Chapter 8

Non-metric Methods

8.1 Introduction

We have considered pattern recognition based on feature vectors of real-valued and discrete-valued numbers, and in all cases there has been a natural measure of distance between such vectors. For instance in the nearest-neighbor classifier the notion figures conspicuously — indeed it is the core of the technique — while for neural networks the notion of similarity appears when two input vectors sufficiently "close" lead to similar outputs. Most practical pattern recognition methods address problems of this sort, where feature vectors are real-valued and there exists some notion of metric.

NOMINAL DATA

But suppose a classification problem involves *nominal* data — for instance descriptions that are discrete and without any natural notion of similarity or even ordering. Consider the use of information about teeth in the classification of fish and sea mammals. Some teeth are small and fine (as in baleen whales) for straining tiny prey from the sea. Others (as in sharks) coming in multiple rows. Some sea creatures, such as walruses, have tusks. Yet others, such as squid, lack teeth altogether. There is no clear notion of similarity (or metric) for this information about teeth: it is meaningless to consider the teeth of a baleen whale any more similar to or different from the tusks of a walrus, than it is the distinctive rows of teeth in a shark from their absence in a squid, for example.

PROPERTY D-TUPLE

Thus in this chapter our attention turns away from describing patterns by vectors of real numbers and towardusing *lists* of attributes. A common approach is to specify the values of a fixed number of properties by a *property d-tuple* For example, consider describing a piece of fruit by the four properties of color, texture, taste and smell. Then a particular piece of fruit might be described by the 4-tuple $\{red, shiny, sweet, small\}$, which is a shorthand for `color = red`, `texture = shiny`, `taste = sweet` and `size = small`. Another common approach is to describe the pattern by a variable length *string* of nominal attributes, such as a sequence of base pairs in a segment of DNA, e.g., "`AGCTTCAGATTCCA`."[*] Such lists or strings might be themselves the output of other component classifiers of the type we have seen elsewhere. For instance, we might train a neural network to recognize different component brush

STRING

[*] We often put strings between quotation marks, particularly if this will help to avoid ambiguities.

strokes used in Chinese and Japanese characters (roughly a dozen basic forms); a classifier would then accept as inputs a list of these nominal attributes and make the final, full character classification.

How can we best use such nominal data for classification? Most importantly, how can we efficiently *learn* categories using such non-metric data? If there is structure in strings, how can it be represented? In considering such problems, we move beyond the notion of continuous probability distributions and metrics toward discrete problems that are addressed by rule-based or syntactic pattern recognition methods.

8.2 Decision trees

It is natural and intuitive to classify a pattern through a sequence of questions, in which the next question asked depends on the answer to the current question. This "20-questions" approach is particularly useful for non-metric data, since all of the questions can be asked in a "yes/no" or "true/false"or "value(property) $\in$ set_of_values" style that does not require any notion of metric.

Such a sequence of questions is displayed in a directed *decision tree* or simply *tree*, where by convention the first or *root node* is displayed at the top, connected by successive (directional) *links* or *branches* to other nodes. These are similarly connected until we reach terminal or *leaf* nodes, which have no further links (Fig. 8.1). Sections 8.3 & 8.4 describe some generic methods for *creating* such trees, but let us first understand how they are used for classification. The classification of a particular pattern begins at the root node, which asks for the value of a particular property of the pattern. The different links from the root node corresopnd to the different possible values. Based on the answer we follow the appropriate link to a subsequent or *descendent* node. In the trees we shall discuss, the links must be mutually distinct and exhaustive, i.e., one and only one link will be followed. The next step is to make the decision at the appropriate subsequent node, which can be considered the root of a *sub-tree*. We continue this way until we reach a leaf node, which has no further question. Each leaf node bears a category label and the test pattern is assigned the category of the leaf node reached.

ROOT NODE

LINK

BRANCH

LEAF

DESCENDENT

SUB-TREE

The simple decision tree in Fig. 8.1 illustrates one benefit of trees over many other classifiers such as neural networks: interpretability. It is a straightforward matter to render the information in such a tree as logical expressions. Such interpretability has two manifestations. First, we can easily interpret the decision for any *particular* test pattern as the conjunction of decisions along the path to its corresponding leaf node. Thus if the properties are {`taste`, `color`, `shape`, `size`}, the pattern $\mathbf{x}$ = {`sweet`, `yellow`, `thin`, `medium`} is classified as **Banana** because it is `(color = yellow) AND (shape = thin)`.* Second, we can occasionally get clear interpretations of the *categories* themselves, by creating logical descriptions using conjunctions and disjunctions (Problem 8). For instance the tree shows **Apple** = `(green AND medium) OR (red AND medium)`.

Rules derived from trees — especially large trees — are often quite complicated and must be reduced to aid interpretation. For our example, one simple rule describes **Apple** = `(medium AND NOT yellow)`. Another benefit of trees is that they lead to

* We retain our convention of representing patterns in boldface even though they need not be true vectors, i.e., they might contain nominal data that cannot be added or multiplied the way vector components can. For this reason we use the terms "attribute" to represent both nominal data and real-valued data, and reserve "feature" for real-valued data.

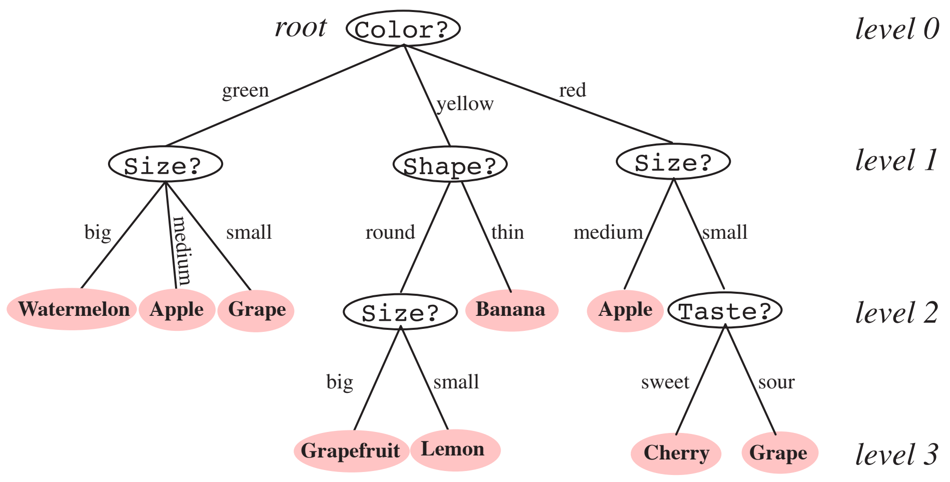


Figure 8.1: Classification in a basic decision tree proceeds from top to bottom. The questions asked at each node concern a particular property of the pattern, and the downward links correspond to the possible values. Successive nodes are visited until a terminal or leaf node is reached, where the category label is read. Note that the same question, `Size?`, appears in different places in the tree, and that different questions can have different numbers of branches. Moreover, different leaf nodes, shown in pink, can be labeled by the same category (e.g., **Apple**).

rapid classification, employing a sequence of typically simple queries. Finally, we note that trees provide a natural way to incorporate prior knowledge from human experts. In practice, though, such expert knowledge if of greatest use when the classification problem is fairly simple and the training set is small.

8.3 CART

Now we turn to the matter of using training data to create or "grow" a decision tree. We assume that we have a set $\mathcal{D}$ of labeled training data and we have decided on a set of properties that can be used to discriminate patterns, but do not know how to organize the tests into a tree. Clearly, any decision tree will progressively split the set of training examples into smaller and smaller subsets. It would be ideal if all the samples in each subset had the same category label. In that case, we would say that each subset was *pure*, and could terminate that portion of the tree. Usually, however, there is a mixture of labels in each subset, and thus for each branch we will have to decide either to stop splitting and accept an imperfect decision, or instead select another property and grow the tree further.

