which causes the method of computing δ_i approximately to abort. If this indicator of an eigenvalue meltdown occurs, the coded message "Chernobyl" is sent up to the S user.

Finally, when the interpolation method is used, the FORTRAN code must allocate space based on a prediction from the number of observations, the number of numeric predictors, and the specification of the surface and errors. If this allocated space is too small, the *k-d* tree division is truncated and a warning message sent up. In some cases the problem is extreme enough that the fit is not carried out; this necessitates increasing the value of α.

Bibliographic Notes

Local regression models are treated in detail in a new book by Cleveland and Grosse (forthcoming). But methods of local fitting date back at least to the 1920s. Initial applications were to smooth a time series (Macauley, 1931). An early use of local fitting for the general regression problem was investigated by Watson (1964). The method amounted to fitting a constant locally—in other words, taking the polynomial degree λ to be zero. This came to be known as kernel smoothing. It leads to very interesting theoretical work but is not of use in practice since it is hard to coax the method into following the patterns in most datasets. More serious attempts at local fitting were suggested by McLain (1974), who fitted quadratic polynomials, and Stone (1977), who fitted linear polynomials. The method of fitting used here was described by Cleveland (1979) for one predictor, and is the basis of the S function `lowess`, which has now been upgraded to the function `scatter.smooth()`. Cleveland and Devlin (1988) extended the method to two or more predictors and investigated the sampling properties in the Gaussian case. (Sampling properties in the symmetric case are still under development.) The computational methods described in Sections 8.3 and 8.4, which are crucial to local regression being useful in practice, are due to Cleveland and Grosse (1991).

Chapter 9

Tree-Based Models

Linda A. Clark
Daryl Pregibon

This chapter describes S functions for tree-based modeling. Tree-based models provide an alternative to linear and additive models for regression problems and to linear logistic and additive logistic models for classification problems. The models are fitted by binary recursive partitioning whereby a dataset is successively split into increasingly homogeneous subsets until it is infeasible to continue. The implementation described in this chapter consists of a number of functions for growing, displaying, and interacting with tree-based models. This approach to tree-based models is consistent with the data-analytic approach to other models, and consists primarily of fits, residual analyses, and interactive graphical inspection.

9.1 Tree-Based Models in Statistics

Tree-based modeling is an exploratory technique for uncovering structure in data. Specifically, the technique is useful for classification and regression problems where one has a set of classification or predictor variables ($\boldsymbol{x}$) and a single-response variable (y). When y is a factor, decision or classification rules are determined from the data—for example,

if $(x_1 \le 2.3)$ and $(x_3 \in \{A, B\})$
then y is most likely to be in level 5.

When y is numeric, regression rules for description or prediction are of the form

if $(x_2 \le 413)$ and $(x_9 \in \{C, D, F\})$ and $(x_5 \le 3.5)$
then the predicted value of y is 4.75.

A classification or regression tree is the collection of many such rules determined by a procedure known as *recursive partitioning*, which is discussed in detail in Section 9.4. This form of classification or prediction rule is very different from that given by more classical models, such as logistic and linear regression analyses, where *linear combinations* are the primary mode of expressing relationships between variables. Indeed, this difference is both the strength of the method and also its weakness.

Statistical inference for tree-based models is in its infancy and far behind that for logistic and linear regression analyses. This is partly because a particular type of *variable selection* underlies tree-based models (e.g., each rule contains only a subset of the available classification or predictor variables, and some may not be used at all). Despite the lack of formal procedures for inference, the method is gaining widespread popularity as a means of devising prediction rules for rapid and repeated evaluation, as a screening method for variables, as a diagnostic technique to assess the adequacy of linear models, and simply for summarizing large multivariate datasets. Some possible reasons for its recent popularity are that:

- in certain applications, especially where the set of predictors contains a mix of numeric variables and factors, tree-based models are sometimes easier to interpret and discuss than linear models;
- tree-based models are invariant to monotone reexpressions of predictor variables so that the precise form in which these appear in a model formula is irrelevant;
- the treatment of missing values (NAs) is more satisfactory for tree-based models than for linear models; and
- tree-based models are more adept at capturing nonadditive behavior; the standard linear model does not allow *interactions* between variables unless they are prespecified and of a particular multiplicative form.

Among the other models covered in this book, tree-based models provide the only means of analysis for factor response variables at more than two levels.

Tree-based models are so-called because the primary method of displaying the fit is in the form of a binary tree. We now provide several examples to motivate the range of application of the methods. The examples are organized according to the type (numeric or factor) of the response variable (y) and the classification or predictor variables ($\boldsymbol{x}$) involved.

9.1.1 Numeric Response and a Single Numeric Predictor

Figure 9.1 displays two views of a tree-based model relating mileage to weight of automobiles in the `car.test.frame` data frame. The left panel of the figure is the

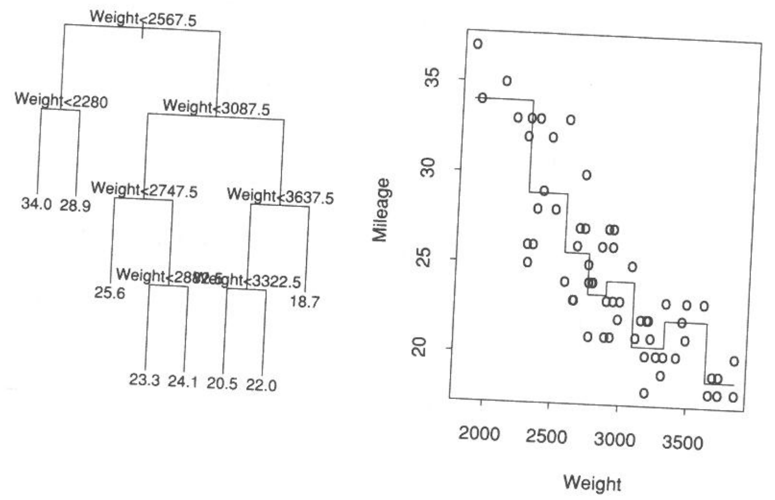


Figure 9.1: *Displays of a tree-based model relating mileage to automobile weight. The plot in the left panel shows how a tree is typically displayed, whereby successive partitions of the data into homogeneous subsets are shown with the rule labeling each split. The overplotting of labels is a common occurrence with this type of display. The plot in the right panel shows a function plot of the same tree together with the actual data values. This representation is only practical for at most two predictor variables.*

standard method of displaying a tree-based model. The idea is that in order to predict mileage from weight, one follows the path from the top node of the tree, called the *root*, to a terminal node, called a *leaf*, according to the rules, called *splits*, at the interior nodes. Automobiles are first split depending on whether they weigh less than 2567.5 pounds. If so, they are again split according to weight being less than 2280 pounds, with the lighter cars having predicted mileage of 34 miles/gallon and the heavier cars having slightly lower mileage of 28.9 miles/gallon. For those automobiles weighing more than 2567.5 pounds, six weight classes are ultimately formed, with predicted mileage varying from 25.6 miles/gallon to a gas-guzzling low of 18.7 miles/gallon. The relationship between mileage and weight seems to behave according to intuition, with heavier cars having poorer mileage than the lighter cars. It appears that doubling the weight of an automobile roughly halves its mileage.

The right panel displays the tree-based model in a more specialized form and one that is more conventional for data of this sort. Here the data themselves and the fitted model are displayed together. As a function of automobile weight, the fitted model is a step function. The height of each step corresponds to the average mileage for automobiles in the weight range under that step. There are a total of eight steps, one for each of the terminal nodes in the tree in the left panel.

9.1.2 Factor Response and Numeric Predictors

The data in this section are from the `kyphosis` data frame introduced in Chapter 6 and analyzed further in Chapter 7. Recall that in those chapters linear and additive logistic models predict the probability of developing `Kyphosis` from the variables `Age`, `Start`, and `Number`. The resulting prediction equations are smooth functions of the first two predictors. By contrast, we now demonstrate tree-based prediction equations that are not smooth but share the essential features of these more traditional analyses.

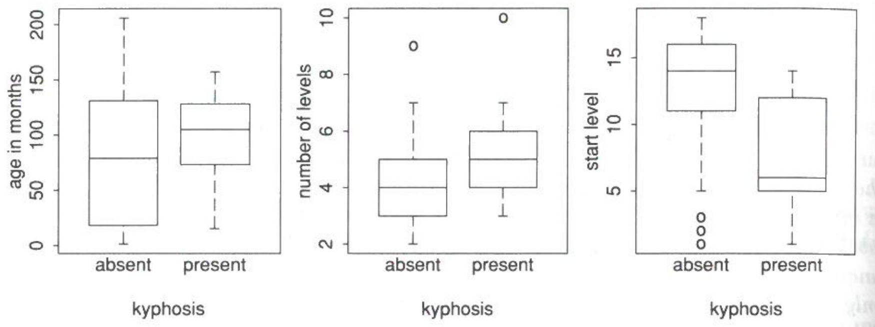


Figure 9.2: *Boxplots of the three numeric predictor variables in the* `kyphosis` *data frame. For each variable, the distribution of individuals with and without* `Kyphosis` *are displayed side by side. The predictor* `Start` *exhibits the greatest difference in these distributions since the lower quartile of those without* `Kyphosis` *is just below the upper quartile of those with* `Kyphosis`.

The distributions of the predictor variables are plotted as a function of `Kyphosis` in Figure 9.2. Of the three predictors, `Start` appears to be the best single predictor since there is a much greater propensity of `Kyphosis` for individuals having `Start`$\leq$ 12 than those with `Start`$>$ 12. The algorithm underlying tree-based prediction determines this cutoff more objectively (by optimization) as 12.5. Moreover, the method then applies the same principle separately to individuals with `Start`$\leq$ 12.5 and those with `Start`$>$ 12.5—namely, comparing the distributions of the predictors as functions of `Kyphosis`. The result of repeated application of this idea leads to the tree displayed in Table 9.1. This semigraphical representation is different from those used in Figure 9.1. It is most useful when the details of the fitting procedure are of interest.

```
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

 1) root 81 83.234001 absent (0.790 0.2100)
   2) Start<12.5 35 47.804001 absent (0.571 0.4290)
     4) Age<34.5 10 6.5019999 absent (0.900 0.1000) *
     5) Age>34.5 25 34.296001 present (0.440 0.5600)
      10) Number<4.5 12 16.301001 absent (0.583 0.4170)
        20) Age<127.5 7 8.3760004 absent (0.714 0.2860) *
        21) Age>127.5 5 6.73 present (0.400 0.6000) *
      11) Number>4.5 13 16.048 present (0.308 0.6920)
        22) Start<8.5 8 6.0279999 present (0.125 0.8750) *
        23) Start>8.5 5 6.73 absent (0.600 0.4000) *
   3) Start>12.5 46 16.454 absent (0.957 0.0435)
     6) Start<14.5 17 12.315 absent (0.882 0.1180)
      12) Age<59 5 0 absent (1.000 0.0000) *
      13) Age>59 12 10.813 absent (0.833 0.1670)
        26) Age<157.5 7 8.3760004 absent (0.714 0.2860) *
        27) Age>157.5 5 0 absent (1.000 0.0000) *
     7) Start>14.5 29 0 absent (1.000 0.0000) *
```

Table 9.1: *A tree-based model for predicting* `Kyphosis`. *The first number after the split is the number of observations. The second number is the* deviance, *which is the measure of node heterogeneity used in the tree-growing algorithm. A deviance of zero corresponds to a perfectly homogeneous node. This term is defined more precisely in Section 9.4.*

The split on `Start` partitions the 81 observations into groups of 35 and 46 individuals (nodes 2 and 3), with probability of `Kyphosis` of 0.429 and 0.0435, respectively. This first group is then partitioned into groups of 10 and 25 individuals (nodes 4 and 5), depending on whether `Age` is less than 34.5 years or not. The former group, with probability of `Kyphosis` of 0.10, is not subdivided further. The latter group is subdivided into groups of 12 and 13 individuals (nodes 10 and 11), depending on whether or not `Number` is less than 4.5. The respective probabilities for these groups are 0.417 and 0.692. This procedure continues, yielding nine distinct probabilities of `Kyphosis` ranging from 0.0 to 0.875. Clearly, as the partitioning continues, our trust in the individual estimated probabilities decreases as they are based on less and less data. Many of the tools discussed in Section 9.2 are aimed at assessing the degree of over- or underfitting of a tree-based model.

