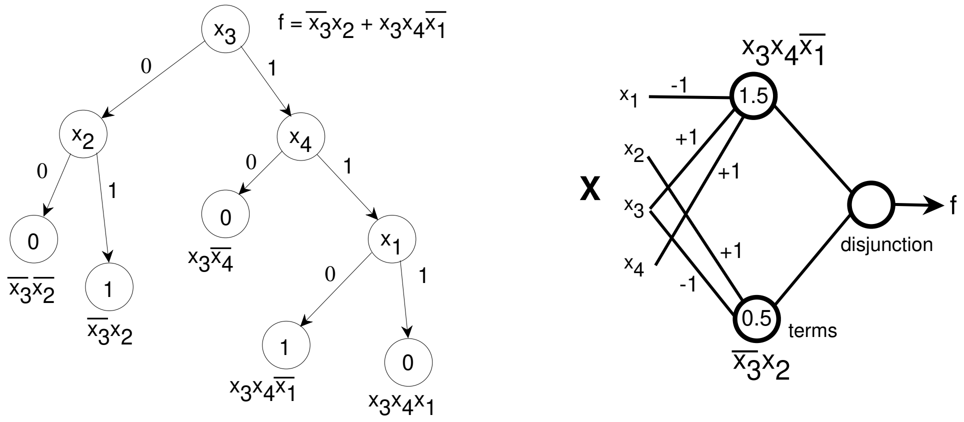


Figure 6.6: A Univariate Decision Tree and its Equivalent Network

Multivariate decision trees with linearly separable functions at each node can also be implemented by feedforward networks—in this case three-layer ones. We show an example in Fig. 6.7 in which the linearly separable functions, each implemented by a TLU, are indicated by L_1, L_2, L_3, and L_4. Again, the final layer has fixed weights, but the weights in the first two layers must be trained. Different approaches to training procedures have been discussed by [Brent, 1990], by [John, 1995], and (for a special case) by [Marchand & Golea, 1993].

6.4 Overfitting and Evaluation

6.4.1 Overfitting

In supervised learning, we must choose a function to fit the training set from among a set of hypotheses. We have already showed that generalization is impossible without bias. When we know a priori that the function we are trying to guess belongs to a small subset of all possible functions, then, even with an incomplete set of training samples, it is possible to reduce the subset of functions that are consistent with the training set sufficiently to make useful guesses about the value of the function for inputs not in the training set. And, the larger the training set, the more likely it is that even a randomly selected consistent function will have appropriate outputs for

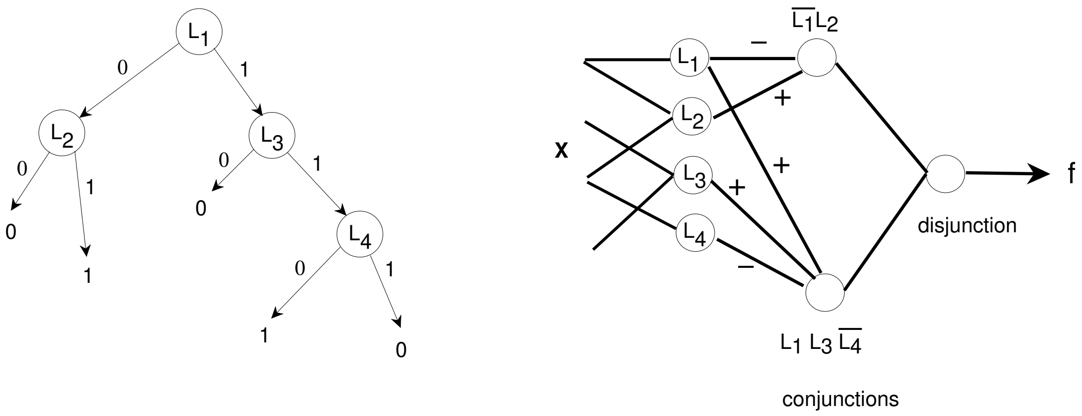


Figure 6.7: A Multivariate Decision Tree and its Equivalent Network

patterns not yet seen.