This suggests an obvious recursive tree-growing process: given the data represented at a node, either declare that node to be a leaf (and state what category to assign to it), or find another property to use to split the data into subsets. However, this is only one example of a more generic tree-growing methodology know as CART (Classification and Regression Trees). CART provides a general framework that can be instatiated in various ways to produce different decision trees. In the CART approach, six general kinds of questions arise:

1. Should the properties be restricted to binary-valued or allowed to be multi-

SPLIT

valued? That is, how many decision outcomes or *splits* will there be at a node?

2. Which property should be tested at a node?
3. When should a node be declared a leaf?
4. If the tree becomes "too large," how can it be made smaller and simpler, i.e., pruned?
5. If a leaf node is impure, how should the category label be assigned?
6. How should missing data be handled?

We consider each of these questions in turn.

8.3.1 Number of splits

Each decision outcome at a node is called a *split*, since it corresponds to splitting a subset of the training data. The root node splits the full training set; each successive decision splits a proper subset of the data. The number of splits at a node is closely related to question 2, specifying *which* particular split will be made at a node. In general, the number of splits is set by the designer, and could vary throughout the tree, as we saw in Fig. 8.1. The number of links descending from a node is sometimes called the node's *branching factor* or *branching ratio*, denoted B. However, every decision (and hence every tree) can be represented using just *binary* decisions (Problem 2). Thus the root node querying fruit color ($B = 3$) in our example could be replaced by two nodes: the first would ask `fruit = green?`, and at the end of its "no" branch, another node would ask `fruit = yellow?`. Because of the universal expressive power of binary trees and the comparative simplicity in training, we shall concentrate on such trees (Fig. 8.2).

BRANCHING FACTOR

8.3.2 Test selection and node impurity

Much of the work in designing trees focuses on deciding which property test or query should be performed at each node.[*] With non-numeric data, there is no geometrical interpretation of how the test at a node splits the data. However, for numerical data, there is a simple way to visualize the decision boundaries that are produced by decision trees. For example, suppose that the test at each node has the form "is $x_i \leq x_{is}$?" This leads to hyperplane decision boundaries that are perpendicular to the coordinate axes, and to decision regions of the form illustrated in Fig. 8.3.

The fundamental principle underlying tree creation is that of simplicity: we prefer decisions that lead to a simple, compact tree with few nodes. This is a version of Occam's razor, that the simplest model that explains data is the one to be preferred (Chap. **??**). To this end, we seek a property test T at each node N that makes the data reaching the immediate descendent nodes as "pure" as possible. In formalizing this notion, it turns out to be more conveninet to define the *im*purity, rather than

PURITY

[*] The problem is further complicated by the fact that there is no reason why the test at a node has to involve only one property. One might well consider logical combinations of properties, such as using `(size = medium) AND (NOT (color = yellow))?` as a test. Trees in which each test is based on a single property are called *monothetic*; if the query at any of the nodes involves two or more properties, the tree is called *polythetic*. For simplicity, we generally restrict our treatment to monothetic trees. In all cases, the key requirement is that the decision at a node be well-defined and unambiguous so that the response leads down one and only one branch.

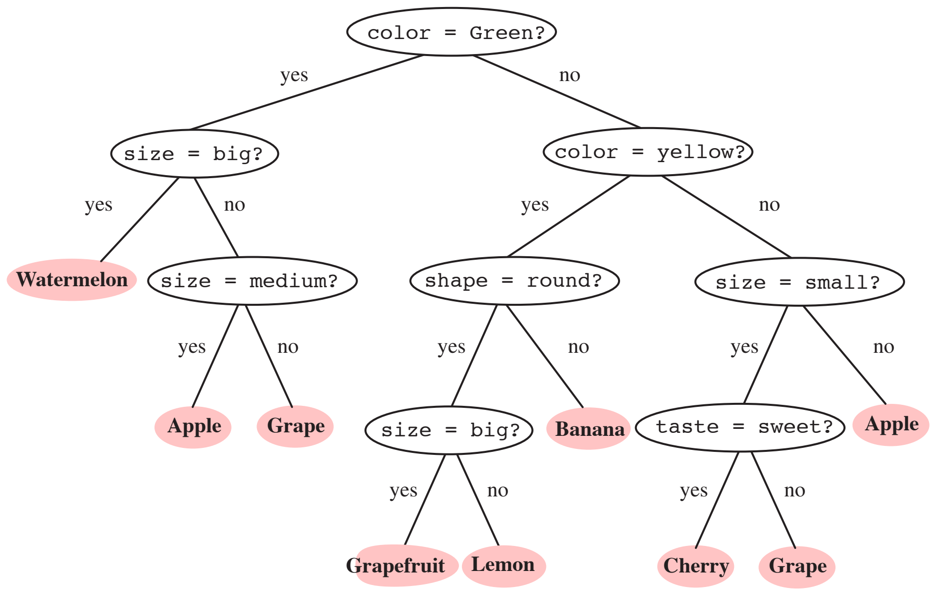


Figure 8.2: A tree with arbitrary branching factor at different nodes can always be represented by a functionally equivalent binary tree, i.e., one having branching factor $B = 2$ throughout. By convention the "yes" branch is on the left, the "no" branch on the right. This binary tree contains the same information and implements the same classification as that in Fig. 8.1.

the purity of a node. Several different mathematical measures of impurity have been proposed, all of which have basically the same behavior. Let $i(N)$ denote the impurity of a node N. In all cases, we want $i(N)$ to be 0 if all of the patterns that reach the node bear the same category label, and to be large if the categories are equally represented.

The most popular measure is the *entropy impurity* (or occasionally *information impurity*):

ENTROPY IMPURITY

$$i(N) = -\sum_j P(\omega_j) \log_2 P(\omega_j), \tag{1}$$

where $P(\omega_j)$ is the fraction of patterns at node N that are in category ω_j.* By the well-known properties of entropy, if all the patterns are of the same category, the impurity is 0; otherwise it is positive, with the greatest value occuring when the different classes are equally likely.

Another definition of impurity is particularly useful in the two-category case. Given the desire to have zero impurity when the node represents only patterns of a single category, the simplest polynomial form is:

$$i(N) = P(\omega_1)P(\omega_2). \tag{2}$$

This can be interpreted as a *variance impurity* since under reasonable assumptions it

VARIANCE IMPURITY

* Here we are a bit sloppy with notation, since we normally reserve P for probability and $\hat{P}$ for frequency ratios. We could be even more precise by writing $\hat{P}(\mathbf{x} \in \omega_j|N)$ — i.e., the fraction of training patterns $\mathbf{x}$ at node N that are in category ω_j, given that they have survived all the previous decisions that led to the node N — but for the sake of simplicity we sill avoid such notational overhead.