9.1.3 Factor Response and Mixed Predictor Variables

The data are from the `market.survey` data frame introduced in Chapter 3 and subsequently analyzed in Chapters 6 and 7. Here we briefly review the available data, which were obtained from a survey of 1000 people; for now, we concentrate on the 759 individuals for whom complete data were obtained. The aim of the survey was to identify segments of the residential long-distance market, where AT&T should concentrate its marketing efforts. The variables collected include household income (`income`), number of household moves in the past five years (`moves`), age of respondent (`age`), education level (`education`), employment category (`employment`), average monthly usage (`usage`), whether the respondent has a nonpublished phone number (`nonpub`), whether the respondent participates in the Reach Out America Plan (`reach.out`), whether the respondent holds a calling card (`card`), and the respondent's chosen long-distance carrier (`pick`).

The tree in Figure 9.3 provides a particularly simple prediction rule for long-distance carrier. For average usage of more than \$12.50 per month, the preferred choice is AT&T. For average usage of less than \$12.50 per month, the choice depends on whether the respondent has a nonpublished directory listing. If so, then AT&T is again the preferred choice, but if the directory listing is published, then an "other common carrier" (OCC) is preferred. (Evidently the OCC folks did some telemarketing themselves!)

9.2 S Functions and Objects

Our approach is not to have a single function for tree-based modeling, but rather a collection of functions, which, together with existing S functions, form a basis for building and assessing this new class of models. Our implementation centers around the idea of a tree object. This object provides commonality among functions to grow, manipulate, and display trees.

9.2.1 Growing a Tree

There is a single function to grow a tree, named `tree()`. The expression

```
> z.auto <- tree(Mileage ~ Weight, car.test.frame)
```

grows a regression tree using the variables `Mileage` and `Weight` from the data frame `car.test.frame` and gives the name `z.auto` to the resulting tree object. Similarly, the expression

```
> z.kyph <- tree(Kyphosis ~ Age + Number + Start, kyphosis)
```

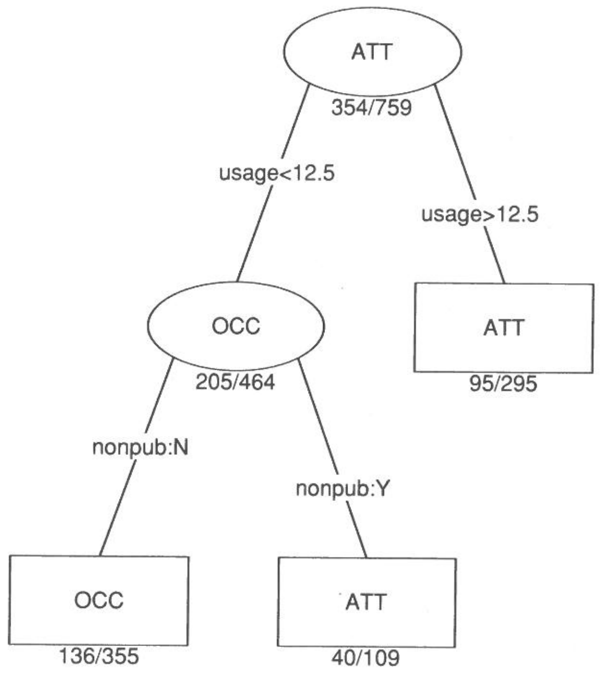


Figure 9.3: *A display of a tree fitted to the long-distance marketing data. This form of tree display is primarily for presentation purposes as it conceals the details of the tree-growing process. The edges connecting the nodes are labeled by the left and right splits. Interior nodes are denoted by ellipses and terminal nodes by rectangles, with the predicted value of the response variable centered in the node. The number under each terminal node is the misclassification error rate; for example, in the rightmost node, which is labeled ATT, 95 out of the 295 respondents in the node actually picked OCC.*

grows a classification tree using the variables from the data frame `kyphosis` and gives the name `z.kyph` to the resulting tree object. The function `tree()` automatically distinguishes between regression and classification trees according to whether the response variable is numeric or a factor. It implements a binary recursive partitioning algorithm described in Section 9.4. The only detail relevant to the present discussion is that the algorithm adds nodes until they are homogeneous or contain too few observations (≤ 5, by default).

The function `tree()` takes two arguments, a `formula` object and a `data.frame`, either of which can be missing. As with all modeling functions, a missing `data.frame` argument simply means that the functions expect the variables named in `formula` to be in the search list. If `formula` is missing, then it is constructed automatically from the data.frame using the first variable as the response. For example, an equivalent expression defining z.kyph is tree(kyphosis). Valid formulas for trees allow all standard manipulations of variables such as cut(), log(), I(), etc. These are seldom used on the right side of a formula since trees are invariant to monotone reexpressions of individual predictor variables. The only meaningful operator in a formula for trees is " + ," indicating which variables are to be included as predictors. This is so because trees capture interactions without explicit specification. Given these points, it may seem that formulas for trees are a gross overkill as a means of specifying the terms used in the model. Nonetheless, they provide a convenient means to specify reexpressions of the response variable and, more importantly, to facilitate applying quite different models to the same data.

A tree object contains information regarding the partitioning of the predictor variables into homogeneous regions that is required by subsequent functions for manipulating and displaying trees. Predictably, a tree object has class "tree". Generic functions such as summary(), print(), plot(), residuals(), and predict() work as expected for objects of class "tree". A summary of a fitted tree-based model is available by the summary() function:

```
> summary(z.auto)

Regression tree:
tree(formula = Mileage ~  Weight, car.test.frame)
Number of terminal nodes:  8
Residual mean deviance:   4.208 = 218.819 / 52
Distribution of residuals:
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 -3.889 -1.111   0.000  0.000  1.167   4.375

> summary(z.kyph)

Classification tree:
tree(formula = Kyphosis ~ Age + Number + Start, kyphosis)
Number of terminal nodes:  9
Residual mean deviance:  0.594 = 42.742 / 72
Misclassification error rate: 0.123 = 10 / 81
```

Notice that there is some difference in the summary depending on whether the tree is a classification or a regression tree.

A tree prints using indentation as a key to the underlying structure. Since print() is invoked upon typing the name of an object, a tree can be printed simply by typing its name. The example given in Table 9.1 was constructed with the expression z.kyph. The amount of information displayed by print() relative to summary() might seem disproportionate for objects of class "tree", but the philosophy that print() should provide a quick look at the object is maintained, as it does

little more than format the contents of a tree object. The summary() function on the other hand does involve computation that can result in less than instantaneous response.

Subtrees

A subtree of a tree object can be selected or deleted in a natural way through subscripting; for example, a positive subscript corresponds to selecting a subtree and a negative subscript corresponds to deleting a subtree. This implies that there is an ordering or index to tree objects that permits identification by number. Indeed, nodes of a tree object are numbered to succinctly capture the tree topology and to provide quick reference. An example of the numbering scheme is that given in Table 9.1 for the tree grown to the kyphosis data. Descendants of node number 3 can be removed, or a new subtree can be rooted at node 3, as follows:

```
> z.kyph[-3]
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

 1) root 81 83.234 absent (0.790 0.2100)
   2) Start<12.5 35 47.804 absent (0.571 0.4290)
     4) Age<34.5 10 6.502 absent (0.900 0.1000) *
     5) Age>34.5 25 34.296 present (0.440 0.5600)
      10) Number<4.5 12 16.301 absent (0.583 0.4170)
        20) Age<127.5 7 8.376 absent (0.714 0.2860) *
        21) Age>127.5 5 6.73 present (0.400 0.6000) *
      11) Number>4.5 13 16.048 present (0.308 0.6920)
        22) Start<8.5 8 6.028 present (0.125 0.8750) *
        23) Start>8.5 5 6.73 absent (0.600 0.4000) *
   3) Start>12.5 46 16.454 absent (0.957 0.0435) *

> z.kyph[3]
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

 3) Start>12.5 46 16.454 absent (0.957 0.0435)
   6) Start<14.5 17 12.315 absent (0.882 0.1180)
    12) Age<59 5 0 absent (1.000 0.0000) *
    13) Age>59 12 10.813 absent (0.833 0.1670)
      26) Age<157.5 7 8.376 absent (0.714 0.2860) *
      27) Age>157.5 5 0 absent (1.000 0.0000) *
   7) Start>14.5 29 0 absent (1.000 0.0000) *
```

Implicit in our discussion above is that a subtree of a tree object is itself a tree object. This allows a subtree to be printed with the same ease as the original tree.

The importance of tree subscripting becomes apparent as tree size gets larger. For example, consider growing a tree to the long-distance marketing data:

```
> z.survey <- tree(market.survey, na.action = na.omit)
```

The tree displayed earlier in Figure 9.3 is a particularly terse summary of this tree obtained with the expression `z.survey[-c(4,5,3)]`. The complete tree, `z.survey`, is displayed using the `plot()` function in Figure 9.4. The function displays a tree as an unlabeled dendrogram, rooted at the top of the figure. The `plot.tree()` method takes an optional argument, `type=`, which controls node placement. The default is nonuniform spacing whereby the vertical position of a node pair is a function of the importance of the parent split. It is particularly appropriate during analysis where the primary consideration is often one of tree simplification. The alternate (`type="u"`) behavior uses node depth to guide vertical placement of nodes. This results in a uniform layout that is useful for subsequent labeling. The tree displayed in the left panel of Figure 9.4 was obtained with the default node spacing, e.g., `plot(z.survey)`, while that in the right panel was obtained by `plot(z.survey, type = "u")`. In the former plot, the importance of the first few splits is readily apparent. This insight is at the expense of reduced resolution at the leaves of the tree, where detail is arguably of lesser importance.

Labeling a tree is distinct from plotting a tree. The size of the tree displayed in Figure 9.4 demonstrates why two separate functions are required; once the tree is plotted, labeling may or may not follow depending on its topology. The `text()` method for trees provides a means to label the dendrogram displayed by `plot()`. The user has control over what components of the tree object are used as labels at interior or leaf nodes. The tree displayed in the left panel of Figure 9.1 was labeled with `text(z.auto)`.

Tree-based modeling is similar in many ways to that discussed in previous chapters. An important similarity is the degree to which tools to diagnose model adequacy are applied. Figure 9.5 displays two commonly used plots for regression models as applied to the automobile mileage example—namely, a scatterplot of residuals versus fitted values and a normal probability plot of residuals. The fitted values are obtained with the expression `predict(z.auto)`. The residuals, *observed–fitted*, are obtained by subtracting the fitted values from the response variable, or directly with the expression `residuals(z.auto)`. The normal probability plot does not suggest any unusual patterns, but the plot of residuals versus fitted values demonstrates heteroscedasticity. This pattern, together with the moderate curvature demonstrated in Figure 9.1, suggests that a reexpression of the response variable, say from miles per gallon to gallons per mile, might be more appropriate.

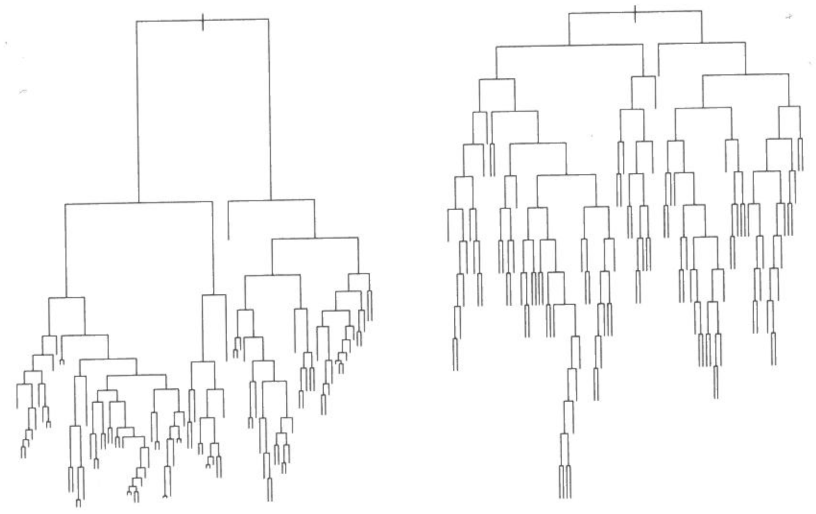
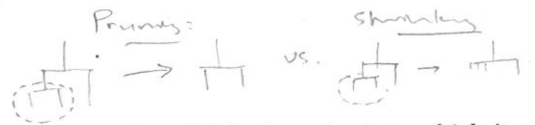


Figure 9.4: *Dendrograms of the tree* `z.survey` *grown to the long-distance marketing data. The dendrogram on the left uses the change in deviance to guide the vertical positioning of each pair of nodes. Resolution at the leaves of the tree is sacrificed to provide a visual cue of split importance. The dendrogram on the right uses node depth to guide the placement of each node. (The root has depth 0.)*



Pruning and Shrinking

Another aspect of assessing a fitted tree-based model is the extent to which it can be simplified without sacrificing goodness-of-fit. This is also an important consideration for prediction. Since tree size is intentionally not limited in the growing process, a certain degree of overfitting has occurred. There are two ways to address this problem; the one to choose depends upon whether the primary concern is parsimonious description or accurate prediction.

