Contents

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.

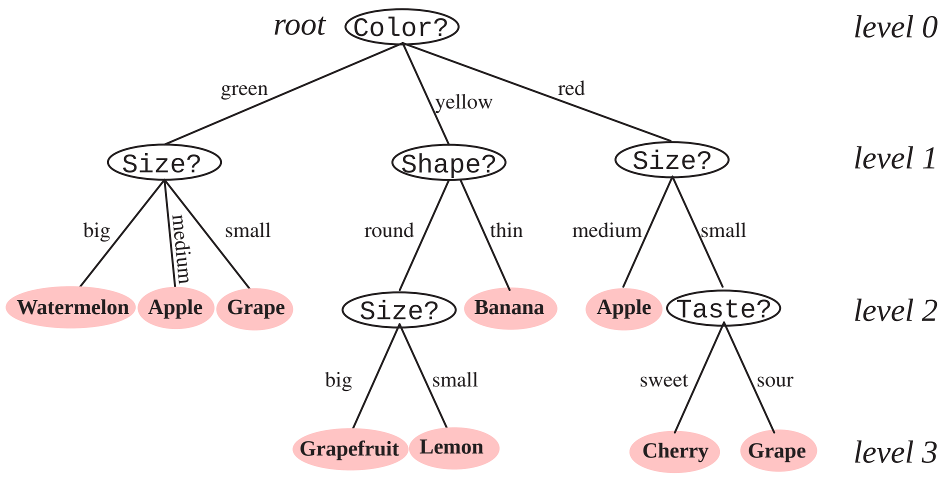


Figure 8.1: Classification in a basic decision tree proceeds from top to bottom. The questions asked at each node concern a particular property of the pattern, and the downward links correspond to the possible values. Successive nodes are visited until a terminal or leaf node is reached, where the category 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.

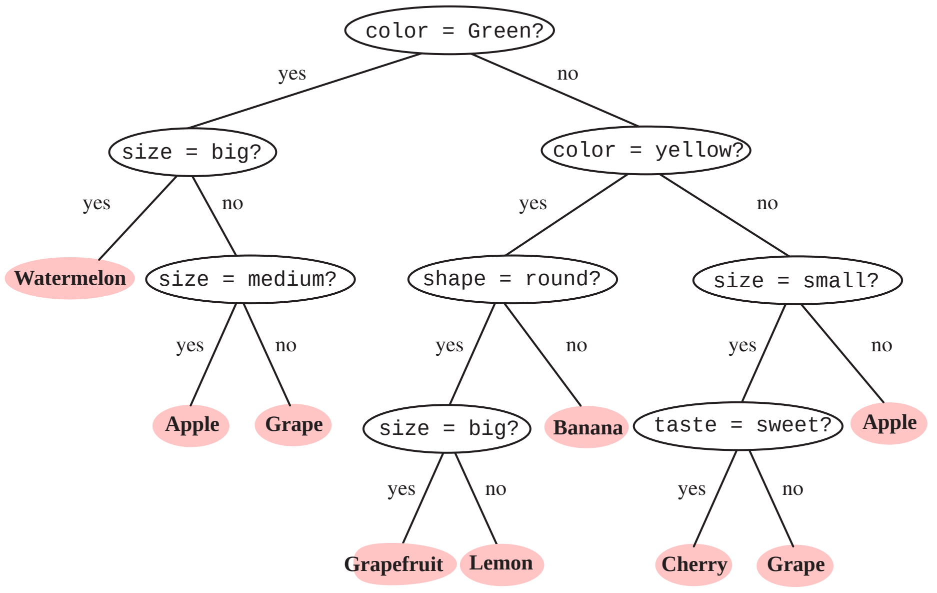


Figure 8.2: A tree with arbitrary branching factor at different nodes can always be represented by a functionally equivalent binary tree, i.e., one having branching factor $B = 2$ throughout. By convention the "yes" branch is on the left, the "no" branch on the right. This binary tree contains the same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

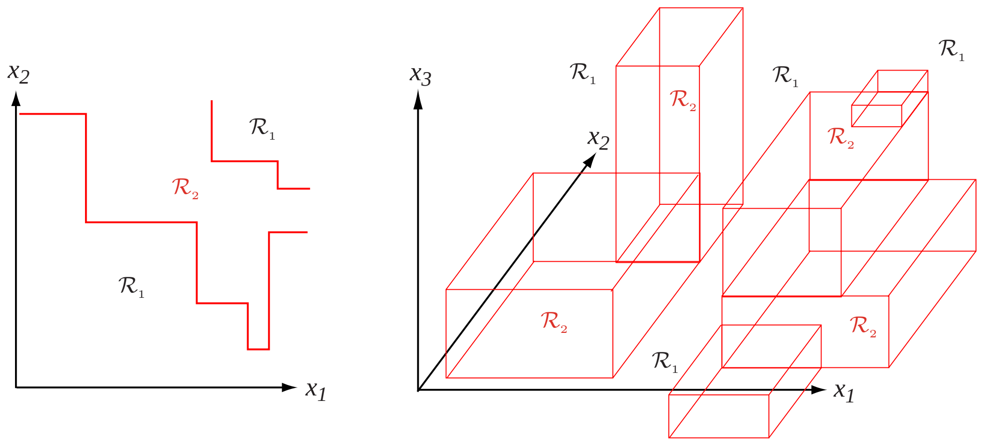


Figure 8.3: Monothetic decision trees create decision boundaries with portions perpendicular to the feature axes. The decision regions are marked $\mathcal{R}_1$ and $\mathcal{R}_2$ in these two-dimensional and three-dimensional two-category examples. With a sufficiently large tree, any decision boundary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

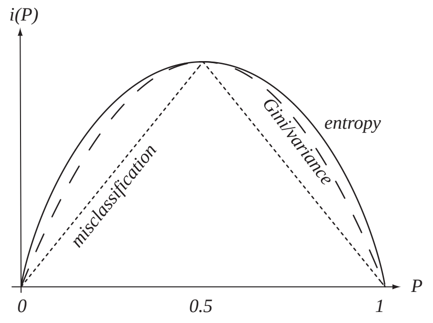


Figure 8.4: For the two-category case, the impurity functions peak at equal class frequencies and the variance and the Gini impurity functions are identical. To facilitate comparisons, the entropy, variance, Gini and misclassification impurities (given by Eqs. 1 – 4, respectively) have been adjusted 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

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

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.

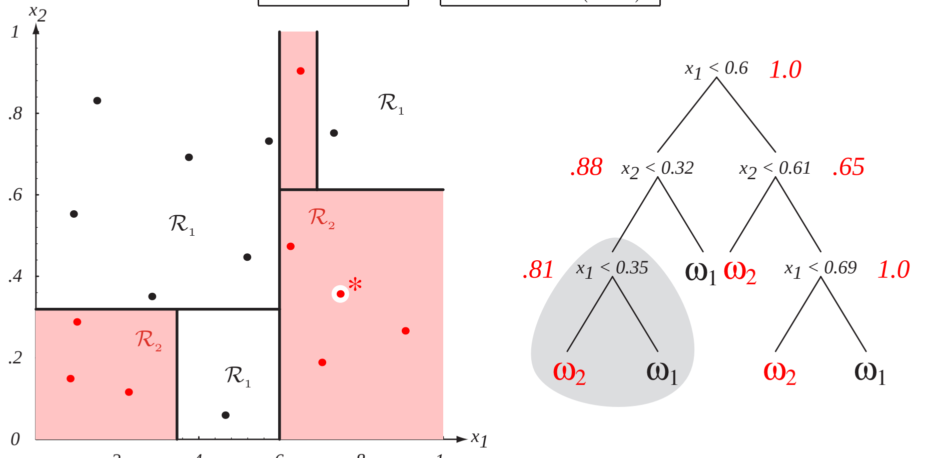
Example 1 illustrates some of these steps.

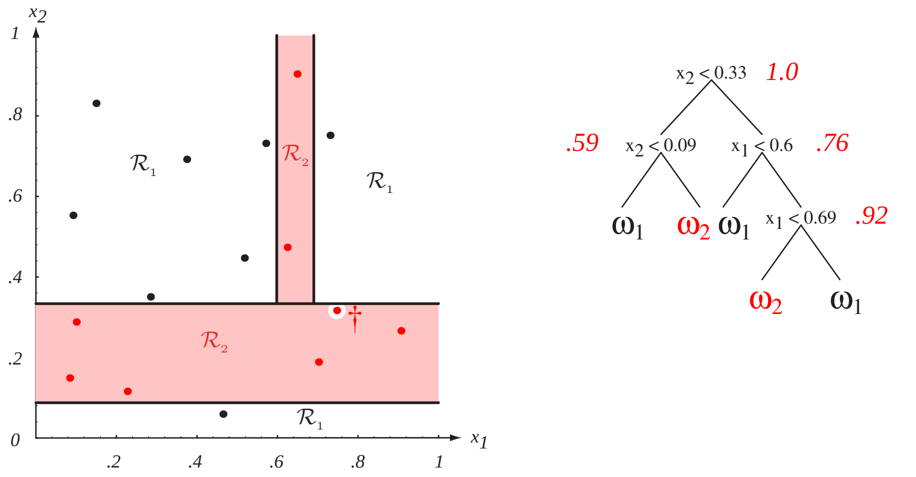
Example 1: A simple tree classifier

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

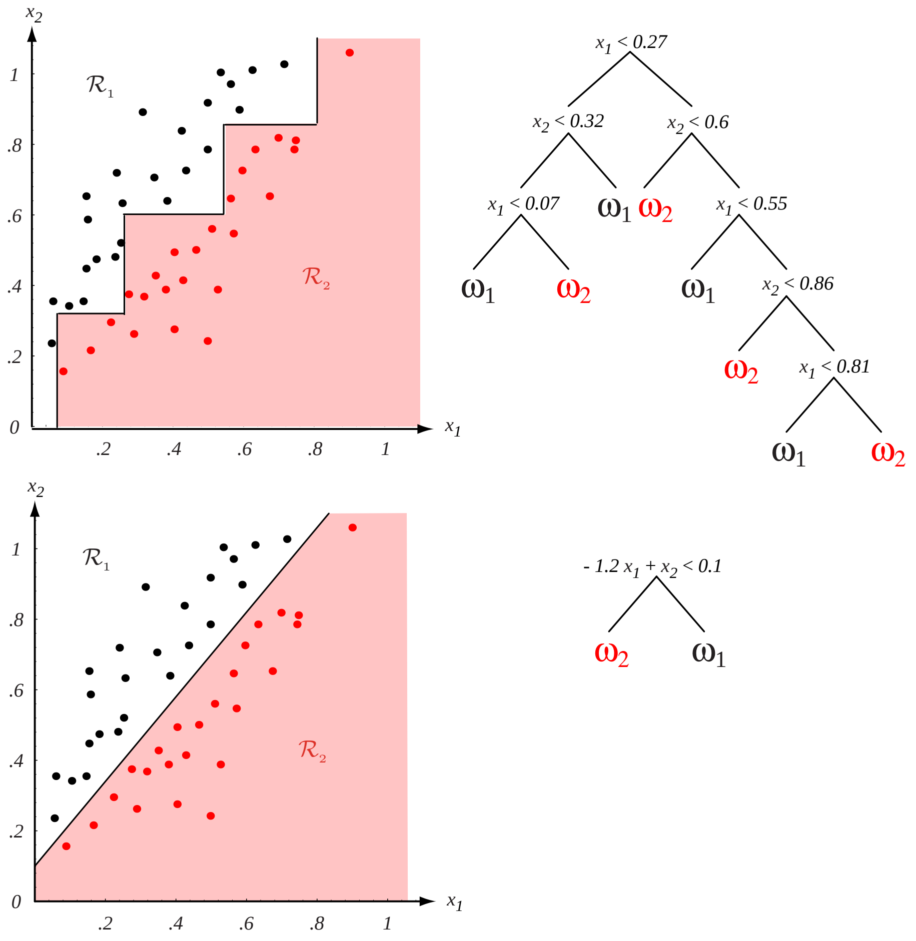


Figure 8.5: If the class of node decisions does not match the form of the training data, a very complicated decision tree will result, as shown at the top. Here decisions are parallel to the axes while in fact the data is better split by boundaries along another direction. If however "proper" decision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

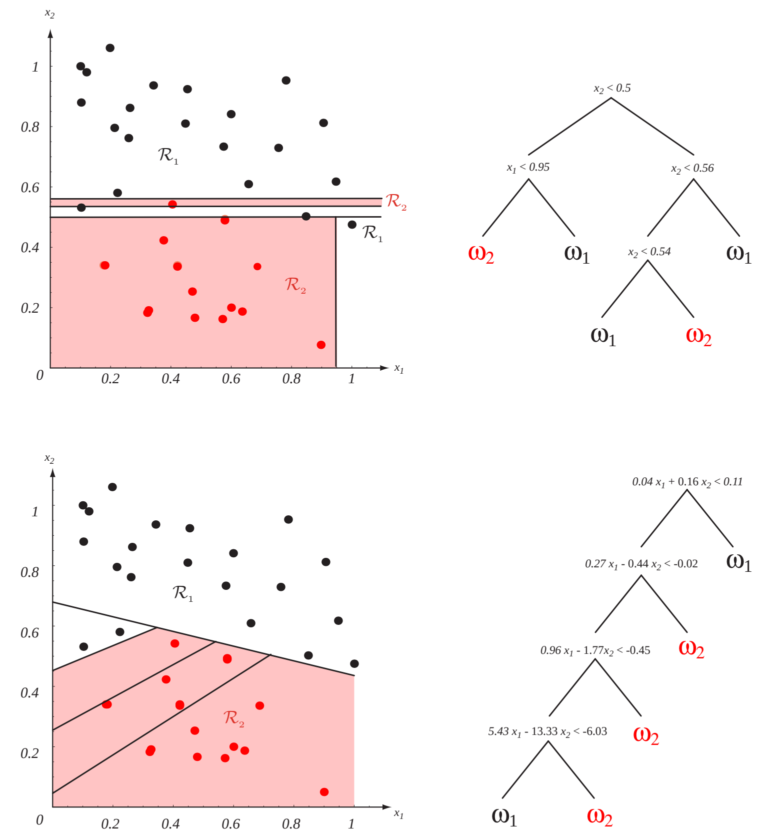


Figure 8.6: One form of multivariate tree employs general linear decisions at each node, giving splits along arbitrary directions in the feature space. In virtually all interesting cases the training data is not linearly separable, and thus the LMS algorithm is more useful than methods that require the 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

DEFICIENT PATTERN

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 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

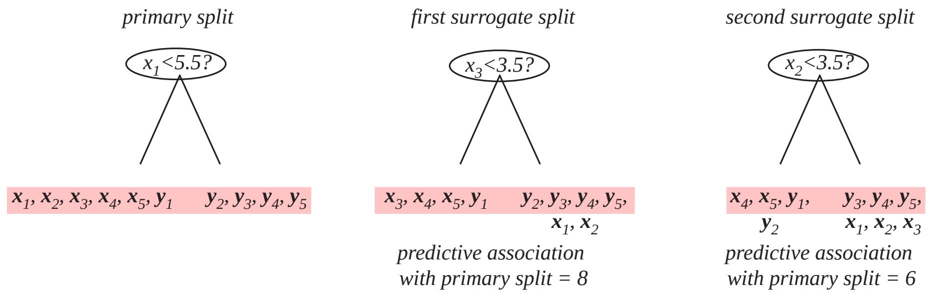
Consider the creation of a monothetic tree using an entropy impurity and the following ten training points. Since the tree will be used to classify test patterns with missing features, we will give each node surrogate splits.