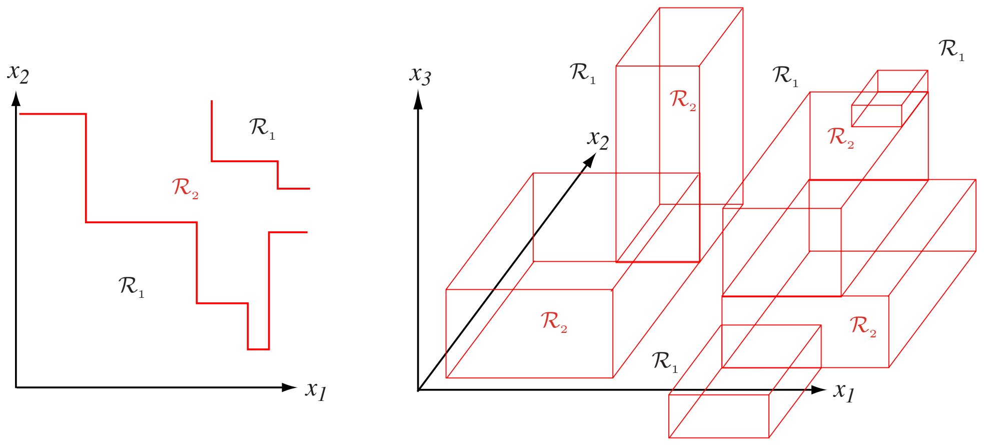


Figure 8.3: Monothetic decision trees create decision boundaries with portions perpendicular to the feature axes. The decision regions are marked $\mathcal{R}_1$ and $\mathcal{R}_2$ in these two-dimensional and three-dimensional two-category examples. With a sufficiently large tree, any decision boundary can be approximated arbitrarily well.

is related to the variance of a distribution associated with the two categories (Problem 10). A generalization of the variance impurity, applicable to two or more categories, is the *Gini impurity*:

GINI IMPURITY

$$i(N) = \sum_{i \neq j} P(\omega_i)P(\omega_j) = 1 - \sum_j P^2(\omega_j). \tag{3}$$

This is just the expected error rate at node N if the category label is selected randomly from the class distribution present at N. This criterion is more strongly peaked at equal probabilities than is the entropy impurity (Fig. 8.4).

MISCLASSIFICATION IMPURITY

The *misclassification impurity* can be written as

$$i(N) = 1 - \max_j P(\omega_j), \tag{4}$$

and measures the minimum probability that a training pattern would be misclassified at N. Of the impurity measures typically considered, this measure is the most strongly peaked at equal probabilities. It has a discontinuous derivative, though, and this can present problems when searching for an optimal decision over a continuous parameter space. Figure 8.4 shows these impurity functions for a two-category case, as a function of the probability of one of the categories.

We now come to the key question — given a partial tree down to node N, what value s should we choose for the property test T? An obvious heuristic is to choose the test that decreases the impurity as much as possible. The drop in impurity is defined by

$$\Delta i(N) = i(N) - P_L i(N_L) - (1 - P_L)i(N_R), \tag{5}$$

where N_L and N_R are the left and right descendent nodes, $i(N_L)$ and $i(N_R)$ their impurities, and P_L is the fraction of patterns at node N that will go to N_L when property test T is used. Then the "best" test value s is the choice for T that maximizes $\Delta i(T)$. If the entropy impurity is used, then the impurity reduction corresponds to an

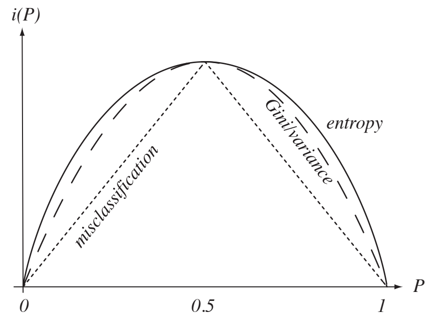


Figure 8.4: For the two-category case, the impurity functions peak at equal class frequencies and the variance and the Gini impurity functions are identical. To facilitate comparisons, the entropy, variance, Gini and misclassification impurities (given by Eqs. 1 – 4, respectively) have been adjusted in scale and offset to facilitate comparison; such scale and offset does not directly affect learning or classification.

information gain provided by the query. Since each query in a binary tree is a single "yes/no" one, the reduction in entropy impurity due to a split at a node cannot be greater than one bit (Problem 5).

The way to find an optimal decision for a node depends upon the general form of decision. Since the decision criteria are based on the extrema of the impurity functions, we are free to change such a function by an additive constant or overall scale factor and this will not affect which split is found. Designers typically choose functions that are easy to compute, such as those based on a *single* feature or attribute, giving a monothetic tree. If the form of the decisions is based on the nominal attributes, we may have to perform extensive or exhaustive search over all possible subsets of the training set to find the rule maximizing Δi. If the attributes are real-valued, one could use gradient descent algorithms to find a splitting hyperplane (Sect. 8.3.8), giving a polythetic tree. An important reason for favoring binary trees is that the decision at any node can generally be cast as a one-dimensional optimization problem. If the branching factor B were instead greater than 2, a two- or higher-dimensional optimization would be required; this is generally much more difficult (Computer exercise **??**).

Sometimes there will be several decisions s that lead to the same reduction in impurity and the question arises how to choose among them. For example, if the features are real-valued and a split lying anywhere in a range $x_l < x_s < x_u$ for the x variable leads to the same (maximum) impurity reduction, it is traditional to choose either the midpoint or the weighted average — $x_s = (x_l + x_u)/2$ or $x_s = (1 - P)x_l + x_u P$, respectively — where P is the probability a pattern goes to the "left" under the decision. Computational simplicity may be the determining factor as there are rarely deep theoretical reasons to favor one over another.

Note too that the optimization of Eq. 5 is *local* — done at a single node. As with the vast majority of such *greedy methods*, there is no guarantee that successive locally optimal decisions lead to the *global* optimum. In particular, there is no guarantee that after training we have the smallest tree (Computer exercise **??**). Nevertheless, for every reasonable impurity measure and learning method, we can always continue to split further to get the lowest possible impurity at the leafs (Problem **??**). There is no assurance that the impurity at a leaf node will be the zero, however: if two

GREEDY METHOD

patterns have the same attribute description yet come from different categories, the impurity will be greater than zero.

Occasionally during tree creation the misclassification impurity (Eq. 4) will not decrease whereas the Gini impurity would (Problem **??**); thus although classification is our final goal, we may prefer the Gini impurity because it "anticipates" later splits that will be useful. Consider a case where at node N there are 90 patterns in ω_1 and 10 in ω_2. Thus the misclassification impurity is 0.1. Suppose there are no splits that guarantee a ω_2 majority in either of the two descendent nodes. Then the misclassification remains at 0.1 for all splits. Now consider a split which sends 70 ω_1 patterns to the right along with 0 ω_2 patterns, and sends 20 ω_1 and 10 ω_2 to the left. This is an attractive split but the misclassification impurity is still 0.1. On the other hand, the Gini impurity for this split is less than the Gini for the parent node. In short, the Gini impurity shows that this as a good split while the misclassification rate does not.

TWOING CRITERION

In multiclass binary tree creation, the *twoing criterion* may be useful.* The overall goal is to find the split that best splits *groups* of the c categories, i.e., a candidate "supercategory" $\mathcal{C}_1$ consisting of all patterns in some subset of the categories, and candidate "supercategory" $\mathcal{C}_2$ as all remaining patterns. Let the class of categories be $\mathcal{C} = \{\omega_1, \omega_2, \ldots, \omega_c\}$. At each node, the decision splits the categories into $\mathcal{C}_1 = \{\omega_{i_1}, \omega_{i_2}, \ldots, \omega_{i_k}\}$ and $\mathcal{C}_2 = \mathcal{C} - \mathcal{C}_1$. For every candidate split s, we compute a change in impurity $\Delta i(s, \mathcal{C}_1)$ as though it corresponded to a standard two-class problem. That is, we find the split $s^*(\mathcal{C}_1)$ that maximizes the change in impurity. Finally, we find the supercategory $\mathcal{C}_1^*$ which maximizes $\Delta i(s^*(\mathcal{C}_1), \mathcal{C}_1)$. The benefit of this impurity is that it is *strategic* — it may learn the largest scale structure of the overall problem (Problem 4).

It may be surprising, but the particular choice of an impurity function rarely seems to affect the final classifier and its accuracy. An entropy impurity is frequently used because of its computational simplicity and basis in information theory, though the Gini impurity has received significant attention as well. In practice, the stopping criterion and the pruning method — when to stop splitting nodes, and how to merge leaf nodes — are more important than the impurity function itself in determining final classifier accuracy, as we shall see.