Figure 9.6 displays three variations of `z.kyph`, the classification tree grown to the `kyphosis` data. The first panel is the dendrogram for the full tree with nine terminal nodes. The second panel is a *pruned* version with three terminal nodes. The third panel is a *shrunken* version with nine *actual* terminal nodes and about three *effective* terminal nodes. Note that the pruned tree shares the same estimated probabilities as the full tree but that apart from the root node, those of the shrunken tree are completely different. Summaries of the pruned and shrunken trees are:

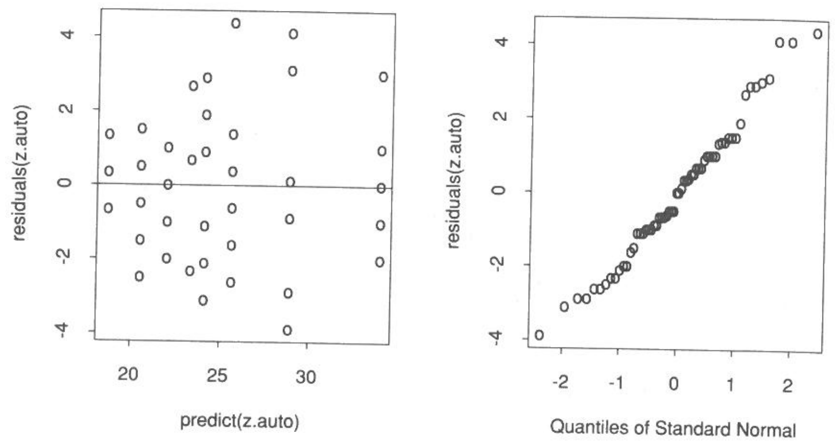


Figure 9.5: *Two standard diagnostic plots for regression data as applied to the fit described by* `z.auto`*. The plot in the left panel is that of residuals versus fitted values. The plot in the right panel is a normal probability plot of residuals. These plots suggest that there are no apparent outliers but that the variance seems to increase with level.*

```
> summary(zp.kyph)

Classification tree:
prune.tree(tree = z.kyph, k = 5)
Variables actually used in tree construction:
[1] "Age"    "Start"
Number of terminal nodes:  3
Residual mean deviance:  0.734 = 57.252 / 78
Misclassification error rate: 0.173 = 14 / 81

> summary(zs.kyph)

Classification tree:
shrink.tree(tree = z.kyph, k = 0.25)
Number of terminal nodes:  9
Effective number of terminal nodes:  2.8
Residual mean deviance:  0.739 = 57.754 / 78.2
Misclassification error rate: 0.136 = 11 / 81
```

Which tree is better? In one sense, the pruned tree, since it provides a much more

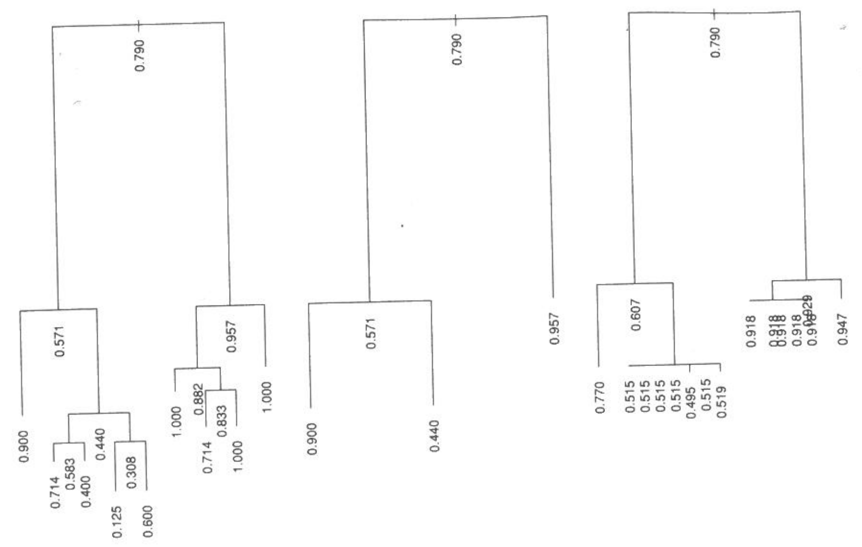


Figure 9.6: *Three variations of the tree grown to the* `kyphosis` *data. All plots are on a common scale, and nonuniform (vertical) spacing of nodes is used. The nodes are labeled with the estimated probability that* `Kyphosis==absent`*. The node labels have been rotated to improve readability. The first panel is the full tree* `z.kyph`*; it is a graphical representation of the tabular version presented in Table 9.1. The second panel is a pruned version of* `z.kyph`*, whereby the least important splits have been pruned off. Note that the estimated probabilities and node heights match those of the full tree. The third panel is a shrunken version of* `z.kyph`*, whereby the estimated probabilities have been pulled back or shrunken toward the root. Apart from the root, neither the estimated probabilities nor the node heights match those of the full tree. The squashing of the dendrogram at the bottom indicates that these nodes have been shrunk completely to their parents.*

succinct description of the data (note that only two out of the three predictors remain). In another sense, the shrunken tree, since its misclassification error rate is lower than that of the pruned tree. Thus, there is no hard and fast rule on which is better; the choice depends on where your priorities lie (simplicity versus accuracy). We now proceed to describe these methods in more detail.

The function `prune.tree()` takes a tree object as a required argument. If no additional arguments are supplied, it determines a nested sequence of subtrees of the supplied tree by recursively snipping off the least important splits. Importance is captured by the cost-complexity measure:

$$D_\alpha(T') = D(T') + \alpha size(T')$$

T' = subtree   size(T') = number of terminal nodes

[illegible] of subtree

where $D(T')$ is the deviance of the subtree T', $size(T')$ is the number of terminal nodes of T', and α is the cost-complexity parameter. For any specified α, cost-complexity pruning determines the subtree T' that minimizes $D_\alpha(T')$ over all subtrees of T. The optimal subtree for a given α is obtained by supplying `prune.tree()` with the argument `k=`α. For example, the tree displayed in the second panel of Figure 9.6 was obtained by `prune.tree(z.kyph, 5)`. If `k=`α is a vector, the sequence of subtrees that minimize the cost-complexity measure is returned rather than a tree object.

The function `shrink.tree()` takes a tree object as a required argument. If no additional arguments are supplied, it determines a sequence of trees of the supplied tree that differ in their fitted values. A particular tree in the sequence is indexed by α, which defines *shrunken* fitted values according to the recursion:

$$\hat{y}(node) = \alpha\bar{y}(node) + (1-\alpha)\hat{y}(parent)$$

where $\bar{y}(node)$ is the usual fitted value for a node, and $\hat{y}(parent)$ is the *shrunken* fitted value for the node's parent—that is, it was obtained by applying the same recursion. The function `shrink.tree()` uses a particular parametrization of α that *optimally* shrinks children nodes to their parent based on the magnitude of the difference between $\bar{y}(node)$ and $\bar{y}(parent)$. The sequence is anchored between the full tree ($\alpha = 1$) and the root node tree ($\alpha = 0$). A heuristic argument allows one to map α into the number of *effective* terminal nodes, thereby facilitating comparison with pruning. The tree for a given α is obtained by supplying `shrink.tree()` with the argument `k=`α. For example, the tree displayed in the third panel of Figure 9.6 was obtained by `shrink.tree(z.kyph, .25)`. If `k=`α is a vector, the sequence of trees that are determined by these shrinkage parameters is returned rather than a tree object.

Figure 9.7 displays the sequences for pruning and shrinking `z.survey`. These are obtained by omitting the `k=` argument and plotting the resulting object. These objects have class `"tree.sequence"` for which a `plot()` method exists. Each panel displays the deviance versus size (the number of terminal nodes or the number of *effective* terminal nodes) for each tree in the sequence. An additional (upper) axis shows the mapping between size and `k` for each method. By construction, the deviance decreases as tree size increases, a common phenomenon in model-fitting (i.e., the fit improves as parameters are added to the model). This limits the usefulness of the plot except in those situations where a dramatic change in deviance occurs at a particular value of `k`.

It should not be surprising that the sequences produced by these methods provide little guidance on what size tree is adequate. The same data that were used to grow the tree are being asked to provide this additional information. But since the tree was optimized for the supplied data, the tree sequences have no possible alternative but to behave as observed. There are two ways out of this dilemma:

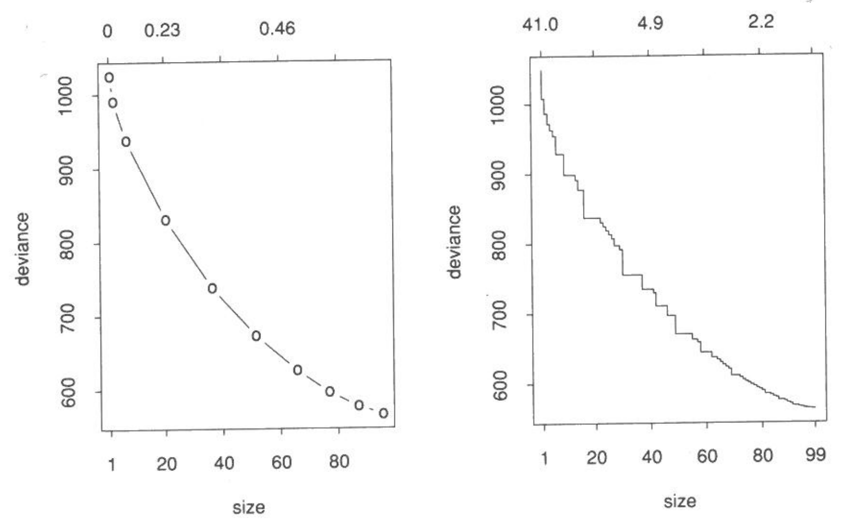


Figure 9.7: *Plots of deviance versus size (number of terminal nodes) for sequences of subtrees of* `z.survey`. *The left panel is based on optimal shrinking while the right panel is based on cost-complexity pruning. The former is plotted as a continuous function to reinforce its continuous behavior. The latter is plotted as a step function because optimal subtrees remain constant between adjacent values of* `k`. *Each panel has an additional axis along the top indicating the values of* `k` *that correspond to the different sized subtrees in the sequence.*

one is to use new (independent) data to guide the selection of the right size tree, and the other is to reuse the existing data by the method of *cross-validation*. In either case, the issue of tree-based prediction of new data arises. Let's pursue this diversion before returning and concluding our discussion of choosing the right size tree.

Prediction

An important use of tree-based models is predicting the value of a response variable for a known set of predictor variables. By prediction we mean to evaluate the splits describing a tree-based model for a set of predictor variables and defining the `yval` at the deepest node reached as the prediction. Normally this corresponds to a leaf node of the tree, but we adopt the convention that a prediction may reside in a nonterminal node if, in following along the path defined by the set of predictor variables for a new observation, a value of a predictor is encountered that has never been seen at that node in the tree-growing process. The classic case of this is encountering a missing value (`NA`) when only complete observations were used to grow the tree. More generally and more subtly, this condition occurs for factor predictors whenever a split is encountered where the value goes neither left nor right (e.g., if $x = B$ and the left and right splits at a node are, respectively, $x \in \{A, C\}$ and $x \in \{D, E\}$).

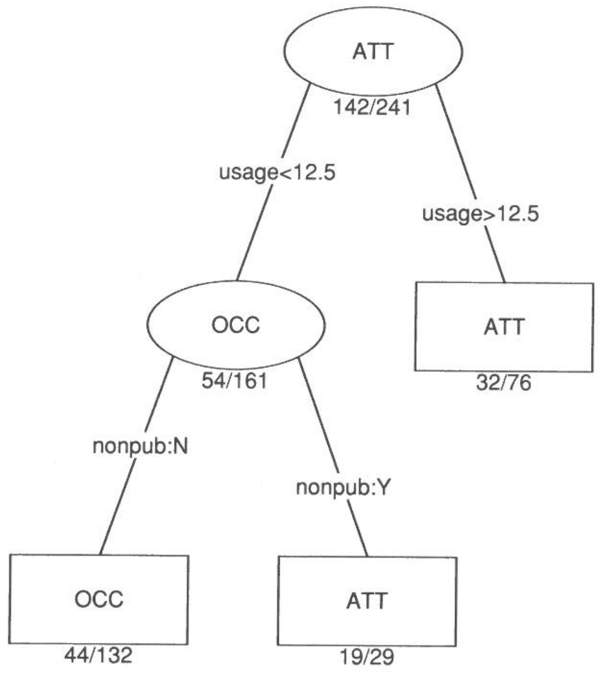


Figure 9.8: *Tree representation of prediction from the classification tree* `zs.survey`. *The node labels are the predicted values of* `pick`. *The numbers displayed under each node represent the misclassification error rate for the new data* `na.market.survey`. *The overall misclassification error rate is quite high (0.41). Four of the respondents remain at the root node due to missing values in the predictor* `usage`.