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

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

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

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

This rule can be simplified to give

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

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

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

there is no requirement that these be meaningful words in a natural language such as English or French.) A particularly long string is denoted *text*. Any contiguous string that is part of **x** is called a substring, segment, or more frequently a *factor* of **x**. For example, "`GCT`" is a factor of "`AGCTTC`."

TEXT

FACTOR

There is a large number of problems in computations on strings. The ones that are of greatest importance in pattern recognition are:

String matching: Given **x** and *text*, test whether **x** is a factor of *text*, and if so, where it appears.

Edit distance: Given two strings **x** and **y**, compute the minimum number of basic operations — character insertions, deletions and exchanges — needed to transform **x** into **y**.

String matching with errors: Given **x** and *text*, find the locations in *text* where the "cost" or "distance" of **x** to any factor of *text* is minimal.

String matching with the "don't care" symbol: This is the same as basic string matching, but with a special symbol, ∅, the *don't care* symbol, which can match any other symbol.

DON'T CARE SYMBOL

We should begin by understanding the several ways in which these string operations are used in pattern classification. Basic string matching can be viewed as an extreme case of template matching, as in finding a particular English word within a large electronic corpus such as a novel or digital repository. Alternatively, suppose we have a large text such as Herman Melville's **Moby Dick**, and we want to classify it as either most relevant to the topic of fish or to the topic of hunting. Test strings or *keywords* for the fish topic might include "`salmon`," "`whale`," "`fishing`," "`ocean`," while those for hunting might include "`gun`," "`bullet`," "`shoot`," and so on. String matching would determine the number of occurrences of such keywords in the text. A simple count of the keyword occurrences could then be used to classify the text according to topic. (Other, more sophisticated methods for this latter stage would generally be preferable.)

KEYWORD

The problem of string matching with the don't care symbol is closely related to standard string matching, even though the best algorithms for the two types of problems differ, as we shall see. Suppose, for instance, that in DNA sequence analysis we have a segment of DNA, such as **x** = "`AGCCG`∅∅∅∅∅`GACTG`," where the first and last sections (called motifs) are important for coding a protein while the middle section, which consists of five characters, is nevertheless known to be inert and to have no function. If we are given an extremely long DNA sequence (the *text*), string matching with the don't care symbol using the pattern **x** containing ∅ symbols would determine if *text* is in the class of sequences that could yield the particular protein.

The string operation that finds greatest use in pattern classification is based on edit distance, and is best understood in terms of the nearest-neighbor algorithm (Chap. **??**). Recall that in that algorithm each training pattern or prototype is stored along with its category label; an unknown test pattern is then classified by its nearest prototype. Suppose now that the prototypes are strings and we seek to classify a novel test string by its "nearest" stored string. For instance an acoustic speech recognizer might label every 10-ms interval with the most likely phoneme present in an utterance, giving a string of discrete phoneme labels such as "`tttoooonn`." Edit distance would then be used to find the "nearest" stored training pattern, so that its category label can be read.

The difficulty in this approach is that there is no obvious notion of metric or distance between strings. In order to proceed, then, we must introduce some measure of distance between the strings. The resulting *edit distance* is the minimum number of fundamental operations needed to transform the test string into a prototype string, as we shall see.

The string-matching-with-errors problem contains aspects of both the basic string matching and the edit distance problems. The goal is to find all locations in *text* where $\mathbf{x}$ is "close" to the substring or factor of *text*. This measure of closeness is chosen to be an edit distance. Thus the string-matching-with-errors problem finds use in the same types of problems as basic string matching, the only difference being that there is a certain "tolerance" for a match. It finds use, for example, in searching digital texts for possibly misspelled versions of a given target word.

Naturally, deciding *which* strings to consider is highly problem-dependent. Nevertheless, given target strings and the relevance of tolerances, and so on, the string matching problems just outlined are conceptually very simple; the challenge arises when the problems are large, such as searching for a segment within the roughly 3×10^9 base pairs in the human genome, the 3×10^7 characters in an electronic version of **War and Peace** or the more than 10^{13} characters in a very large digital repository. For such cases, the effort is in finding tricks and heuristics that make the problem computationally tractable.

We now consider these four string operations in greater detail.

8.5.1 String matching

The most fundamental and useful operation in string matching is testing whether a candidate string $\mathbf{x}$ is a factor of *text*. Naturally we assume the number of characters in *text*, denoted length[*text*] or $|text|$, is greater than that in $\mathbf{x}$, and for most computationally interesting cases $|text| \gg |\mathbf{x}|$. Each discrete character is taken from an ALPHABET *alphabet* $\mathcal{A}$, for example binary or decimal numerals, the English letters, or four DNA bases, i.e., $\mathcal{A} = \{0, 1\}$ or {`0,1,2,...,9`} or {`a,b,c,...,z`} or {`A,G,C,T`}, respectively. SHIFT A *shift*, s, is an offset needed to align the first character of $\mathbf{x}$ with character number $s + 1$ in *text*. The basic string matching problem is to find whether there VALID SHIFT exists a *valid shift*, i.e., one where there is a perfect match between each character in $\mathbf{x}$ and the corresponding one in *text*. The general string-matching problem is to list *all* valid shifts (Fig. 8.7).

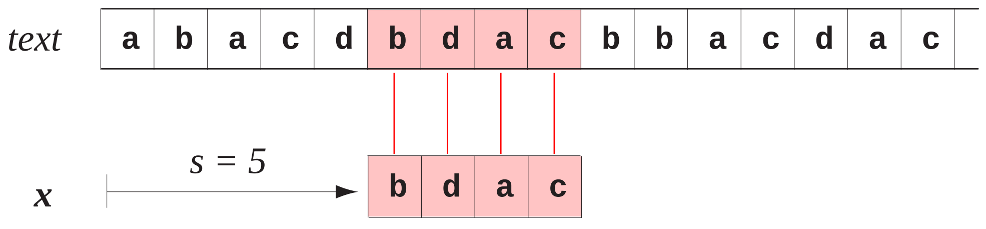


Figure 8.7: The general string-matching problem is to find all shifts s for which the pattern $\mathbf{x}$ appears in *text*. Any such shift is called *valid*. In this case $\mathbf{x}$ = "`bdac`" is indeed a factor of *text*, and $s = 5$ is the only valid shift.

The most straightforward approach in string matching is to test each possible shift s in turn, as given in the naive string-matching algorithm.

Algorithm 1 (Naive string matching)

```
1  begin initialize $\mathcal{A}, \mathbf{x}, text, n \leftarrow$ length$[text], m \leftarrow$ length$[\mathbf{x}]$
2          $s \leftarrow 0$
3          while $s \leq n - m$
4                  if $\mathbf{x}[1...m] = text[s+1...s+m]$
5                     then print "pattern occurs at shift" $s$
6                  $s \leftarrow s + 1$
7          return
8    end
```

Algorithm 1 is hardly optimal — it takes time $\mathbf{\Theta}((n-m+1)m)$ in the worst case; if $\mathbf{x}$ and *text* are random, however, the algorithm is efficient (Problem 18). The weakness in the naive string-matching algorithm is that information from one candidate shift s is not exploited when seeking a subsequent candidate shift. A more sophisticated method, the Boyer-Moore algorithm, uses such information in a clever way.

Algorithm 2 (Boyer-Moore string matching)

```
1  begin initialize $\mathcal{A}, \mathbf{x}, text, n \leftarrow$ length$[text], m \leftarrow$ length$[\mathbf{x}]$
2          $\mathcal{F}(\mathbf{x}) \leftarrow$ last-occurrence function
3          $\mathcal{G}(\mathbf{x}) \leftarrow$ good-suffix function
4          $s \leftarrow 0$
5          while $s \leq n - m$
6                  do $j \leftarrow m$
7                  while $j > 0$ and $\mathbf{x}[j] = text[s+j]$
8                          do $j \leftarrow j - 1$
9                          if $j = 0$
10                            then print "pattern occurs at shift" $s$
11                                   $s \leftarrow s + \mathcal{G}(0)$
12                             else $s \leftarrow s + \max\big[\mathcal{G}(j), j - \mathcal{F}(text[s+j])\big]$
13   return
14 end
```

Postponing for the moment considerations of the functions $\mathcal{F}$ and $\mathcal{G}$, we can see that the Boyer-Moore algorithm resembles the naive string-matching algorithm, but with two exceptions. First, at each candidate shift s, the character comparisons are done in reverse order, i.e., from right to left (line 8). Second, according to lines 11 & 12, the increment to a new shift apparently need not be 1.

The power of Algorithm 2 lies in two heuristics that allow it to skip the examination of a large number shifts and hence character comparisons: the *good-suffix heuristic* and the *bad-character heuristic* operate independently and in parallel. After a mismatch is detected, each heuristic proposes an amount by which s can be safely increased without missing a valid shift; the larger of these proposed shifts is selected and s is increased accordingly.

BAD-CHARACTER HEURISTIC

The *bad-character heuristic* utilizes the rightmost character in *text* that does not match the aligned character in $\mathbf{x}$. Because character comparisons proceed right-to-left, this "bad character" is found as efficiently as possible. Since the current shift s is invalid, no more character comparisons are needed and a shift increment can be made. The bad-character heuristic proposes incrementing the shift by an amount to align the rightmost occurrence of the bad character in $\mathbf{x}$ with the bad character identified in *text*. This guarantees that no valid shifts have been skipped (Fig. 8.8).

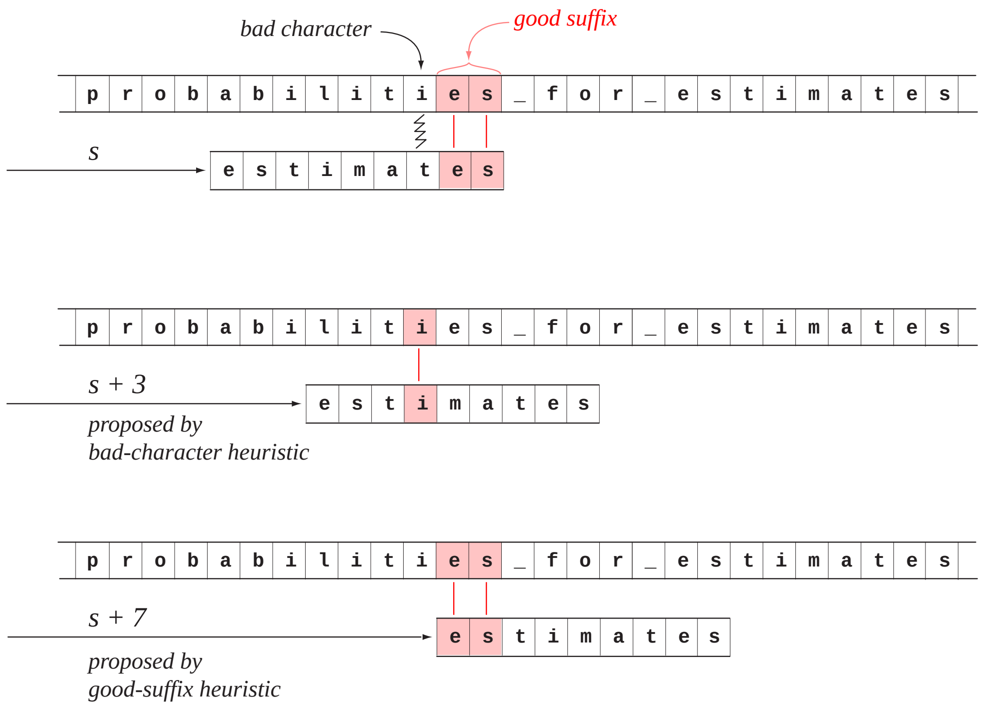


Figure 8.8: String matching by the Boyer-Moore algorithm takes advantage of information obtained at one shift s to propose the next shift; the algorithm is generally much less computationally expensive than naive string matching, which always increments shifts by a single character. The top figure shows the alignment of *text* and pattern $\mathbf{x}$ for an invalid shift s. Character comparisons proceed right to left, and the first two such comparisons are a match — the good suffix is "`es`." The first (right-most) mismatched character in *text*, here "`i`," is called the *bad character*. The bad-character heuristic proposes incrementing the shift to align the right-most "`i`" in $\mathbf{x}$ with the bad character "`i`" in *text* — a shift increment of 3, as shown in the middle figure. The bottom figure shows the effect of the good-suffix heuristic, which proposes incrementing the shift the least amount that will align the good suffix, "`es`" in $\mathbf{x}$, with that in *text* — here an increment of 7. Lines 11 & 12 of the Boyer-Moore algorithm select the larger of the two proposed shift increments, i.e., 7 in this case. Although not shown in this figure, after the mismatch is detected at shift $s + 7$, both the bad-character and the good-suffix heuristics propose an increment of yet another 7 characters, thereby finding a valid shift.

Now consider the *good-suffix heuristic*, which operates in parallel with the bad-character heuristic, and also proposes a safe shift increment. A general *suffix* of $\mathbf{x}$ is a factor or substring of $\mathbf{x}$ that contains the final character in $\mathbf{x}$. (Likewise, a *prefix* contains the initial character in $\mathbf{x}$.) At shift s the rightmost contiguous characters in *text* that match those in $\mathbf{x}$ are called the *good suffix*, or "matching suffix." As before, because character comparisons are made right-to-left, the good suffix is found with the minimum number of comparisons. Once a character mismatch has been found, the good-suffix heuristic proposes to increment the shift so as to align the next occurrence of the good suffix in $\mathbf{x}$ with that identified in *text*. This insures that no valid shift has been skipped. Given the two shift increments proposed by the two heuristics, line 12

GOOD-SUFFIX HEURISTIC

SUFFIX

PREFIX

GOOD SUFFIX

of the Boyer-Moore algorithm chooses the larger.

These heuristics rely on the functions $\mathcal{F}$ and $\mathcal{G}$. The *last-occurrence function*, $\mathcal{F}(\mathbf{x})$, is merely a table containing every letter in the alphabet and the position of its rightmost occurrence in $\mathbf{x}$. For the pattern in Fig. 8.8, the table would contain: `a`, 6; `e`, 8; `i`, 4; `m`, 5; `s`, 9; and `t`, 8. All 20 other letters in the English alphabet are assigned a value 0, signifying that they do not appear in $\mathbf{x}$. The construction of this table is simple (Problem 22) and need be done just once; it does not significanly affect the computational cost of the Boyer-Moore algorithm.

LAST-OCCURRENCE FUNCTION

The *good-suffix function*, $\mathcal{G}(\mathbf{x})$, creates a table which for each suffix gives the location of its other occurrences in $\mathbf{x}$. In the example in Fig. 8.8, the suffix `s` (the last character in "`estimates`") also occurs at position 2 in $\mathbf{x}$. Further, the suffix "`es`" occurs at position 1 in $\mathbf{x}$. The suffix "`tes`" does not appear elsewhere in $\mathbf{x}$ and hence it, and all other suffixes, are assigned the value 0. In sum, then, the table of $\mathcal{G}(\mathbf{x})$ would have just two non-zero entries: `s`, 2 and `es`, 1.

GOOD-SUFFIX FUNCTION

In practice, these heuristics make the Boyer-Moore one of the most attractive string-matching algorithms on serial computers. Other powerful methods quickly become conceptually more involved and are generally based on precomputing functions of $\mathbf{x}$ that enable efficient shift increments, or dividing the problem for efficient parallel computation.