Multi-way splits

Although we shall concentrate on binary trees, we briefly mention the matter of allowing the branching ratio at each node to be set during training, a technique will return to in a discussion of the ID3 algorithm (Sect. 8.4.1). In such a case, it is tempting to use a multi-branch generalization of Eq. 5 of the form

$$\Delta i(s) = i(N) - \sum_{k=1}^{B} P_k i(N_k), \tag{6}$$

where P_k is the fraction of training patterns sent down the link to node N_k, and $\sum\limits_{k=1}^{B} P_k = 1$. However, the drawback with Eq. 6 is that decisions with large B are inherently favored over those with small B whether or not the large B splits in fact represent meaningful structure in the data. For instance, even in random data, a

* The twoing criterion is not a true impurity measure.

high-B split will reduce the impurity more than will a low-B split. To avoid this drawback, the candidate change in impurity of Eq. 6 must be scaled, according to

$$\Delta i_B(s) = \frac{\Delta i(s)}{-\sum\limits_{k=1}^{B} P_k \log_2 P_k}. \tag{7}$$

a method based on the *gain ratio impurity* (Problem 17). Just as before, the optimal split is the one maximizing $\Delta i_B(s)$.

GAIN RATIO IMPURITY

8.3.3 When to stop splitting

Consider now the problem of deciding when to stop splitting during the training of a binary tree. If we continue to grow the tree fully until each leaf node corresponds to the lowest impurity, then the data has typically been overfit (Chap. **??**). In the extreme but rare case, each leaf corresponds to a single training point and the full tree is merely a convenient implementation of a lookup table; it thus cannot be expected to generalize well in (noisy) problems having high Bayes error. Conversely, if splitting is stopped too early, then the error on the training data is not sufficiently low and hence performance may suffer.

How shall we decide when to stop splitting? One traditional approach is to use techniques of Chap. **??**, in particular cross-validation. That is, the tree is trained using a subset of the data (for instance 90%), with the remaining (10%) kept as a validation set. We continue splitting nodes in successive layers until the error on the validation data is minimized.

Another method is to set a (small) threshold value in the reduction in impurity; splitting is stopped if the best candidate split at a node reduces the impurity by less than that pre-set amount, i.e., if $\max_s \Delta i(s) \leq \beta$. This method has two main benefits. First, unlike cross-validation, the tree is trained directly using *all* the training data. Second, leaf nodes can lie in different levels of the tree, which is desirable whenever the complexity of the data varies throughout the range of input. (Such an *unbalanced* tree requires a different number of decisions for different test patterns.) A fundamental drawback of the method, however, is that it is often difficult to know how to set the threshold because there is rarely a simple relationship between β and the ultimate performance (Computer exercise 2). A very simple method is to stop when a node represents fewer than some threshold number of points, say 10, or some fixed percentage of the total training set, say 5%. This has a benefit analogous to that in k-nearest-neighbor classifiers (Chap. **??**); that is, the size of the partitions is small in regions where data is dense, but large where the data is sparse.

BALANCED TREE

Yet another method is to trade complexity for test accuracy by splitting until a minimum in a new, global criterion function,

$$\alpha \cdot size + \sum_{leaf\ nodes} i(N), \tag{8}$$

is reached. Here *size* could represent the number of nodes or links and α is some positive constant. (This is analogous to regularization methods in neural networks that penalize connection weights or nodes.) If an impurity based on entropy is used for $i(N)$, then Eq. 8 finds support from *minimum description length* (MDL), which we shall consider again in Chap. **??**. The sum of the impurities at the leaf nodes is a measure of the uncertainty (in bits) in the training data given the model represented

MINIMUM DESCRIPTION LENGTH

by the tree; the size of the tree is a measure of the complexity of the classifier itself (which also could be measured in bits). A difficulty, however, is setting α, as it is not always easy to find a simple relationship between α and the final classifier performance (Computer exercise 3).

An alternative approach is to use a stopping criterion based on the statistical significance of the reduction of impurity. During tree construction, we estimate the distribution of all the Δi for the current collection of nodes; we assume this is the full distribution of Δi. For any candidate node split, we then determine whether it is statistically different from zero, for instance by a chi-squared test (cf. Sect. **??**). If a candidate split does not reduce the impurity *significantly*, splitting is stopped (Problem 15).

HYPOTHESIS TESTING

A variation in this technique of *hypothesis testing* can be applied even without strong assumptions on the distribution of Δi. We seek to determine whether a candidate split is "meaningful," that is, whether it differs significantly from a random split. Suppose n patterns survive at node N (with n_1 in ω_1 and n_2 in ω_2); we wish to decide whether a candidate split s differs significantly from a random one. Suppose a particular candidate split s sends Pn patterns to the left branch, and $(1-P)n$ to the right branch. A random split having this probability (i.e., the null hypothesis) would place Pn_1 of the ω_1 patterns and Pn_2 of the ω_2 patterns to the left, and the remaining to the right. We quantify the deviation of the results due to candidate split s from the (weighted) random split by means of the *chi-squared statistic*, which in this two-category case is

CHI-SQUARED STATISTIC

$$\chi^2 = \sum_{i=1}^{2} \frac{(n_{iL} - n_{ie})^2}{n_{ie}}, \tag{9}$$

where n_{iL} is the number of patterns in category ω_i sent to the left under decision s, and $n_{ie} = Pn_i$ is the number expected by the random rule. The chi-squared statistic vanishes if the candidate split s gives the same distribution as the random one, and is larger the more s differs from the random one. When χ^2 is greater than a critical value, as given in a table (cf. Table **??**), then we can reject the null hypothesis since s differs "significantly" at some probability or *confidence level*, such as .01 or .05. The critical values of the confidence depend upon the number of degrees of freedom, which in the case just described is 1, since for a given probability P the *single* value n_{1L} specifies all other values (n_{1R}, n_{2L} and n_{2R}). If the "most significant" split at a node does not yield a χ^2 exceeding the chosen confidence level threshold, splitting is stopped.

CONFIDENCE LEVEL

8.3.4 Pruning

Occassionally, stopped splitting suffers from the lack of sufficient look ahead, a phenomenon called the *horizon effect*. The determination of the optimal split at a node N is not influenced by decisions at N's descendent nodes, i.e., those at subsequent levels. In stopped splitting, node N might be declared a leaf, cutting off the possibility of beneficial splits in subsequent nodes; as such, a stopping condition may be met "too early" for overall optimal recognition accuracy. Informally speaking, the stopped splitting biases the learning algorithm toward trees in which the greatest impurity reduction is near the root node.

HORIZON EFFECT

The principal alternative approach to stopped splitting is *pruning*. In pruning, a tree is grown fully, that is, until leaf nodes have minimum impurity — beyond any putative “horizon.” Then, all pairs of neighboring leaf nodes (i.e., ones linked to a common antecedent node, one level above) are considered for elimination. Any pair whose elimination yields a satisfactory (small) increase in impurity is eliminated, and the common antecedent node declared a leaf. (This antecedent, in turn, could itself be pruned.) Clearly, such *merging* or *joining* of the two leaf nodes is the inverse of splitting. It is not unusual that after such pruning, the leaf nodes lie in a wide range of levels and the tree is unbalanced.

MERGING

Although it is most common to prune starting at the leaf nodes, this is not necessary: cost-complexity pruning can replace a complex subtree with a leaf directly. Further, C4.5 (Sect. 8.4.2) can eliminate an arbitrary test node, thereby replacing a subtree by one of its branches.

The benefits of pruning are that it avoids the horizon effect; further, since there is no training data held out for cross-validation, it directly uses *all* information in the training set. Naturally, this comes at a greater computational expense than stopped splitting, and for problems with large training sets, the expense can be prohibitive (Computer exercise **??**). For small problems, though, these computational costs are low and pruning is generally to be preferred over stopped splitting. Incidentally, what we have been calling stopped training and pruning are sometimes called pre-pruning and post-pruning, respectively.

