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

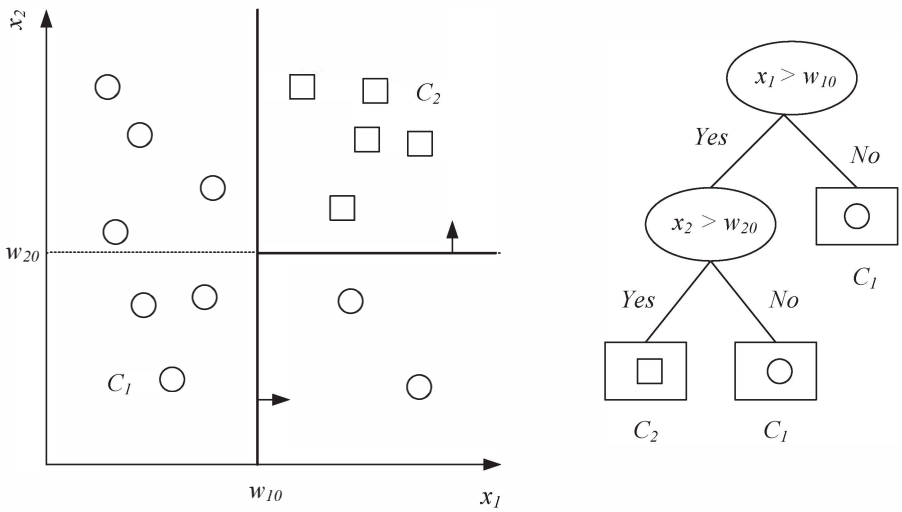

Figure 9.1 Example of a dataset and the corresponding decision tree. Oval nodes are the decision nodes and rectangles are leaf nodes. The univariate decision node splits along one axis, and successive splits are orthogonal to each other. After the first split, $\{\boldsymbol{x}|x_1 < w_{10}\}$ is pure 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

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

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

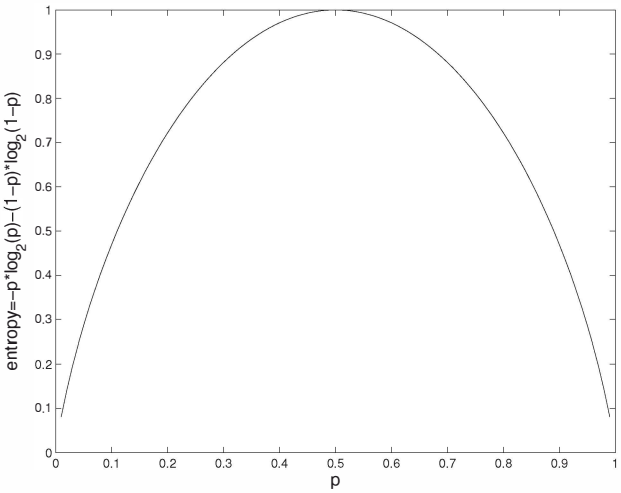

Figure 9.2 Entropy function for a two-class problem.

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 $\mathbf{x}^t$ for which the test $f_m(\mathbf{x}^t)$ 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

