9 *Decision Trees*

A decision tree is a hierarchical data structure implementing the divide-and-conquer strategy. It is an efficient nonparametric method, which can be used for both classification and regression. We discuss learning algorithms that build the tree from a given labeled training sample, as well as how the tree can be converted to a set of simple rules that are easy to understand. Another possibility is to learn a rule base directly.

9.1 Introduction

IN PARAMETRIC estimation, we define a model over the whole input space and learn its parameters from all of the training data. Then we use the same model and the same parameter set for any test input. In nonparametric estimation, we divide the input space into local regions, defined by a distance measure like the Euclidean norm, and for each input, the corresponding local model computed from the training data in that region is used. In the instance-based models we discussed in chapter 8, given an input, identifying the local data defining the local model is costly; it requires calculating the distances from the given input to all of the training instances, which is $\mathcal{O}(N)$.

DECISION TREE

A *decision tree* is a hierarchical model for supervised learning whereby the local region is identified in a sequence of recursive splits in a smaller number of steps. A decision tree is composed of internal decision nodes and terminal leaves (see figure 9.1). Each *decision node m* implements a test function $f_m(\boldsymbol{x})$ with discrete outcomes labeling the branches. Given an input, at each node, a test is applied and one of the branches is taken depending on the outcome. This process starts at the root and is repeated

DECISION NODE

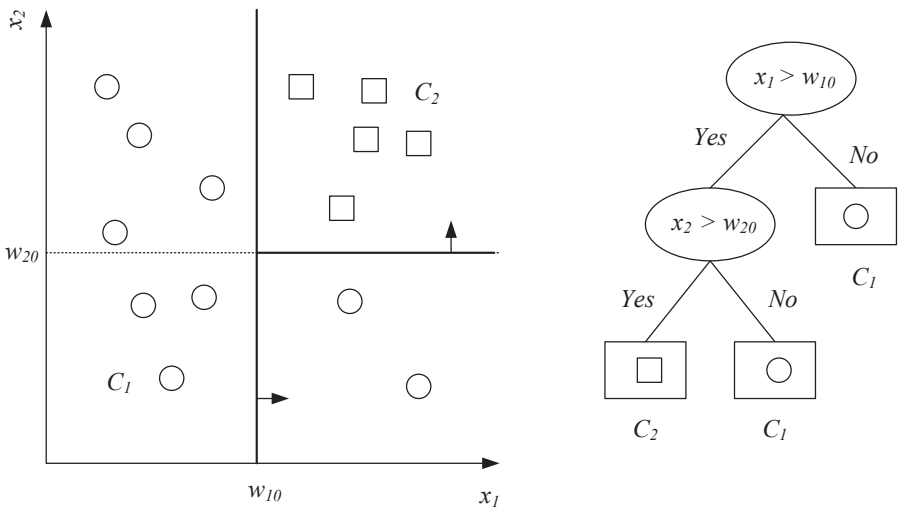

Figure 9.1 Example of a dataset and the corresponding decision tree. Oval nodes are the decision nodes and rectangles are leaf nodes. The univariate decision node splits along one axis, and successive splits are orthogonal to each other. After the first split, $\{\boldsymbol{x}|x_1 < w_{10}\}$ is pure and is not split further.

LEAF NODE

recursively until a *leaf node* is hit, at which point the value written in the leaf constitutes the output.

A decision tree is also a nonparametric model in the sense that we do not assume any parametric form for the class densities and the tree structure is not fixed a priori but the tree grows, branches and leaves are added, during learning depending on the complexity of the problem inherent in the data.

Each $f_m(\boldsymbol{x})$ defines a discriminant in the d-dimensional input space dividing it into smaller regions that are further subdivided as we take a path from the root down. $f_m(\cdot)$ is a simple function and when written down as a tree, a complex function is broken down into a series of simple decisions. Different decision tree methods assume different models for $f_m(\cdot)$, and the model class defines the shape of the discriminant and the shape of regions. Each leaf node has an output label, which in the case of classification is the class code and in regression is a numeric value. A leaf node defines a localized region in the input space where instances falling in this region have the same labels (in classification), or very similar numeric outputs (in regression). The boundaries of the

regions are defined by the discriminants that are coded in the internal nodes on the path from the root to the leaf node.