A conceptually different pruning method is based on *rules.* Each leaf has an associated rule — the conjunction of the individual decisions from the root node, through the tree, to the particular leaf. Thus the full tree can be described by a large list of rules, one for each leaf. Occasionally, some of these rules can be simplified if a series of decisions is redundant. Eliminating the *irrelevant* precondition rules simplifies the description, but has no influence on the classifier function, including its generalization ability. The predominant reason to prune, however, is to improve generalization. In this case we therefore eliminate rules so as to improve accuracy on a validation set (Computer exercise 6). This technique may even allow the elimination of a rule corresponding to a node near the root.

One of the benefits of rule pruning is that it allows us to distinguish between the contexts in which any particular node N is used. For instance, for some test pattern $\mathbf{x}_1$ the decision rule at node N is necessary; for another test pattern $\mathbf{x}_2$ that rule is irrelevant and thus N could be pruned. In traditional node pruning, we must either keep N or prune it away. In rule pruning, however, we can eliminate it where it is not necessary (i.e., for patterns such as $\mathbf{x}_1$) and retain it for others (such as $\mathbf{x}_2$).

A final benefit is that the reduced rule set may give improved interpretability. Although rule pruning was not part of the original CART approach, such pruning can be easily applied to CART trees. We shall consider an example of rule pruning in Sect. 8.4.2.

8.3.5 Assignment of leaf node labels

Assigning category labels to the leaf nodes is the simplest step in tree construction. If successive nodes are split as far as possible, and each leaf node corresponds to patterns in a single category (zero impurity), then of course this category label is assigned to the leaf. In the more typical case, where either stopped splitting or pruning is used and the leaf nodes have positive impurity, each leaf should be labeled by the category that has most points represented. An extremely small impurity is not necessarily desirable, since it may be an indication that the tree is overfitting the training data.

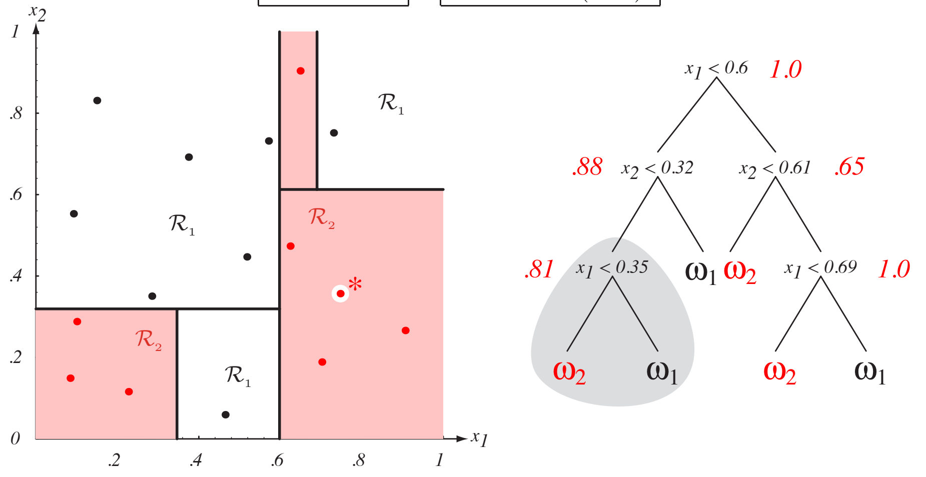
Example 1 illustrates some of these steps.

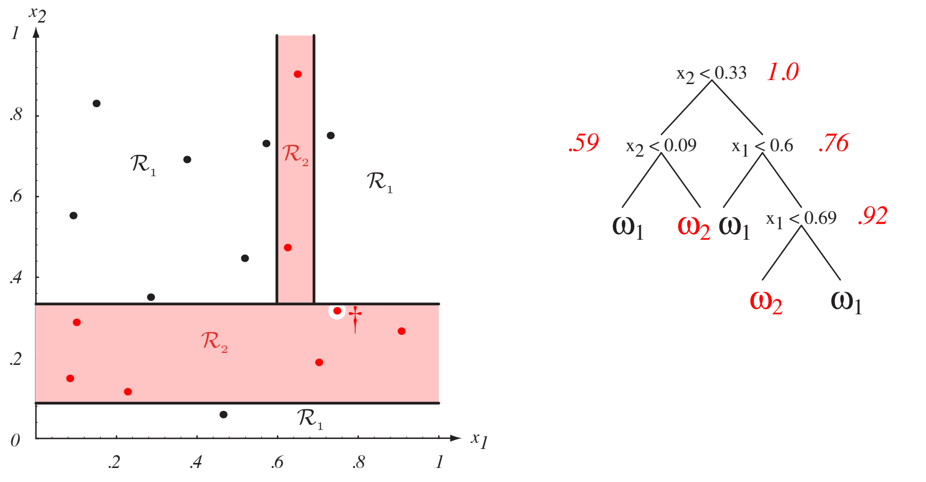
Example 1: A simple tree classifier

Consider the following $n = 16$ points in two dimensions for training a binary CART tree ($B = 2$) using the entropy impurity (Eq. 1).

ω_1 (black)	
x_1	x_2
.15	.83
.09	.55
.29	.35
.38	.70
.52	.48
.57	.73
.73	.75
.47	.06

ω_2 (red)	
x_1	x_2
.10	.29
.08	.15
.23	.16
.70	.19
.62	.47
.91	.27
.65	.90
.75	.36* (.32†)





Training data and associated (unpruned) tree are shown at the top. The entropy impurity at non-terminal nodes is shown in red and the impurity at each leaf is 0. If the single training point marked * were instead slightly lower (marked †), the resulting tree and decision regions would differ significantly, as shown at the bottom.

The impurity of the root node is

$$i(N_{root}) = -\sum_{i=1}^{2} P(\omega_i)\log_2\ P(\omega_i) = -[.5\log_2.5 + .5\log_2.5] = 1.0.$$

For simplicity we consider candidate splits parallel to the feature axes, i.e., of the form "is $x_i < x_{is}$?". By exhaustive search of the $n-1$ positions for the x_1 feature and $n-1$ positions for the x_2 feature we find by Eq. 5 that the greatest reduction in the impurity occurs near $x_{1s} = 0.6$, and hence this becomes the decision criterion at the root node. We continue for each sub-tree until each final node represents a single category (and thus has the lowest impurity, 0), as shown in the figure. If pruning were invoked, the pair of leaf nodes at the left would be the first to be deleted (gray shading) since there the impurity is increased the least. In this example, stopped splitting with the proper threshold would also give the same final network. In general, however, with large trees and many pruning steps, pruning and stopped splitting need not lead to the same final tree.

This particular training set shows how trees can be sensitive to details of the training points. If the ω_2 point marked * in the top figure is moved slightly (marked †), the tree and decision regions differ significantly, as shown at the bottom. Such instability is due in large part to the discrete nature of decisions early in the tree learning.

Example 1 illustrates the informal notion of *instability* or sensitivity to training points. Of course, if we train any common classifier with a slightly different training set the final classification decisions will differ somewhat. If we train a CART classifier, however, the alteration of even a single training point can lead to radically different decisions overall. This is a consequence of the discrete and inherently greedy nature of such tree creation. Instability often indicates that incremental and off-line versions of the method will yield significantly different classifiers, even when trained on the same data.

STABILITY

8.3.6 Computational complexity

Suppose we have n training patterns in d dimensions in a two-category problem, and wish to construct a binary tree based on splits parallel to the feature axes using an entropy impurity. What are the time and the space complexities?