Many applications require a *text* to be searched for several strings, as in the case of keyword search through a digital text. Occasionally, some of these search strings are themselves factors of other search strings. Presumably we would not want to acknowledge a match of a short string if it were also part of a match for a longer string. Thus if our keywords included "`beat`," "`eat`," and "`be`," we would want our search to return only the string match of "`beat`" from *text* = "`when_chris_beats_the_drum`," not the shorter strings "`eat`" and "`be`," which are nevertheless "there" in *text*. This is an example of the *subset-superset problem*. Although there may be much bookkeeping associated with imposing such a strict bias for longer sequences over shorter ones, the approach is conceptually straightforward (Computer exercise 9).

SUBSET-SUPERSET PROBLEM

8.5.2 Edit distance

The fundamental idea underlying pattern recognition using edit distance is based on the nearest-neighbor algorithm (Chap. **??**). We store a full training set of strings and their associated category labels. During classification, a test string is compared to each stored string and a "distance" or score is computed; the test string is then assigned the category label of the "nearest" string in the training set.

Unlike the case using real-valued vectors discussed in Chap. **??**, there is no single obvious measure of the similarity or difference between two strings. For instance, it is not clear whether "`abbccc`" is closer to "`aabbcc`" or to "`abbcccb`." To proceed, then, we introduce a measure of the difference between two strings. Such an *edit distance* between $\mathbf{x}$ and $\mathbf{y}$ describes how many fundamental operations are required to transform $\mathbf{x}$ into $\mathbf{y}$. These fundamental operations are:

substitutions: A character in $\mathbf{x}$ is replaced by the corresponding character in $\mathbf{y}$.

insertions: A character in $\mathbf{y}$ is inserted into $\mathbf{x}$, thereby increasing the length of $\mathbf{x}$ by one character.

deletions: A character in $\mathbf{x}$ is deleted, thereby decreasing the length of $\mathbf{x}$ by one character.

INTERCHANGE Occasionally we also consider a fourth operation, *interchange*, or “twiddle,” or *transposition*, which interchanges two neighboring characters in $\mathbf{x}$. Thus, one could transform $\mathbf{x}$ = “<u>as</u>p” into $\mathbf{y}$ = “<u>sa</u>p” with a single interchange. Because such an interchange can always be expressed as two substitutions, for simplicity we shall not consider interchanges.

Let $\mathbf{C}$ be an $m \times n$ matrix of integers associated with a cost or “distance” and let $\delta(\cdot, \cdot)$ denote a generalization of the Kronecker delta function, having value 1 if the two arguments (characters) match and 0 otherwise. The basic edit-distance algorithm is then:

Algorithm 3 (Edit distance)

```
1  begin initialize $\mathcal{A}, \mathbf{x}, \mathbf{y}, m \leftarrow \text{length}[\mathbf{x}], n \leftarrow \text{length}[\mathbf{y}]$
2          $\mathbf{C}[0,0] \leftarrow 0$
3          $i \leftarrow 0$
4          do $i \leftarrow i+1$
5              $\mathbf{C}[i,0] \leftarrow i$
6          until $i = m$
7          $j \leftarrow 0$
8          do $j \leftarrow j+1$
9              $\mathbf{C}[0,j] \leftarrow j$
10         until $j = n$
11         $i \leftarrow 0;\ j \leftarrow 0$
12         do $i \leftarrow i+1$
13             do $j \leftarrow j+1$
14                 $\mathbf{C}[i,j] = \min\big[\underbrace{\mathbf{C}[i-1,j]+1}_{insertion}, \underbrace{\mathbf{C}[i,j-1]+1}_{deletion}, \underbrace{\mathbf{C}[i-1,j-1]+1-\delta(\mathbf{x}[i],\mathbf{y}[j])}_{no\ change/exchange}\big]$
15             until $j = n$
16         until $i = m$
17    return $\mathbf{C}[m,n]$
18 end
```

Lines 4 – 10 initialize the left column and top row of $\mathbf{C}$ with the integer number of “steps” away from $i = 0, j = 0$. The core of this algorithm, line 14, finds the minimum cost in each entry of $\mathbf{C}$, column by column (Fig. 8.9). Algorithm 3 is thus greedy in that each column of the distance or cost matrix is filled using merely the costs in the previous column. Linear programming techniques can also be used to find a global minimum, though this nearly always requires greater computational effort (Problem 27).

If insertions and deletions are equally costly, then the symmetry property of a metric holds. However, we can broaden the applicability of the algorithm by allowing in line 14 different costs for the fundamental operations; for example insertions might cost twice as much as substitutions. In such a broader case, properties of symmetry and the triangle inequality no longer hold and edit distance is not a true metric (Problem 28).

As shown in Fig. 8.9, $\mathbf{x}$ = “`excused`” can be transformed to $\mathbf{y}$ = “`exhausted`” through one substitution and two insertions. The table shows the steps of this transformation, along with the computed entries of the cost matrix $\mathbf{C}$. For the case shown, where each fundamental operation has a cost of 1, the edit distance is given by the value of the cost matrix at the sink, i.e., $\mathbf{C}[7,9] = 3$.

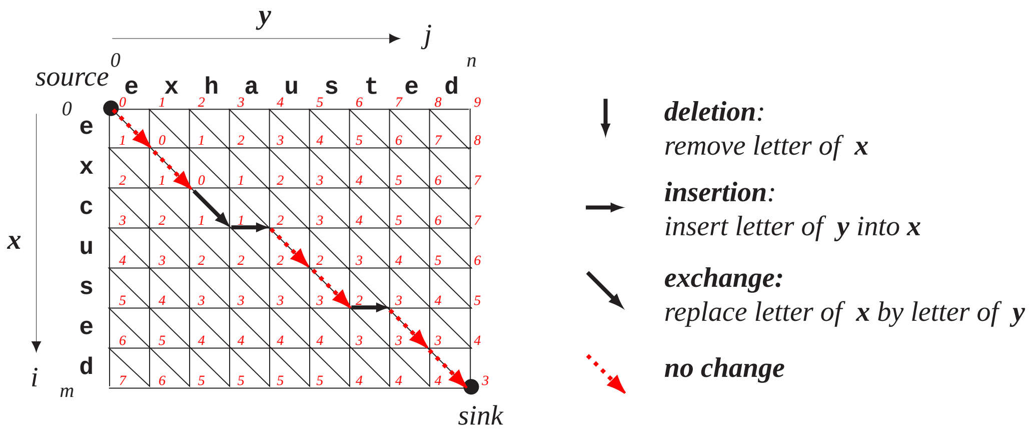


Figure 8.9: The edit distance calculation for strings **x** and **y** can be illustrated in a table. Algorithm 3 begins at *source*, $i = 0, j = 0$, and fills in the cost matrix **C**, column by column (shown in red), until the full edit distance is placed at the *sink*, $\mathbf{C}[i = m, j = n]$. The edit distance between "`excused`" and "`exhausted`" is thus 3.

x	`excused`	source string	$\mathbf{C}[0,0] = 0$
	`exhused`	substitute `h` for `c`	$\mathbf{C}[3,3] = 1$
	`exhaused`	insert `a`	$\mathbf{C}[3,4] = 2$
	`exhausted`	insert `t`	$\mathbf{C}[5,7] = 3$
y	`exhausted`	target string	$\mathbf{C}[7,9] = 3$

8.5.3 Computational complexity

Algorithm 3 is $\mathcal{O}(mn)$ in time, of course; it is $\mathcal{O}(m)$ in space (memory) since only the entries in the previous column need be stored when computing $\mathbf{C}[i, j]$ for $i = 0$ to m. Because of the importance of string matching and edit distance throughout computer science, a number of algorithms have been proposed. We need not delve into the details here (but see the Bibliography) except to say that there are sophisticated string-matching algorithms with time complexity $\mathcal{O}(m + n)$.

8.5.4 String matching with errors

There are several versions of the *string-matching-with-errors problem*; the one that concerns us is this: given a pattern **x** and *text*, find the shift for which the edit distance between **x** and a factor of *text* is minimum. The algorithm for the string-matching-with-errors problem is very similar to that for edit distance. Let **E** be a matrix of costs, analogous to **C** in Algorithm 3. We seek a shift for which the edit distance to a factor of *text* is minimum, or formally $\min[\mathbf{C}(\mathbf{x}, \mathbf{y})]$ where **y** is any factor of *text*. To this end, the algorithm must compute its new cost **E** whose entries are $\mathbf{E}[i, j] = \min[\mathbf{C}(\mathbf{x}[1...i], \mathbf{y}[1...j])]$.

The principal difference between the algorithms for the two problems (i.e., with or without errors) is that we initialize **E**[0,j] to 0 in the string matching with errors problem, instead of to j in lines $4 - 10$ of the basic string matching algorithm. This initialization of **E** expresses the fact that the "empty" prefix of **x** matches an empty factor of *text*, and contributes no cost.

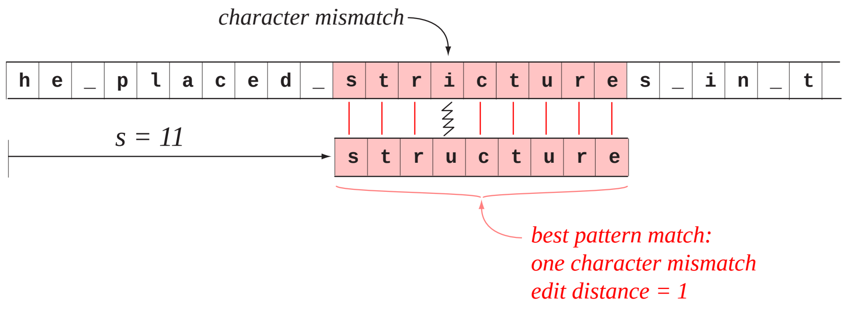


Figure 8.10: The string matching with errors problem is to find the shift s for which the edit distance between $\mathbf{x}$ and an aligned factor of *text* is minimum. In this illustration, the minimum edit distance is 1, corresponding to the character exchange $\texttt{u} \rightarrow \texttt{i}$ and the shift $s = 11$ is the location.

Two minor heuristics for reducing computational effort are relevant to the string-matching-with-errors problem. The first is that except in highly unusual cases, the length of the candidate factors of *text* that need be considered are roughly equal to length[$\mathbf{x}$]. Second, for each candidate shift, the edit-distance calculation can be terminated if it already exceeds the current minimum. In practice, this latter heuristic can reduce the computational burden significantly. Otherwise, the algorithm for string matching with errors is virtually the same as that for edit distance (Computer exercise 10).

8.5.5 String matching with the "don't-care" symbol

String matching with the "don't-care" symbol, $\emptyset$, is formally the same as basic string matching, but the $\emptyset$ in either $\mathbf{x}$ or *text* is said to match any character (Fig. 8.11).

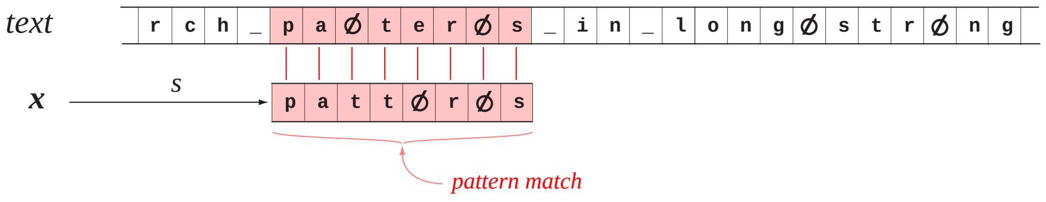


Figure 8.11: String matching with don't care symbol is the same as basic string matching except the $\emptyset$ symbol — in either *text* or $\mathbf{x}$ — matches any character. The figure shows the only valid shift.

An obvious approach to string matching with the don't care symbol is to modify the naive string-matching algorithm to include a condition for matching the don't care symbol. Such an approach, however, retains the computational inefficiencies of naive string matching (Problem 29). Further, extending the Boyer-Moore algorithm to include $\emptyset$ is somewhat difficult and inefficient. The most effective methods are based on fundamental methods in computer arithmetic and, while fascinating, would take us away from our central concerns of pattern recognition (cf. Bibliography). The use of this technique in pattern recognition is the same as string matching, with a particular type of "tolerance."

While learning is a general and fundamental technique throughout pattern recognition, it has found limited use in recognition with basic string matching. This is because the designer typically knows precisely which strings are being sought — they do not need to be learned. Learning can, of course, be based on the *outputs* of a string-matching algorithm, as part of a larger pattern recognition system.

8.6 Grammatical methods

Up to here, we have not been concerned with any detailed models that might underly the generation of the sequence of characters in a string. We now turn to the case where rules of a particular sort were used to generate the strings and thus where their structure is fundamental. Often this structure is hierarchical, where at the highest or most abstract level a sequence is very simple, but at subsequent levels there is greater and greater complexity. For instance, at its most abstract level, the string "`The history book clearly describes several wars`" is merely a sentence. At a somewhat more detailed level it can be described as a noun phrase followed by a verb phrase. The noun phrase can be expanded at yet a subsequent level, as can the verb phrase. The expansion ends when we reach the words "`The`," "`history`," and so forth — items that are considered the "characters," atomic and without further structure. Consider too strings representing valid telephone numbers — local, national and international. Such numbers conform to a strict structure: either a country code is present or it is not; if not, then the domestic national code may or may not be present; if a country code is present, then there is a set of permissible city codes and for each city there is a set of permissible area codes and individual local numbers, and so on.

As we shall see, such structure is easily specified in a *grammar*, and when such structure is present the use of a grammar for recognition can improve accuracy. For instance, grammatical methods can be used to provide constraints for a full system that uses a statistical recognizer as a component. Consider an optical character recognition system that recognizes and interprets mathematical equations based on a scanned pixel image. The mathematical symbols often have specific "slots" that can be filled with certain other symbols; this can be specified by a grammar. Thus an integral sign has two slots, for upper and lower limits, and these can be filled by only a limited set of symbols. (Indeed, a grammar is used in many mathematical typesetting programs in order to prevent authors from creating meaningless "equations.") A full system that recognizes the integral sign could use a grammar to limit the number of candidate categories for a particular slot, and this increases the accuracy of the full system. Similarly, consider the problem of recognizing phone numbers within acoustic speech in an automatic dialing application. A statistical or Hidden-Markov-Model acoustic recognizer might perform word spotting and pick out number words such as "eight" and "hundred." A subsequent stage based on a formal grammar would then exploit the fact that telephone numbers are highly constrained, as mentioned.

We shall study the case when crisp rules specify how the representation at one level leads to a more expanded and complicated representation at the next level. We sometimes call a string generated by a set of rules a *sentence*; the rules are specified by a grammar, denoted G. (Naturally, there is no requirement that these be related in any way to sentences in natural language such as English.) In pattern recognition, we are given a sentence and a grammar, and seek to determine whether the sentence was generated by G.

SENTENCE

8.6.1 Grammars

The notion of a grammar is very general and powerful. Formally, a grammar G consists of four components:

NULL STRING

symbols: Every sentence consists of a string of characters (which are also called primitive symbols, terminal symbols or letters), taken from an alphabet $\mathcal{A}$. For bookkeeping, it is also convenient to include the *null* or *empty string* denoted ϵ, which has length zero; if ϵ is appended to any string $\mathbf{x}$, the result is again $\mathbf{x}$.

variables: These are also called non-terminal symbols, intermediate symbols or occasionally internal symbols, and are taken from a set $\mathcal{I}$.

ROOT SYMBOL

root symbol: The *root symbol* or starting symbol is a special internal symbol, the source from which all sequences are derived. The root symbol is taken from a set $\mathcal{S}$.