The hierarchical placement of decisions allows a fast localization of the region covering an input. For example, if the decisions are binary, then in the best case, each decision eliminates half of the cases. If there are b regions, then in the best case, the correct region can be found in $\log_2 b$ decisions. Another advantage of the decision tree is interpretability. As we will see shortly, the tree can be converted to a set of *IF-THEN rules* that are easily understandable. For this reason, decision trees are very popular and sometimes preferred over more accurate but less interpretable methods.

We start with univariate trees where the test in a decision node uses only one input variable and we see how such trees can be constructed for classification and regression. We later generalize this to multivariate trees where all inputs can be used in an internal node.

9.2 Univariate Trees

UNIVARIATE TREE

In a *univariate tree*, in each internal node, the test uses only one of the input dimensions. If the used input dimension, x_j, is discrete, taking one of n possible values, the decision node checks the value of x_j and takes the corresponding branch, implementing an n-way split. For example, if an attribute is color $\in$ {red, blue, green}, then a node on that attribute has three branches, each one corresponding to one of the three possible values of the attribute.

A decision node has discrete branches and a numeric input should be discretized. If x_j is numeric (ordered), the test is a comparison

$$f_m(\boldsymbol{x}) : x_j > w_{m0} \tag{9.1}$$

where w_{m0} is a suitably chosen threshold value. The decision node divides the input space into two: $L_m = \{\boldsymbol{x}|x_j > w_{m0}\}$ and $R_m = \{\boldsymbol{x}|x_j \leq w_{m0}\}$; this is called a *binary split*. Successive decision nodes on a path from the root to a leaf further divide these into two using other attributes and generating splits orthogonal to each other. The leaf nodes define hyperrectangles in the input space (see figure 9.1).

BINARY SPLIT

Tree induction is the construction of the tree given a training sample. For a given training set, there exists many trees that code it with no error, and, for simplicity, we are interested in finding the smallest among

them, where tree size is measured as the number of nodes in the tree and the complexity of the decision nodes. Finding the smallest tree is NP-complete (Quinlan 1986), and we are forced to use local search procedures based on heuristics that give reasonable trees in reasonable time.

Tree learning algorithms are greedy and, at each step, starting at the root with the complete training data, we look for the best split. This splits the training data into two or n, depending on whether the chosen attribute is numeric or discrete. We then continue splitting recursively with the corresponding subset until we do not need to split anymore, at which point a leaf node is created and labeled.

9.2.1 Classification Trees

CLASSIFICATION TREE
IMPURITY MEASURE

In the case of a decision tree for classification, namely, a *classification tree*, the goodness of a split is quantified by an *impurity measure*. A split is pure if after the split, for all branches, all the instances choosing a branch belong to the same class. Let us say for node m, N_m is the number of training instances reaching node m. For the root node, it is N. N_m^i of N_m belong to class C_i, with $\sum_i N_m^i = N_m$. Given that an instance reaches node m, the estimate for the probability of class C_i is

$$\hat{P}(C_i|\boldsymbol{x}, m) \equiv p_m^i = \frac{N_m^i}{N_m} \tag{9.2}$$

Node m is pure if p_m^i for all i are either 0 or 1. It is 0 when none of the instances reaching node m are of class C_i, and it is 1 if all such instances are of C_i. If the split is pure, we do not need to split any further and can add a leaf node labeled with the class for which p_m^i is 1. One possible function to measure impurity is *entropy* (Quinlan 1986) (see figure 9.2):

ENTROPY

$$\mathcal{I}_m = -\sum_{i=1}^{K} p_m^i \log_2 p_m^i \tag{9.3}$$

where $0 \log 0 \equiv 0$. Entropy in information theory specifies the minimum number of bits needed to encode the class code of an instance. In a two-class problem, if $p^1 = 1$ and $p^2 = 0$, all examples are of C^1, and we do not need to send anything, and the entropy is 0. If $p^1 = p^2 = 0.5$, we need to send a bit to signal one of the two cases, and the entropy is 1. In between these two extremes, we can devise codes and use less than a bit per message by having shorter codes for the more likely class and