At the root node (level 0) we must first sort the training data, $\mathcal{O}(n\log n)$ for each of the d features or dimensions. The entropy calculation is $\mathcal{O}(n) + (n-1)\mathcal{O}(d)$ since we examine $n-1$ possible splitting points. Thus for the root node the time complexity is $\mathcal{O}(dn\log n)$. Consider an average case, where roughly half the training points are sent to each of the two branches. The above analysis implies that splitting each node in level 1 has complexity $\mathcal{O}(d\ n/2\ \log(n/2))$; since there are two such nodes at that level, the total complexity is $\mathcal{O}(dn\log(n/2))$. Similarly, for the level 2 we have $\mathcal{O}(dn\log(n/4))$, and so on. The total number of levels is $\mathcal{O}(\log\ n)$. We sum the terms for the levels and find that the total average time complexity is $\mathcal{O}(dn\ (\log\ n)^2)$. The time complexity for recall is just the depth of the tree, i.e., the total number of levels, is $\mathcal{O}(\log\ n)$. The space complexity is simply the number of nodes, which, given some simplifying assumptions (such as a single training point per leaf node), is $1 + 2 + 4 + ... + n/2 \approx n$, that is, $\mathcal{O}(n)$ (Problem 9).

We stress that these assumptions (for instance equal splits at each node) rarely hold exactly; moreover, heuristics can be used to speed the search for splits during training. Nevertheless, the result that for fixed dimension d the training is $\mathcal{O}(dn^2 \log n)$ and classification $\mathcal{O}(\log n)$ is a good rule of thumb; it illustrates how training is far more computationally expensive than is classification, and that on average this discrepancy grows as the problem gets larger.

There are several techniques for reducing the complexity during the training of trees based on real-valued data. One of the simplest heuristics is to begin the search for splits x_{is} at the "middle" of the range of the training set, moving alternately to progressively higher and lower values. Optimal splits always occur for decision thresholds between adjacent points from *different* categories and thus one should test only such ranges. These and related techniques generally provide only moderate reductions in computation (Computer exercise **??**). When the patterns consist of nominal data, candidate splits could be over every subset of attributes, or just a single entry, and the computational burden is best lowered using insight into features (Problem 3).

8.3.7 Feature choice

As with most pattern recognition techniques, CART and other tree-based methods work best if the "proper" features are used (Fig. 8.5). For real-valued vector data, most standard preprocessing techniques can be used before creating a tree. Preprocessing by principal components (Chap. **??**) can be effective, since it finds the "important" axes, and this generally leads to simple decisions at the nodes. If however the principal axes in one region differ significantly from those in another region, then no single choice of axes overall will suffice. In that case we may need to employ the techniques of Sect. 8.3.8, for instance allowing splits to be at arbitrary orientation, often giving smaller and more compact trees.

8.3.8 Multivariate decision trees

If the "natural" splits of real-valued data do not fall parallel to the feature axes or the full training data set differs significantly from simple or accommodating distributions, then the above methods may be rather inefficient and lead to poor generalization (Fig. 8.6); even pruning may be insufficient to give a good classifier. The simplest solution is to allow splits that are not parallel to the feature axes, such as a general linear classifier trained via gradient descent on a classification or sum-squared-error criterion (Chap. **??**). While such training may be slow for the nodes near the root if the training set is large, training will be faster at nodes closer to the leafs since less training data is used. Recall can remain quite fast since the linear functions at each node can be computed rapidly.

8.3.9 Priors and costs

Up to now we have tacitly assumed that a category ω_i is represented with the same frequency in both the training and the test data. If this is not the case, we need a method for controlling tree creation so as to have lower error on the actual final classification task when the frequencies are different. The most direct method is to "weight" samples to correct for the prior frequencies (Problem 16). Furthermore, we may seek to minimize a general cost, rather than a strict misclassification or 0-1

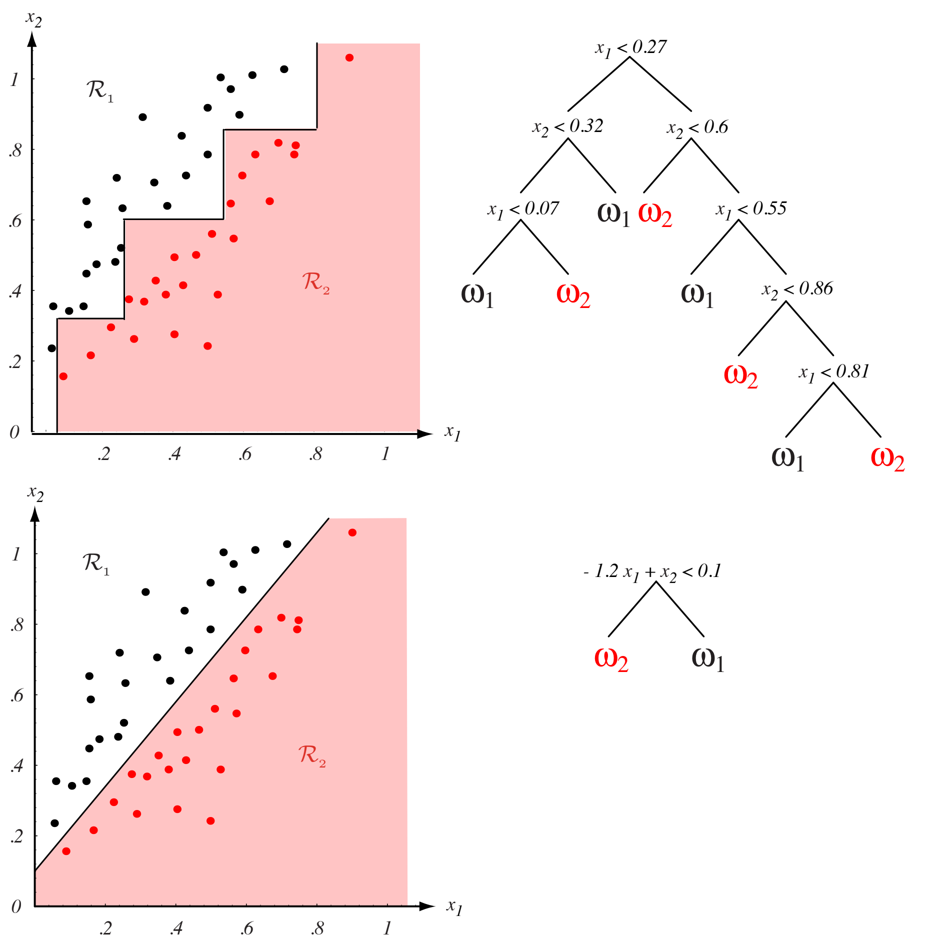


Figure 8.5: If the class of node decisions does not match the form of the training data, a very complicated decision tree will result, as shown at the top. Here decisions are parallel to the axes while in fact the data is better split by boundaries along another direction. If however "proper" decision forms are used (here, linear combinations of the features), the tree can be quite simple, as shown at the bottom.

cost. As in Chap. **??**, we represent such information in a cost matrix λ_{ij} — the cost of classifying a pattern as ω_i when it is actually ω_j. Cost information is easily incorporated into a Gini impurity, giving the following weighted Gini impurity,

WEIGHTED GINI IMPURITY

$$i(N) = \sum_{ij} \lambda_{ij} P(\omega_i) P(\omega_j), \tag{10}$$

which should be used during training. Costs can be incorporated into other impurity measures as well (Problem 11).