PRODUCTION RULE

productions: The set of *production rules*, rewrite rules, or simply rules, denoted $\mathcal{P}$, specify how to transform a set of variables and symbols into other variables and symbols. These rules determine the core structures that can be produced by the grammar. For instance if A is an internal symbol and $\mathtt{c}$ a terminal symbol, the rewrite rule $\mathtt{c}A \rightarrow \mathtt{cc}$ means that any time the segment $\mathtt{c}A$ appears in a string, it can be replaced by $\mathtt{cc}$.

LANGUAGE

Thus we denote a general grammar by its alphabet, its variables, its particular root symbol, and the rewrite rules: $G = (\mathcal{A}, \mathcal{I}, \mathcal{S}, \mathcal{P})$. The *language* generated by grammar, denoted $\mathcal{L}(G)$, is the set of all strings (possibly infinite in number) that can be generated by G.

Consider two examples; the first is quite simple and abstract. Let $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$,

$$\mathcal{S} = S, \mathcal{I} = \{A, B, C\}, \text{and } \mathcal{P} = \left\{ \begin{array}{ll} \mathbf{p}_1\text{: } S \rightarrow \mathtt{a}SBA \; OR \; \mathtt{a}BA & \mathbf{p}_2\text{: } AB \rightarrow BA \\ \mathbf{p}_3\text{: } \mathtt{b}B \rightarrow \mathtt{bb} & \mathbf{p}_4\text{: } \mathtt{b}A \rightarrow \mathtt{bc} \\ \mathbf{p}_5\text{: } \mathtt{c}A \rightarrow \mathtt{cc} & \mathbf{p}_6\text{: } \mathtt{a}B \rightarrow \mathtt{ab} \end{array} \right\}.$$

(In order to make the list of rewrite rules more compact, we shall condense rules having the same left hand side by means of the OR on the right hand side. Thus rule $\mathbf{p}_1$ is a condensation of the two rules $S \rightarrow \mathtt{a}SBA$ and $S \rightarrow \mathtt{a}BA$.) If we start with S and apply the rewrite rules in the following orders, we have the following two cases:

root	S
$\mathbf{p}_1$	$\mathtt{a}BA$
$\mathbf{p}_6$	$\mathtt{ab}A$
$\mathbf{p}_4$	abc

root	S
$\mathbf{p}_1$	aSBA
$\mathbf{p}_1$	aaBABA
$\mathbf{p}_6$	aabABA
$\mathbf{p}_2$	aabBAA
$\mathbf{p}_3$	aabbAA
$\mathbf{p}_4$	aabbcA
$\mathbf{p}_5$	aabbcc

PRODUCTION

After the rewrite rules have been applied in these sequences, no more symbols match the left-hand side of any rewrite rule, and the process is complete. Such a transformation from the root symbol to a final string is called a *production*. These two productions show that abc and aabbcc are in the language generated by G. In fact, it can be shown (Problem 38) that this grammar generates the language $\mathcal{L}(G) = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n | n \geq 1\}$.

A much more complicated grammar underlies the English language, of course. The alphabet consists of all English words, $\mathcal{A} = \{$`the`, `history`, `book`, `sold`, `over`, `1000`, `copies`, $\ldots\}$, and the intermediate symbols are the parts of speech: $\mathcal{I} = \{\langle noun\rangle, \langle verb\rangle, \langle noun\ phrase\rangle, \langle verb\ phrase\rangle, \langle adjective\rangle, \langle adverb\rangle, \langle adverbial\ phrase\rangle\}$. The root symbol here is $\mathcal{S} = \langle sentence\rangle$. A restricted set of the production rules in English includes:

$$\mathcal{P} = \left\{ \begin{array}{r} \langle sentence\rangle \rightarrow \langle noun\ phrase\rangle\langle verb\ phrase\rangle \\ \langle noun\ phrase\rangle \rightarrow \langle adjective\rangle\langle noun\ phrase\rangle \\ \langle verb\ phrase\rangle \rightarrow \langle verb\ phrase\rangle\langle adverbial\ phrase\rangle \\ \langle noun\rangle \rightarrow \texttt{book}\ OR\ \texttt{theorem}\ OR\ \ldots \\ \langle verb\rangle \rightarrow \texttt{describes}\ OR\ \texttt{buys}\ OR\ \texttt{holds}\ OR\ \ldots \\ \langle adverb\rangle \rightarrow \texttt{over}\ OR\ \ldots \end{array} \right\}$$

This subset of the rules of English grammar does not prevent the generation of meaningless sentences, of course. For instance, the nonsense sentence "`Squishy green dreams hop heuristically`" can be derived in this subset of English grammar. Figure 8.12 shows the steps of a production in a *derivation tree*, where the root symbol is displayed at the top and the terminal symbols at the bottom.

DERIVATION TREE

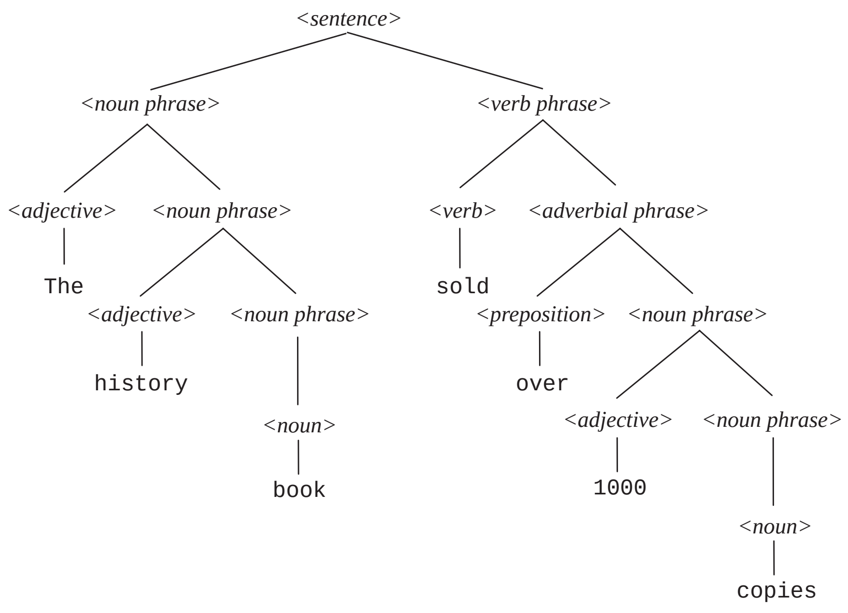


Figure 8.12: This derivation tree illustrates how a portion of English grammar can transform the root symbol, here $\langle sentence\rangle$, into a particular sentence or string of elements, here English words, which are read from left to right.

8.6.2 Types of string grammars

There are four main types of grammar, arising from different types of structure in the productions. As we have seen, a rewrite rule is of the form $\alpha \rightarrow \beta$, where α and β are strings made up of intermediate and terminal symbols.

Type 0: Free or unrestricted Free grammars have no restrictions on the rewrite rules and thus they provide no constraints or structure on the strings they can produce. While in principle they can express an arbitrary set of rules, this generality comes at the tremendous expense of possibly unbounded learning time. Knowing that a string is derived from a type 0 grammar provides no information and as such, type 0 grammars in general have but little use in pattern recognition.

Type 1: Context-sensitive A grammar is called context-sensitive if every rewrite rule is of the form

$$\alpha I \beta \rightarrow \alpha x \beta$$

where α and β are any strings made up of intermediate and terminal symbols, I is an intermediate symbol and x is an intermediate or terminal symbol (other than ϵ). We say that "I can be rewritten as x in the context of α on the left and β on the right."

Type 2: Context-free A grammar is called context free if every production is of the form

$$I \rightarrow x$$

where I is an intermediate symbol and x an intermediate or terminal symbol (other than ϵ). Clearly, unlike a type 1 grammar, here there is no need for a "context" for the rewriting of I by x.

Type 3: Finite State or Regular A grammar is called regular if every rewrite rule is of the form

$$\alpha \rightarrow z\beta \qquad OR \qquad \alpha \rightarrow z$$

where α and β are made up of intermediate symbols and z is a terminal symbol (other than ϵ). Such grammars are also called finite state because they can be generated by a finite state machine, which we shall see in Fig. 8.16.

A language generated by a grammar of type i is called a type i language. It can be shown that the class of grammars of type i includes all grammars of type $i+1$; thus there is a strict hierarchy in grammars.

Chomsky normal form

Any context-free grammar can be converted into one in *Chomsky normal form* (CNF). Such a grammar has all rules of the form

$$A \rightarrow BC \qquad \text{and} \qquad A \rightarrow z$$

where A, B and C are intermediate symbols (that is, they are in $\mathcal{I}$) and z is a terminal symbol. For every context-free grammar G, there is another G' in Chomsky normal form such that $\mathcal{L}(G) = \mathcal{L}(G')$ (Problem 36).

Example 3: A grammar for pronouncing numbers

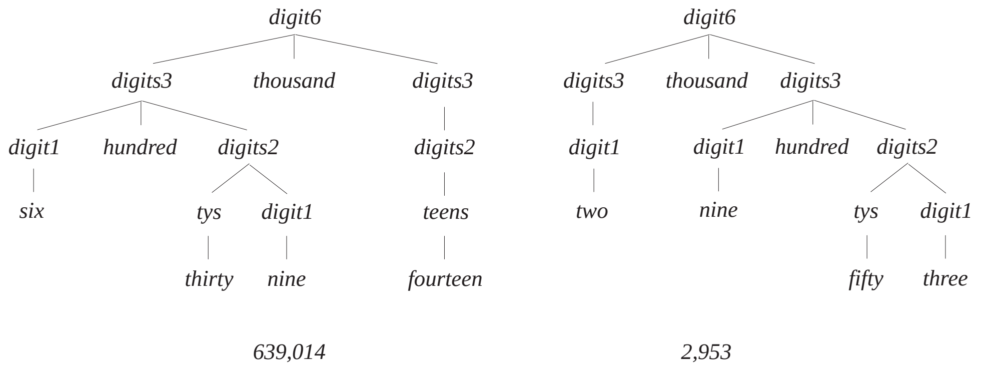
In order to understand these issues better, consider a grammar that yields pronunciation of any number between 1 and 999,999. The alphabet has 29 basic terminal symbols, i.e., the spoken words
$\mathcal{A} = \{$*one*, *two*, …, *ten*, *eleven*, …, *twenty*, *thirty*, …, *ninety*, *hundred*, *thousand*$\}$.

There are six non-terminal symbols, corresponding to general six-digit, three-digit, and two-digit numbers, the numbers between ten and nineteen, and so forth, as will be clear below:
$\mathcal{I} = \{digits6,\ digits3,\ digits2,\ digit1,\ teens,\ tys\}$.
The root node corresponds to a general number up to six digits in length:
$\mathcal{S} = digits6$.

The set of rewrite rules is based on a knowledge of English:

$$\mathcal{P} = \left\{ \begin{array}{r} digits6 \rightarrow digits3\ thousand\ digits3 \\ digits6 \rightarrow digits3\ thousand\ OR\ digits3 \\ digits3 \rightarrow digit1\ hundred\ digits2 \\ digits3 \rightarrow digit1\ hundred\ OR\ digits2 \\ digits2 \rightarrow teens\ OR\ tys\ OR\ tys\ digit1\ OR\ digit1 \\ digit1 \rightarrow one\ OR\ two\ OR\ \ldots\ nine \\ teens \rightarrow ten\ OR\ eleven\ OR\ \ldots\ nineteen \\ tys \rightarrow twenty\ OR\ thirty\ OR\ \ldots\ OR\ ninety \end{array} \right\}$$

The grammar takes *digit*6 and applies the productions until the elements in the final alphabet are produced, as shown in the figure. Because it contains rewrite rules such as $digits6 \rightarrow digits3\ thousand$, this grammar cannot be type 3. It is easy to confirm that it is type 2.



These two derivation trees show how the grammar G yields the pronunciation of 639,014 and 2,953. The final string of terminal symbols is read from left to right.

8.6.3 Recognition using grammars

Recognition using grammars is formally very similar to the general approaches used throughout pattern recognition. Suppose we suspect that a test sentence was generated by one of c different grammars, $G_1, G_2, \ldots, G_c$, which can be considered as different models or classes. A test sentence $\mathbf{x}$ is classified according to which grammar could have produced it, or equivalently, the language $\mathcal{L}(G_i)$ of which $\mathbf{x}$ is a member.

Up to now we have worked forward — forming a derivation from a root node to a final sentence. For recognition, though, we must employ the inverse process: that is, given a particular $\mathbf{x}$, find a derivation in G that leads to $\mathbf{x}$. This process, called *parsing*, is virtually always much more difficult than forming a derivation. We now PARSING discuss one general approach to parsing, and briefly mention two others.

Bottom-up parsing

Bottom-up parsing starts with the test sentence $\mathbf{x}$, and seeks to simplify it, so as to represent it as the root symbol. The basic approach is to use candidate productions from $\mathcal{P}$ "backwards," i.e., find rewrite rules whose right hand side matches part of the current string, and replace that part with a segment that could have produced it. This is the general method in the Cocke-Younger-Kasami algorithm, which fills a *parse table* from the "bottom up." The grammar must be expressed in Chomsky normal form and thus the productions $\mathcal{P}$ must all be of the form $A \rightarrow BC$, a broad but not all inclusive category of grammars. Entries in the table are candidate strings in a portion of a valid derivation. If the table contains the source symbol S, then indeed we can work forward from S and derive the test sentence, and hence $\mathbf{x} \in \mathcal{L}(G)$. In the following, x_i (for $i = 1, \ldots n$) represents the individual terminal characters in the string to be parsed.

PARSE TABLE

Algorithm 4 (Bottom-up parsing)

```
1  begin initialize $G = (\mathcal{A}, \mathcal{I}, S, \mathcal{P}), \mathbf{x} = x_1 x_2 \ldots x_n$
2        $i \leftarrow 0$
3        do $i \leftarrow i + 1$
4           $V_{i1} \leftarrow \{A \mid A \rightarrow x_i\}$
5        until $i = n$
6        $j \leftarrow 1$
7        do $j \leftarrow j + 1$
8           $i \leftarrow 0$
9           do $i \leftarrow i + 1$
10             $V_{ij} \leftarrow \emptyset$
11             $k \leftarrow 0$
12             do $k \leftarrow k + 1$
13                $V_{ij} \leftarrow V_{ij} \cup \{A \mid A \rightarrow BC \in \mathcal{P}, B \in V_{ik}$ and $C \in V_{i+k,j-k}\}$
14             until $k = j - 1$
15          until $i = n - j + 1$
16       until $j = n$
17       if $S \in V_{1n}$ then print "parse of" $\mathbf{x}$ "successful in $G$"
18    return
19 end
```

Consider the operation of Algorithm 4 in the following simple abstract example. Let the grammar G have two terminal and three intermediate symbols: $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$, and $\mathcal{I} = \{A, B, C\}$. The root symbol is S, and there are just four production rules:

$$\mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & S & \rightarrow AB\ OR\ BC \\ \mathbf{p}_2: & A & \rightarrow BA\ OR\ \mathtt{a} \\ \mathbf{p}_3: & B & \rightarrow CC\ OR\ \mathtt{b} \\ \mathbf{p}_4: & C & \rightarrow AB\ OR\ \mathtt{a} \end{array} \right\}.$$

Figure 8.13 shows the parse table generated by Algorithm 4 for the input string $\mathbf{x}$ = "baaba." Along the bottom are the characters x_i of this string. Lines 2 through 5 of the algorithm fill in the first ($j = 1$) row with any internal symbols that derive the corresponding character in $\mathbf{x}$. The $i = 1$ and $i = 4$ entries of that bottom row are filled with B, since rewrite rule $\mathbf{p}_3$: $B \rightarrow \mathtt{b}$. Likewise the remaining entries are filled with both A and C, as a result of rewrite rules $\mathbf{p}_2$ and $\mathbf{p}_4$.