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

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

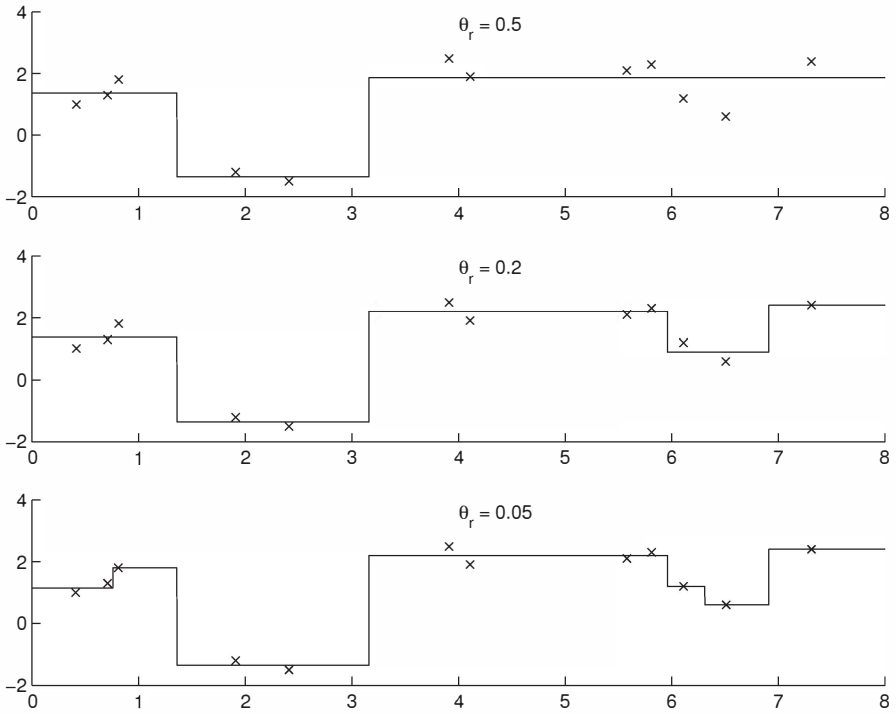

Figure 9.4 Regression tree smooths for various values of θ_r. The corresponding trees are given in figure 9.5.

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.

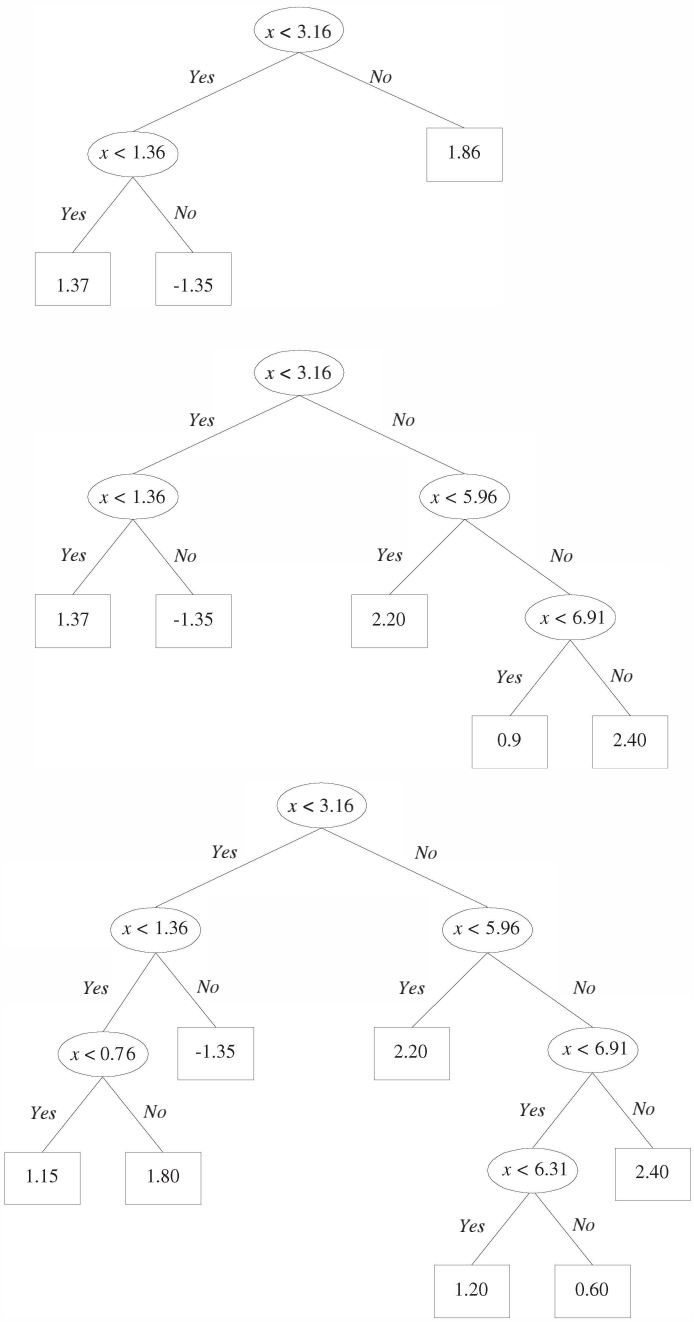

Figure 9.5 Regression trees implementing the smooths of figure 9.4 for various values of θ_r.

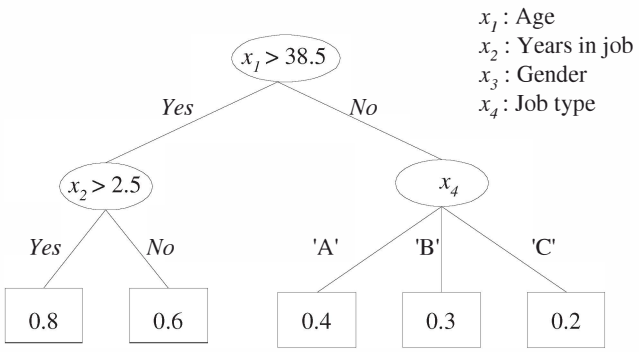

Figure 9.6 Example of a (hypothetical) decision tree. Each path from the root to a leaf can be written down as a conjunctive rule, composed of conditions defined by the decision nodes on the path.

