As you've already seen, decision trees can naturally handle variables of any type without special encoding, although we did see that a local form of mean/target encoding can be used to reduce the computational burden imposed by nominal categorical splits. Nonetheless, using an encoding strategy, like OHE, can sometimes improve the predictive performance or interpretability of a tree-based model; see Kuhn and Johnson [2013, Sec. 14.7] for a brief discussion on the use of OHE in tree-based methods. Further, some tree-based software, like Scikit-learn's **sklearn.tree** module, require all features to be numeric—forcing users to employ different encoding schemes for categorical features. See Boehmke and Greenwell [2020, Chap. 3] for details on different encoding strategies (with examples in R), and further references.

2.5 Building a decision tree

In the previous sections, we talked about the basics of splitting a node (i.e., partitioning some subset of the learning sample). Building a CART-like decision tree starts by splitting the root node, and then recursively applying the same splitting procedure to every resulting child node until a saturated tree is obtained (i.e., all terminal nodes are pure) or other stopping criteria are met. In essence, the partitioning stops when at least one of the following conditions are met:

- all the terminal nodes are pure;
- the specified maximum tree depth has been reached;
- the minimum number of observations that must exist in a node in order for a split to be attempted has been reached;
- no further splits are able to decrease the overall lack of fit by a specified factor;
- and so forth.

This often results in an overly complex tree structure that overfits the learning sample; that is, it has low bias, but high variance.

To illustrate, consider a random sample of size $N = 500$, generated from the following sine wave with Gaussian noise:

$$Y = \sin(X) + \epsilon,$$

where $X \sim \mathcal{U}(0, 2\pi)$ and $\epsilon \sim \mathcal{N}(0, \sigma = 0.3)$. A scatterplot of the data, along with the true response function, is shown in Figure 2.12.

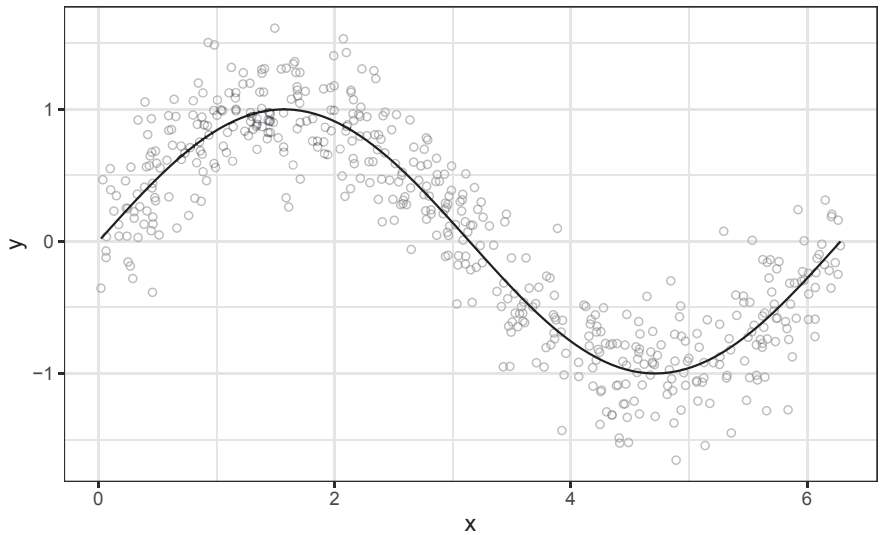

FIGURE 2.12: Data generated from a simple sine wave with Gaussian noise. The black curve shows the true mean response $\mathrm{E}(Y|X = x) = \sin(x)$.

Figure 2.13 shows the prediction function from two regression trees fit to the same data.[i] The tree on the left is too complex and has too many splits, and exhibits high variance, but low bias (i.e., it fits the current sample well, but the tree structure will vary wildly from one sample to the next because it's mostly fitting the noise here); unstable models, like this one are often referred to as *unstable learners* (more on this in Section 5.1). The tree on the right, which is a simple decision stump (i.e., a tree with only a single split), is too simple, and will also not be useful for prediction because it has extremely high bias, but low variance (i.e., it doesn't fit the data too well, but the tree structure will be more stable from sample to sample); such a weak performing model is often referred to as a *weak learner* (more on this in Section 5.2).