The core computation in the algorithm is performed in line 13, which fills entries throughout the table with symbols that could produce segments in lower rows, and hence might be part of a valid derivation (if indeed one is found). For instance, the $i = 1, j = 2$ entries contain any symbols that could produce segments in the row beneath it. Thus this entry contains S because by rule $\mathbf{p}_1$: $S \rightarrow BC$, and also contains A because by rule $\mathbf{p}_2$: $A \rightarrow BA$. According to the innermost loop over k (lines 12 – 14), we seek the left hand side for rules that span a range. For instance, the $i = 3, j = 3$ entry contains B because for $k = 2$ and rule $\mathbf{p}_3$: $B \rightarrow CC$ (as shown in Fig. 8.14).

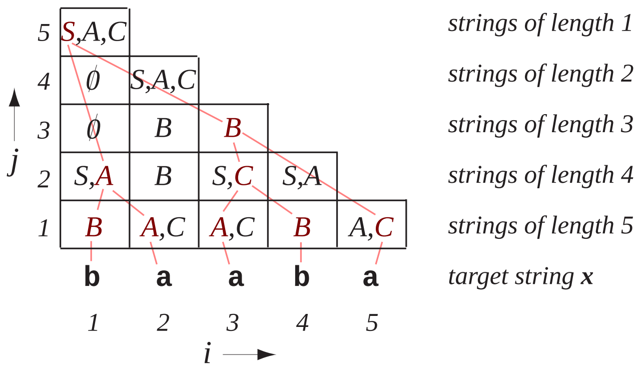


Figure 8.13: The bottom-up parsing algorithm fills the parse table with symbols that might be part of a valid derivation. The pink lines are not provided by the algorithm, but when read downward from the root symbol confirm that a valid derivation exists.

Figure 8.14 shows the cells that are searched when filling a particular cell in the parse table. The sequence sweeps vertically up to the cell in question, while diagonally down from the cell in question; this guarantees that the all paths from the top cell in a valid derivation can be found. If the top cell contains the root symbol S (and possibly other symbols), then indeed the string is successfully parsed. That is, there exists a valid production leading from S to the target string $\mathbf{x}$.

To understand how this table is filled, consider first the $j = 1$ row. The $j = 4, i = 1$ cell contains B, because according to rewrite rule $\mathbf{p}_3$, B is the only intermediate symbol that could yield `b` in the query sentence, directly below. The same logic holds for the $i = 1, j = 1$ cell. The remaining three cells for $j = 1$ contain A and C, since these are the only intermediate variables that can derive **a**. Incidentally, the derivation in Fig. 8.15 confirms that the parse is valid.

The computational complexity of bottom-up parsing performed by Algorithm 4 is high. The innermost loop of line 13 is executed n or fewer times, while lines 7 & 9 are $\mathcal{O}(n^2)$, which is also the space complexity. The time complexity is $\mathcal{O}(n^3)$.

Top-down and other methods of parsing

As its name suggests, top-down parsing starts with the root node and successively applies productions from $\mathcal{P}$, with the goal of finding a derivation of the test sentence $\mathbf{x}$. Since it is rare that the sentence is derived in a single production, it is necessary to specify some criteria to guide the choice of which rewrite rule to apply. Such criteria could include beginning the parse at the first (left) character in the sentence (that is, finding a small set of rewrite rules that yield the first character), then iteratively expanding the production to derive subsequent characters.

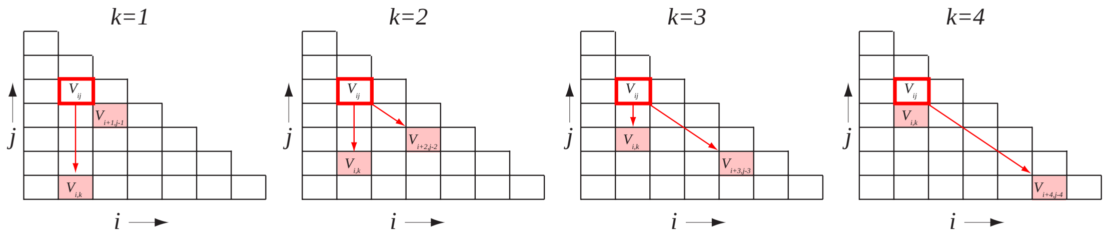


Figure 8.14: The innermost loop of Algorithm 4 seeks to fill a cell V_{ij} (outlined in red) by the left-hand side of any rewrite rule whose right-hand side corresponds to symbols in the two shaded cells. As k is incremented, the cells queried move vertically upward to the cell in question, and diagonally down from that cell. The shaded cells show the possible right-hand sides in a derivation, as illustrated by the pink lines in Fig. 8.13.

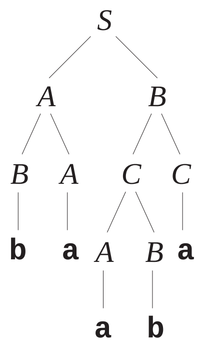


Figure 8.15: This valid derivation of "`babaa`" in G can be read from the pink lines in the parse table of Fig. 8.13 generated by the bottom-up parse algorithm.

The bottom-up and top-down parsers just described are quite general and there are a number of parsing algorithms which differ in space and time complexities. Many parsing methods depend upon the model underlying the grammar. One popular such model is *finite state machines.* Such a machine consists of nodes and transition links; each node can emit a symbol, as shown in Fig. 8.16.

FINITE STATE MACHINE

8.7 Grammatical inference

In many applications, the grammar is designed by hand. Nevertheless, learning plays an extremely important role in pattern recognition research and it is natural that we attempt to learn a grammar from example sentences it generates. When seeking to follow that general approach we are immediately struck by differences between the areas addressed by grammatical methods and those that can be described as statistical. First, for most languages there are many — often an infinite number of — grammars that can produce it. If two grammars G_1 and G_2 generate the same language (and no other sentences), then this ambiguity is of no consequence; recognition will be the same. However, since training is always based on a *finite* set of samples, the problem is underspecified. There are an infinite number of grammars consistent with the training data, and thus we cannot recover the source grammar uniquely.

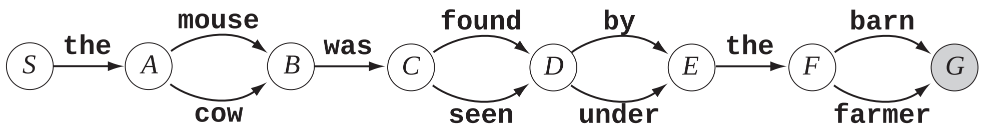


Figure 8.16: One type of finite state machine consists of nodes that can emit terminal symbols ("the," "mouse," etc.) and transition to another node. Such operation can be described by a grammar. For instance, the rewrite rules for this finite state machine include $S \rightarrow \texttt{the}A$, $A \rightarrow \texttt{mouse}B$ OR $\texttt{cow}B$, and so on. Clearly these rules imply this finite state machine implements a type 3 grammar. The final internal node (shaded) would lead to the null symbol ϵ.

There are two main techniques used to make the problem of inferring a grammar from instances tractable. The first is to use both *positive* and *negative* instances. That is, we use a set $\mathcal{D}^+$ of sentences known to be derivable in the grammar; we also use a set $\mathcal{D}^-$ that are known to be *not* derivable in the grammar. In a multicategory case, it is common to take the positive instances in G_i and use them for negative examples in G_j for $j \neq i$. Even with both positive and negative instances, a finite training set rarely specifies the grammar uniquely. Thus our second technique is to impose conditions and constraints. A trivial illustration is that we demand that the alphabet of the candidate grammar contain only those symbols that appear in the training sentences. Moreover, we demand that every production rule in the grammar be used. We seek the "simplest" grammar that explains the training instances where "simple" generally refers to the total number of rewrite rules, or the sum of their lengths, or other natural criterion. These are versions of Occam's razor, that the simplest explanation of the data is to be preferred (Chap **??**).

In broad overview, learning proceeds as follows. An initial grammar G^0 is guessed. Often it is useful to specify the type of grammar (1, 2 or 3), and thus place constraints on the forms of the candidate rewrite rules. In the absence of other prior information, it is traditional to make G^0 as simple as possible and gradually expand the set of productions as needed. Positive training sentences $\mathbf{x}_i^+$ are selected from $\mathcal{D}^+$ one by one. If $\mathbf{x}_i^+$ cannot be parsed by the grammar, then new rewrite rules are proposed for $\mathcal{P}$. A new rule is accepted if and only if it is used for a successful parse of $\mathbf{x}_i^+$ *and* does not allow any negative samples to be parsed.

In greater detail, then, an algorithm for inferring the grammar is:

Algorithm 5 (Grammatical inference (overview))

```
1 begin initialize $\mathcal{D}^+, \mathcal{D}^-, G^0$
2         $n^+ \leftarrow |\mathcal{D}^+|$  (number of instances in $\mathcal{D}^+$)
3         $\mathcal{S} \leftarrow S$
4         $\mathcal{A} \leftarrow$ set of characters in $\mathcal{D}^+$
5         $i \leftarrow 0$
6         do $i \leftarrow i + 1$
7             read $\mathbf{x}_i^+$ from $\mathcal{D}^+$
8             if $\mathbf{x}_i^+$ cannot be parsed by $G$
9                 then do propose additional productions to $\mathcal{P}$ and variables to $\mathcal{I}$
10                        accept updates if $G$ parses $\mathbf{x}_i^+$ but no string in $\mathcal{D}^-$
11            until $i = n^+$
12            eliminate redundant productions
```

```
13          return $G \leftarrow \{\mathcal{A}, \mathcal{I}, \mathcal{S}, \mathcal{P}\}$
14    end
```

Informally, Algorithm 5 continually adds new rewrite rules as required by the successive sentences selected from $\mathcal{D}^+$ so long as the candidate rewrite rule does not allow a sentence in $\mathcal{D}^-$ to be parsed. Line 9 does not state how to choose the specific candidate rewrite rule, but in practice the rule may be chosen from a predefined set (with simpler rules selected first), or based on specific knowledge of the underlying models generating the sentences.

Example 4: Grammar inference

Consider inferring a grammar G from the following positive and negative examples: $\mathcal{D}^+ = \{\texttt{a}, \texttt{aaa}, \texttt{aaab}, \texttt{aab}\}$, and $\mathcal{D}^- = \{\texttt{ab}, \texttt{abc}, \texttt{abb}, \texttt{aabb}\}$. Clearly the alphabet of G is $\mathcal{A} = \{\texttt{a}, \texttt{b}\}$. We posit a single internal symbol for G^0, and the simplest rewrite rule $\mathcal{P} = \{S \to A\}$.

i	$\mathbf{x}_i^+$	$\mathcal{P}$	$\mathcal{P}$ produces $\mathcal{D}^-$?
1	a	$S \to A$ $A \to \texttt{a}$	No
2	aaa	$S \to A$ $A \to \texttt{a}$ $A \to \texttt{a}A$	No
3	aaab	$S \to A$ $A \to \texttt{a}$ $A \to \texttt{a}A$ $A \to \texttt{ab}$	Yes: $\texttt{ab} \in \mathcal{D}^-$
3	aaab	$S \to A$ $A \to \texttt{a}$ $A \to \texttt{a}A$ $A \to \texttt{aab}$	No
4	aab	$S \to A$ $A \to \texttt{a}$ $A \to \texttt{a}A$ $A \to \texttt{aab}$	No

The table shows the progress of the algorithm. The first positive instance, a, demands a rewrite rule $A \to \texttt{a}$. This rule does not allow any sentences in $\mathcal{D}^-$ to be derived, and thus is accepted for $\mathcal{P}$. When $i = 3$, the proposed rule $A \to \texttt{ab}$ indeed allows $\mathbf{x}_3^+$ to be derived, but the rule is rejected because it also derives a sentence in $\mathcal{D}^-$. Instead, the next proposed rule, $A \to \texttt{aab}$ is accepted. The final grammar inferred has the four rewrite rules shown at the bottom of the table.

The method of grammatical inference just described is quite general. It is made more specialized by placing restrictions on the types of candidate rewrite rules, corresponding to the designer's assumptions about the type of grammar (1, 2 or 3). For a type 3 grammar, we can consider learning in terms of the finite state machine. In that case, learning consists of adding nodes and links (cf. Bibliography).

8.8 *Rule-based methods

In problems where classes can be characterized by general *relationships* among entities, rather than by instances per se, it becomes attractive to build classifiers based on rules. Rule-based methods are integral to expert systems in artificial intelligence, but since they have found only modest use in pattern recognition, we shall give merely a short overview. We shall focus on a broad class of *if-then rules* for representing and learning such relationships.

IF-THEN RULE

A very simple if-then rule is

$$IF\ \texttt{Swims(x)}\ AND\ \texttt{HasScales(x)}\ THEN\ \texttt{Fish(x)},$$

which means, of course, that if an object `x` has the property that it swims, and the property that it has scales, then it is a fish. Rules have the great benefits that they are easily interpreted and can be used in database applications where information is encoded in relations. A drawback is that there is no natural notion of probability and it is somewhat difficult, therefore, to use rules when there is high noise and a large Bayes error.

PREDICATE

A *predicate*, such as `Man(·)`, `HasTeeth(·)` and `AreMarried(·,·)`, is a test that returns a value of logical `True` or `False`.[*] Such predicates can apply to problems where the data are numerical non-numerical, linguistic, strings, or any of a broad class of types. The choice of predicates and their evaluation depend strongly on the problem, of course, and in practice these are generally more difficult tasks than learning the rules. For instance, Fig. 8.17 below illustrates the use of rules in categorizing a structure as an arch. Such a rule might involve predicates such as `Touch(·, ·)` or `Supports(·, ·, ·)` which address whether two blocks touch, or whether two blocks support a third. It is a very difficult problem in computer vision to evaluate such predicates based on a pixel image taken of the scene.

PROPOSITIONAL LOGIC

There are two main types of if-then rules: *propositional* (variable-free) and *first-order*. A propositional rule describes a particular instance, as in

FIRST-ORDER LOGIC

$$IF\ \texttt{Male(Bill)}\ AND\ \texttt{IsMarried(Bill)}\ THEN\ \texttt{IsHusband(Bill)},$$

CONSTANT

where `Bill` is a particular atomic item. Because its properties are fixed, `Bill` is an example of a (logical) *constant*. The deficiency of propositional logic is that it provides no general way to represent general relations among a large number of instances. For example, even if we knew `Male(Edward)` and `IsMarried(Edward)` are both `True`, the above rule would not allow us to infer that `Edward` is is a husband, since that rule applies only to the particular constant `Bill`.

VARIABLE

This deficiency is overcome in first-order logic, which permit rules with *variables*, such as

$$IF\ \texttt{Eats(x,y)}\ AND\ \texttt{HasFlesh(x)}\ THEN\ \texttt{Carnivore(y)},$$

where here `x` and `y` are the variables. This rule states that for any items `x` and `y`, if `y` eats `x` and `x` has flesh, then `y` is a carnivore. Clearly this is a very powerful summary of an enormous wealth of examples — first-order rules are far more expressive

[*] We shall ignore cases where a predicate is `Undefined`.

than classical propositional logic. The power of first-order logic is illustrated in the following rules:

$$\begin{aligned} &IF\ \texttt{Male(x)}\ AND\ \texttt{IsMarried(y,z)}\ THEN\ \texttt{IsHusband(x)},\\ &IF\ \texttt{Parent(x,y)}\ AND\ \texttt{Parent(y,z)}\ THEN\ \texttt{GrandParent(x,z)}\\ &\text{and}\\ &IF\ \texttt{Spouse(x,y)}\ THEN\ \texttt{Spouse(y,x)}. \end{aligned}$$

A rule from first-order logic can also apply to constants, for instance:

$$IF\ \texttt{Eats(Mouse,Cat)}\ AND\ \texttt{Mammal(Mouse)}\ THEN\ \texttt{Carnivore(Cat)},$$

where `Cat` and `Mouse` are two particular constants.