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 $y = 0.2$

KNOWLEDGE EXTRACTION

Such a rule base allows *knowledge extraction*; it can be easily understood and allows experts to verify the model learned from data. For each rule, one can also calculate the percentage of training data covered by the rule, namely, *rule support*. The rules reflect the main characteristics of the dataset: They show the important features and split positions. For instance, in this (hypothetical) example, we see that in terms of our purpose (y), people who are thirty-eight years old or less are different from people who are thirty-nine or more years old. And among this latter group, it is the job type that makes them different, whereas in the former group, it is the number of years in a job that is the best discriminating characteristic.

RULE SUPPORT

In the case of a classification tree, there may be more than one leaf labeled with the same class. In such a case, these multiple conjunctive expressions corresponding to different paths can be combined as a disjunction (OR). The class region then corresponds to a union of these multiple patches, each patch corresponding to the region defined by one leaf. For example, class C_1 of figure 9.1 is written as

$$\text{IF } (x \leq w_{10}) \text{ OR } ((x_1 > w_{10}) \text{ AND } (x_2 \leq w_{20})) \text{ THEN } C_1$$

PRUNING RULES

Pruning rules is possible for simplification. Pruning a subtree corresponds to pruning terms from a number of rules at the same time. It may be possible to prune a term from one rule without touching other rules. For example, in the previous rule set, for R3, if we see that all whose `job-type='A'` have outcomes close to 0.4, regardless of `age`, $R3$ can be pruned as

$$R3' : \text{IF (job-type='A') THEN } y = 0.4$$

Note that after the rules are pruned, it may not be possible to write them back as a tree anymore.

9.5 Learning Rules from Data

RULE INDUCTION

As we have just seen, one way to get IF-THEN rules is to train a decision tree and convert it to rules. Another is to learn the rules directly. *Rule induction* works similar to tree induction except that rule induction does a depth-first search and generates one path (rule) at a time, whereas tree induction goes breadth-first and generates all paths simultaneously.

Rules are learned one at a time. Each rule is a conjunction of conditions on discrete or numeric attributes (as in decision trees) and these

conditions are added one at a time, to optimize some criterion, for example, minimize entropy. A rule is said to *cover* an example if the example satisfies all the conditions of the rule. Once a rule is grown and pruned, it is added to the rule base and all the training examples covered by the rule are removed from the training set, and the process continues until enough rules are added. This is called *sequential covering*. There is an outer loop of adding one rule at a time to the rule base and an inner loop of adding one condition at a time to the current rule. These steps are both greedy and do not guarantee optimality. Both loops have a pruning step for better generalization.

SEQUENTIAL COVERING

RIPPER

IREP

One example of a rule induction algorithm is *Ripper* (Cohen 1995), based on an earlier algorithm *Irep* (Fürnkranz and Widmer 1994). We start with the case of two classes where we talk of positive and negative examples, then later generalize to $K > 2$ classes. Rules are added to explain positive examples such that if an instance is not covered by any rule, then it is classified as negative. So a rule when it matches is either correct (true positive), or it causes a false positive. The pseudocode of the outer loop of Ripper is given in figure 9.7.

FOIL

In Ripper, conditions are added to the rule to maximize an information gain measure used in Quinlan's (1990) *Foil* algorithm. Let us say we have rule R and R' is the candidate rule after adding a condition. Change in gain is defined as

$$Gain(R', R) = s \cdot \left(\log_2 \frac{N'_+}{N'} - \log_2 \frac{N_+}{N} \right) \tag{9.17}$$

where N is the number of instances that are covered by R and N_+ is the number of true positives in them. N' and N'_+ are similarly defined for R'. s is the number of true positives in R, which are still true positives in R', after adding the condition. In terms of information theory, the change in gain measures the reduction in bits to encode a positive instance.