We return to the long-distance marketing example where we illustrate prediction using an additional 241 survey respondents. These respondents were part of the initial survey but were omitted from the preliminary analysis because of missing values in the variables. The data are collected in the data frame `na.market.survey`. Predictions from the tree in Figure 9.3 are displayed in Figure 9.8. The figure shows the disposition of the 241 observations along the prediction paths of the tree. Of the 241 observations, 161 are directed to the left (OCC), 76 to the right (ATT), and 4 remain at the root node (due to missing values for `usage`). Of the 161, 132 are directed to the left (OCC) and 29 to the right (ATT). The misclassification error rate associated with these predictions is quite high (41%). This error rate varies from leaf node to leaf node, from 33% (leftmost leaf), to 66% (middle leaf), to 42% (rightmost leaf).

The tree displayed in Figure 9.8 was obtained with the expression:

```
zd.survey <- predict(zs.survey, na.market.survey, type = "tree")
```

The `predict()` method takes a tree object and a data frame. The tree object is likely to be a simplified version of that provided by `tree()`. The names of the variables in the data frame must include the predictors in the formula used to construct the tree. The function returns the values predicted by the tree for the data in the data frame, either as a vector (the default) or as a tree object, `type="tree"`. If a data frame is not supplied, `predict()` returns the fitted values for the data used to construct the tree; we used this feature in our earlier discussion of residual plots.

Cross-validation

We now return to the topic of choosing the right size tree based on data not used to grow the tree. Test data can be supplied to the functions `prune.tree()` and `shrink.tree()` with the `newdata=` argument. The functions return an object of class `"tree.sequence"` containing the sequence evaluated on the test data. Figure 9.9 illustrates this functionality for the market survey data, where the new data consist of those held back due to missing values. These plots span a wide range of tree sizes, but the most promising are those with fewer than a dozen nodes. The range can be restricted by suitable specification of the argument `k`. Panel 1 of Figure 9.10 demonstrates such a restriction for `k` in the range 0.05 to 0.20 for the optimal shrinking sequence. Evidently, either a very small tree is called for or the data with NAs are not drawn from the same population as those without.

The function `cv.tree()` can be used to address this ambiguity by applying a procedure described in Section 9.3 called *cross-validation*. The basic idea is to divide the original data into mutually exclusive sets. For each set, a tree is grown to the remaining sets and a subtree sequence obtained; the set *held out* is then used to evaluate the sequence. Deviances from each set are accumulated (as a function of `k`) and returned as an object of class `"tree.sequence"`. A plot of the cross-validated deviance versus tree size is seldom monotone decreasing since data used to evaluate the sequences were not used to construct them. A common feature of the plot is a fairly flat minimum, and trees in this region are candidates for further consideration. The result of tenfold cross-validation of the tree `z.survey` is displayed in the right panel of Figure 9.10. The plot was obtained by the expressions

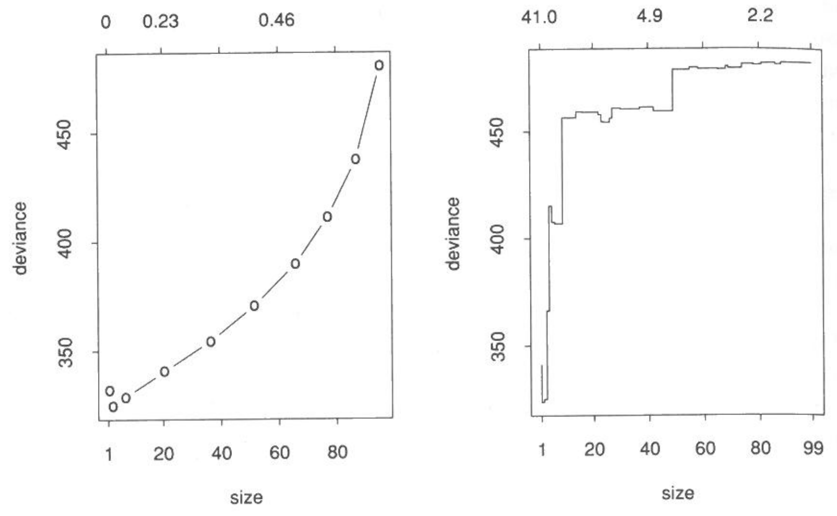


Figure 9.9: *Plots of deviance versus size for sequences of subtrees of* `z.survey` *evaluated on new data. The new data are from the data frame* `market.survey` *but were omitted from the fit due to missing values in some of the predictors. The* `na.tree.replace()` *function was used to replace NAs with an additional factor level. Since the original tree was constructed from data without missing values, this in effect means that when the new level* `"NA"` *is encountered, the deviance at that node is used. The left panel is based on optimal shrinking while the right panel is based on cost-complexity pruning. Comparison with Figure 9.7 highlights the differences in these sequences when based on training and independent test data. This figure suggests that either a very simple tree (at most three nodes) be used to summarize these data, or that the two datasets, those with and those without NAs, are qualitatively different.*

```
> k <- seq(.05, .20, length = 10)
> cv.survey <- cv.tree(z.survey, r.survey, k = k)
> plot(cv.survey, type = "b")
```

The dataset `r.survey` contains a random permutation of the integers 1 to 10, of length `length(pick)`, denoting the assignment of the observations into 10 mutually exclusive sets. The function `cv.tree()` will determine a permutation by default, but it is often useful to specify one, especially if comparison with another sequencing method is desired. The final argument, `FUN=`, specifies which sequencing function is to be used; the default is `shrink.tree`.

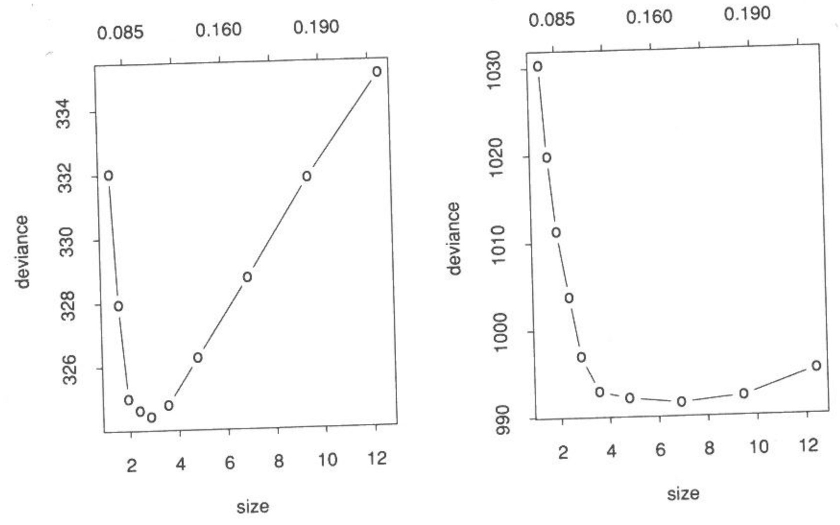


Figure 9.10: *Plots of deviance versus size for sequences of shrunken trees of* `z.survey`. *The range of trees considered was restricted to values of* `k` *between 0.05 and 0.20, corresponding to trees with effective size from 1 to 12. The left panel is based on evaluating the sequence on new data while the right panel is based on cross-validation. The left panel provides sharp discrimination in tree size, strongly suggesting a three-node tree. The right panel is not so sharp and is typical of sequences computed by cross-validation. Even so, a modest seven-node tree is suggested.*

9.2.2 Functions for Diagnosis

Residual analysis is important and not peculiar to a single class of models. In the case of trees, it is natural to exploit the very representation that is used to capture and describe the fitted model—namely, the dendrogram—as the primary means of diagnosis. We now introduce functions that utilize the tree metaphor to facilitate and guide diagnosis. The functions divide themselves along the natural components of a tree-based model—namely, *subtrees, nodes, splits,* and *leaves.* Most of the methods involve *interacting* with trees, and by this we usually mean *graphical interaction.* We note parenthetically, and sometimes explicitly below, that all the functions can be used noninteractively (by including a list of node numbers as an argument), but their usefulness seems to be significantly enhanced when used interactively.

In certain of the figures in this section, a (new) general mechanism to obtain multiple figures within the S graphics model is used. The *split-screen mode* is an alternative to par(mfrow) that allows arbitrary rectangular regions (called *screens*) to be specified for graphics input and output. We use this mechanism rather than the standard multifigure format not only to attain a more flexible layout style, but also because the order in which screens are accessed is under user control. It is able, for example, to arbitrarily receive graphics input from one screen and send graphics output to another. We have attempted to restrict our use of the split-screen mode to minimize the introduction of too much ancillary material. A single function tree.screens(), called without arguments, will set up a generic partition of the figure region used by the tree-specific functions that we provide. See the detailed documentation of split.screen() for further information.

9.2.3 Examining Subtrees

The function snip.tree() allows the analyst to *snip off* branches of a tree either through a specified list of nodes, or interactively by graphic input. For the former, the subset method for tree objects described earlier, "[.tree"(), is a convenient shorthand. For example, the expression z.auto[-2] is equivalent to the expression snip.tree(z.auto, 2). This usage requires knowing the number of the node or nodes in question; the interactive approach obviates this need. It is most convenient when working at a high-resolution graphics terminal and provides a type of *what-if* analysis on the displayed tree. The graphical interface is such that a single click of the graphics input device (e.g., a mouse) informs the user of the change in tree deviance that would result if the subtree rooted at the selected node is snipped off; a second click on the same node actually does the snipping. By snipping, we mean that the tree object is modified to reflect the deleted subtree and also that the portion of the plotted dendrogram corresponding to the subtree rooted at the selected node is "erased." The process can be continued, and, on exit, what remains of the original tree is returned as a tree object. An example of the textual information displayed during this process is as follows:

```
> zsnip.survey <- snip.tree(z.survey)
node number:  4
   tree deviance =  562.518
   subtree deviance =  741.663
node number:  10
   tree deviance =  741.663
   subtree deviance =  786.214
node number:  7
   tree deviance =  786.214
   subtree deviance =  962.767
```

Here we first selected and then reselected nodes 4, 10, and 7 of the tree z.survey. Note how the subtree deviance at one stage becomes the tree deviance at the next stage. The graphical result of this process is displayed in Figure 9.11. The second

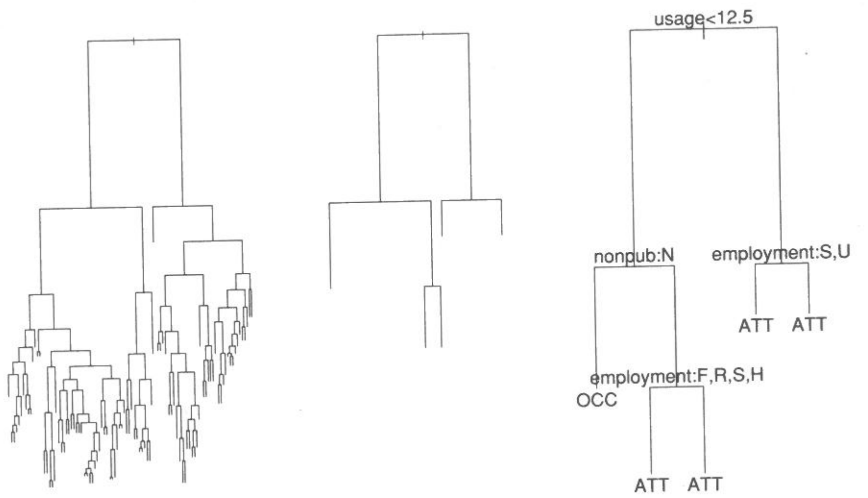


Figure 9.11: *An illustration of interactive snipping of subtrees. The full tree* z.survey *is plotted in the first panel. Upon selection of a node, the change in deviance that would result by snipping off the subtree rooted at that node is displayed. If it is reselected, the subtree is* snipped off, *which has the side effect of erasing the subtree from the dendrogram. The second panel shows what remains of the tree after the subtrees rooted at nodes 4, 10, and 7 are snipped off. The final panel replots and labels the snipped tree.*

panel shows the result of snipping off the subtrees rooted at nodes 4, 10, and 7. The final panel replots the snipped tree zsnip.survey and labels it. This points out one reason for snipping—gaining resolution at the top of the tree so that it can be usefully labeled. The node numbers of the branches that were snipped off are collected together and pasted into the call component of the tree object to inform the user that the result was obtained by snipping nodes so-and-so from tree such-and-such. For example, the call component of zsnip.survey is

```
> zsnip.survey$call
snip.tree(tree = z.survey, nodes = c(4, 10, 7))
```