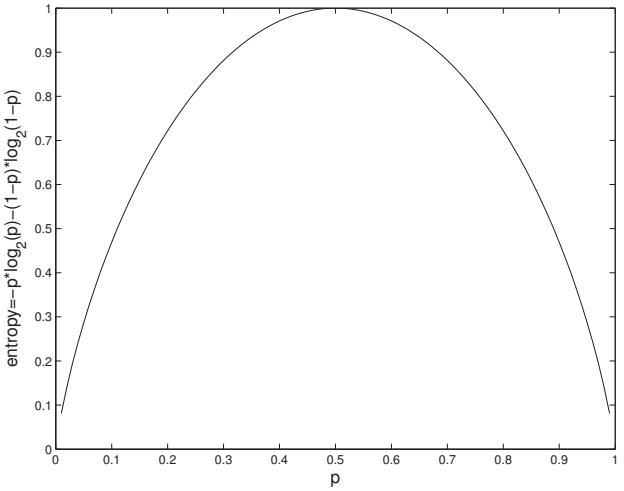

Figure 9.2 Entropy function for a two-class problem.

longer codes for the less likely. When there are $K > 2$ classes, the same discussion holds and the largest entropy is $\log_2 K$ when $p^i = 1/K$.

But entropy is not the only possible measure. For a two-class problem where $p^1 \equiv p$ and $p^2 = 1 - p$, $\phi(p, 1-p)$ is a nonnegative function measuring the impurity of a split if it satisfies the following properties (Devroye, Györfi, and Lugosi 1996):

- $\phi(1/2, 1/2) \geq \phi(p, 1-p)$, for any $p \in [0, 1]$.
- $\phi(0, 1) = \phi(1, 0) = 0$.
- $\phi(p, 1-p)$ is increasing in p on $[0, 1/2]$ and decreasing in p on $[1/2, 1]$.

Examples are

1. Entropy

$$\phi(p, 1-p) = -p \log_2 p - (1-p) \log_2 (1-p) \tag{9.4}$$

Equation 9.3 is the generalization to $K > 2$ classes.

GINI INDEX 2. *Gini index* (Breiman et al. 1984)

$$\phi(p, 1-p) = 2p(1-p) \tag{9.5}$$

3. Misclassification error

$$\phi(p, 1-p) = 1 - \max(p, 1-p) \tag{9.6}$$

These can be generalized to $K > 2$ classes, and the misclassification error can be generalized to minimum risk given a loss function (exercise 1). Research has shown that there is not a significant difference between these three measures.

If node m is not pure, then the instances should be split to decrease impurity, and there are multiple possible attributes on which we can split. For a numeric attribute, multiple split positions are possible. Among all, we look for the split that minimizes impurity after the split because we want to generate the smallest tree. If the subsets after the split are closer to pure, fewer splits (if any) will be needed afterward. Of course this is locally optimal, and we have no guarantee of finding the smallest decision tree.

Let us say at node m, N_{mj} of N_m take branch j; these are $\boldsymbol{x}^t$ for which the test $f_m(\boldsymbol{x}^t)$ returns outcome j. For a discrete attribute with n values, there are n outcomes, and for a numeric attribute, there are two outcomes ($n = 2$), in either case satisfying $\sum_{j=1}^{n} N_{mj} = N_m$. N_{mj}^i of N_{mj} belong to class C_i: $\sum_{i=1}^{K} N_{mj}^i = N_{mj}$. Similarly, $\sum_{j=1}^{n} N_{mj}^i = N_m^i$.