Conditions are added to a rule until it covers no negative example. Once a rule is grown, it is pruned back by deleting conditions in reverse order, to find the rule that maximizes the *rule value metric*

RULE VALUE METRIC

$$rvm(R) = \frac{p - n}{p + n} \tag{9.18}$$

where p and n are the number of true and false positives, respectively, on the pruning set, which is one-third of the data, having used two-thirds as the growing set.

```
Ripper(Pos,Neg,k)
    RuleSet ← LearnRuleSet(Pos,Neg)
    For k times
        RuleSet ← OptimizeRuleSet(RuleSet,Pos,Neg)
LearnRuleSet(Pos,Neg)
    RuleSet ← ∅
    DL ← DescLen(RuleSet,Pos,Neg)
    Repeat
        Rule ← LearnRule(Pos,Neg)
        Add Rule to RuleSet
        DL' ← DescLen(RuleSet,Pos,Neg)
        If DL'>DL+64
          PruneRuleSet(RuleSet,Pos,Neg)
          Return RuleSet
        If DL'<DL DL ← DL'
          Delete instances covered by Rule from Pos and Neg
    Until Pos = ∅
    Return RuleSet
PruneRuleSet(RuleSet,Pos,Neg)
    For each Rule ∈ RuleSet in reverse order
        DL ← DescLen(RuleSet,Pos,Neg)
        DL' ← DescLen(RuleSet-Rule,Pos,Neg)
        IF DL'<DL Delete Rule from RuleSet
    Return RuleSet
OptimizeRuleSet(RuleSet,Pos,Neg)
    For each Rule ∈ RuleSet
        DL0 ← DescLen(RuleSet,Pos,Neg)
        DL1 ← DescLen(RuleSet-Rule+
                ReplaceRule(RuleSet,Pos,Neg),Pos,Neg)
        DL2 ← DescLen(RuleSet-Rule+
                ReviseRule(RuleSet,Rule,Pos,Neg),Pos,Neg)
        If DL1=min(DL0,DL1,DL2)
                Delete Rule from RuleSet and
                        add ReplaceRule(RuleSet,Pos,Neg)
        Else If DL2=min(DL0,DL1,DL2)
                Delete Rule from RuleSet and
                        add ReviseRule(RuleSet,Rule,Pos,Neg)
    Return RuleSet
```

Figure 9.7 Ripper algorithm for learning rules. Only the outer loop is given; the inner loop is similar to adding nodes in a decision tree.

Once a rule is grown and pruned, all positive and negative training examples covered by the rule are removed from the training set. If there are remaining positive examples, rule induction continues. In the case of noise, we may stop early, namely, when a rule does not explain enough number of examples. To measure the worth of a rule, minimum description length (section 4.8) is used (Quinlan 1995). Typically, we stop if the description of the rule is not shorter than the description of instances it explains. The description length of a rule base is the sum of the description lengths of all the rules in the rule base, plus the description of instances not covered by the rule base. Ripper stops adding rules when the description length of the rule base is more than 64 bits larger than the best description length so far. Once the rule base is learned, we pass over the rules in reverse order to see if they can be removed without increasing the description length.

Rules in the rule base are also optimized after they are learned. Ripper considers two alternatives to a rule: One, called the replacement rule, starts from an empty rule, is grown, and is then pruned. The second, called the revision rule, starts with the rule as it is, is grown, and is then pruned. These two are compared with the original rule, and the shortest of three is added to the rule base. This optimization of the rule base can be done k times, typically twice.

When there are $K > 2$ classes, they are ordered in terms of their prior probabilities such that C_1 has the lowest prior probability and C_K has the highest. Then a sequence of two-class problems are defined such that, first, instances belonging to C_1 are taken as positive examples and instances of all other classes are taken as negative examples. Then, having learned C_1 and all its instances removed, it learns to separate C_2 from $C_3, \ldots, C_K$. This process is repeated until only C_K remains. The empty default rule is then labeled C_K, so that if an instance is not covered by any rule, it will be assigned to C_K.

For a training set of size N, Ripper's complexity is $\mathcal{O}(N \log^2 N)$ and is an algorithm that can be used on very large training sets (Dietterich 1997). The rules we learn are *propositional rules*. More expressive, *first-order rules* have variables in conditions, called *predicates*. A *predicate* is a function that returns true or false depending on the value of its argument. Predicates therefore allow defining relations between the values of attributes, which cannot be done by propositions (Mitchell 1997):

PROPOSITIONAL RULES
FIRST-ORDER RULES

IF Father(y, x) AND Female(y) THEN Daughter(x, y)

INDUCTIVE LOGIC PROGRAMMING

Such rules can be seen as programs in a logic programming language, such as Prolog, and learning them from data is called *inductive logic programming*. One such algorithm is Foil (Quinlan 1990).