FUNCTION If-then rules can also incorporate *functions*, which return numerical values, as illustrated in the following:

$$IF\ \texttt{Male(x)}\ AND\ (\texttt{Age(x)} < \texttt{16})\ THEN\ \texttt{Boy(x)},$$

TERM where `(Age(x)` is a function that returns a numerical age in years while the expression or *term* `(Age(x) < 16)` returns either logical `True` or `False`. In sum, the above rule states that a male younger than 16 years old is a boy. If we were to use decision trees or statistical techniques, we would not be able to learn this rule perfectly, even given a tremendously large number of examples.

It is clear given a set of first-order rules how to use them in pattern classification: we merely present the unknown item and evaluate the propositions and rules. Thus consider the long rule

$$\begin{aligned} &IF\ \texttt{IsBlock(x)}\ AND\ \texttt{IsBlock(y)}\ AND\ \texttt{IsBlock(z)}\\ &AND\ \texttt{Touch(x,y)}\ AND\ \texttt{Touch(x,z)}\ AND\ \texttt{NotTouch(y,z)}\\ &AND\ \texttt{Supports(x,y,z)}\ THEN\ \texttt{Arch(x,y,z)}, \end{aligned} \tag{11}$$

where `Supports(x,y,z)` means that `x` is supported by both `y` and `z`. We stress that designing algorithms to implement `IsBlock(·)`, `Supports(·,·,·)` and so on can be extremely difficult; there is little we can say about them here other than that nearly always building these component algorithms represents the greatest effort in designing the overall classifier. Nevertheless, given reliable such algorithms, the rule could be used to classify simple structures as an arch or non-arch (Fig. 8.17).

8.8.1 Learning rules

Now we turn briefly to the learning of such if-then rules. We have already seen several ways to learn rules. For instance, we can train a decision tree via CART, ID3, C4.5 or other algorithm, and then simplify the tree to extract rules (Sect. 8.4). For cases where the underlying data arises from a grammar, we can infer the particular rules via the methods in Sect. 8.7. A key distinction in the approach we now discuss is that they can learn sets of first-order rules containing variables. As in grammatical inference, our approach to learning rules from a set of positive and negative examples, $\mathcal{D}^+$ and

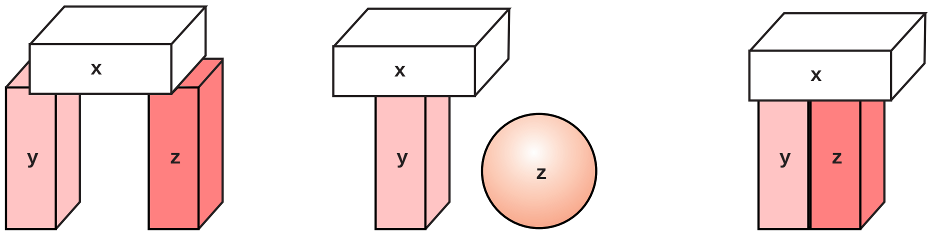


Figure 8.17: The rule in Eq. 11 identifies the figure on the left as an example of `Arch`, but not the other two figures. In practice, it is very difficult to develop subsystems that evaluate the propositions themselves, for instance `Touch(x,y)` and `Supports(x,y,z)`.

SEQUENTIAL COVERING

$\mathcal{D}^-$, is to learn a single rule, delete the examples that it explains, and iterate. Such *sequential covering* learning algorithms lead to a disjunctive set of rules that "cover" the training data. After such training it is traditional to simplify the resulting logical rule by means of standard logical methods.

The designer must specify the predicates and functions, based on a prior knowledge of the problem domain. The algorithm begins by considering the most general rules using these predicates and functions, and finds the "best" simple rule. Here, "best" means that the rule describes the largest number of training examples. Then, the algorithm searches among all refinements of the best rule, choosing the refinement that too is "best." This process is iterated until no more refinements can be added, or when the number of items described is maximum. In this way a single, though possibly complex, if-then rule has been learned (Fig. 8.18). The sequential covering algorithm iterates this process and returns a set of rules. Because of its greedy nature, the algorithm need not learn the smallest set of rules.

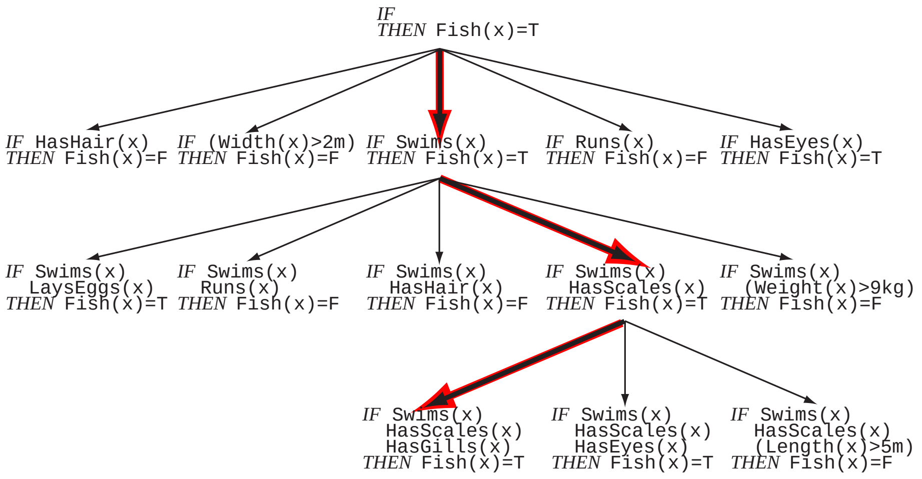

IF
THEN Fish(x)=T
IF HasHair(x) THEN Fish(x)=F
IF (Width(x)>2m) THEN Fish(x)=F
IF Swims(x) THEN Fish(x)=T
IF Runs(x) THEN Fish(x)=F
IF HasEyes(x) THEN Fish(x)=T
IF Swims(x) LaysEggs(x) THEN Fish(x)=T
IF Swims(x) Runs(x) THEN Fish(x)=F
IF Swims(x) HasHair(x) THEN Fish(x)=F
IF Swims(x) HasScales(x) THEN Fish(x)=T
IF Swims(x) (Weight(x)>9kg) THEN Fish(x)=F
IF Swims(x) HasScales(x) HasGills(x) THEN Fish(x)=T
IF Swims(x) HasScales(x) HasEyes(x) THEN Fish(x)=T
IF Swims(x) HasScales(x) (Length(x)>5m) THEN Fish(x)=F

Figure 8.18: In sequential covering, candidate rules are searched through successive refinements. First, the "best" rule having a single conditional predicate is found, i.e., the one explaining most training data. Next, other candidate predicates are added, the best compound rule selected, and so forth.

A general approach is to search first through all rules having a *single* attribute.

Next, consider the rule having a single conjunction of two predicates, then these conjunctions, and so on. Note that this greedy algorithm need not be optimal — that is, it need not yield the most compact rule.

Summary

Non-metric data consists of lists of nominal attributes; such lists might be unordered or ordered (strings). Tree-based methods such as CART, ID3 and C4.5 rely on answers to a series of questions (typically binary) for classification. The designer selects the form of question and the tree is grown, starting at the root node, by finding splits of data that make the representation more "pure." There are several acceptable impurity measures, such as misclassification, variance and Gini; the entropy impurity, however, has found greatest use. To avoid overfitting and to improve generalization, one must either employ stopped splitting (declaring a node with non-zero impurity to be a leaf), or instead prune a tree trained to minimum impurity leafs. Tree classifiers are very flexible, and can be used in a wide range of applications, including those with data that is metric, non-metric or in combination.

When comparing patterns that consist of strings of non-numeric symbols, we use edit distance — a measure of the number of fundamental operations (insertions, deletions, exchanges) that must be performed to transform one string into another. While the general edit distance is not a true metric, edit distance can nevertheless be used for nearest-neighbor classification. String matching is finding whether a test string appears in a longer *text*. The requirement of a perfect match in basic string matching can be relaxed, as in string matching with errors, or with the don't care symbol. These basic string and pattern recognition ideas are simple and straightforward, addressing them in large problems requires algorithms that are computationally efficient.

Grammatical methods assume the strings are generated from certain classes of rules, which can be described by an underlying grammar. A grammar G consists of an alphabet, intermediate symbols, a starting or root symbol and most importantly a set of rewrite rules, or productions. The four different types of grammars — free, context-sensitive, context-free, and regular — make different assumptions about the nature of the transformations. Parsing is the task of accepting a string $\mathbf{x}$ and determining whether it is a member of the language generated by G, and if so, finding a derivation. Grammatical methods find greatest use in highly structured environments, particularly where structure lies at many levels. Grammatical inference generally uses positive and negative example strings (i.e., ones in the language generated by G and ones not in that language), to infer a set of productions.

Rule-based systems use either propositional logic (variable-free) or first-order logic to describe a category. In broad overview, rules can be learned by sequentially "covering" elements in the training set by successively more complex compound rules.

Bibliographical and Historical Remarks

Most work on decision trees addresses problems in continuous features, though a key property of the method is that they apply to nominal data too. Some of the foundations of tree-based classifiers stem from the Concept Learning System described in [42], but the important book on CART [10] provided a strong statistics foundation and revived interest in the approach. Quinlan has been a leading exponent of tree classifiers, introducing ID3 [66], C4.5 [69], as well as the application of minimum description length for pruning [71, 56]. A good overview is [61], and a comparison of multivariate decision tree methods is given in [11]. Splitting and pruning criteria based on probabilities are explored in [53], and the use of an interesting information metric for this end is described in [52]. The Gini index was first used in analysis of variance in categorical data [47]. Incremental or on-line learning in decision trees is explored in [85]. The missing variable problem in trees is addressed in [10, 67], which describe methods more general than those presented here. An unusual parallel "neural" search through trees was presented in [78].

The use of edit distance began in the 1970s [64]; a key paper by Wagner and Fischer proposed the fundamental Algorithm 3 and showed that it was optimal [88]. The explosion of digital information, especially natural language text, has motivated work on string matching and related operations. An excellent survey is [5] and two thorough books are [23, 82]. The computational complexity of string algorithms is presented in [21, Chapter 34]. The fast string matching method of Algorithm 2 was introduced in [9]; its complexity and speedups and improvements were discussed in [18, 35, 24, 4, 40, 83]. String edit distance that permits block-level transpositions is discussed in [48]. Some sophisticated string operations — two-dimensional string matching, longest common subsequence and graph matching — have found only occasionally use in pattern recognition. Statistical methods applied to strings are discussed in [26]. Finite-state automata have been applied to several problems in string matching [23, Chapter 7], as well as time series prediction and switching, for instance converting from an alphanumeric representation to a binary representation [43]. String matching has been applied to the recognition DNA sequences and text, and is essential in most pattern recognition and template matching involving large databases of text [14]. There is a growing literature on special purpose hardware for string operations, of which the Splash-2 system [12] is a leading example.

The foundations of a formal study of grammar, including the classification of grammars, began with the landmark book by Chomsky [16]. An early exposition of grammatical inference [39, Chapter 6] was the source for much of the discussion here. Recognition based on parsing (Latin *pars orationis* or "part of speech") has been fundamental in automatic language recognition. Some of the earliest work on three-dimensional object recognition relied on complex grammars which described the relationships of corners and edges, in block structures such arches and towers. It was found that such systems were very brittle; they failed whenever there were errors in feature extraction, due to occlusion and even minor misspecifications of the model. For the most part, then, grammatical methods have been abandoned for object recognition and scene analysis [60, 25]. Grammatical methods have been applied to the recognition of some simple, highly structured diagrams, such as electrical circuits, simple maps and even Chinese/Japanese characters. For useful surveys of the basic ideas in syntactic pattern recognition see [33, 34, 32, 13, 62, 14], for parsing see [28, 3], for grammatical inference see [59]. The complexity of parsing type 3 is linear in the length of the string, type 2 is low-order polynomial, type 1 is exponential; pointers to the relevant literature appear in [76]. There has been a great deal of work on parsing natural language and speech, and a good textbook on artificial intelligence addressing this topic and much more is [75]. There is much work on inferring grammars from instances, such as Crespi-Reghizzi algorithm (context free) [22]. If queries can be presented interactively, the learning of a grammar can be speeded [81].

The methods described in this chapter have been expanded to allow for stochastic grammars, where there are probabilities associated with rules [20]. A grammar can be considered a specification of a prior probability for a class; for instance, a uniform prior over all (legal) strings in the language $\mathcal{L}$. Error-correcting parsers have been used when random variations arise in an underlying stochastic grammar [50, 84]. One can also apply probability measures to languages [8].

Rule-based methods have formed the foundation of expert systems, and have been applied extensively through many branches of artificial intelligence such as planning, navigation and prediction; their use in pattern recognition has been modest, however. Early influential systems include DENDRAL, for inferring chemical structure from mass spectra [29], PROSPECTOR, for finding mineral deposits [38], and MYCIN, for medical diagnosis [79]. Early use of rule induction for pattern recognition include that of Michalski [57, 58]. Figure 8.17 was inspired by Winston's influential work on learning simple geometrical structures and relationships [91]. Learning rules can be called inductive logic programming; Clark and Niblett have made a number of contributions to the field, particularly their CN2 induction algorithm [17]. Quinlan, who has contributed much to the theory and application of tree-based classifiers, describes his FOIL algorithm, which uses a minimum description length criterion to stop the learning of first-order rules [68]. Texts on inductive logic include [46, 63] and general machine learning, including inferencing [44, 61].

Problems

⊕ *Section 8.2*

1. When a test pattern is classified by a decision tree, that pattern is subjected to a sequence of queries, corresponding to the nodes along a path from root to leaf. Prove that for any decision tree, there is a functionally equivalent tree in which every such path consists of *distinct* queries. That is, given an arbitrary tree prove that it is always possible to construct an equivalent tree in which no test pattern is ever subjected to the same query twice.

⊕ *Section 8.3*

2. Consider classification trees that are non-binary.

(a) Prove that for any arbitrary tree, with possibly unequal branching ratios throughout, there exists a binary tree that implements the same classification function.

(b) Consider a tree with just two levels — a root node connected to B leaf nodes ($B \geq 2$). What are the upper and the lower limits on the number of *levels* in a functionally equivalent binary tree, as a function of B?

(c) As in part (b), what are the upper and lower limits on the number of *nodes* in a functionally equivalent binary tree?

3. Compare the computational complexities of a monothetic and a polythetic tree classifier trained on the same data as follows. Suppose there are $n/2$ training patterns in each of two categories. Every pattern has d attributes, each of which can take on k discrete values. Assume that the best split evenly divides the set of patterns.

(a) How many levels will there be in the monothetic tree? The polythetic tree?

(b) In terms of the variables given, what is the complexity of finding the optimal split at the root of a monothetic tree? A polythetic tree?

(c) Compare the total complexities for training the two trees fully.

4. The task here is to find the computational complexity of training a tree classifier using the twoing impurity where candidate splits are based on a single feature. Suppose there are c classes, $\omega_1, \omega_2, ..., \omega_c$, each with n/c patterns that are d-dimensional. Proceed as follows:

(a) How many possible non-trivial divisions into two supercategories are there at the root node?