Neither tree is likely to be accurate when applied to a different sample from the same model; the ensemble methods discussed in Part II of this book can improve the performance of both weak and unstable learners. When using a single decision tree, however, the question we need to answer is, How complex should we make the tree? Ideally, we should have stopped splitting nodes at some *subtree* along the way, but where?

A rather careless approach is to build a tree by only splitting nodes that meet some threshold on prediction error. However, this is shortsighted because a low-quality split early on may lead to a very good split later in the tree. The standard approach to finding an optimal subtree—basically, determining when

[i]The associated tree diagrams are shown in the top left and bottom right of Figure 2.14 (p. 73), respectively.

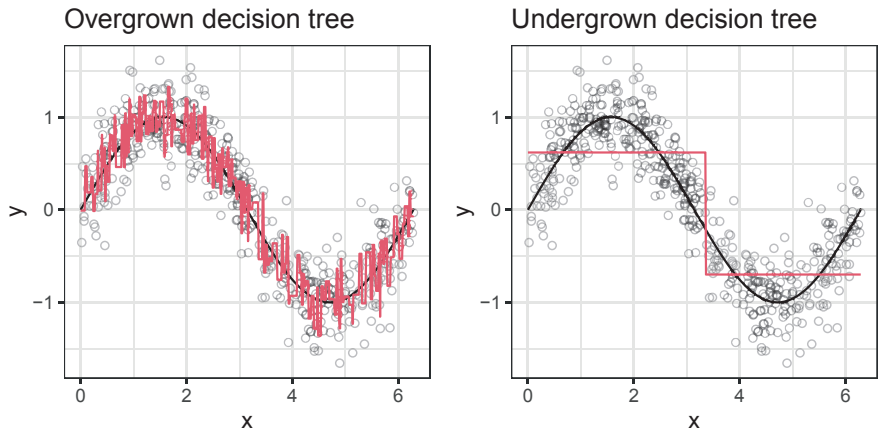

FIGURE 2.13: Regression trees applied to the sine wave example. Left: this tree is too complex (i.e., low bias and high variance). Right: this tree is too simple (i.e., high bias and low variance).

we should have stopped splitting nodes—is called *cost-complexity pruning*, or *weakest link pruning* [Breiman et al., 1984], or just pruning for short. Other pruning procedures are discussed in Ripley [1996, pp. 226–231] and Zhang and Singer [2010, pp. 44–49]. Pruning a decision tree is quite analogous to the process of *backward elimination* in multiple linear regression—start with a complex tree with too many splits, and *prune off* leaves whose contributions aren't enough to offset the added complexity. The details are covered in the next section.

2.5.1 Cost-complexity pruning

The idea of pruning a decision tree is similar to the process of backward elimination in multiple linear regression. In essence, we build a large tree with too many splits, denoted $\mathcal{T}_0$, and then prune it back by collapsing internal nodes until we find some optimal subtree, denoted $\mathcal{T}_{opt}$, that meets a certain criterion, like having the smallest cross-validation error.

Let $\{A_k\}_{k=1}^{K}$ be the terminal nodes of some tree $\mathcal{T}$, where $|\mathcal{T}| = K$ is the number of terminal nodes, or size of $\mathcal{T}$. Recall that the overall goal of CART is to extract homogenous subgroups (i.e., terminal nodes). In this sense, the overall quality (or *risk*) of the tree depends on the quality of its terminal nodes. We define the risk of the tree to be $R(\mathcal{T}) = \sum_{k=1}^{K} p(A_k) \times r(A_k)$, where $r(A_k)$ is some measure of the quality of the k-th terminal node; see (2.4) on page 55. For regression trees, $R(\mathcal{T})$ is the error sum of squares (SSE). For classification trees based on the observed class priors and equal misclassification costs

(i.e., $L_{i,j} = 1$ for all $i \neq j$), $R(\mathcal{T})$ is simply the proportion of observations misclassified in the learning sample.