BINDING

Assigning a value to a variable is called *binding*. A rule matches if there is a set of bindings to the variables existing in the training set. Learning first-order rules is similar to learning propositional rules with an outer loop of adding rules, and an inner loop of adding conditions to a rule, with prunings at the end of each loop. The difference is in the inner loop, where at each step we consider one predicate to add (instead of a proposition) and check the increase in the performance of the rule (Mitchell 1997). To calculate the performance of a rule, we consider all possible bindings of the variables, count the number of positive and negative bindings in the training set, and use, for example, equation 9.17. In this first-order case, we have predicates instead of propositions, so they should be previously defined, and the training set is a set of predicates known to be true.

9.6 Multivariate Trees

MULTIVARIATE TREE

In the case of a univariate tree, only one input dimension is used at a split. In a *multivariate tree*, at a decision node, all input dimensions can be used and thus it is more general. When all inputs are numeric, a binary linear multivariate node is defined as

$$f_m(\mathbf{x}) : \mathbf{w}_m^T \mathbf{x} + w_{m0} > 0 \tag{9.19}$$

Because the linear multivariate node takes a weighted sum, discrete attributes should be represented by 0/1 dummy numeric variables. Equation 9.19 defines a hyperplane with arbitrary orientation (see figure 9.8). Successive nodes on a path from the root to a leaf further divide these, and leaf nodes define polyhedra in the input space. The univariate node with a numeric feature is a special case when all but one of w_{mj} are 0. Thus the univariate numeric node of equation 9.1 also defines a linear discriminant but one that is orthogonal to axis x_j, intersecting it at w_{m0} and parallel to all other x_i. We therefore see that in a univariate node there are d possible orientations ($\mathbf{w}_m$) and $N_m - 1$ possible thresholds ($-w_{m0}$), making an exhaustive search possible. In a multivariate node, there are $2^d \begin{pmatrix} N_m \\ d \end{pmatrix}$ possible hyperplanes (Murthy, Kasif, and Salzberg 1994) and an exhaustive search is no longer practical.

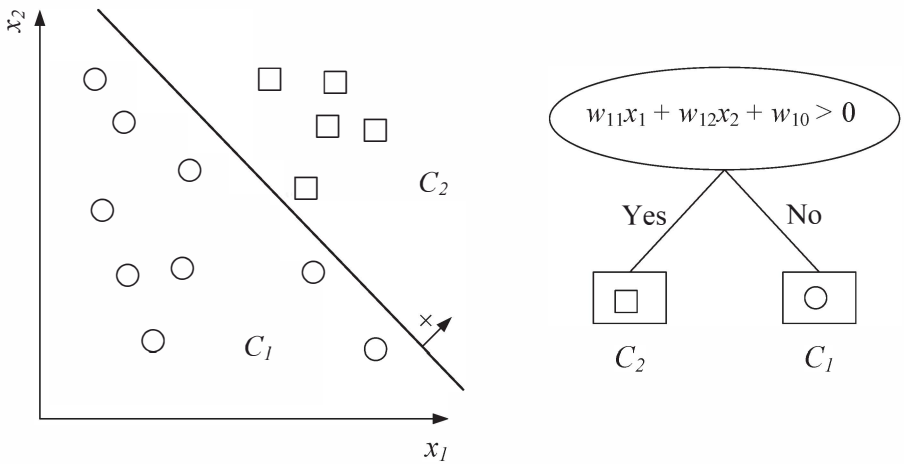

Figure 9.8 Example of a linear multivariate decision tree. The linear multivariate node can place an arbitrary hyperplane and thus is more general, whereas the univariate node is restricted to axis-aligned splits.

When we go from a univariate node to a linear multivariate node, the node becomes more flexible. It is possible to make it even more flexible by using a nonlinear multivariate node. For example, with a quadratic, we have

$$f_m(\mathbf{x}) : \mathbf{x}^T \mathbf{W}_m \mathbf{x} + \mathbf{w}_m^T \mathbf{x} + w_{m0} > 0 \tag{9.20}$$

Guo and Gelfand (1992) propose to use a multilayer perceptron (chapter 11) that is a linear sum of nonlinear basis functions, and this is another way of having nonlinear decision nodes. Another possibility is a

SPHERE NODE *sphere node* (Devroye, Györfi, and Lugosi 1996)

$$f_m(\mathbf{x}) : \|\mathbf{x} - \mathbf{c}_m\| \leq \alpha_m \tag{9.21}$$

where $\mathbf{c}_m$ is the center and α_m is the radius.