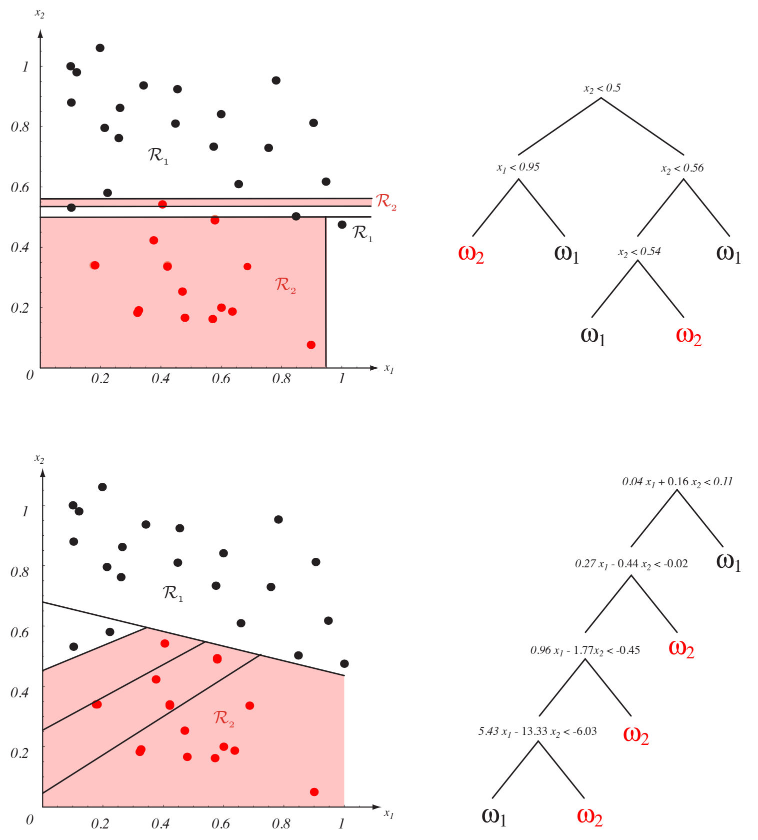


Figure 8.6: One form of multivariate tree employs general linear decisions at each node, giving splits along arbitrary directions in the feature space. In virtually all interesting cases the training data is not linearly separable, and thus the LMS algorithm is more useful than methods that require the data to be linearly separable, even though the LMS need not yield a minimum in classification error (Chap. **??**). The tree at the bottom can be simplified by methods outlined in Sect. 8.4.2.

8.3.10 Missing attributes

Classification problems might have missing attributes during training, during classification, or both. Consider first training a tree classifier despite the fact that some training patterns are missing attributes. A naive approach would be to delete from consideration any such *deficient patterns*; however, this is quite wasteful and should be employed only if there are many complete patterns. A better technique is to proceed as otherwise described above (Sec. 8.3.2), but instead calculate impurities at a node N using only the attribute information present. Suppose there are n training points at N and that each has three attributes, except one pattern that is missing attribute x_3. To find the best split at N, we calculate possible splits using all n points using attribute x_1, then all n points for attribute x_2, then the $n-1$ non-deficient points for attribute x_3. Each such split has an associated reduction in impurity, calculated as before, though here with different numbers of patterns. As always, the desired split is the one which gives the greatest decrease in impurity. The generalization of this procedure to more features, to multiple patterns with missing attributes, and even to

DEFICIENT PATTERN

patterns with several missing attributes is straightforward, as is its use in classifying non-deficient patterns (Problem 14).

Now consider how to create and use trees that can *classify* a deficient pattern. The trees described above cannot directly handle test patterns lacking attributes (but see Sect. 8.4.2), and thus if we suspect that such deficient test patterns will occur, we must modify the training procedure discussed in Sect. 8.3.2. The basic approach during classification is to use the traditional ("primary") decision at a node whenever possible (i.e., when the queries involves a feature that is present in the deficient test pattern) but to use alternate queries whenever the test pattern is missing that feature.

During training then, in addition to the primary split, each non-terminal node N is given an ordered set of *surrogate splits*, consisting of an attribute label and a rule. The first such surrogate split maximizes the "predictive association" with the primary split. A simple measure of the predictive association of two splits s_1 and s_2 is merely the numerical count of patterns that are sent to the "left" by both s_1 and s_2 plus the count of the patterns sent to the "right" by both the splits. The second surrogate split is defined similarly, being the one which uses another feature and best approximates the primary split in this way. Of course, during classification of a deficient test pattern, we use the first surrogate split that does not involve the test pattern's missing attributes. This missing value strategy corresponds to a linear model replacing the pattern's missing value by the value of the non-missing attribute most strongly correlated with it (Problem **??**). This strategy uses to maximum advantage the (local) associations among the attributes to decide the split when attribute values are missing. A method closely related to surrogate splits is that of *virtual values*, in which the missing attribute is assigned its most likely value.

SURROGATE SPLIT

PREDICTIVE ASSOCIATION

VIRTUAL VALUE

Example 2: Surrogate splits and missing attributes

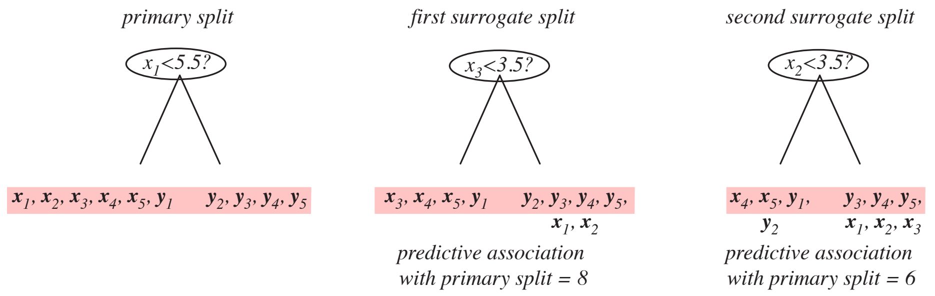
Consider the creation of a monothetic tree using an entropy impurity and the following ten training points. Since the tree will be used to classify test patterns with missing features, we will give each node surrogate splits.

$$\omega_1: \overset{\mathbf{x}_1}{\begin{pmatrix}0\\7\\8\end{pmatrix}}, \overset{\mathbf{x}_2}{\begin{pmatrix}1\\8\\9\end{pmatrix}}, \overset{\mathbf{x}_3}{\begin{pmatrix}2\\9\\0\end{pmatrix}}, \overset{\mathbf{x}_4}{\begin{pmatrix}4\\1\\1\end{pmatrix}}, \overset{\mathbf{x}_5}{\begin{pmatrix}5\\2\\2\end{pmatrix}}$$

$$\omega_2: \overset{\mathbf{y}_1}{\begin{pmatrix}3\\3\\3\end{pmatrix}}, \overset{\mathbf{y}_2}{\begin{pmatrix}6\\0\\4\end{pmatrix}}, \overset{\mathbf{y}_3}{\begin{pmatrix}7\\4\\5\end{pmatrix}}, \overset{\mathbf{y}_4}{\begin{pmatrix}8\\5\\6\end{pmatrix}}, \overset{\mathbf{y}_5}{\begin{pmatrix}9\\6\\7\end{pmatrix}}.$$

Through exhaustive search along all three features, we find the primary split at the root node should be "$x_1 < 5.5$?", which sends $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5, \mathbf{y}_1\}$ to the left and $\{\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \mathbf{y}_5\}$ to the right, as shown in the figure.