Building a tree to minimize $R(\mathcal{T})$ will always lead to a saturated tree, resulting in a model with little or no bias but often high variance (i.e., overfitting the learning sample). Instead, we penalize the complexity (or size) of the tree by minimizing

$$R_\alpha(\mathcal{T}) = R(\mathcal{T}) + \alpha|\mathcal{T}|,$$

where $\alpha \geq 0$ is a tuning parameter controlling the trade-off between the complexity of the tree, $|\mathcal{T}|$, and how well it fits the training data, $R(\mathcal{T})$. In this sense, $R_\alpha(\mathcal{T})$ can be viewed as a penalized objective function similar to what's used in *regularized regression*; see, for example, Hastie et al. [2009, Chap. 3] or Boehmke and Greenwell [2020, Chap. 6]. When $\alpha = 0$, no penalty is incurred, resulting in the most complex tree $\mathcal{T}_0$. On the other extreme, we can always find a large enough value of α that results in a decision tree with no splits (i.e., the root node). Choosing the right value of α is important and can be done using cross-validation or other methods; a specific cross-validation approach is covered in Section 2.5.2.

Breiman et al. [1984, Chap. 10] showed that for each α, there exists a unique smallest subtree, denoted $\mathcal{T}_\alpha$, that minimizes $R_\alpha(\mathcal{T})$. This result is important because it guarantees that no two equally sized subtrees of $\mathcal{T}_0$ will have the same value of $R_\alpha(\mathcal{T})$. To obtain $\mathcal{T}_\alpha$, start pruning $\mathcal{T}_0$ by successively collapsing the internal node that produces the smallest per-node increase to $R(\mathcal{T})$, and continue until reaching the root node. This process results in a (finite) sequence of nested subtrees (see Figure 2.14 on page 73 for an example) that contains $\mathcal{T}_\alpha$; for details, see Breiman et al. [1984, Chap. 10] or Ripley [1996, Sec. 7.2].

To illustrate, take $\mathcal{T}_0$ to be the left tree in Figure 2.12, which has a total of 154 splits. The corresponding tree diagram is displayed in the top left of Figure 2.14. The rest of the tree diagrams in Figure 2.14 correspond to the last 15 trees in the pruning sequence (minus the root node), ending with a decision stump. The optimal subtree, $\mathcal{T}_\alpha$, which has a total of 20 splits (or 21 terminal nodes), was found using 10-fold cross-validation and is highlighted in green.

For comparison, I compared how each subtree performed on an independent test set of 500 new observations. For each subtree in the pruned sequence, the prediction error on the test set, measured as $1 - R^2$, where R^2 is the squared Pearson correlation between the observed and fitted values, was computed. Both the test and cross-validation errors are displayed in Figure 2.15. Here, the results are similar, but the test error suggests a slightly simpler tree with only 18 splits.

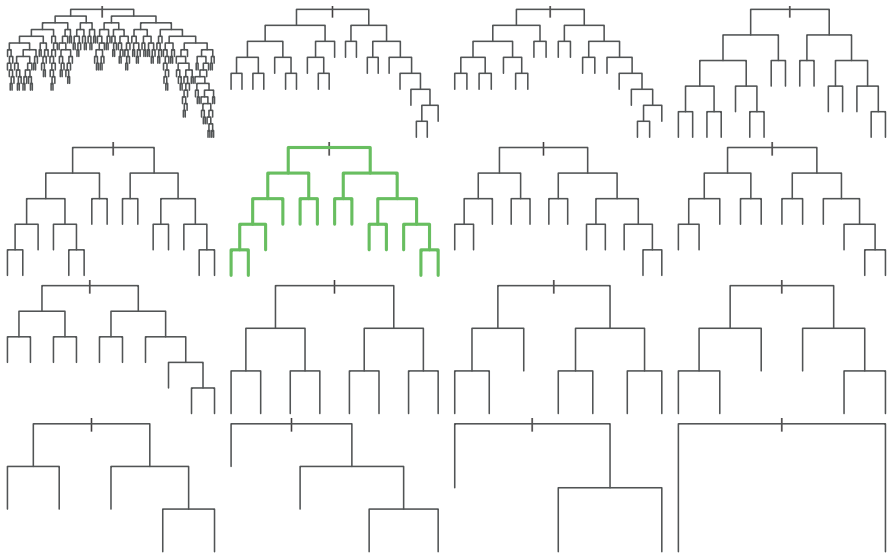

FIGURE 2.14: Nested subtrees for the sine wave example. The optimal subtree, chosen via 10-fold cross-validation, is highlighted in green.