Then given that at node m, the test returns outcome j, the estimate for the probability of class C_i is

$$\hat{P}(C_i|\boldsymbol{x}, m, j) \equiv p_{mj}^i = \frac{N_{mj}^i}{N_{mj}} \tag{9.7}$$

and the total impurity after the split is given as

$$\mathcal{I}'_m = -\sum_{j=1}^{n} \frac{N_{mj}}{N_m} \sum_{i=1}^{K} p_{mj}^i \log_2 p_{mj}^i \tag{9.8}$$

In the case of a numeric attribute, to be able to calculate p_{mj}^i using equation 9.1, we also need to know w_{m0} for that node. There are $N_m - 1$ possible w_{m0} between N_m data points: We do not need to test for all (possibly infinite) points; it is enough to test, for example, at halfway between points. Note also that the best split is always between adjacent points belonging to different classes. So we try them, and the best in terms of purity is taken for the purity of the attribute. In the case of a discrete attribute, no such iteration is necessary.

```
GenerateTree(X)
    If NodeEntropy(X)< θ_I /* equation 9.3 */
      Create leaf labelled by majority class in X
      Return
    i ← SplitAttribute(X)
    For each branch of x_i
      Find X_i falling in branch
      GenerateTree(X_i)

SplitAttribute(X)
    MinEnt← MAX
    For all attributes i = 1,...,d
        If x_i is discrete with n values
          Split X into X_1,...,X_n by x_i
          e ← SplitEntropy(X_1,...,X_n) /* equation 9.8 */
          If e<MinEnt MinEnt ← e; bestf ← i
        Else /* x_i is numeric */
          For all possible splits
              Split X into X_1, X_2 on x_i
              e←SplitEntropy(X_1, X_2)
              If e<MinEnt MinEnt ← e; bestf ← i
    Return bestf
```

Figure 9.3 Classification tree construction.

So for all attributes, discrete and numeric, and for a numeric attribute for all split positions, we calculate the impurity and choose the one that has the minimum entropy, for example, as measured by equation 9.8. Then tree construction continues recursively and in parallel for all the branches that are not pure, until all are pure. This is the basis of the *classification and regression tree* (CART) algorithm (Breiman et al. 1984), *ID3* algorithm (Quinlan 1986), and its extension *C4.5* (Quinlan 1993). The pseudocode of the algorithm is given in figure 9.3.

CLASSIFICATION AND REGRESSION TREE
ID3
C4.5

It can also be said that at each step during tree construction, we choose the split that causes the largest decrease in impurity, which is the difference between the impurity of data reaching node m (equation 9.3) and the total entropy of data reaching its branches after the split (equation 9.8).

One problem is that such splitting favors attributes with many values. When there are many values, there are many branches, and the impurity can be much less. For example, if we take training index t as an attribute, the impurity measure will choose that because then the impurity of each branch is 0, although it is not a reasonable feature. Nodes with many branches are complex and go against our idea of splitting class discriminants into simple decisions. Methods have been proposed to penalize such attributes and to balance the impurity drop and the branching factor.

When there is noise, growing the tree until it is purest, we may grow a very large tree and it overfits; for example, consider the case of a mislabeled instance amid a group of correctly labeled instances. To alleviate such overfitting, tree construction ends when nodes become pure enough, namely, a subset of data is not split further if $\mathcal{I} < \theta_I$. This implies that we do not require that p^i_{mj} be exactly 0 or 1 but close enough, with a threshold θ_p. In such a case, a leaf node is created and is labeled with the class having the highest p^i_{mj}.

θ_I (or θ_p) is the complexity parameter, like h or k of nonparametric estimation. When they are small, the variance is high and the tree grows large to reflect the training set accurately, and when they are large, variance is lower and a smaller tree roughly represents the training set and may have large bias. The ideal value depends on the cost of misclassification, as well as the costs of memory and computation.

It is generally advised that in a leaf, one stores the posterior probabilities of classes, instead of labeling the leaf with the class having the highest posterior. These probabilities may be required in later steps, for example, in calculating risks. Note that we do not need to store the instances reaching the node or the exact counts; just ratios suffice.

9.2.2 Regression Trees

REGRESSION TREE

A *regression tree* is constructed in almost the same manner as a classification tree, except that the impurity measure that is appropriate for classification is replaced by a measure appropriate for regression. Let us say for node m, $\mathcal{X}_m$ is the subset of $\mathcal{X}$ reaching node m; namely, it is the set of all $\boldsymbol{x} \in \mathcal{X}$ satisfying all the conditions in the decision nodes on the path from the root until node m. We define

$$b_m(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \mathcal{X}_m\text{: } \boldsymbol{x} \text{ reaches node } m \\ 0 & \text{otherwise} \end{cases} \tag{9.9}$$

In regression, the goodness of a split is measured by the mean square error from the estimated value. Let us say g_m is the estimated value in node m.

$$E_m = \frac{1}{N_m} \sum_t (r^t - g_m)^2 b_m(\mathbf{x}^t) \tag{9.10}$$

where $N_m = |\mathcal{X}_m| = \sum_t b_m(\mathbf{x}^t)$.