There are a number of algorithms proposed for learning multivariate decision trees for classification: The earliest is the multivariate version of the CART algorithm (Breiman et al. 1984), which fine-tunes the weights w_{mj} one by one to decrease impurity. CART also has a preprocessing stage to decrease dimensionality through subset selection (chapter 6) and reduce the complexity of the node. An algorithm with some extensions

OC1 to CART is the *OC1* algorithm (Murthy, Kasif, and Salzberg 1994). One

possibility (Loh and Vanichsetakul 1988) is to assume that all classes are Gaussian with a common covariance matrix, thereby having linear discriminants separating each class from the others (chapter 5). In such a case, with K classes, each node has K branches and each branch carries the discriminant separating one class from the others. Brodley and Utgoff (1995) propose a method where the linear discriminants are trained to minimize classification error (chapter 10). Guo and Gelfand (1992) propose a heuristic to group $K > 2$ classes into two supergroups, and then binary multivariate trees can be learned. Loh and Shih (1997) use 2-means clustering (chapter 7) to group data into two. Yıldız and Alpaydın (2000) use LDA (chapter 6) to find the discriminant once the classes are grouped into two.

Any classifier approximates the real (unknown) discriminant choosing one hypothesis from its hypothesis class. When we use univariate nodes, our approximation uses piecewise, axis-aligned hyperplanes. With linear multivariate nodes, we can use arbitrary hyperplanes and do a better approximation using fewer nodes. If the underlying discriminant is curved, nonlinear nodes work better. The branching factor has a similar effect in that it specifies the number of discriminants that a node defines. A binary decision node with two branches defines one discriminant separating the input space into two. An n-way node separates into n. Thus, there is a dependency among the complexity of a node, the branching factor, and tree size. With simple nodes and low branching factors, one may grow large trees, but such trees, for example, with univariate binary nodes, are more interpretable. Linear multivariate nodes are more difficult to interpret. More complex nodes also require more data and are prone to overfitting as we get down the tree and have less and less data. If the nodes are complex and the tree is small, we also lose the main idea of the tree, which is that of dividing the problem into a set of simple problems. After all, we can have a very complex classifier in the root that separates all classes from each other, but then this will not be a tree!

9.7 Notes

Divide-and-conquer is a frequently used heuristic that has been used since the days of Caesar to break a complex problem, for example, Gaul, into a group of simpler problems. Trees are frequently used in computer science to decrease complexity from linear to log time. Decision trees

were made popular in statistics in Breiman et al. 1984 and in machine learning in Quinlan 1986 and Quinlan 1993. Multivariate tree induction methods became popular more recently; a review and comparison on many datasets are given in Yıldız and Alpaydın 2000. Many researchers (e.g., Guo and Gelfand 1992), proposed to combine the simplicity of trees with the accuracy of multilayer perceptrons (chapter 11). Many studies, however, have concluded that the univariate trees are quite accurate and interpretable, and the additional complexity brought by linear (or nonlinear) multivariate nodes is hardly justified. A recent survey is given by Rokach and Maimon (2005).

OMNIVARIATE DECISION TREE

The *omnivariate decision tree* (Yıldız and Alpaydın 2001) is a hybrid tree architecture where the tree may have univariate, linear multivariate, or nonlinear multivariate nodes. The idea is that during construction, at each decision node, which corresponds to a different subproblem defined by the subset of the training data reaching that node, a different model may be appropriate and the appropriate one should be found and used. Using the same type of nodes everywhere corresponds to assuming that the same inductive bias is good in all parts of the input space. In an omnivariate tree, at each node, candidate nodes of different types are trained and compared using a statistical test (chapter 19) on a validation set to determine which one generalizes the best. The simpler one is chosen unless a more complex one is shown to have significantly higher accuracy. Results show that more complex nodes are used early in the tree, closer to the root, and as we go down the tree, simple univariate nodes suffice. As we get closer to the leaves, we have simpler problems and, at the same time, we have less data. In such a case, complex nodes overfit and are rejected by the statistical test. The number of nodes increases exponentially as we go down the tree; therefore, a large majority of the nodes are univariate and the overall complexity does not increase much.

Decision trees are used more frequently for classification than for regression. They are very popular: They learn and respond quickly, and are accurate in many domains (Murthy 1998). It is even the case that a decision tree is preferred over more accurate methods, because it is interpretable. When written down as a set of IF-THEN rules, the tree can be understood and the rules can be validated by human experts who have knowledge of the application domain.