We now seek the first surrogate split at the root node; such a split must be based on either the x_2 or the x_3 feature. Through exhaustive search we find that the split "$x_3 < 3.5$?" has the highest predictive association with the primary split — a value of 8, since 8 patterns are sent to matching directions by each rule, as shown in the figure. The second surrogate split must be along the only remaining feature, x_2. We find that for this feature the rule "$x_2 < 3.5$?" has the highest predictive association with the primary split, a value of 6. (This, incidentally, is not the optimal x_2 split for impurity reduction — we use it because it best approximates the preferred, primary split.) While the above describes the training of the root node, training of other nodes is conceptually the same, though computationally less complex because fewer points need be considered.


primary split
x_1<5.5?
$x_1, x_2, x_3, x_4, x_5, y_1$
y_2, y_3, y_4, y_5
first surrogate split
x_3<3.5?
x_3, x_4, x_5, y_1
$y_2, y_3, y_4, y_5, x_1, x_2$
predictive association with primary split = 8
second surrogate split
x_2<3.5?
x_4, x_5, y_1, y_2
$y_3, y_4, y_5, x_1, x_2, x_3$
predictive association with primary split = 6

Of all possible splits based on a single feature, the primary split, "$x_1 < 5.5$?", minimizes the entropy impurity of the full training set. The first surrogate split at the root node must use a feature other than x_1; its threshold is set in order to best approximate the action of the primary split. In this case "$x_3 < 3.5$?" is the first surrogate split. Likewise, here the second surrogate split must use the x_2 feature; its threshold is chosen to best approximate the action of the primary split. In this case "$x_2 < 3.5$?" is the second surrogate split. The pink shaded band marks those patterns sent to the matching direction as the primary split. The number of patterns in the shading is thus the predictive association with the primary split.

During classification, any test pattern containing feature x_1 would be queried using the primary split, "$x_1 \leq 5.5$?" Consider though the deficient test pattern $(*, 2, 4)^t$, where * is the missing x_1 feature. Since the primary split cannot be used, we turn instead to the first surrogate split, "$x_3 \leq 3.5$?", which sends this point to the right. Likewise, the test pattern $(*, 2, *)^t$ would be queried by the second surrogate split, "$x_2 \leq 3.5$?", and sent to the left.

Sometimes the fact that an attribute is missing can be informative. For instance, in medical diagnosis, the fact that an attribute (such as blood sugar level) is missing might imply that the physician had some reason not to measure it. As such, a missing attribute could be represented as a new feature, and used in classification.

8.4 Other tree methods

Virtually all tree-based classification techniques can incorporate the fundamental techniques described above. In fact that discussion expanded beyond the core ideas in the earliest presentations of CART. While most tree-growing algorithms use an entropy impurity, there are many choices for stopping rules, for pruning methods and for the treatment of missing attributes. Here we discuss just two other popular tree algorithms.

8.4.1 ID3

ID3 received its name because it was the third in a series of identification or "ID" procedures. It is intended for use with nominal (unordered) inputs only. If the problem involves real-valued variables, they are first binned into intervals, each interval being treated as an unordered nominal attribute. Every split has a branching factor B_j, where B_j is the number of discrete attribute bins of the variable j chosen for splitting. In practice these are seldom binary and thus a gain ratio impurity should be used (Sect. 8.3.2). Such trees have their number of levels equal to the number of input variables. The algorithm continues until all nodes are pure or there are no more variables to split on. While there is thus no pruning in standard presentations of the ID3 algorithm, it is straightforward to incorporate pruning along the ideas presented above (Computer exercise 4).

8.4.2 C4.5

The C4.5 algorithm, the successor and refinement of ID3, is the most popular in a series of "classification" tree methods. In it, real-valued variables are treated the same as in CART. Multi-way ($B > 2$) splits are used with nominal data, as in ID3 with a gain ratio impurity based on Eq. 7. The algorithm uses heuristics for pruning derived based on the statistical significance of splits.

A clear difference between C4.5 and CART involves classifying patterns with missing features. During training there are no special accommodations for subsequent classification of deficient patterns in C4.5; in particular, there are no surrogate splits precomputed. Instead, if node N with branching factor B queries the missing feature in a deficient test pattern, C4.5 follows *all* B possible answers to the descendent nodes and ultimately B leaf nodes. The final classification is based on the labels of the B leaf nodes, weighted by the decision probabilities at N. (These probabilities are simply those of decisions at N on the training data.) Each of N's immediate descendent nodes can be considered the root of a sub-tree implementing part of the full classification model. This missing-attribute scheme corresponds to weighting these sub-models by the probability *any* training pattern at N would go to the corresponding outcome of the decision. This method does not exploit statistical correlations between different features of the training points, whereas the method of surrogate splits in CART does. Since C4.5 does not compute surrogate splits and hence does not need to store them, this algorithm may be preferred over CART if space complexity (storage) is a major concern.

The C4.5 algorithm has the provision for pruning based on the rules derived from the learned tree. Each leaf node has an associated rule — the conjunction of the decisions leading from the root node, through the tree, to that leaf. A technique called C4.5*Rules* deletes redundant antecedents in such rules. To understand this, consider the left-most leaf in the tree at the bottom of Fig. 8.6, which corresponds to the rule

C4.5RULES

$$\begin{array}{lll} IF\big[& (0.40x_1 + 0.16x_2 & < \ 0.11) \\ \quad AND & (0.27x_1 - 0.44x_2 & < -0.02) \\ \quad AND & (0.96x_1 - 1.77x_2 & < -0.45) \\ \quad AND & (5.43x_1 - 13.33x_2 & < -6.03)\big] \\ THEN & \quad \mathbf{x} \in \omega_1. & \end{array}$$

This rule can be simplified to give

$$IF\big[\quad (0.40x_1 + \ 0.16x_2 \quad < 0.11)$$

$$\begin{aligned} AND\ (5.43x_1 - 13.33x_2 &< -6.03)\big] \\ THEN \quad & \mathbf{x} \in \omega_1, \end{aligned}$$

as should be evident in that figure. Note especially that information corresponding to nodes near the root can be pruned by C4.5Rules. This is more general than impurity based pruning methods, which instead merge leaf nodes.

8.4.3 Which tree classifier is best?

In Chap. **??** we shall consider the problem of comparing different classifiers, including trees. Here, rather than directly comparing typical implementations of CART, ID3, C4.5 and other numerous tree methods, it is more instructive to consider variations within the different component steps. After all, with care one can generate a tree using any reasonable feature processing, impurity measure, stopping criterion or pruning method. Many of the basic principles applicable throughout pattern classification guide us here. Of course, if the designer has insight into feature preprocessing, this should be exploited. The binning of real-valued features used in early versions of ID3 does not take full advantage of order information, and thus ID3 should be applied to such data only if computational costs are otherwise too high. It has been found that an entropy impurity works acceptably in most cases, and is a natural default. In general, pruning is to be preferred over stopped training and cross-validation, since it takes advantage of more of the information in the training set; however, pruning large training sets can be computationally expensive. The pruning of rules is less useful for problems that have high noise and are at base statistical in nature, but such pruning can often simplify classifiers for problems where the data were generated by rules themselves. Likewise, decision trees are poor at inferring simple concepts, for instance whether more than half of the binary (discrete) attributes have value +1. As with most classification methods, one gains expertise and insight through experimentation on a wide range of problems. No single tree algorithm dominates or is dominated by others.

It has been found that trees yield classifiers with accuracy comparable to other methods we have discussed, such as neural networks and nearest-neighbor classifiers, especially when specific prior information about the appropriate form of classifier is lacking. Tree-based classifiers are particularly useful with non-metric data and as such they are an important tool in pattern recognition research.

8.5 *Recognition with strings

Suppose the patterns are represented as ordered sequences or *strings* of discrete items, as in a sequence of letters in an English word or in DNA bases in a gene sequence, such as "AGCTTCGAATC." (The letters A, G, C and T stand for the nucleic acids adenine, guanine, cytosine and thymine.) Pattern classification based on such strings of discrete symbols differs in a number of ways from the more commonly used techniques we have addressed up to here. Because the string elements — called *characters*, letters or symbols — are nominal, there is no obvious notion of distance between strings. There is a further difficulty arising from the fact that strings need not be of the same length. While such strings are surely not vectors, we nevertheless broaden our familiar boldface notation to now apply to strings as well, e.g., $\mathbf{x}$ = "AGCTTC," though we will often refer to them as patterns, strings, templates or general *words*. (Of course,

CHARACTER

WORD