However, even with bias, if the training set is not sufficiently large compared with the size of the hypothesis space, there will still be too many consistent functions for us to make useful guesses, and generalization performance will be poor. When there are too many hypotheses that are consistent with the training set, we say that we are *overfitting* the training data. Overfitting is a problem that we must address for all learning methods.

Since a decision tree of sufficient size can implement *any* Boolean function there is a danger of overfitting—especially if the training set is small. That is, even if the decision tree is synthesized to classify all the members of the training set correctly, it might perform poorly on new patterns that were not used to build the decision tree. Several techniques have been proposed to avoid overfitting, and we shall examine some of them here. They make use of methods for estimating how well a given decision tree might generalize—methods we shall describe next.

6.4.2 Validation Methods

The most straightforward way to estimate how well a hypothesized function (such as a decision tree) performs on a test set is to test it on the test set! But, if we are comparing several learning systems (for example, if we are comparing different decision trees) so that we can select the one that performs the best on the test set, then such a comparison amounts to "training on the test data." True, training on the test data enlarges the training set, with a consequent expected improvement in generalization, but there is still the danger of overfitting if we are comparing several different learning systems. Another technique is to split the training set—using (say) two-thirds for training and the other third for estimating generalization performance. But splitting reduces the size of the training set and thereby increases the possibility of overfitting. We next describe some validation techniques that attempt to avoid these problems.

Cross-Validation

In *cross-validation*, we divide the training set Ξ into K mutually exclusive and exhaustive equal-sized subsets: $\Xi_1, \ldots, \Xi_K$. For each subset, Ξ_i, train on the union of all of the other subsets, and empirically determine the error rate, ε_i, on Ξ_i. (The error rate is the number of classification errors made on Ξ_i divided by the number of patterns in Ξ_i.) An estimate of the error rate that can be expected on new patterns of a classifier trained on *all* the patterns in Ξ is then the average of the ε_i.

Leave-one-out Validation

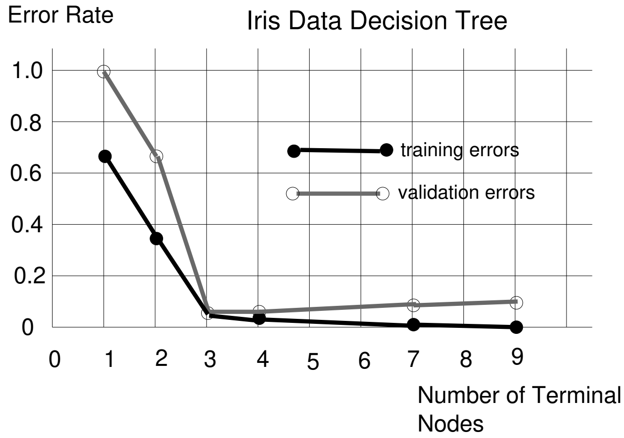
Leave-one-out validation is the same as cross validation for the special case in which K equals the number of patterns in Ξ, and each Ξ_i consists of a single pattern. When testing on each Ξ_i, we simply note whether or not a mistake was made. We count the total number of mistakes and divide by K to get the estimated error rate. This type of validation is, of course, more expensive computationally, but useful when a more accurate estimate of the error rate for a classifier is needed.

Describe "bootstrapping" also [Efron, 1982].

6.4.3 Avoiding Overfitting in Decision Trees

Near the tips of a decision tree there may be only a few patterns per node. For these nodes, we are selecting a test based on a very small sample, and thus we are likely to be overfitting. This problem can be dealt with by terminating the test-generating procedure before all patterns are perfectly split into their separate categories. That is, a leaf node may contain patterns of more than one class, but we can decide in favor of the most numerous class. This procedure will result in a few errors but often accepting a small number of errors on the training set results in fewer errors on a testing set.

This behavior is illustrated in Fig. 6.8.