(b) For any one of these candidate supercategory divisions, what is the computational complexity of finding the split that minimizes the entropy impurity?

(c) Use your results from parts (a) & (b) to find the computational complexity of finding the split at the root node.

(d) Suppose for simplicity that each split divides the patterns into equal subsets and furthermore that each leaf node corresponds to a single pattern. In terms of the variables given, what will be the expected number of levels of the tree?

(e) Naturally, the number of classes represented at any particular node will depend upon the level in the tree; at the root all c categories must be considered, while at the level just above the leaves, only 2 categories must be considered. (The pairs of particular classes represented will depend upon the particular node.) State some natural simplifying assumptions, and determine the number of candidate classes at any node as a function of level. (You may need to use the floor or ceiling notation, $\lfloor x \rfloor$ or $\lceil x \rceil$, in your answer, as described in the Appendix.)

(f) Use your results from part (e) and the number of patterns to find the computational complexity at an arbitrary level L.

(g) Use all your results to find the computational complexity of training the full tree classifier.

(h) Suppose there $n = 2^{10}$ patterns, each of which is $d = 6$ dimensional, evenly divided among $c = 16$ categories. Suppose that on a uniprocessor a fundamental computation requires roughly 10^{-10} seconds. Roughly how long will it take to train your classifier using the twoing criterion? How long will it take to classify a single test pattern?

5. Consider training a binary tree using the entropy impurity, and refer to Eqs. 1 & 5.

(a) Prove that the decrease in entropy impurity provided by a single yes/no query can never be greater than one bit.

(b) For the two trees in Example 1, verify that each split reduces the impurity and that this reduction is never greater than 1 bit. Explain nevertheless why the impurity at a node *can* be lower than at its descendent, as occurs in that Example.

(c) Generalize your result from part (a) to the case with arbitrary branching ratio $B \geq 2$.

6. Let $P(\omega_1), \ldots, P(\omega_c)$ denote the probabilities of c classes at node N of a binary classification tree, and $\sum\limits_{j=1}^{c} P(\omega_j) = 1$. Suppose the impurity $i(P(\omega_1), \ldots, P(\omega_c))$ at N is a strictly concave function of the probabilities. That is, for any probabilities

$$\begin{aligned} i_a &= i(P^a(\omega_1), \ldots, P^a(\omega_c)) \\ i_b &= i(P^b(\omega_1), \ldots, P^b(\omega_c)) \\ &\ \text{and} \\ i^*(\boldsymbol{\alpha}) &= i(\alpha_1 P^a(\omega_1) + (1-\alpha_1)P^b(\omega_1), \ldots, \alpha_c P^a(\omega_c) + (1-\alpha_c)P^b(\omega_c)), \end{aligned}$$

then for $0 \leq \alpha_j \leq 1$ and $\sum\limits_{j=1}^{c} \alpha_j = 1$, we have

$$i_a \leq i^* \leq i_b.$$

(a) Prove that for any split, we have $\Delta i(s,t) \geq 0$, with equality if and only if $P(\omega_j|T_L) = P(\omega_j|T_R) = P(\omega_j|T)$, for $j = 1, \ldots, c$. In other words, for a concave impurity function, splitting never increases the impurity.

(b) Prove that entropy impurity (Eq. 1) is a concave function.

(c) Prove that Gini impurity (Eq. 3) is a concave function.

7. Show that the surrogate split method described in the text corresponds to the assumption that the missing feature (attribute) is the one most informative.

8. Consider a two-category problem and the following training patterns, each having four binary attributes:

ω_1	ω_2
0110	1011
1010	0000
0011	0100
1111	1110

(a) Use the entropy impurity (Eq. 1) to create by hand an unpruned classifier for this data.

(b) Apply simple logical reduction methods to your tree in order to express each category by the simplest logical expression, i.e., with the fewest ANDs and ORs.

9. Show that the time complexity of recall in an unpruned, fully trained tree classifier with uniform branching ratio is $\mathcal{O}(\log n)$ where n is the number of training patterns. For uniform branching factor, B, state the exact functional form of the number of queries applied to a test pattern as a function of B.

10. Consider impurity functions for a two-category classification problem as a function of $P(\omega_1)$ (and implicitly $P(\omega_2) = 1 - P(\omega_1)$). Show that the simplest reasonable polynomial form for the impurity is related to the sample variance as follows:

(a) Consider restricting impurity functions to the family of polynomials in $P(\omega_1)$. Explain why i must be at least quadratic in $P(\omega_1)$.

(b) Write the simplest quadratic form for $P(\omega_1)$ given the boundary conditions $i(P(\omega_1)=0)=i(P(\omega_1)=1)=0$; show that your impurity function can be written $i \propto P(\omega_1)P(\omega_2)$.

(c) Suppose all patterns in category ω_1 are assigned the value 1.0, while all those in ω_2 the value 0.0, thereby giving a bimodal distribution. Show that your impurity measure is proportional to the sample variance of this full distribution. Interpret your answer in words.

11. Show how general costs, represented in a cost matrix λ_{ij}, can be incorporated into the misclassification impurity (Eq. 4) during the training of a multicategory tree.

12. In this problem you are asked to create tree classifiers for a one-dimensional two-category problem in the limit of large number of points, where $P(\omega_1)=P(\omega_2)=1/2$, $p(x|\omega_1) \sim N(0,1)$ and $p(x|\omega_2) \sim N(1,2)$, and all nodes have decisions of the form "is $x \leq x_s$" for some threshold x_s. Each binary tree will be small — just a root node plus two other (non-terminal) nodes and four leaf nodes. For each of the four impurity measures below, state the splitting criteria (i.e., the value x_s at each of the three non-terminal nodes), as well as the final test error. Whenever possible, express your answers functionally, possibly using the error function $erf(\cdot)$, as well as numerically.

(a) Entropy impurity (Eq. 1).

(b) Gini impurity (Eq. 3).

(c) Misclassification impurity (Eq. 4).

(d) Another splitting rule is based on the so-called *Kolmogorov-Smirnov test.* Let the cumulative distributions for a single variable x for each categories be $F_i(x)$ for $i=1,2$. The splitting criterion is the maximum difference in the cumulative distributions, i.e.,

Kolmogorov-Smirnov test

$$\max_{x_s} |F_1(x_s) - F_2(x_s)|.$$

(e) Using the methods of Chap. **??**, calculate the Bayes decision rule, and the Bayes error.

13. Repeat Problem 12 but for two one-dimensional Cauchy distributions,

$$p(x|\omega_i) = \frac{1}{\pi b_i} \cdot \frac{1}{1+\left(\frac{x-a_i}{b_i}\right)^2}, \qquad i=1,2,$$

with $P(\omega_1)=P(\omega_2)=1/2$, $a_1=0$, $b_1=1$, $a_2=1$ and $b_2=2$. (Here error functions are not needed.)

14. Generalize the missing attribute problem to the case of *several* missing features, and to several deficient patterns. Specifically, write pseudocode for creating a binary decision tree where each d-dimensional pattern can have multiple missing features.

15. During the growing of a decision tree, a node represents the following six-dimensional binary patterns:

ω_1	ω_2
110101	011100
101001	010100
100001	011010
101101	010000
010101	001000
111001	010100
100101	111000
011000	110101

Candidate decision are based on single feature values.

(a) Which feature should be used for splitting?

(b) Recall the use of statistical significance for stopped splitting. What is the null hypothesis in this example?

(c) Calculate chi-squared for your decision in part (a). Does it differ significantly from the null hypothesis at the 0.01 confidence level? Should splitting be stopped?

(d) Repeat part (c) for the 0.05 level.

16. Consider the following patterns, each having four binary-valued attributes:

ω_1	ω_2
1100	1100
0000	1111
1010	1110
0011	0111

Note especially that the first patterns in the two categories are the same.

(a) Create by hand a binary classification tree for this data. Train your tree so that the leaf nodes have the lowest impurity possible.

(b) Suppose it is known that during testing the prior probabilities of the two categories will not be equal, but instead $P(\omega_1) = 2P(\omega_2)$. Modify your training method and use the above data to form a tree for this case.

⊕ *Section 8.4*

17. Consider training a binary decision tree to classify two-component patterns from two categories. The first component is binary, `0` or `1`, while the second component has six possible values, `A` through `F`:

ω_1	ω_2
1A	0A
0E	0C
0B	1C
1B	0F
1F	0B
0D	1D

Compare splitting the root node based on the first feature with splitting it on the second feature in the following way.

(a) Use an entropy impurity with a two-way split (i.e., $B = 2$) on the first feature and a six-way split on the second feature.

(b) Repeat (a) but using a gain ratio impurity.

(c) In light of your above answers discuss the value of gain ratio impurity in cases where splits have different branching ratios.

⊕ *Section 8.5*

18. Consider strings $\mathbf{x}$ and *text*, of length m and n, respectively, from an alphabet $\mathcal{A}$ consisting of d characters. Assume that the naive string-matching algorithm (Algorithm 1) exits the implied loop in line 4 as soon as a mismatch occurs. Prove that the number of character-to-character comparisons made on average for random strings is

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1).$$

19. Consider string matching using the Boyer-Moore algorithm (Algorithm 2) based on the trinary alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Apply the good-suffix function $\mathcal{G}$ and the last-occurrence function $\mathcal{F}$ to each of the following strings:

(a) "`acaccacbac`"

(b) "`abababcbcbaaabcbaa`"

(c) "`cccaaababaccc`"

(d) "`abbabbabbcbbabbcbba`"

20. Consider the string-matching problem illustrated in the top of Fig. 8.8. Assume *text* began at the first character of "`probabilities`."

(a) How many basic character comparisons are required by the naive string-matching algorithm (Algorithm 1) to find a valid shift?

(b) How many basic character comparisons are required by the Boyer-Moore string matching algorithm (Algorithm 2)?

21. For each of the *text*s below, determine the number of fundamental character comparisons needed to find all valid shifts for the test string $\mathbf{x}$ = "`abcca`" using the naive string-matching algorithm (Algorithm 1) and the Boyer-Moore algorithm (Algorithm 2).

(a) "`abcccdabacabbca`"

(b) "`dadadadadadadad`"

(c) "`abcbcabcabcabc`"

(d) "`accabcababacca`"

(e) "bbccacbccabbcca"

22. Write pseudocode for an efficient construction of the last-occurrence function $\mathcal{F}$ used in the Boyer-Moore algorithm (Algorithm 2). Let d be the number of elements in the alphabet $\mathcal{A}$, and m the length of string $\mathbf{x}$.

(a) What is the time complexity of your algorithm in the worst case?

(b) What is the space complexity of your algorithm in the worst case?

(c) How many fundamental operations are required to compute $\mathcal{F}$ for the 26-letter English alphabet for $\mathbf{x}$ = "bonbon"? For $\mathbf{x}$ = "marmalade"? For $\mathbf{x}$ = "abcdabdabcaabcda"?

23. Consider the training data from the trinary alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ in the table

ω_1	ω_2	ω_3
aabbc	bccba	caaaa
ababcc	bbbca	cbcaab
babbcc	cbbaaaa	baaca

Use the simple edit distance to classify each of the below strings. If there are ambiguities in the classification, state which two (or all three) categories are candidates.

(a) "abacc"

(b) "abca"

(c) "ccbba"

(d) "bbaaac"

24. Repeat Problem 23 using its training data but the following test data:

(a) "ccab"

(b) "abdca"

(c) "abc"

(d) "bacaca"

25. Repeat Problem 23 but assume that the cost of different string transformations are not equal. In particular, assume that an interchange is twice as costly as an insertion or a deletion.

26. Consider edit distance with positive but otherwise arbitrary costs associated with each of the fundamental operations of insertion, deletion and substitution.

(a) Which of the criteria for a metric are always obeyed and which not necessarily obeyed?

(b) For any criteria that are not always obeyed, construct a counterexample.

27. Algorithm 3 employs a greedy heuristic for computing the edit distance between two strings $\mathbf{x}$ and $\mathbf{y}$; it need not give a global optimum. In the following, let $|\mathbf{x}| = n_1$ and $|\mathbf{y}| = n_2$.

(a) State the computational complexity of an exhaustive examination of all transformations of $\mathbf{x}$ into $\mathbf{y}$. (Assume that no transformation need be considered if it leads to a string shorter than $Min[n_1, n_2]$ or longer than $Max[n_1, n_2]$.)

(b) Recall from Chap. **??** the basic approach of linear programming. Write pseudocode that would apply linear programming to the calculation of edit distances.

28. Consider the general edit distance with positive costs and whether it has the four properties of a true metric: non-negativity, reflexivity, symmetry and the triangle inequality.

29. Consider strings $\mathbf{x}$ and *text*, of length m and n, respectively, from an alphabet $\mathcal{A}$ consisting of d characters.

(a) Modify the pseudocode of the naive string-matching algorithm to include the don't care symbol.

(b) Employ the assumptions of Problem 18 but also that $\mathbf{x}$ has exactly k don't care symbols while *text* has none. Find the number of character-to-character comparisons made on average for otherwise random strings.

(c) Show that in the limit of $k = 0$ your answer is closely related to that of Problem 18.

(d) What is your answer in part (b) in the limit $k = m$?

⊕ *Section 8.6*

30. Mathematical expressions in the computer language Lisp are of the form `(operation` *operand*$_1$ *operand*$_2$`)` where spaces delineate potentially ambiguous symbols and expressions can be nested, for example `(quotient (plus 4 9) 6)`.

(a) Write a simple grammar for the four basic arithmetic operations `plus`, `difference`, `times` and `quotient`, applied to positive single-digit integers. Be sure to include parentheses in your alphabet.

(b) Determine by hand whether each of the following candidate Lisp expressions can be derived in your grammar, and if so, show a corresponding derivation tree.

- `(times (plus (difference 5 9)(times 3 8))(quotient 2 6))`
- `(7 difference 2)`
- `(quotient (7 plus 2) (plus 6 3))`
- `((plus) (6 2))`
- `(difference (plus 5 9) (difference 6 8))` .

31. Consider the language $\mathcal{L}(G) = \{\mathtt{a}^n\mathtt{b} | n \geq 1\}$.

(a) Construct by hand a grammar that generates this language.

(b) Use G to form derivation trees for the strings "`ab`" and "`aaaaab`."

32. Consider the grammar G: $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\mathcal{S} = S$, $\mathcal{I} = \{A, B\}$ and $\mathcal{P} = \{S \rightarrow \mathtt{c}A\mathtt{b},\ A \rightarrow \mathtt{a}B\mathtt{a},\ B \rightarrow \mathtt{a}B\mathtt{a},\ B \rightarrow \mathtt{cb}\}$.

(a) What type of grammar is G?

(b) Prove that this grammar generates the language $\mathcal{L}(G) = \{\mathtt{ca}^n\mathtt{cba}^n\mathtt{b} | n \geq 1\}$.

(c) Draw the derivation tree the following two strings: "caacbaab" and "cacbab."

PALINDROME **33.** A *palindrome* is a sequence of characters that reads the same forward and backward, such as "i," "tat," "boob," and "sitonapotatopanotis."

(a) Write a grammar that generates all palindromes using 26 English letters (no spaces). Use your grammar to show a derivation tree for "noon" and "bib."

(b) What type is your grammar (0, 1, 2 or 3)?

(c) Write a grammar that generates all words that consist of a single letter followed by a palindrome. Use your grammar to show a derivation tree for "pi," "too," and "stat."

34. Consider the grammar G in Example 3.

(a) How many possible derivations are there in G for numbers 1 through 999?