The function select.tree() is the dual of snip.tree(). It allows individual subtrees of a specified tree to be selected and assigned. For each node number supplied, the function returns a tree object rooted at that node. If no nodes are supplied, the function expects them to be selected by graphical interaction. When more than one node is specified or selected, the subtrees are organized as a list, with the node number naming the individual elements. One might reasonably call

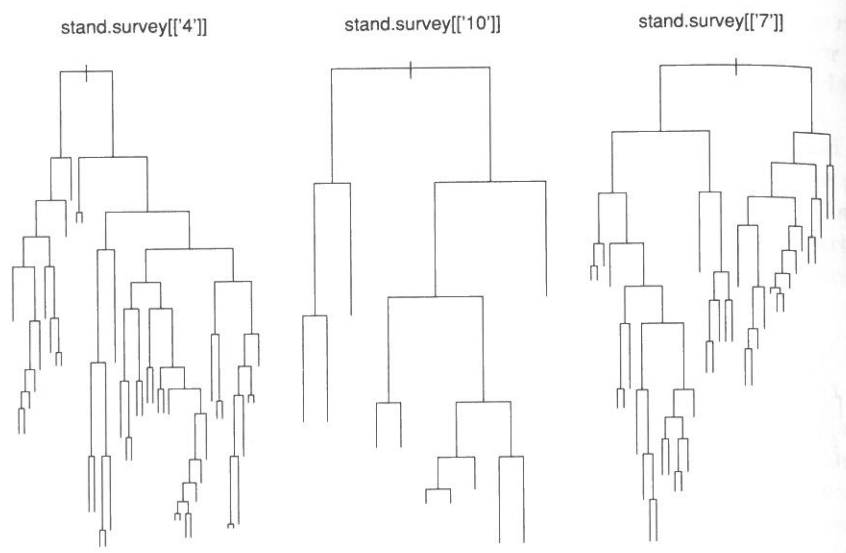


Figure 9.12: *An illustration of a stand of trees. The three panels contain the subtrees of* `z.survey` *that were* snipped off *in Figure 9.11. Each tree in the stand is amenable to all methods for tree objects, including plot methods. The panels in the figure were obtained by applying the* `plot()` *method to the stand* `stand.survey`.

such a list a *stand* (of trees). An interesting feature of stands results from the fact that the trees it contains are bona fide tree objects. Thus, they are amenable to any and all display and analysis functions for trees. A useful way to peruse a stand is by applying a function to it using `apply()`. For example, Figure 9.12 is obtained by the expression

```
> stand.survey <- select.tree(z.survey, nodes = c(4, 10, 7))
> sapply(stand.survey, plot)
```

Like `snip.tree()`, the subset method for tree objects, `"[.tree"()`, is a convenient shorthand for `select.tree()`. For example, `z.survey[c(4, 10, 7)]` is equivalent to the expression given above for `stand.survey`. Also like `snip.tree()`, the `call` component of a selected subtree is constructed to inform the user that the result was obtained by selecting subtree so-and-so from tree such-and-such.

9.2.4 Examining Nodes

Much information concerning a fitted tree resides in the nodes. It is important that this information be readily available, and yet, there is too much information to

usefully label a dendrogram with. We now introduce some tree-specific functions to encourage users to browse the nodes of a fitted tree-based model. Let's introduce a new example based on the data frame `cu.summary` described in Section 3.1.1. The data are summarized as follows:

```
summary(cu.summary)
      Price              Country            Reliability
 Min.   : 5866     USA       :49     Much worse :18
 1st Qu.:10090     Japan     :31     worse      :12
 Median :13150     Germany   :11     average    :26
 Mean   :15740     Japan/USA: 9      better     : 8
 3rd Qu.:19160     Sweden    : 5     Much better:21
 Max.   :41990     Korea     : 5     NAs        :32
                   (Other)   : 7

      Mileage             Type
 Min.   :18.00     Compact:22
 1st Qu.:21.00     Large  : 7
 Median :23.00     Medium :30
 Mean   :24.58     Small  :22
 3rd Qu.:27.00     Sporty :26
 Max.   :37.00     Van    :10
 NAs    :57
```

The model we entertain addresses the relationship of automobile characteristics to automobile reliability. The fitted tree-based model is obtained by the expression

```
> f.cu <- formula(Reliability ~ Price + Country + Mileage + Type)
> z.cu <- tree(f.cu, cu.summary, na.action = na.tree.replace)
```

and is plotted in Figure 9.13. Since this is a classification tree with a five-level response variable, much information has been suppressed in the labeled dendrogram. Node contents may be inspected with the `browser()` method for trees, which takes a tree object as a required argument and an optional list of nodes. If the latter is omitted, the function waits for the user to select nodes with the graphics input device. For example, clicking on the left-child of the root node of the tree `z.cu` yields:

```
> browser(z.cu)
node number: 2
 split: Country:Japan,Japan/USA
 n: 27
 dev: 36.9219
 yval: Much better
 Much worse worse   average    better Much better
          0     0 0.1111111 0.1111111   0.7777778
```

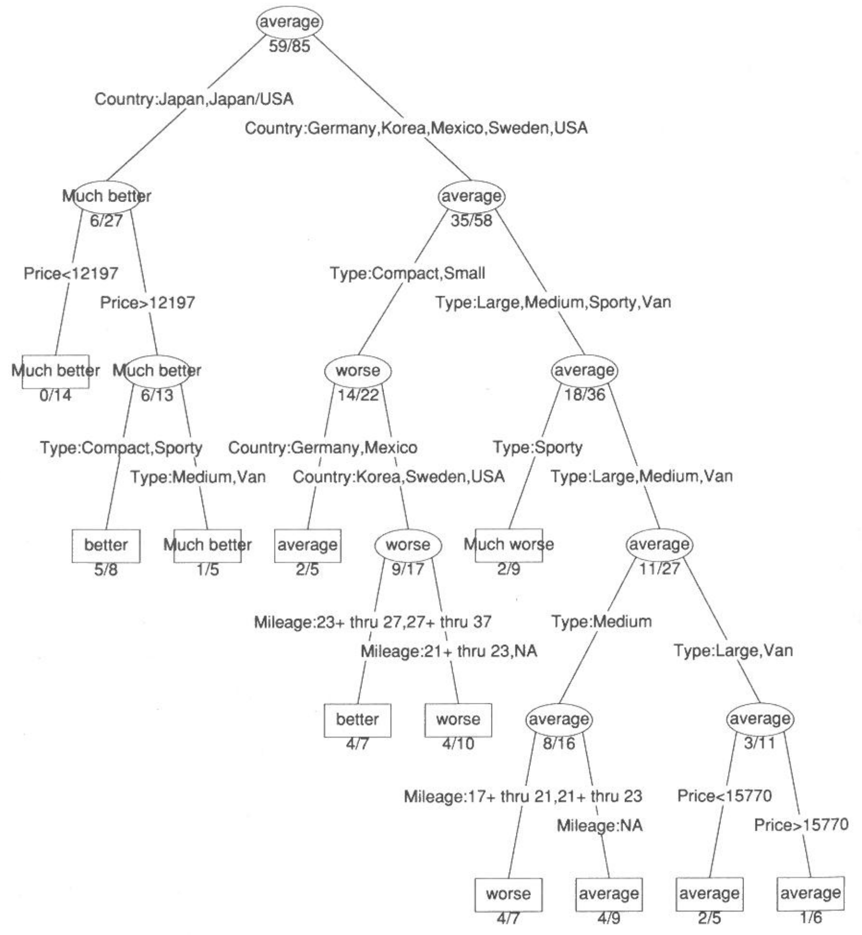


Figure 9.13: *A display of a tree fitted to the automobile reliability data. The response variable has levels* `Much Worse`, `worse`, `average`, `better`, `Much Better`. *The predicted value of the response variable is centered in the node. The number under each terminal node is the misclassification error rate. The split at the root node suggests that Japanese cars, whether manufactured here or abroad, have much better perceived reliability than cars of other nationalities.*

The `identify()` method also takes a tree object as a required argument and an optional list of nodes. If the latter is omitted, the function waits for the user to select nodes from the dendrogram. The function returns a list, with one component for each node selected, containing the names of the observations falling in the node. For example, clicking on the leftmost node of the tree `z.cu` yields:

```
> identify(z.cu)
node number: 4
   Acura Integra 4
   GEO Prizm  4
   Honda Civic 4
   Mazda Protege 4
   Nissan Sentra 4
   Subaru Loyale 4
   Toyota Corolla 4
   Toyota Tercel 4
   Honda Civic CRX Si 4
   Honda Accord 4
   Nissan Stanza 4
   Subaru Legacy 4
   Toyota Camry 4
```

The "4" following each automobile name is actually part of the name (these are all four-cylinder cars) and has nothing to do with the fact that node 4 was selected. If the result of `identify()` is assigned, these names can then be used as subscripts to examine data specific to individual nodes. The following expressions demonstrate how the predictor `Price` varies for observations in nodes 2 and 3:

```
> node2.3 <- identify(z.cu, 2:3)
> quantile(Price[node2.3[["2"]]])
[1]  6488.00  9730.50 12145.00 17145.25 24760.00
> quantile(Price[node2.3[["3"]]])
[1]  5899.0  9995.0 13072.5 20225.0 39950.0
```

Nodes 2 and 3 are the left and right children, respectively, of the root node. Given that the more reliable cars follow the left path rather than the right, apart from the least expensive automobiles, it appears that you pay more for more troublesome cars!

The function `path.tree()` allows the user to obtain the *path* (sequence of splits) from the root to any node of a tree. It takes a tree object as a required argument and an optional list of nodes. If the latter is omitted, the function waits for the user to select nodes from the dendrogram. The function returns a list, with one component for each node specified or selected. The component contains the sequence of splits leading to that node. In interactive mode, the individual paths are (optionally) printed out as nodes are selected. The function is useful in those cases where tree size or label lengths are such that severe overplotting results if the tree is labeled indiscriminately. For example, selecting one of the deep nodes of the tree `z.cu` yields:

```
> path.tree(z.cu)
node number: 26
   root
   Country:Germany,Korea,Mexico,Sweden,USA
   Type:Compact,Small
   Country:Korea,Sweden,USA
   Mileage:23+ thru 27,27+ thru 37
```

By examining the path, we can see that the automobiles in this node consist of those manufactured in Korea, Sweden, and USA, which are compact or small, and for which the reported mileage is between 23 and 37 mpg.

9.2.5 Examining Splits

The tree grown to the automobile reliability data suggests that Japanese cars, whether manufactured here or abroad, are more reliable than cars of other nationalities. Should we believe this? The answer in general is no; the recursive partitioning algorithm underlying the `tree()` function is just that: an algorithm. There may well be other variables, or even other partitions of the variable `Country`, that discriminate reliable from unreliable cars, but these just miss out being the "best" split among all possible. The function `burl.tree()` allows the user to select nodes and observe the competition for the best split at that node. For numeric predictors, a high density plot is used to show the *goodness-of-split* at each possible cut-point split. For factor predictors, a scatterplot plot displays goodness-of-split versus a decimal equivalent of the binary representation of each possible subset split; the plotting character is a string labeling the left split. Figure 9.14 provides an example for the tree `z.cu`. The plots under the dendrogram show a clear preference for splits involving the variable `Country`. Figure 9.15 is an enlargement of the scatterplot for `Country`. We see that the candidate splits divide into two groups, one of which (top) discriminates better than the other (bottom). Among those in the top portion, that labeled `ef=Japan, Japan/USA` is the best; moreover, it is the common intersection of all the candidate splits in the top portion. Given this information, we are more likely to believe that this split is meaningful.