(From Weiss, S., and Kulikowski, C., *Computer Systems that Learn*, Morgan Kaufmann, 1991)

Figure 6.8: Determining When Overfitting Begins

One can use cross-validation techniques to determine when to stop splitting nodes. If the cross validation error increases as a consequence of a node split, then don't split. One has to be careful about when to stop, though, because underfitting usually leads to more errors on test sets than does overfitting. There is a general rule that the lowest error-rate attainable by a sub-tree of a fully expanded tree can be no less than 1/2 of the error rate of the fully expanded tree [Weiss & Kulikowski, 1991, page 126].

Rather than stopping the growth of a decision tree, one might grow it to its full size and then prune away leaf nodes and their ancestors until cross-validation accuracy no longer increases. This technique is called *post-pruning*. Various techniques for pruning are discussed in [Weiss & Kulikowski, 1991].

6.4.4 Minimum-Description Length Methods

An important tree-growing and pruning technique is based on the *minimum-description-length (MDL)* principle. (MDL is an important idea that extends beyond decision-tree methods [Rissanen, 1978].) The idea is that the simplest decision tree that can predict the classes of the training patterns is the best one. Consider the problem of transmitting just the labels of a training set of patternsassuming that the receiver of this information already has the ordered set of patterns. If there are m patterns, each labeled by one of R classes, one could transmit a list of m R-valued numbers. Assuming equally probable classes, this transmission would require $m \log_2 R$ bits. Or, one could transmit a decision tree that correctly labelled all of the patterns. The number of bits that this transmission would require depends on the technique for encoding decision trees and on the size of the tree. If the tree is small and accurately classifies all of the patterns, it might be more economical to transmit the tree than to transmit the labels directly. In between these extremes, we might transmit a tree plus a list of labels of all the patterns that the tree misclassifies.

In general, the number of bits (or description length of the binary encoded message) is $t + d$, where t is the length of the message required to transmit the tree, and d is the length of the message required to transmit the labels of the patterns misclassified by the tree. In a sense, that tree associated with the smallest value of $t + d$ is the best or most economical tree. The MDL method is one way of adhering to the Occam's razor principle.

Quinlan and Rivest [Quinlan & Rivest, 1989] have proposed techniques for encoding decision trees and lists of exception labels and for calculating the description length $(t + d)$ of these trees and labels. They then use the description length as a measure of quality of a tree in two ways:

1. In growing a tree, they use the reduction in description length to select tests (instead of reduction in uncertainty).

2. In pruning a tree after it has been grown to zero error, they prune away those nodes (starting at the tips) that achieve a decrease in the description length.

These techniques compare favorably with the uncertainty-reduction method, although they are quite sensitive to the coding schemes used.

6.4.5 Noise in Data

Noise in the data means that one must inevitably accept some number of errors—depending on the noise level. Refusal to tolerate errors on the training set when there is noise leads to the problem of "fitting the noise." Dealing with noise, then, requires accepting some errors at the leaf nodes just as does the fact that there are a small number of patterns at leaf nodes.

6.5 The Problem of Replicated Subtrees

Decision trees are not the most economical means of implementing some Boolean functions. Consider, for example, the function $f = x_1x_2 + x_3x_4$. A decision tree for this function is shown in Fig. 6.9. Notice the replicated subtrees shown circled. The DNF-form equivalent to the function implemented by this decision tree is $f = x_1x_2 + x_1\overline{x_2}x_3x_4 + \overline{x_1}x_3x_4$. This DNF form is non-minimal (in the number of disjunctions) and is equivalent to $f = x_1x_2 + x_3x_4$.