In a node, we use the mean (median if there is too much noise) of the required outputs of instances reaching the node

$$g_m = \frac{\sum_t b_m(\mathbf{x}^t) r^t}{\sum_t b_m(\mathbf{x}^t)} \tag{9.11}$$

Then equation 9.10 corresponds to the variance at m. If at a node, the error is acceptable, that is, $E_m < \theta_r$, then a leaf node is created and it stores the g_m value. Just like the regressogram of chapter 8, this creates a piecewise constant approximation with discontinuities at leaf boundaries.

If the error is not acceptable, data reaching node m is split further such that the sum of the errors in the branches is minimum. As in classification, at each node, we look for the attribute (and split threshold for a numeric attribute) that minimizes the error, and then we continue recursively.

Let us define $\mathcal{X}_{mj}$ as the subset of $\mathcal{X}_m$ taking branch j: $\cup_{j=1}^n \mathcal{X}_{mj} = \mathcal{X}_m$. We define

$$b_{mj}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \mathcal{X}_{mj}\text{: } \mathbf{x} \text{ reaches node } m \text{ and takes branch } j \\ 0 & \text{otherwise} \end{cases} \tag{9.12}$$

g_{mj} is the estimated value in branch j of node m.

$$g_{mj} = \frac{\sum_t b_{mj}(\mathbf{x}^t) r^t}{\sum_t b_{mj}(\mathbf{x}^t)} \tag{9.13}$$

and the error after the split is

$$E'_m = \frac{1}{N_m} \sum_j \sum_t (r^t - g_{mj})^2 b_{mj}(\mathbf{x}^t) \tag{9.14}$$

The drop in error for any split is given as the difference between equation 9.10 and equation 9.14. We look for the split such that this drop is maximum or, equivalently, where equation 9.14 takes its minimum. The code given in figure 9.3 can be adapted to training a regression tree by

replacing entropy calculations with mean square error and class labels with averages.

Mean square error is one possible error function; another is worst possible error

$$E_m = \max_j \max_t |r^t - g_{mj}| b_{mj}(\boldsymbol{x}^t) \tag{9.15}$$

and using this, we can guarantee that the error for any instance is never larger than a given threshold.

The acceptable error threshold is the complexity parameter; when it is small, we generate large trees and risk overfitting; when it is large, we underfit and smooth too much (see figures 9.4 and 9.5).

Similar to going from running mean to running line in nonparametric regression, instead of taking an average at a leaf that implements a constant fit, we can also do a linear regression fit over the instances choosing the leaf:

$$g_m(\boldsymbol{x}) = \boldsymbol{w}_m^T \boldsymbol{x} + w_{m0} \tag{9.16}$$

This makes the estimate in a leaf dependent on $\boldsymbol{x}$ and generates smaller trees, but there is the expense of extra computation at a leaf node.

9.3 Pruning

Frequently, a node is not split further if the number of training instances reaching a node is smaller than a certain percentage of the training set—for example, 5 percent—regardless of the impurity or error. The idea is that any decision based on too few instances causes variance and thus generalization error. Stopping tree construction early on before it is full is called *prepruning* the tree.

PREPRUNING

POSTPRUNING

Another possibility to get simpler trees is *postpruning*, which in practice works better than prepruning. We saw before that tree growing is greedy and at each step, we make a decision, namely, generate a decision node, and continue further on, never backtracking and trying out an alternative. The only exception is postpruning where we try to find and prune unnecessary subtrees.