The function `hist.tree()` also focuses on splits at specified or interactively selected nodes by displaying side-by-side histograms of supplied variables. Specifically, the histogram on the left displays the distribution of the observations on that variable following the left split, while the histogram on the right displays the distribution of the observations following the right split. It is similar to `burl.tree()` in that it displays a variable's discriminating ability, but is different in that it allows

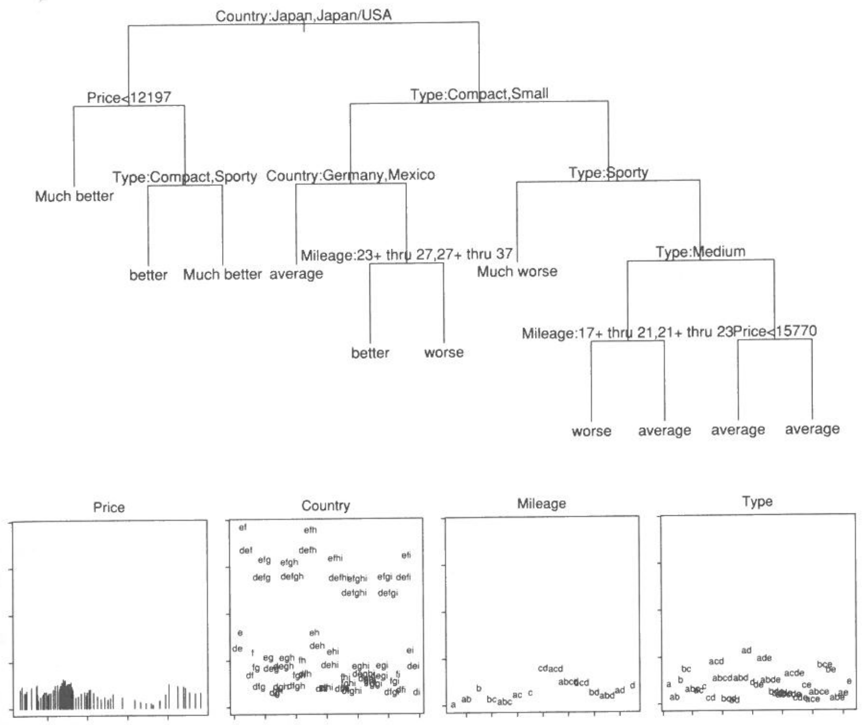


Figure 9.14: *An illustration of burling a tree-based model. The top panel displays the labeled dendrogram of* `z.cu`*; initially, the lower portion is empty. Upon selection of the root node, the plots in the lower four panels are displayed. These show, for each predictor in the model formula, the goodness-of-split criterion for each possible split. The goodness-of-split criterion is the difference in deviance between the parent (in this case the root node) and its children (defined by the tentative split); large deviance differences correspond to important splits. For numeric predictors, a high-density plot conveys the importance of each possible cut-point split. For factor predictors, an arbitrary ordering is used along the abscissa (x-axis) to separate different subset splits; the left split is used as a plotting character. The ordinate (y-axis) of all plots is identical. These plots show that, at the root node,* `Country` *is the best discriminator of automobile reliability. It also shows that there are many good subset splits on* `Country`*, the "best" being the one labeled* `ef` *in the upper left. Upon selection of another node in the dendrogram, the lower portion of the screen is erased and refreshed with four new panels displaying the splits relevant at that node.*

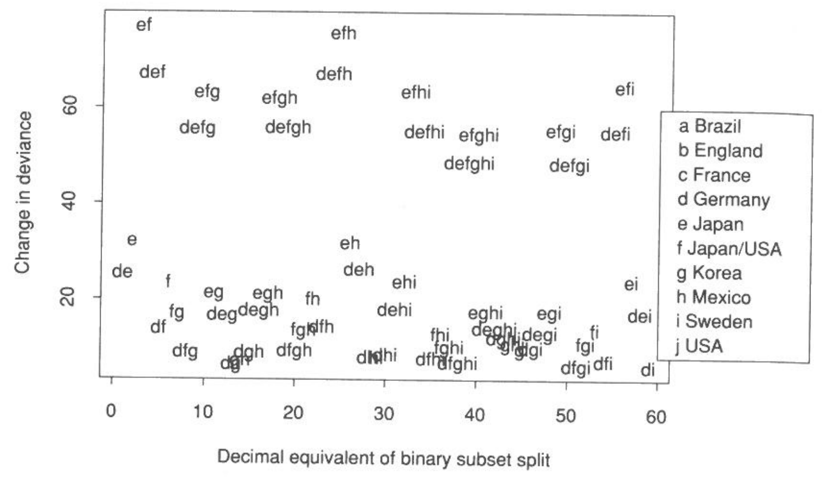


Figure 9.15: *A scatterplot of the competing subset splits on* `Country` *at the root node of the tree* `z.cu`*. The plotted character strings are the left splits; none contain* `j` *since it is the last level of* `Country` *and, by construction, resides in right splits only. No subsets contain* `abc` *since automobiles from these countries were omitted due to missing values of the response variable* `Reliability`*; this occurred silently by* `na.tree.replace()` *when* `z.cu` *was grown. There are no singleton splits for* `d, g, h,` *or* `i` *since these countries have fewer than five automobiles in the model frame and the algorithm has a minimum subset size of five. The splits seem to divide into two groups: those having good discriminating power (upper portion), and those having mediocre to poor power (lower portion). The former all contain* `ef`*, supporting its selection as the best discriminating subset.*

variables other than predictors to be displayed. Figure 9.16 provides an example for the tree `z.cu` fitted to the automobile reliability data. This example resulted from the expression:

```
> hist.tree(z.cu, Reliability, Price, Mileage, nodes = 1)
```

At a glance we see the complete distribution of the response variable `Reliability` for nodes 2 and 3 (the children nodes of the root). It is interesting that not a single `Much Better` car follows the right split. The second panel (`Price`) graphically conveys what our earlier analysis using `identify()` suggested: that the most reliable cars are not the most expensive ones. It appears that status and reliability are incompatible in these data.

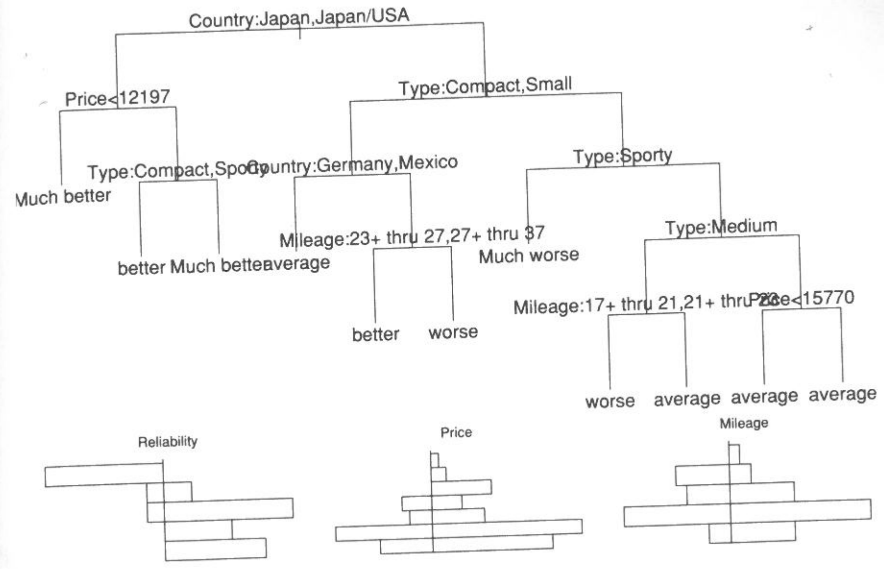


Figure 9.16: *A illustration of the function* `hist.tree()` *at the root node of the automobile reliability tree* `z.cu`*. The upper portion of the plot contains the labeled dendrogram. The lower portion displays a side-by-side histogram for each of the variables* `Reliability`*,* `Price`*, and* `Mileage`*. The left-side histogram summarizes the observations following the left split, and similarly for the right. The figure shows that Japanese cars manufactured here or abroad tend to be more reliable, less expensive, and more fuel efficient than others.*

9.2.6 Examining Leaves

Often it is useful to observe the distribution of a variable over the leaves of a tree. Two related (noninteractive) functions encourage this functionality. They are noninteractive since they do not depend on user selection of a particular node; their intended effect is across all terminal nodes. The function `tile.tree()` augments the bottom of a dendrogram with a plot that shows the distribution of a specified factor for observations in each leaf. These distributions are encoded into the widths of tiles that are lined up with each leaf. If numeric variables are supplied, they are automatically quantized. One use of this function is for displaying class probabilities across the leaves of a tree. An example is displayed in Figure 9.17. A related function `rug.tree()` augments the bottom of a dendrogram with a (high-density) plot that shows the average value of the specified variable for observations in each leaf. These averages are encoded into lengths of line segments that are lined up with each leaf. The function takes an optional argument, `FUN=`, so that summaries other than simple averages (e.g., trimmed means) can be obtained. Figure 9.18 displays

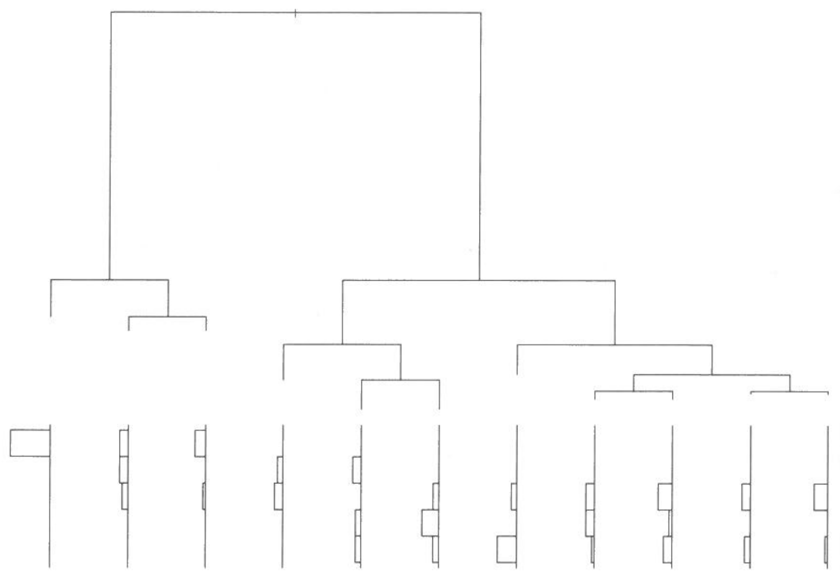


Figure 9.17: *The dendrogram of the automobile reliability tree* `z.cu` *enhanced with a tiling of the variable* `Reliability`. *The distribution of* `Reliability` *over the leaves of the tree is readily discerned. Successive calls to* `tile.tree()` *with other variables is encouraged by not replotting the dendrogram—only the new tiling is plotted after the bottom screen is "erased".*

the distribution of the variable `usage` for the tree grown to the market survey data. Recalling that the split at the root node was `usage` ≤ 12.5, the general shape of the rug is as expected: lower on the left and higher on the right. Somewhat unexpected is the fact that the heavier users are, by and large, much heavier users.

9.3 Specializing the Computations

As described in the preceding section, the tree object is a repository for a number of by-products of the tree-growing algorithm. The named components of a tree object are

```
> names(z.survey)
[1] "frame"     "where"      "terms"  "call"
```

The `frame` component is a data frame, one row for each node in the tree. The row labels, `row.names(frame)`, are node numbers defining the topology of the tree. Nodes of a (full) binary tree are laid out in a regular pattern:

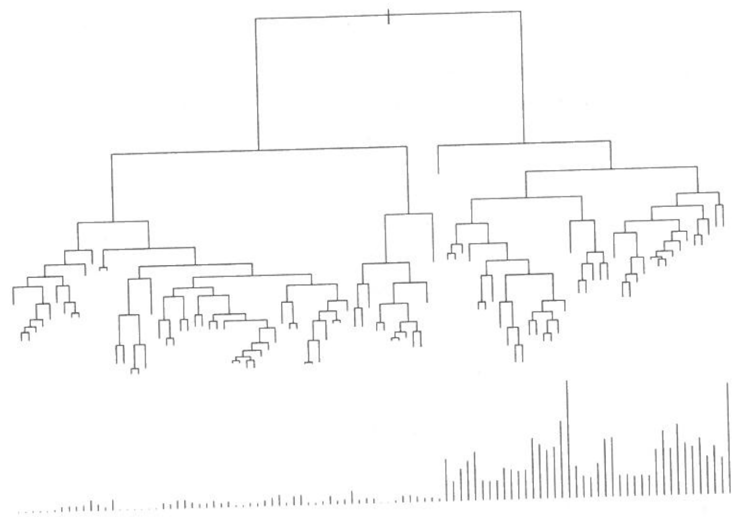


Figure 9.18: *The dendrogram of the long-distance marketing tree* `z.survey` *enhanced with a rug of the variable* `usage`. *The distribution of this variable over the leaves of the tree is readily discerned. Successive calls to* `rug.tree()` *with other variables is encouraged by not replotting the dendrogram—only the new rug is plotted after the bottom screen is erased.*

```
              1
        2           3
   4       5  6        7
  ...    ...  ...   ...
```

More generally, nodes at depth d are integers n, $2^d \leq n < 2^{d+1}$. Of course, any specific tree is not full and consists of a subset of all possible nodes. The ordering of the nodes in the `frame` corresponds to a depth-first traversal of the tree according to this numbering scheme.

The elements (columns) of `frame` contain the following node-specific information:

- the variable used in the split at that node (`var`)
- the number of observations in the node (`n`)
- a measure of node heterogeneity (`dev`)
- the fitted value of the node (`yval`)
- the matrix of left and right split values (`splits`).