It is generally recommended that a decision tree be tested and its accuracy be taken as a benchmark before more complicated algorithms are employed. Analysis of the tree also allows an understanding of the im-

portant features, and the univariate tree does its own automatic feature extraction. Another big advantage of the univariate tree is that it can use numeric and discrete features together, without needing to convert one type into the other.

The decision tree is a nonparametric method, similar to the instance-based methods discussed in chapter 8, but there are a number of differences:

- Each leaf node corresponds to a "bin," except that the bins need not be the same size (as in Parzen windows) or contain an equal number of training instances (as in k-nearest neighbor).
- The bin divisions are not done based only on similarity in the input space, but supervised output information through entropy or mean square error is also used.
- Another advantage of the decision tree is that, thanks to the tree structure, the leaf ("bin") is found much faster with smaller number of comparisons.
- The decision tree, once it is constructed, does not store all the training set but only the structure of the tree, the parameters of the decision nodes, and the output values in leaves; this implies that the space complexity is also much less, as opposed to instance-based nonparametric methods that store all training examples.

With a decision tree, a class need not have a single description to which all instances should match. It may have a number of possible descriptions that can even be disjoint in the input space.

The decision tree we discussed until now have *hard* decision nodes; that is, we take one of the branches depending on the test. We start from the root and follow a single path and stop at a leaf where we output the response value stored in that leaf. In a *soft decision tree*, however, we take *all* the branches but with different probabilities, and we follow in parallel all the paths and reach all the leaves, but with different probabilities. The output is the weighted average of all the outputs in all the leaves where the weights correspond to the probabilities accumulated over the paths; we will discuss this in section 12.9.

SOFT DECISION TREE

In chapter 17, we talk about combining multiple learners; one of the most popular models combined is a decision tree, and an ensemble of decision trees is called a *decision forest*. We will see that if we train not

DECISION FOREST

one but many decision trees, each on a random subset of training set or a random subset of the input features, and combine their predictions, overall accuracy can be significantly increased. This is the idea behind the *random forest* method.

RANDOM FOREST

The tree is different from the statistical models discussed in previous chapters. The tree codes directly the discriminants separating class instances without caring much for how those instances are distributed in the regions. The decision tree is *discriminant-based*, whereas the statistical methods are *likelihood-based* in that they explicitly estimate $p(\boldsymbol{x}|C_i)$ before using Bayes' rule and calculating the discriminant. Discriminant-based methods directly estimate the discriminants, bypassing the estimation of class densities. We further discuss such discriminant-based methods in the chapters ahead.

9.8 Exercises

1. Generalize the Gini index (equation 9.5) and the misclassification error (equation 9.6) for $K > 2$ classes. Generalize misclassification error to risk, taking a loss function into account.

   SOLUTION:

   - Gini index with $K > 2$ classes: $\phi(p_1, p_2, \ldots, p_K) = \sum_{i=1}^{K} \sum_{j<i} p_i p_j$
   - Misclassification error: $\phi(p_1, p_2, \ldots, p_K) = 1 - \max_{i=1}^{K} p_i$
   - Risk: $\phi_\Lambda(p_1, p_2, \ldots, p_K) = \min_{i=1}^{K} \sum_{k=1}^{K} \lambda_{ik} p_k$ where Λ is the $K \times K$ loss matrix.

2. For a numeric input, instead of a binary split, one can use a ternary split with two thresholds and three branches as

   $$x_j < w_{ma}, \; w_{ma} \leq x_j < w_{mb}, \; x_j \geq w_{mb}$$

   Propose a modification of the tree induction method to learn the two thresholds, w_{ma}, w_{mb}. What are the advantages and the disadvantages of such a node over a binary node?

   SOLUTION: For the numeric attributes, instead of one split threshold, we need to try all possible pairs of split thresholds and choose the best. When there are two splits, there are three children, and in calculating the entropy after the splits, we need to sum up over the three sets corresponding to the instances taking the three branches.

   The complexity of finding the best pair is $\mathcal{O}(N_m^2)$ instead of $\mathcal{O}(N_m)$ and each node stores two thresholds instead of one and has three branches instead

of two. The advantage is that one ternary node splits an input into three, whereas this requires two successive binary nodes. Which one is better depends on the data at hand; if we have hypotheses that require bounded intervals (e.g., rectangles), a ternary node may be advantageous.