In postpruning, we grow the tree full until all leaves are pure and we have no training error. We then find subtrees that cause overfitting and we prune them. From the initial labeled set, we set aside a *pruning set*, unused during training. For each subtree, we replace it with a leaf node

PRUNING SET

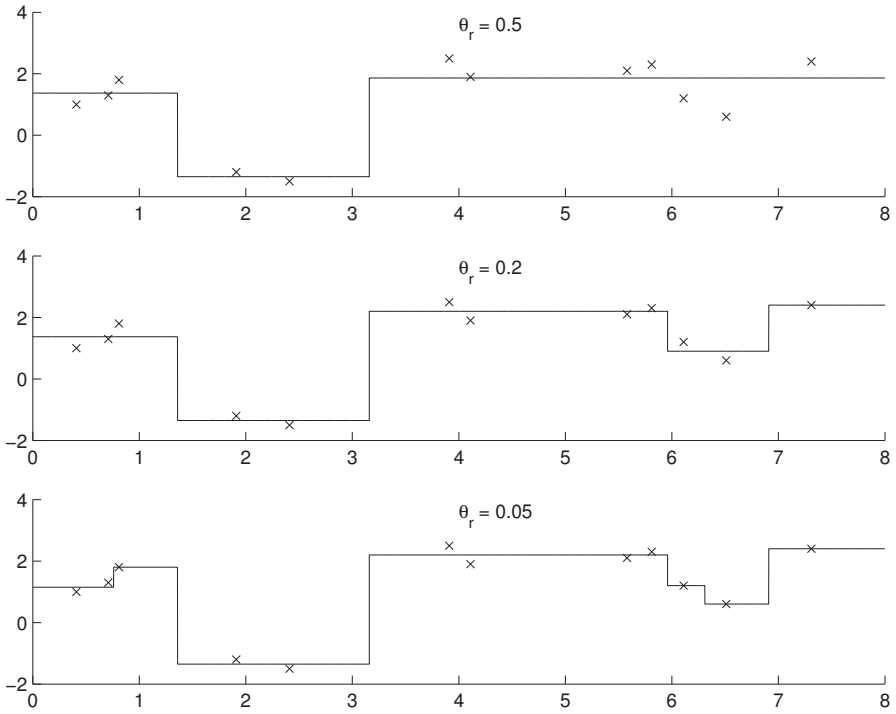

Figure 9.4 Regression tree smooths for various values of θ_r. The corresponding trees are given in figure 9.5.

labeled with the training instances covered by the subtree (appropriately for classification or regression). If the leaf node does not perform worse than the subtree on the pruning set, we prune the subtree and keep the leaf node because the additional complexity of the subtree is not justified; otherwise, we keep the subtree.

For example, in the third tree of figure 9.5, there is a subtree starting with condition $x < 6.31$. This subtree can be replaced by a leaf node of $y = 0.9$ (as in the second tree) if the error on the pruning set does not increase during the substitution. Note that the pruning set should not be confused with (and is distinct from) the validation set.

Comparing prepruning and postpruning, we can say that prepruning is faster but postpruning generally leads to more accurate trees.