So how is the sequence of α values determined? For any internal node A, we can find α using

$$\alpha = \frac{R(A) - R(\mathcal{T}_A)}{|\mathcal{T}_A| - 1},$$

where $\mathcal{T}_A$ is the subtree rooted at node A. To start pruning, we need to find the first threshold value α_1, which is just the smallest α value among the $|\mathcal{T}| - 1$ internal nodes of the tree $\mathcal{T}$. Once α_1 is obtained, we prune the tree by collapsing one of the $|\mathcal{T}| - 1$ internal nodes and making it a terminal node whenever

$$\alpha_1 \geq \frac{R(A) - R(\mathcal{T}_A)}{|\mathcal{T}_A| - 1}.$$

This results in the optimal subtree, $\mathcal{T}_{\alpha_1}$, associated with $\alpha = \alpha_1$. Starting with $\mathcal{T}_{\alpha_1}$, we then continue this process by finding α_2 in the same way we found α_1 for the full tree $\mathcal{T}$. The process is continued until reaching the root node. It might sound confusing, but we'll walk through the calculations using the mushroom example in the next section.

The **rpart** package, which is used extensively throughout this chapter, employs a slightly friendlier, and rescaled, version of the cost-complexity parameter α, which they denote as *cp*. Specifically, **rpart** uses

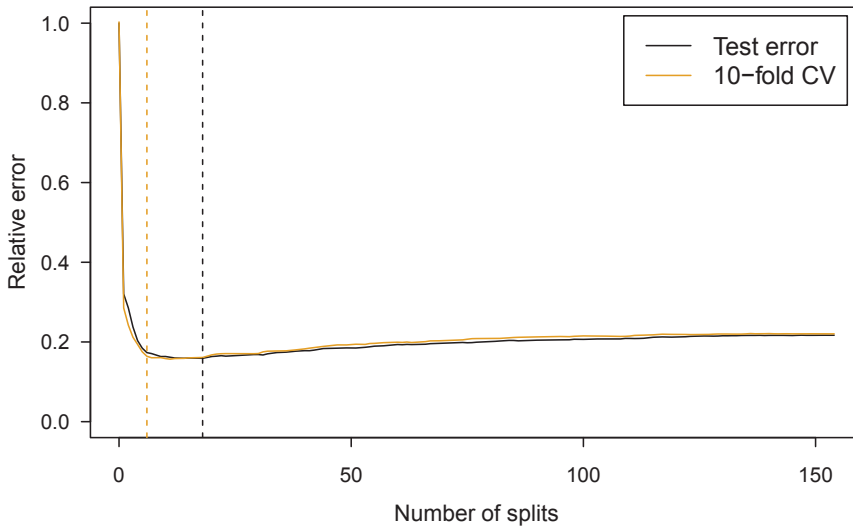

FIGURE 2.15: Relative error based on the test set (black curve) and 10-fold cross-validation (yellow curve) vs. the number of splits for the sine wave example. The vertical yellow line shows the optimal number of splits based on 10-fold cross-validation, while the vertical black line shows the optimal number of splits based on the independent test set.

$$R_{cp}(\mathcal{T}) \equiv R(\mathcal{T}) + cp \times |\mathcal{T}| \times R(\mathcal{T}_1),$$

where $\mathcal{T}_1$ is the tree with zero splits (i.e., the root node). Compared to α, cp is unitless, and a value of $cp = 1$ will always result in a tree with zero splits. The complexity parameter, cp, can also be used as a stopping rule during tree construction. In many open source implementations of CART, whenever $cp > 0$, any split that does not decrease the overall lack of fit by a factor of cp is not attempted. In a regression tree, for instance, this means that the overall R^2 must increase by cp at each step for a split to occur. The main idea is to reduce computation time by avoiding potentially unworthy splits. However, this runs the risk of not finding potentially much better splits further down the tree.

2.5.1.1 Example: mushroom edibility

Let's drive the main ideas home by calculating a few α values to prune a simple tree for the mushroom edibility data. Consider again a simple decision tree for the mushroom edibility data which is displayed in Figure 2.16. This is a simple tree with only three splits, but we'll use it to illustrate how

pruning works and how the sequence of α values is computed. For clarity, the number of observations in each class is displayed within each node, and the node numbers appear at the top of each node. For example, node 8 contains 4208 edible mushrooms and 24 poisonous ones. The assigned classification, or majority class, is printed above the class frequencies in each node. This tree was also built using the observed class priors and equal misclassification costs; hence, $R(\mathcal{T})$ is just the proportion of misclassifications in the learning sample: $24/8124 \approx 0.003$.