Routine application of the functions in this chapter does not require users to manipulate this object directly, but for concreteness we display the 21 row `z.cu$frame` here:

```
        var  n        dev        yval splits.left splits.right
 1  Country 85 260.997544     average        :ef       :dghij
 2    Price 27  36.921901 Much better     <12197       >12197
 4   <leaf> 14   0.000000 Much better
 5     Type 13  26.262594 Much better        :ae          :cf
10   <leaf>  8  17.315128      better
11   <leaf>  5   5.004024 Much better
 3     Type 58 146.993133     average        :ad        :bcef
 6  Country 22  59.455690       worse        :dh         :gij
12   <leaf>  5   6.730117     average
13  Mileage 17  42.603572       worse        :cd          :be
26   <leaf>  7  15.105891      better
27   <leaf> 10  19.005411       worse
 7     Type 36  68.976020     average         :e         :bcf
14   <leaf>  9   9.534712  Much worse
15     Type 27  50.919255     average         :c          :bf
30  Mileage 16  33.271065     average        :ab           :e
60   <leaf>  7  14.059395       worse
61   <leaf>  9  16.863990     average
31    Price 11  12.890958     average     <15770       >15770
62   <leaf>  5   6.730117     average
63   <leaf>  6   5.406735     average
```

This example illustrates a labeling convention specific to trees whereby levels of factor predictors are assigned successive lower-case letters. Thus, the first right split, `:dghij` (on `Country`), is shorthand for `:Germany,Korea,Mexico,Sweden,USA`. Such a convention is necessary in order to provide meaningful information about splits in a limited amount of space. The problem is particularly acute for labeling plotted dendrograms but is also important in tabular displays such as that resulting from `print()`. The `labels()` method for trees allows full control over which style of labels is desired; it is usually invoked by printing and plotting functions rather than called directly by the user.

In the case of classification trees, an additional component of the frame object is the matrix (`yprob`) containing the class probability vectors of the nodes labeled by the levels of the response variable. We omitted this in the above display of `z.cu$frame` in order to conserve space.

The `where` component of a tree object is a vector containing the row number (in `frame`) of the terminal node that each observation falls into. It has a `names` attribute that corresponds to the `row.names` of the model frame used to grow or otherwise define the tree. Like the `frame` component, it is heavily used in many of the functions that manipulate trees. For example, the vector of fitted values is obtained as `z$frame[z$where, "yval"]`. The remaining components, `"terms"` and `"call"`, are identical to those described in previous chapters.

We emphasize that for the most part you will not have to look directly at the values of these components. However, in order to modify the behavior of any of the supplied functions, or to construct new ones, you should first feel comfortable manipulating these components. For example, consider the following function (provided in the library):

```
meanvar.tree() <- function(tree, xlab = "ave(y)",
  ylab = "ave(deviance)", ...) {
    if(!inherits(tree, "tree"))
            stop("Not legitimate tree")
    if(!is.null(attr(tree, "ylevels")))
            stop("Plot not useful for probability trees")
    frame <- tree$frame
    frame <- frame[frame$var == "<leaf>", ]
    x <- frame$yval
    y <- frame$dev/frame$n
    label <- row.names(frame)
    plot(x, y, xlab = xlab, ylab = ylab, type = "n", ...)
    text(x, y, label)
    invisible(list(x = x, y = y, label = label))
}
```

This function uses only the `frame` component to produce a plot of the within-node variance (`dev/n`) versus the within-node average (`yval`) for numeric responses. The node number is used as the plotting character. This plot is useful for assessing the assumption of constant variability throughout predictor space. If trend is apparent in the plot, a reexpression of the response variable y is recommended for proper trees to be grown.

The functions we provide are intended to make the task of modeling data with binary trees more pleasant and at the same time more powerful. The examples in the previous sections showed how the user might directly use these functions during an analysis. Of course, the functions can also be called by other functions and thus form the building blocks for more specialized functions or even more complicated manipulations of tree-based models.

The single best example illustrating the power of using the functions as primitives in a more complicated function is given by the technique known as *cross-validation*. Specifically, consider the problem of selecting the optimal tree in a pruning or shrinking sequence. The general idea is that the deviances, used as a measure of predictive ability, for any of the trees in the sequence are far too optimistic—that is, too small—as they are based on the same data used to construct the tree. It would be better—that is, less biased—to use an independent sample with which to assess the predictive ability of any specific tree. Cross-validation is an attempt to do just this where the original dataset is carved into K mutually exclusive subsets, each of which will serve as an independent test set for trees grown on learning sets composed of the union of the $K - 1$ remaining subsets. For each of the learning sets, a tree must be grown and a pruning or shrinking sequence determined. The corresponding test set must then be dropped down the trees in the sequence and some measure of goodness computed (e.g., misclassification error rate or deviance—we use the latter). These are then summed over the induced replications and displayed. An implementation is as follows:

```
cv.tree <- function(tree, rand, FUN = shrink.tree, ...)
{
    if(!inherits(object, "tree"))
        stop("Not legitimate tree")
    m <- model.frame(object)
    p <- FUN(object, ...)
    if(missing(rand))
        rand <- sample(10, length(m[[1]]), replace = T)
    which <- unique(rand)
    cvdev <- 0
    for(i in which) {
        tlearn <- tree(model = m[rand != i, ])
        plearn <- FUN(tlearn, newdata = m[rand == i, ], p$k, ...)
        cvdev <- cvdev + plearn$dev
    }
    p$dev <- cvdev
    p
}
```

Apart from some initialization steps, the function first sequences the original tree and assigns the result to `p`. In the `for` loop, we use two different high-level tree manipulation functions. We first use `tree()` to grow a tree to the learning model, `m[rand != i, ]`. This is followed by a call to the sequencing function, `shrink.tree()` by default, to produce the sequence for the learning tree and to evaluate the sequence for the model containing the test data, `m[rand == i, ]`. Finally, the deviances are summed across samples and returned for subsequent plotting.

Other functions for tree-based modeling are included in the library that have not been explicitly mentioned in the text. Some are low-level utility functions that are called by the high-level functions accessed directly by the user. Others are high-level functions that are specialized for certain numerical or graphical purposes. The function `basis.tree()` is an example of the former whereby an orthogonal basis for a fitted tree is computed. There is one basis vector for each split and one for the root (the unit vector). A linear model fitted to this basis yields fitted values identical to those from the tree. This linear model representation of a fitted tree-based model is sometimes useful for suggesting new methods for understanding trees (e.g., shrinkage estimation.) The functions `post.tree()` and `partition.tree()` are examples of special purpose graphics functions. The function `post.tree()` does not require

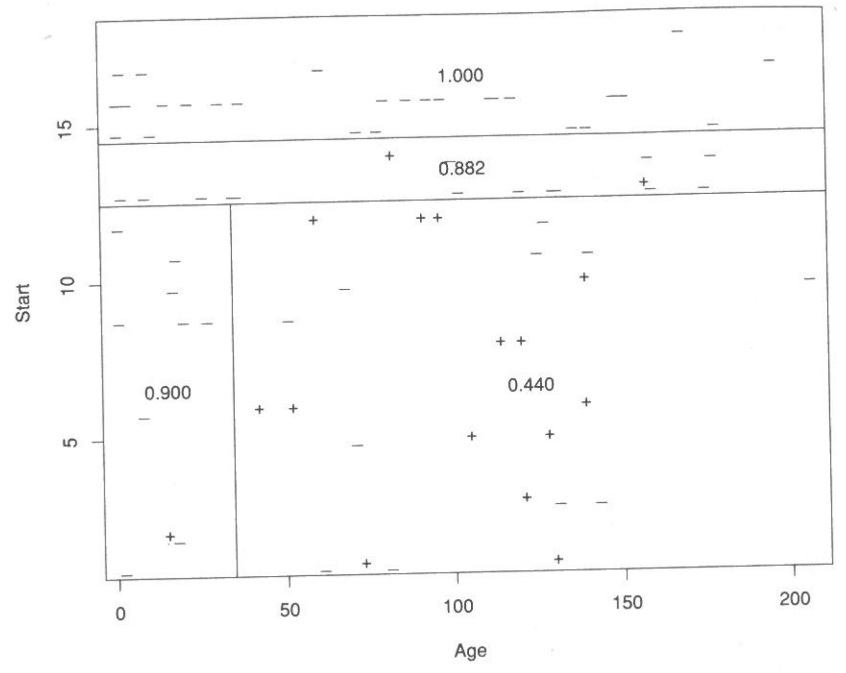


Figure 9.19: *A display of* `z.kyph[-c(5, 6)]`*, a subtree of* `z.kyph` *depending on the variables* `Age` *and* `Start`*. The plot was obtained with the expression* `partition.tree(z.kyph[-c(5, 6)], label = "absent")`*. The data values appear on the plot as the plotting characters "–" and "+." These were added with the expression* `text(Age, Start, ifelse(Kyphosis == "absent", "-", "+"))`*. Three of the four regions are quite homogeneous; no apparent structure is discernible in the remaining one.*

activation of a graphics device, but rather that the user has access to a printer compatible with the `PostScript` page-description language. The trees displayed in Figures 9.3, 9.8, and 9.13 were produced by `post.tree()`. This "pretty printed" display of a tree uses uniform vertical spacing of nodes and is more appropriate for presentation than for diagnosis.

The function `partition.tree()` is peculiar to trees that depend on at most two predictor variables. For a single predictor, `partition.tree()` displays the tree as a step function, each step corresponding to a terminal node of the tree. This display sacrifices the information in the tree object concerning the sequence of splits leading to the leaf nodes, but gains familiarity of expression when one regards y

as a function of x. The example in the right panel of Figure 9.1 was created with `partition.tree()`. For two predictors, `partition.tree()` displays the partition of the plane into homogeneous regions, each rectangular region corresponding to a terminal node of the tree. In certain cases it is possible to reconstruct the sequence of splits giving rise to the partition from the display, although this is not the primary intended purpose. An optional argument, `label`, allows the user to specify the labels associated with the partition, the default being the fitted value `yval`. For classification trees, a specific level of the response factor can be specified. Figure 9.19 demonstrates a two-variable example based on the subtree `z.kyph[-c(5, 6)]`.

Certain enhancements to the display functions are desirable so that more information can be displayed subject to the constraint of minimal overplotting. For example, the `text()` method for trees introduced in Section 9.2 allows an argument, `FUN=`, to encourage users to explore interactive labeling. Suppose a user had a function, say `brush()`, which allowed one to paint on labels (say with button 1) as well as erase them (say with button 2). By paint we mean that buttons are depressed and held rather than simply clicked. Then one could selectively label a plotted dendrogram in those cases where unrestricted labeling would conceal the dendrogram itself.

A somewhat different specific proposal that we considered was displaying a histogram or a boxplot of the distribution of y at each node of the tree. This would allow comparison of scale and shape changes as nodes are split in addition to location differences, as is currently done. A function `zoom.tree()` might then be written so that selecting a node might *zoom in* or otherwise provide an enlargement of the histogram. This would necessitate some device-specific graphics functions, which we have attempted to avoid.

9.4 Numerical and Statistical Methods

Tree-based models are defined most precisely by the algorithm used to fit them. The algorithm attempts to partition the space of predictor variables (X) into homogeneous regions, such that within each region the conditional distribution of y given $\boldsymbol{x}$, $f(y|\boldsymbol{x})$, does not depend on $\boldsymbol{x}$. We first present the algorithm and then discuss the three essential components as regards our implementation.

```
**Initialize:** current node = root = $\{y_i\ , i = 1, \ldots, n\}$
    stack = NULL

**Recurse:** for current node $\neq$ NULL

    **Loop:** for each $x_j$ partition $x$ into two sets $X_{LEFT}$ and $X_{RIGHT}$ such that
        $f(y|X_{LEFT})$ and $f(y|X_{RIGHT})$ are most different

    **Split node:** split current node into $Y_{LEFT}$ and $Y_{RIGHT}$ according to the $x_j$
        and the associated split that is best among all $x$'s

    **Test:** if *ok* to split $Y_{RIGHT}$
            push $Y_{RIGHT}$ onto stack
        if *ok* to split $Y_{LEFT}$
            current node = $Y_{LEFT}$
        else *pop* stack
```

Partitioning the Predictors

Predictor variables appropriate for tree-based models can be of several types: factors, ordered factors, and numeric. Partitions are governed solely by variable type and therefore do not require explicit specification by the user.

If x is a factor, with say k levels, then the class of splits consists of all possible ways to assign the k levels into two subsets. In general, there are $2^{k-1}-1$ possibilities (order is unimportant and the empty set is not allowed). So, for example, if x has three levels (a, b, c), the possible splits consist of $a|bc$, $ab|c$, and $b|ac$.