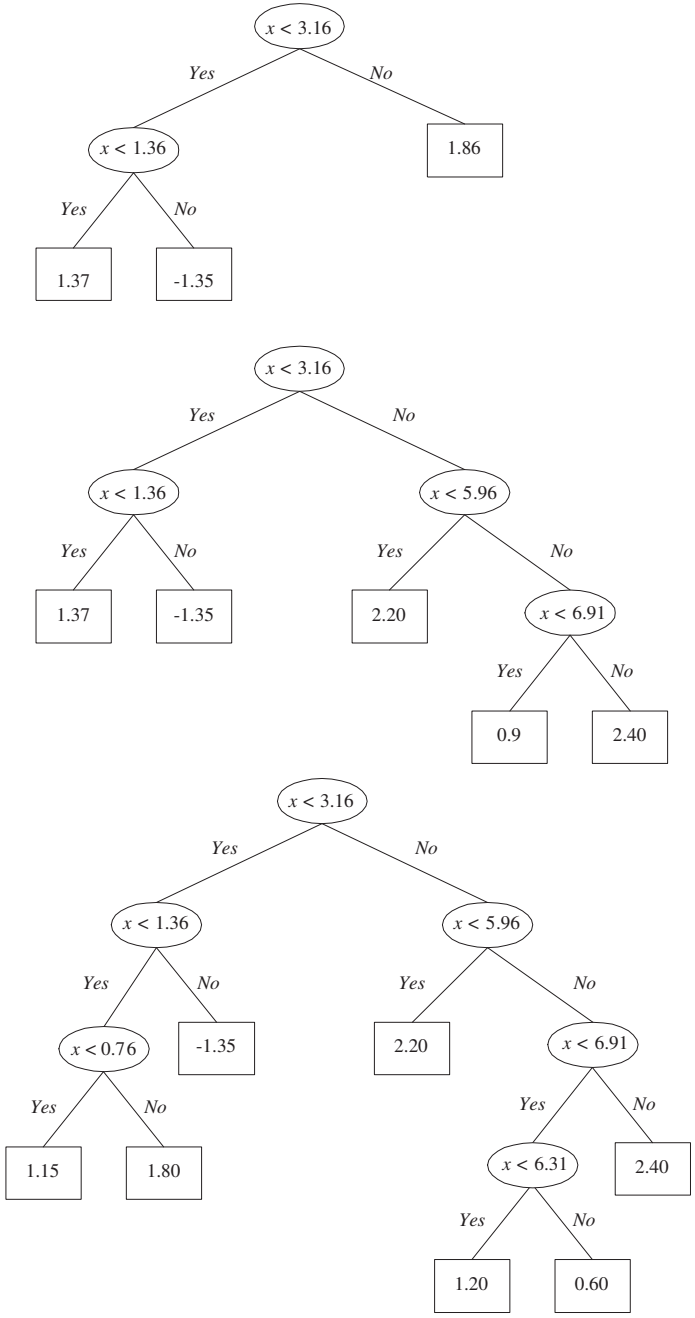

Figure 9.5 Regression trees implementing the smooths of figure 9.4 for various values of θ_r.

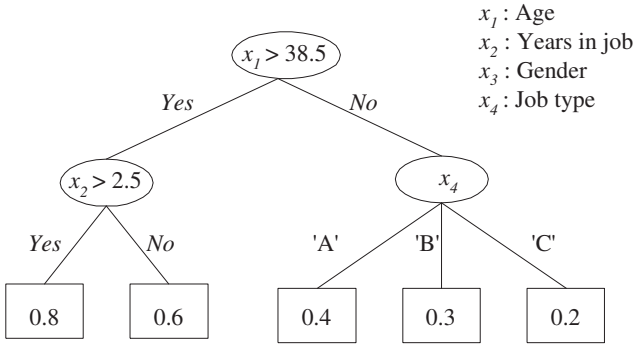

Figure 9.6 Example of a (hypothetical) decision tree. Each path from the root to a leaf can be written down as a conjunctive rule, composed of conditions defined by the decision nodes on the path.

9.4 Rule Extraction from Trees

A decision tree does its own feature extraction. The univariate tree only uses the necessary variables, and after the tree is built, certain features may not be used at all. We can also say that features closer to the root are more important globally. For example, the decision tree given in figure 9.6 uses x_1, x_2, and x_4, but not x_3. It is possible to use a decision tree for feature extraction: we build a tree and then take only those features used by the tree as inputs to another learning method.

INTERPRETABILITY

Another main advantage of decision trees is *interpretability*: The decision nodes carry conditions that are simple to understand. Each path from the root to a leaf corresponds to one conjunction of tests, as all those conditions should be satisfied to reach to the leaf. These paths together can be written down as a set of *IF-THEN rules*, called a *rule base*. One such method is *C4.5Rules* (Quinlan 1993).

IF-THEN RULES

For example, the decision tree of figure 9.6 can be written down as the following set of rules:

R1: IF (age $>$ 38.5) AND (years-in-job $>$ 2.5) THEN $y = 0.8$
R2: IF (age $>$ 38.5) AND (years-in-job $\leq$ 2.5) THEN $y = 0.6$
R3: IF (age $\leq$ 38.5) AND (job-type = 'A') THEN $y = 0.4$
R4: IF (age $\leq$ 38.5) AND (job-type = 'B') THEN $y = 0.3$
R5: IF (age $\leq$ 38.5) AND (job-type = 'C') THEN $y = 0.2$