3. Propose a tree induction algorithm with backtracking.

4. In generating a univariate tree, a discrete attribute with n possible values can be represented by n 0/1 dummy variables and then treated as n separate numeric attributes. What are the advantages and disadvantages of this approach?

5. Derive a learning algorithm for sphere trees (equation 9.21). Generalize to ellipsoid trees.

6. In a regression tree, we discussed that in a leaf node, instead of calculating the mean, we can do a linear regression fit and make the response at the leaf dependent on the input. Propose a similar method for classification trees.

   SOLUTION: This implies that at each leaf, we will have a linear classifier trained with instances reaching there. That linear classifier will generate posterior probabilities for the different classes, and those probabilities will be used in the entropy calculation. That is, it is not necessary for a leaf to be pure, that is, to contain instances of only one class; it is enough that the classifier in that leaf generates posterior probabilities close to 0 or 1.

7. Propose a rule induction algorithm for regression.

8. In regression trees, how can we get rid of discontinuities at the leaf boundaries?

9. Let us say that for a classification problem, we already have a trained decision tree. How can we use it in addition to the training set in constructing a k-nearest neighbor classifier?

   SOLUTION: The decision tree does feature selection, and we can use only the features used by the tree. The average number of instances per leaf also gives us information about a good k value.

10. In a multivariate tree, very probably, at each internal node, we will not be needing all the input variables. How can we decrease dimensionality at a node?

    SOLUTION: Each subtree handles a local region in the input space that can be explained by a small number of features. We can do feature selection or extraction using only the subset of the instances reaching that node. Ideally, as we go down the tree, we would expect to need fewer features.

9.9 References

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Belmont, CA: Wadsworth International Group.

Brodley, C. E., and P. E. Utgoff. 1995. "Multivariate Decision Trees." *Machine Learning* 19:45–77.

Cohen, W. 1995. "Fast Effective Rule Induction." In *Twelfth International Conference on Machine Learning*, ed. A. Prieditis and S. J. Russell, 115–123. San Mateo, CA: Morgan Kaufmann.

Devroye, L., L. Györfi, and G. Lugosi. 1996. *A Probabilistic Theory of Pattern Recognition*. New York: Springer.

Dietterich, T. G. 1997. "Machine Learning Research: Four Current Directions." *AI Magazine* 18:97–136.

Fürnkranz, J., and G. Widmer. 1994. "Incremental Reduced Error Pruning." In *Eleventh International Conference on Machine Learning*, ed. W. Cohen and H. Hirsh, 70–77. San Mateo, CA: Morgan Kaufmann.

Guo, H., and S. B. Gelfand. 1992. "Classification Trees with Neural Network Feature Extraction." *IEEE Transactions on Neural Networks* 3:923–933.

Loh, W.-Y., and Y. S. Shih. 1997. "Split Selection Methods for Classification Trees." *Statistica Sinica* 7:815–840.

Loh, W.-Y., and N. Vanichsetakul. 1988. "Tree-Structured Classification via Generalized Discriminant Analysis." *Journal of the American Statistical Association* 83:715–725.

Mitchell, T. 1997. *Machine Learning*. New York: McGraw-Hill.

Murthy, S. K. 1998. "Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey." *Data Mining and Knowledge Discovery* 4:345–389.

Murthy, S. K., S. Kasif, and S. Salzberg. 1994. "A System for Induction of Oblique Decision Trees." *Journal of Artificial Intelligence Research* 2:1–32.

Quinlan, J. R. 1986. "Induction of Decision Trees." *Machine Learning* 1:81–106.

Quinlan, J. R. 1990. "Learning Logical Definitions from Relations." *Machine Learning* 5:239–266.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Quinlan, J. R. 1995. "MDL and Categorical Theories (continued)." In *Twelfth International Conference on Machine Learning*, ed. A. Prieditis and S. J. Russell, 467–470. San Mateo, CA: Morgan Kaufmann.

Rokach, L., and O. Maimon. 2005. "Top-Down Induction of Decision Trees Classifiers—A Survey." *IEEE Transactions on Systems, Man, and Cybernetics-Part C* 35:476–487.

Yıldız, O. T., and E. Alpaydın. 2000. "Linear Discriminant Trees." In *Seventeenth International Conference on Machine Learning*, ed. P. Langley, 1175–1182. San Francisco: Morgan Kaufmann.

Yıldız, O. T., and E. Alpaydın. 2001. "Omnivariate Decision Trees." *IEEE Transactions on Neural Networks* 12:1539–1546.