If x is an ordered factor with k ordered levels, or if x is numeric with k distinct values, then the class of splits consists of the $k-1$ ways to divide the levels/values into two contiguous, nonoverlapping sets. These splits can be indexed by the midpoints of adjacent levels/values, which we call *cutpoints*. By convention, we implicitly extend the range beyond the observed data, so that at the left-most cutpoint, c_L defines the split $-\infty < x \leq c_L$, and similarly for the right-most cutpoint. Note that the *values* of a numeric predictor are not used in defining splits, only their *ranks*. Indeed, it is this aspect of tree-based models for numeric predictors that render them invariant under monotone transformations of x.

Comparing Distributions at a Node

We depart slightly from most previous authors on recursive partitioning methods in that our view is more closely akin to classical models and methods for regression and classification data. Our view is that we are estimating a step function $\tau(\boldsymbol{x})$ that is simply related to a primary parameter in the conditional distribution of $y|\boldsymbol{x}$. The likelihood function provides the basis for choosing partitions. Specifically, we use the *deviance* (likelihood ratio statistic) to determine which partition of a node is "most likely" given the data. The implementation is such that the type of the response variable is the sole determinant of whether a classification tree (factor response y) or a regression tree (numeric y) is grown. The current implementation ignores any possible ordering of an ordered factor response variable; arguably, this should be exploited in the fitting.

Classification trees

The model we use for classification is based on the multinomial distribution where we use the notation, for example,

$$y = (0, 0, 1, 0)$$

to denote the response y falling into the third level out of four possible. The vector $\mu = (p_1, p_2, p_3, p_4)$, such that $\sum p_k = 1$, denotes the probability that y falls into each of the possible levels. In the terminology of Chapter 6, the model consists of the stochastic component,

$$y_i \sim \mathcal{M}(\mu_i),\ i = 1, \ldots, N$$

and the structural component

$$\mu_i = \tau(\boldsymbol{x}_i).$$

The deviance function for an observation is defined as minus twice the log-likelihood,

$$D(\mu_i; y_i) = -2 \sum_{k=1}^{K} y_{ik} \log(p_{ik}).$$

y_{ik} = should be either 1/0

Regression trees

The model we use for regression is based on the normal (Gaussian) distribution, consisting of the stochastic component,

$$y_i \sim \mathcal{N}(\mu_i, \sigma^2),\ i = 1, \ldots, N$$

and the structural component

$$\mu_i = \tau(\boldsymbol{x}_i).$$

The deviance function for an observation is defined as

$$D(\mu_i; y_i) = (y_i - \mu_i)^2,$$

which is minus twice the log-likelihood scaled by σ^2, which is assumed constant for all i.

At a given node, the mean parameter μ is constant for all observations. The maximum-likelihood estimate of μ, or equivalently the minimum-deviance estimate, is given by the node proportions (classification) or the node average (regression).

The deviance of a node is defined as the sum of the deviances of all observations in the node $D(\hat{\mu}; y) = \sum D(\hat{\mu}; y_i)$. The deviance is identically zero if all the y's are the same (i.e., the node is pure), and increases as the y's deviate from this ideal. Splitting proceeds by comparing this deviance to that of candidate children nodes that allow for separate means in the left and right splits,

$$D(\hat{\mu}_L, \hat{\mu}_R; y) = \sum_L D(\hat{\mu}_L; y_i) + \sum_R D(\hat{\mu}_R; y_i)$$

The split that maximizes the change in deviance (goodness-of-split)

$$\Delta D = D(\hat{\mu}; y) - D(\hat{\mu}_L, \hat{\mu}_R; y)$$

is the split chosen at a given node.

Limiting Node Expansion

The above discussion implies that nodes become more and more pure as splitting progresses. In the limit a tree can have as many terminal nodes as there are observations. In practice this is far too many, and some reasonable constraints should be applied to reduce the number. We use two different criteria for deciding if a node is suitable for splitting. *Do not split:*

- if the node deviance is less than some small fraction of the root node deviance (say 1%); and
- if the node is smaller than some absolute minimum size (say 10).

These limits are implemented through the arguments `mindev` and `minsize`, respectively, in the function `tree.control()`. The current defaults are given above in parentheses.

The default is quite liberal and will still result in an overly large tree with roughly $N/10$ terminal nodes. This is intentional and mimics "best current practice" in recursive partitioning methods. Indeed, the major problem of early tree-building algorithms was deciding when to stop expanding nodes. It was indeed critical as the tree was built in a forward stepwise manner, and once the final node was expanded, modeling was complete. The approach we adopt is not to limit node expansion in the tree-growing process. Instead, an overly large tree is grown, and one must decide which branches to prune off or find some other way account for overfitting (e.g., recursive shrinking). The difference in the approaches is similar to that between forward and backward stepwise selection of variables in linear models. Forward methods can be fooled when the best early split does not meet the criterion of splitting and tree growth is halted—when in fact this split is necessary to clear the field for very important succeeding splits. The example of looking for interactions in linear model residuals provides an illustration.

The design of our functions had this concept in mind from its inception, providing a simple interface to growing a large tree, while providing a collection of interactive functions to inspect nodes, identify observations, snip branches, select subtrees, etc. Our recommended approach to tree building is far less automatic than that provided by other software for the same purpose, as the unbundling of procedures for growing, displaying, and challenging trees requires user initiation in all phases. We now turn to another issue that also requires the user to get involved in the modeling process.

9.4.1 Handling Missing Values

Tree-based models are well suited to handling missing values and several possibilities exist for building trees and predicting from them in the face of NAs. For tree building itself, the current implementation of `tree()` only permits NAs in predictors, and only if requested by the special `na.action()` for trees, `na.tree.replace()`. The effect of this function is to add a new level named `"NA"` to any predictor with missing values; numeric predictors are first quantized. The net effect of using `na.tree.replace()` is that the new variable is treated like any other factor as regards determination of the optimal split. If `x` has three levels (a, b, c), the candidate splits accommodating missing values are $NA|abc$, $NAa|bc$, $NAab|c$, $NAb|ac$, $NAc|ab$, $NAbc|a$, and $NAac|b$. Other possible ways to adapt `tree()` to allow missing values in ordinal and numeric variables would likely require changes in the underlying algorithm.

As described earlier on page 392, the approach we adopt for prediction is that once an NA is detected while dropping a (new) observation down a fitted tree, the observation "stops" at that point where the missing value is required to continue the path down the tree. This is equivalent to sending the observation down both sides of any split requiring the missing value and taking the weighted average of the vector of predictions in the resulting set of terminal nodes. We chose this method over that based on so-called *surrogate splits* because we believe it to be less affected by nonresponse bias. A surrogate split at a given node is a split on a variable other than the optimal one that best predicts the optimal split. If a new observation is being predicted that has a missing value on the split-defining variable, then prediction continues down the tree so long as there is data on the variable given by the surrogate split.

We note in passing another function concerned with missing values. The function `na.pattern()` enumerates the distinct pattern of missing values in a data frame, together with the number of occurrences. For example,

```
> na.pattern(market.survey)
 0000000000 0000000011 0000000100 0000100000 0001000000 0010000000
        759         16          4          3          2          1

 0100000000 0100000011 0100010000 0100100000 0100110000 0101000000
        168          1          2          5          4         10

 0101100000 0101110000 0110000000 0110100000 0111110000
          1          8          2          2         12
```

indicates that all but 241 observations were complete, and of these 168 had information missing on the second variable (`income`!) alone. The remaining 73 observations have a variety of patterns of missing values; of these, all but 26 have `income` among the missing fields.

9.4.2 Some Computational Issues

It should be clear that a fair amount of computation is required to select the best split at a given node. The algorithm underlying tree-based models is computationally intensive. Although it is possible to implement it entirely in the S language, we chose instead to write several of the underlying routines in C. Most have to do with the actual tree-growing process (`grow.c`, `splitvar.c`, and `vsplit.c`), others are for sequencing (`prune.c` and `shrink.c`), another for efficient prediction (`pred.c`), and, finally, others for character manipulation (`btoa.c`) and printing labels (`prlab.c`). Most users will not have to deal with this underlying code, but there are cases where it is unavoidable and even desirable to modify code at this level for some desired effect. Ultimately, such changes need to be compiled and loaded into S.

Our implementation is efficient in the sense that excessive computation is avoided by *updating*, whereby the assessment of split optimality (ΔD) is done incrementally after it has been done once for a particular split. Further computational improvement is possible for splits of factor predictors (where it is needed most!) provided that y is numeric or has at most two levels. If this is the case, then the average value of y in each level of the factor can be used to order the levels so that the best split is among the $k-1$ contiguous splits after reordering. This fails for factor responses with more than two levels since it is unclear how a reordering is to be effected.

9.4.3 Extending the Computations

Tree-based models can be extended to response variables from the exponential family of distributions $f(y;\mu)$ described in Chapter 6. This results in the class of generalized tree-based models (GTMs), whereby the stochastic component of a response is assumed to be an exponential family member and the structural component is described by a tree structure. Thus, for exponential family distributions, there is a logical progression of models of the structural component afforded by linear predictors (GLMs, Chapter 6), additive predictors (GAMs, Chapter 7), and tree-based predictors (GTMs). In principle, the extension is quite straightforward as the only change to the existing software is in the form of the deviance function. Note in particular that specification of a link function is not necessary since the estimate of μ in each node is the within-node average for all exponential family distributions. However, link specification would be necessary in the event that an *offset* is used. More importantly, an offset induces iteration in the calculation of the within-node fitted value. For computational efficiency, one would determine splitting rules using an approximation to the deviance, say the *score function*, and only iterate to convergence once a candidate variable and splitting rule have been determined. This would increase the amount of computation by only a trivial amount relative to the current implementation for classification and regression.

Another possible generalization is the enlargement of the class of splitting rules allowed by our tree-growing algorithm. Specific possibilities include linear combination splits for selected sets of numeric predictors, as well as *boolean combinations* whereby splits on individual factor predictors are *AND*ed and *OR*ed to form a single split at a node. A convenient user interface is obtained by allowing a `matrix` data type in the formula expression supplied to `tree()`, such that columns of the matrix represent the individual variables to be combined: a matrix of numeric variables for linear combination splits, and a logical matrix for boolean combination splits. Thus, splits for these variable types are defined implicitly just as they are for numeric predictors and factors. The computational complexity of such splitting is unwieldy, and only suboptimal selections using *heuristics* are likely to be feasible.

Another interesting possibility is to consider hierarchical or conditional variables that are typical of surveys. For example, depending on whether or not a person is head of household, certain sections of a survey are not completed by the respondent. For others, the values for the entries in these sections are missing, not at random, but because of the structure of the instrument. Tree-based models are particularly adept at capturing these types of data since by decomposing the sample into homogeneous subgroups, the responses to the conditional part of these questions are appropriate once the primary variable has been used in a split. It would seem that a useful way to implement such variables is through an activation bit, which is on for all primary variables, but gets turned on for the secondary ones only when their primary variable is used in a split.

Bibliographic Notes

The introduction of tree-based models in statistics, particularly statistics for the social sciences, is due to Sonquist and Morgan (1964). An implementation of their ideas was realized in the computer program AID (Automatic Interaction Detection), which served to stimulate much subsequent research, such as THAID (Morgan and Messenger, 1973) and CHAID (Kass, 1980). These methods differed primarily in the stopping rules used to halt tree growth.

The inclusion of a chapter on tree-based modeling in this book is due to the influence of the work on classification and regression trees by Breiman et al. (1984). Besides masterfully presenting the material to the mainstream statistical audience, they are responsible for several important pioneering ideas that have redefined the state-of-the-art of tree-based methods. The primary innovation was not to limit node expansion in the tree-growing process. They recommended growing an overly large tree and spending one's effort deciding which branches to prune off. Their method of determining a pruning sequence, based on the concept of *minimal cost complexity*, forms the basis for the function `prune.tree()`. Subsequent work by Chou et al. (1989) generalizes this concept to other tree functionals besides tree

size. Their other important innovation was the introduction of surrogate splits to provide a mechanism to grow trees and make predictions in the presence of NAs and also to provide a measure of variable importance.

Our methodology parallels that of Ciampi et al. (1987) in the use of the likelihood function as the basis for choosing partitions. This is a departure from that of Breiman et al. who use a variety of measures for tree growing and subsequent pruning. The precise definition of the shrinkage scheme discussed in Section 9.2 is also based on the likelihood (deviance) function. Recursive shrinking of tree-based models is a relatively new application of shrinkage estimators due to Hastie and Pregibon (1990). It has not been used as extensively as cost-complexity pruning nor have extensive comparisons been performed with it.

The computational shortcut for enumerating subset splits for factors and numeric responses dates back to Fisher (1958). This shortcut extends to binary responses but not to factor responses with more than two levels. Chou (1988) suggests a heuristic that restricts search to a (possibly) nonoptimal set of partitions. The split produced by the heuristic gets closer to the optimal split as the number of the levels of the factor increase—exactly the case where exhaustive search is infeasible. The current implementation of `tree()` does not incorporate this heuristic.