(b) How many possible derivations are there for numbers 1 through 999,999?

(c) Does the grammar allow any of the numbers (up to six digits) to be pronounced in more than one way?

35. Recall that ϵ is the empty string, defined to have zero length, and no manifestation in a final string. Consider the following grammar G: $\mathcal{A} = \{\mathtt{a}\}$, $\mathcal{S} = S$, $\mathcal{I} = \{A, B, C, D, E\}$ and eight rewrite rules:

$$\mathcal{P} = \left\{ \begin{array}{rlrl} S & \rightarrow AC\mathtt{a}B & C\mathtt{a} & \rightarrow \mathtt{aa}C \\ CB & \rightarrow DB & CB & \rightarrow E \\ \mathtt{a}D & \rightarrow D\mathtt{a} & \mathtt{a}D & \rightarrow AC \\ \mathtt{a}E & \rightarrow E\mathtt{a} & AE & \rightarrow \epsilon \end{array} \right\}.$$

(a) Note how A and B mark the beginning and end of the sentence, respectively, and that C is a marker that doubles the number of as (while moving from left to right through the word). Prove that the language generated by this grammar is $\mathcal{L}(G) = \{\mathtt{a}^{2^n} | n > 0\}$.

(b) Show a derivation tree for "aaaa" and for "aaaaaaaa" (cf. Computer exercise **??**).

36. Explore the notion of Chomsky normal form in the following way.

(a) Show that the grammar G with $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$, $\mathcal{S} = S$, $\mathcal{I} = \{A, B\}$ and rewrite rules:

$$\mathcal{P} = \left\{ \begin{array}{rl} S & \rightarrow \mathtt{b}A \ OR \ \mathtt{a}B \\ A & \rightarrow \mathtt{b}AA \ OR \ \mathtt{a}S \ OR \ \mathtt{a} \\ B & \rightarrow \mathtt{a}BB \ OR \ \mathtt{b}S \ OR \ \mathtt{b} \end{array} \right\},$$

is not in Chomsky normal form.

(b) Show that grammar G' with $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$, $\mathcal{S} = S$, $\mathcal{I} = \{A, B, C_a, C_b, D_1, D_2\}$, and

$$\mathcal{P} = \left\{ \begin{array}{ll} S \rightarrow C_bA \ OR \ C_aB & D_1 \rightarrow AA \\ A \rightarrow C_aS \ OR \ C_bD_1 \ OR \ \mathtt{a} & D_2 \rightarrow BB \\ B \rightarrow C_bS \ OR \ C_aD_2 \ OR \ \mathtt{b} & C_a \rightarrow \mathtt{a} \\ & C_b \rightarrow \mathtt{b} \end{array} \right\}.$$

is in Chomsky normal form.

(c) Show that G and G' are equivalent by converting the rewrite rules of G into those of G' in the following way. Note that the rules $A \rightarrow \mathtt{a}$ and $B \rightarrow \mathtt{b}$ of G are already acceptable. Now convert other rules of G appropriate for Chomsky normal form. First replace $S \rightarrow \mathtt{b}A$ in G by $S \rightarrow C_bA$ and $C_b \rightarrow \mathtt{b}$. Likewise, replace $A \rightarrow \mathtt{a}S$ by $A \rightarrow C_aS$ and $C_a \rightarrow \mathtt{a}$. Continue in this way, keeping in mind the final form of the rewrite rules of G'.

(d) Give a derivation of "aabab" in G and in G'.

37. Prove that each of the following languages are not context-free.

(a) $\mathcal{L}(G) = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | i < j < k\}$.

(b) $\mathcal{L}(G) = \{\mathtt{a}^i | i \text{ a prime}\}$.

38. Consider a grammar with $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\mathcal{S} = S$, $\mathcal{I} = \{A, B\}$, and

$$\mathcal{P} = \left\{ \begin{array}{ll} S \ \rightarrow \mathtt{a}SBA \ OR \ \mathtt{a}BA & AB \rightarrow BA \\ \mathtt{b}B \rightarrow \mathtt{bb} & \mathtt{b}A \rightarrow \mathtt{bc} \\ \mathtt{c}A \rightarrow \mathtt{cc} & \mathtt{a}B \rightarrow \mathtt{ab} \end{array} \right\}.$$

Prove that this grammar generates the language $\mathcal{L}(G) = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n | n \geq 1\}$.
39. Try to parse by hand the following utterances. For each successful parse, show the corresponding derivation tree.

- *three hundred forty two thousand six hundred nineteen*
- *thirteen*
- *nine hundred thousand*
- *two thousand six hundred thousand five*
- *one hundred sixty eleven*

⊕ *Section 8.7*

40. Let $\mathcal{D}_1 = \{\mathtt{ab}, \mathtt{abb}, \mathtt{abbb}\}$ and $\mathcal{D}_2 = \{\mathtt{ba}, \mathtt{aba}, \mathtt{babb}\}$ be positive training examples from two grammars, G_1 and G_2, respectively.

(a) Suppose both grammars are of type 3. Generate some candidate rewrite rules.

(b) Infer grammar G_1 using $\mathcal{D}_2$ as negative examples.

(c) Infer grammar G_2 using $\mathcal{D}_1$ as negative examples.

(d) Use your trained grammars to classify the following sentences; label any sentence that cannot be parsed in either grammar as "ambiguous": "`bba`," "`abab`," "`bbb`" and "`abbbb`."

⊕ *Section 8.8*

41. For each of the below, write an rule giving an equivalent relation using any of the following predicates: `Male(·)`, `Female(·)`, `Parent(·,·)`, `Married(·,·)`,

(a) `Sister(·,·)`, where `Sister(x,y) = True` means that `x` is the sister of `y`.

(b) `Father(·,·)`, where `Father(x,y) = True` means that `x` is the father of `y`.

(c) `Grandmother(·,·)`, where `Grandmother(x,y) = True` means that `x` is the grandmother of `y`.

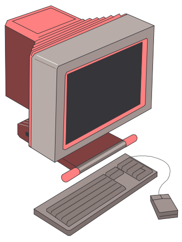
(d) `Husband(·,·)`, where `Husband(x,y) = True` means that `x` is the husband of `y`.

(e) `IsWife(·)`, where `IsWife(x) = True` means that simply that `x` is a wife.

(f) `Siblings(·,·)`

(g) `FirstCousins(·,·)`

Computer exercises



Several exercises will make use of the following data sampled from three categories. Each of the five features takes on a discrete feature, indicated by the range listed at the along the top. Note particularly that there are different number of samples in each category, and that the number of possible values for the features is not the same. For instance, the first feature can take on four values ($A-D$, inclusive), while the last feature can take on just two ($M-N$).

sample	category	$A-D$	$E-G$	$H-J$	$K-L$	$M-N$
1	ω_1	A	E	H	K	M
2	ω_1	B	E	I	L	M
3	ω_1	A	G	I	L	N
4	ω_1	B	G	H	K	M
5	ω_1	A	G	I	L	M
6	ω_2	B	F	I	L	M
7	ω_2	B	F	J	L	N
8	ω_2	B	E	I	L	N
9	ω_2	C	G	J	K	N
10	ω_2	C	G	J	L	M
11	ω_2	D	G	J	K	M
12	ω_2	B	D	I	L	M
13	ω_3	D	E	H	K	N
14	ω_3	A	E	H	K	N
15	ω_3	D	E	H	L	N
16	ω_3	D	F	J	L	N
17	ω_3	A	F	H	K	N
18	ω_3	D	E	J	L	M

⊕ *Section 8.3*

1. Write a general program for growing a binary tree and use it to train a tree fully using the data from the three categories in the table, using an entropy impurity.

(a) Use the (unpruned) tree to classify the following patterns: $\{A, E, I, L, N\}$, $\{D, E, J, K, N\}$, $\{B, F, J, K, M\}$, and $\{C, D, J, L, N\}$.

(b) Prune one pair of leafs, increasing the entropy impurity as little as possible.

(c) Modify your program to allow for non-binary splits, where the branching ratio B as is determined at each node during training. Train a new tree fully using a gain ratio impurity and then classify the points in (a).

2. Recall that one criterion for stopping the growing of a decision tree is to halt splitting when the best split reduces the impurity by less than some threshold value, that is, when $\max_s \Delta i(s) \leq \beta$ where s indicates the split and β is the threshold. Explore the relationship between classifier generalization and β through the following simulations.

(a) Generate 200 training points, 100 each for two two-dimensional Gaussian distributions: $p(\mathbf{x}|\omega_1) \sim N(\binom{-0.25}{0}, \mathbf{I})$ and $p(\mathbf{x}|\omega_2) \sim N(\binom{+0.25}{0}, \mathbf{I})$. Also use your program to generate an independent test set of 100 points, 50 each of the categories.

(b) Write a program to grow a tree classifier, where a node is not split if $\max_s \Delta i(s) \leq \beta$.

(c) Plot the generalization error of your tree classifier versus β for $\beta = 0.01 \rightarrow 1$ in steps of 0.01, as estimated on the test data generated in part (a).

(d) In light of your plot, discuss the relationship between β and generalization error.

3. Repeat all parts of Computer exercise 2, but instead of considering β, focus instead on the role of α as used in Eq. 8.

⊕ *Section 8.4*

4. Write a program for training an ID3 decision tree in which the branching ratio B at each node is equal to the number of discrete "binned" possible values for each attribute. Use a gain ratio impurity.

(a) Use your program to train a tree fully with the ω_1 and ω_2 patterns in the table above.

(b) Use your tree to classify $\{B, G, I, K, N\}$ and $\{C, D, J, L, M\}$.

(c) Write a logical expression which describes the decisions in part (b). Simplify these expressions.

(d) Convert the information in your tree into a single logical expression which describes the ω_1 category. Repeat for the ω_2 category.

5. Consider the issue of tree-based classifiers and deficient patterns.

(a) Write a program to generate a binary decision tree for categories ω_1 and ω_2 using samples points $1-10$ from the table above and an entropy impurity. For each decision node store the primary split and four surrogate splits.

(b) Use your tree to classify the following patterns, where as usual * denotes a missing feature.

- $\{A, F, H, K, M\}$
- $\{*, G, H, K, M\}$
- $\{C, F, I, L, N\}$
- $\{B, *, *, K, N\}$

(c) Now write a program to train a tree using deficient points. Train with sample points $1-10$ from the table, used in part (a), as well as the following four points:

- ω_1: $\{*, F, I, K, N\}$
- ω_1: $\{B, G, H, K, *\}$
- ω_2: $\{C, G, *, L, N\}$
- ω_2: $\{*, F, I, K, N\}$

(d) Use your tree from part (c) to classify the test points in part (b).

6. Train a tree classifier to distinguish all three categories ω_i, $i = 1, 2, 3$, using all 20 sample points in the table above. Use an entropy criterion without pruning or stopping.

(a) Express your tree as a set of rules.

(b) Through exhaustive search, find the rule or rules, which when deleted, lead to the smallest increase in classification error as estimated by the training data.

⊕ *Section 8.5*

7. Write a program to implement the naive string-matching algorithm (Algorithm 1). Insert a conditional branch so as to exit the innermost loop whenever a mismatch occurs (i.e., the shift is found to be invalid). Add a line to count the total number of character-to-character comparisons in the complete string search.

(a) Write a small program to generate a *text* of n characters, taken from an alphabet having d characters. Let $d = 5$ and use your program to generate a *text* of length $n = 1000$ and a test string $\mathbf{x}$ of length $m = 10$.

(b) Compare the number of character-to-character comparisons performed by your program with the theoretical result quoted in Problem 18 for all pairs of the following parameters: $m = \{10, 15, 20\}$ and $n = \{100, 1000, 10000\}$.

8. Write a program to implement the Boyer-Moore algorithm (Algorithm 2) in the following steps. Throughout let the alphabet have d characters.

(a) Write a routine for constructing the good-suffix function $\mathcal{G}$. Let $d = 3$ and apply your routine to the strings $\mathbf{x}_1$ = "`abcbab`"and $\mathbf{x}_2$ = "`babab`."

(b) Write a routine for constructing the last-occurrence function $\mathcal{F}$. Apply it to the strings $\mathbf{x}_1$ and $\mathbf{x}_2$ of part (a).

(c) Write an implementation of the Boyer-Moore algorithm incorporating your routines from parts (a) and (b). Generate a *text* of $n = 10000$ characters chosen from the alphabet $\mathcal{A} = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Use your program to search for $\mathbf{x}_1$ in *text*, and again $\mathbf{x}_2$ in *text*.

(d) Make some statistical assumptions to estimate the number of occurrences of $\mathbf{x}_1$ and $\mathbf{x}_2$ in *text*, and compare that number to your answers in part (c).

9. Write an algorithm for addressing the subset-superset problem in string matching. That is, search a *text* with several strings, some of which are factors of others.

(a) Let $\mathbf{x}_1$ = "beats," $\mathbf{x}_2$ = "beat," $\mathbf{x}_3$ = "be," $\mathbf{x}_4$ = "at," $\mathbf{x}_5$ = "eat," $\mathbf{x}_6$ = "sat." Search for these possible factors in *text* = "$\underbrace{\texttt{beats\_beats\_beats\_}\ldots\texttt{\_beats}}_{100\ \times}$," but do not return any strings that are factors of other test strings found in *text*.

(b) Repeat with *text* consisting of 100 appended copies of "repeatable_," and the test items "repeatable," "pea," "table," "tab," "able," "peat," and "a."

10. String matching with errors. Test on segments xxxx

⊕ *Section 8.6*

11. Write a parser for the grammar described in the text: $\mathcal{A} = \{\texttt{a}, \texttt{b}\}$, $\mathcal{I} = \{A, B\}$,

$$\mathcal{S} = S \text{ and } \mathcal{P} = \left\{ \begin{array}{lll} \mathbf{p}_1: & S & \rightarrow AB \ OR \ BC \\ \mathbf{p}_2: & A & \rightarrow BA \ OR \ \texttt{a} \\ \mathbf{p}_3: & B & \rightarrow CC \ OR \ \texttt{b} \\ \mathbf{p}_4: & C & \rightarrow AB \ OR \ \texttt{a} \end{array} \right\}.$$

Use your program to attempt to parse each of the following strings. In all cases, show the parse tree; for each successful parse show moreover the corresponding derivation tree.

- "aaaabbab"
- "ba"
- "baabab"
- "babab"
- "aaa"
- "baaa"

⊕ *Section 8.7*

12. Write a program to infer a grammar G from the following positive and negative examples:

- $\mathcal{D}^+ = \{\texttt{abc}, \texttt{aabbcc}, \texttt{aaabbbccc}\}$
- $\mathcal{D}^- = \{\texttt{abbc}, \texttt{abcc}, \texttt{aabcc}\}$

Take the following as candidate rewrite rules:

$$\begin{array}{lll}
S \to \texttt{a}SBA & AB \to BA & \texttt{c}B \to \texttt{a}C \\
S \to \texttt{b}SBA & BA \to AB & \texttt{b}A \to \texttt{bc} \\
S \to \texttt{a}BA & \texttt{b}B \to \texttt{bb} & \texttt{b}C \to \texttt{bc} \\
S \to \texttt{a}SB & \texttt{b}C \to \texttt{ba} & \texttt{a}B \to \texttt{ab} \\
S \to \texttt{a}SA & \texttt{c}A \to \texttt{cc} & \texttt{a}B \to \texttt{ca}
\end{array}$$

Proceed as follows:

(a) Implement the general bottom-up parser of Algorithm 4.

(b) Implement the general grammatical inferencing method of Algorithm 5.

(c) Use your programs in conjunction to infer G from the data.