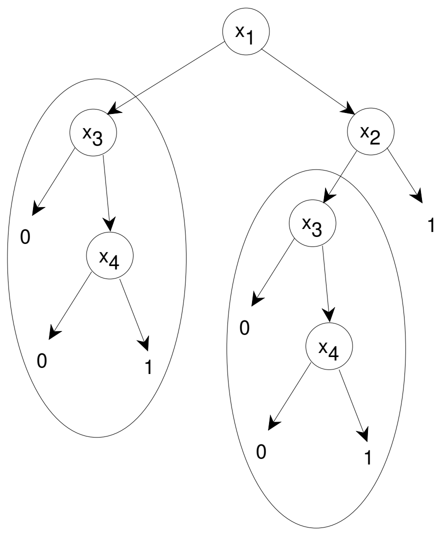


Figure 6.9: A Decision Tree with Subtree Replication

The need for replication means that it takes longer to learn the tree and that subtrees replicated further down the tree must be learned using a smaller training subset. This problem is sometimes called the *fragmentation problem*.

Several approaches might be suggested for dealing with fragmentation. One is to attempt to build a *decision graph* instead of a tree [Oliver, Dowe, & Wallace, 1992, Kohavi, 1994]. A decision graph that implements the same decisions as that of the decision tree of Fig. 6.9 is shown in Fig. 6.10.

Another approach is to use multivariate (rather than univariate tests at each node). In our example of learning $f = x_1x_2 + x_3x_4$, if we had a test

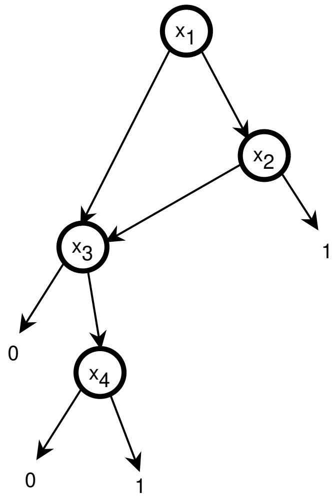


Figure 6.10: A Decision Graph

for $x_1 x_2$ and a test for $x_3 x_4$, the decision tree could be much simplified, as shown in Fig. 6.11. Several researchers have proposed techniques for learning decision trees in which the tests at each node are linearly separable functions. [John, 1995] gives a nice overview (with several citations) of learning such *linear discriminant trees* and presents a method based on "soft entropy."

A third method for dealing with the replicated subtree problem involves extracting propositional "rules" from the decision tree. The rules will have as antecedents the conjunctions that lead down to the leaf nodes, and as consequents the name of the class at the corresponding leaf node. An example rule from the tree with the repeating subtree of our example would be: $x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4 \supset 1$. Quinlan [Quinlan, 1987] discusses methods for reducing a set of rules to a simpler set by 1) eliminating from the antecedent of each rule any "unnecessary" conjuncts, and then 2) eliminating "unnecessary" rules. A conjunct or rule is determined to be unnecessary if its elimination has little effect on classification accuracy—as determined by a chi-square test, for example. After a rule set is processed, it might be the case that more than one rule is "active" for any given pattern, and care must be taken that the active rules do not conflict in their decision about the class of a pattern.

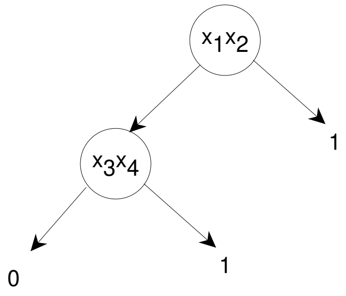


Figure 6.11: A Multivariate Decision Tree

6.6 The Problem of Missing Attributes

To be added.

6.7 Comparisons

Several experimenters have compared decision-tree, neural-net, and nearest-neighbor classifiers on a wide variety of problems. For a comparison of neural nets versus decision trees, for example, see [Dietterich, *et al.*, 1990, Shavlik, Mooney, & Towell, 1991, Quinlan, 1994]. In their *StatLog* project, [Taylor, Michie, & Spiegalhalter, 1994] give thorough comparisons of several machine learning algorithms on several different types of problems. There seems to be no single type of classifier that is best for all problems. And, there do not seem to be any general conclusions that would enable one to say which classifier method is best for which sorts of classification problems, although [Quinlan, 1994] does provide some intuition about properties of problems that might render them ill suited for decision trees, on the one hand, or backpropagation, on the other.

6.8 Bibliographical and Historical Remarks

To be added.