Let A_i, $i \in \{1, 2, 3, 4, 5, 8, 9\}$ denote the seven nodes of the tree in Figure 2.16; in **rpart**, the left and right child nodes for any node numbered x are always numbered $2x$ and $2x+1$, respectively (the root node always corresponds to $x = 1$). We can compute the risk of any terminal node using $R(A_i) = N_{j,A}/N_A$. For example, nodes A_5–A_7 all have a risk of zero (since they are pure nodes).

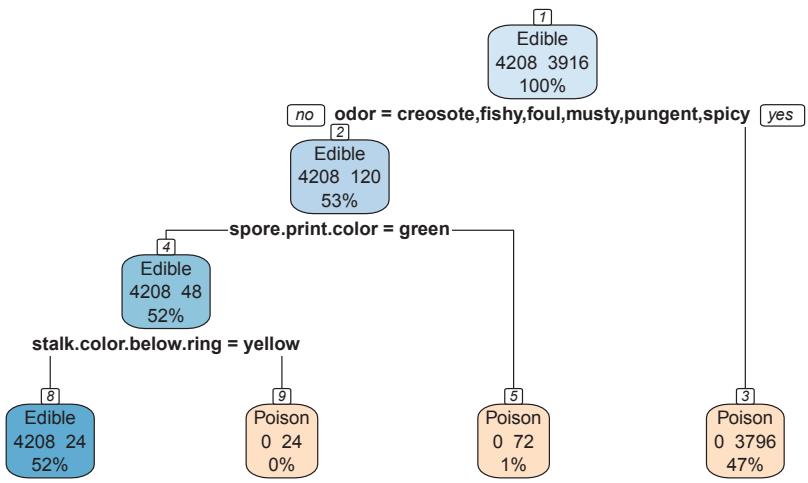

FIGURE 2.16: Classification tree with three splits for the mushroom edibility data. The overall risk of the tree is $24/8124 \approx 0.003$.

To find α_1, we need to first compute α for each of the $|\mathcal{T}_0| - 1 = 3$ internal nodes of the tree, and find which one is the smallest; use the tree diagram in Figure 2.16 to follow along. The α values for the three internal nodes are computed as follows:

$$\begin{aligned}
\alpha_{A_1} &= (3916/8124 - 24/8124) \,/\, (4-1) \approx 0.160 \\
\alpha_{A_2} &= (120/8124 - 24/8124) \,/\, (3-1) \approx 0.006 \\
\alpha_{A_4} &= (48/8124 - 24/8124) \,/\, (2-1) \approx 0.003
\end{aligned} \quad .$$

Since α_{A_4} is the smallest, we collapse node A_4, resulting in the next optimal subtree in the sequence, $\mathcal{T}_{\alpha_1}$, which is displayed in the left side of Figure 2.17. The cost-complexity of this tree is $R_{\alpha_1}(\mathcal{T}_{\alpha_1}) = 0.015$. To find α_2, we start with $\mathcal{T}_{\alpha_1}$ and repeat the process by first finding the smallest α value associated with the $|\mathcal{T}_{\alpha_1}| - 1 = 2$ internal nodes of $\mathcal{T}_{\alpha_1}$. These are given by

$$\begin{aligned} \alpha_{A_1} &= (3916/8124 - 48/8124)/(3-1) \approx 0.238 \\ \alpha_{A_2} &= (120/8124 - 48/8124)/(2-1) \approx 0.009 \end{aligned},$$

making $\alpha_2 = 0.009$. We would then prune the current subtree, $\mathcal{T}_{\alpha_2}$, by collapsing A_2 into a terminal node, resulting in the decision stump displayed in the right side of Figure 2.17. This makes only one possibility for $\alpha_3 = (3916/8124 - 120/8124)/(2-1) \approx 0.467$, which results in the root node after pruning the decision stump, $\mathcal{T}_{\alpha_3}$. In the end, we have the following sequence of α values: $(\alpha_1 = 0.003, \alpha_2 = 0.009, \alpha_3 = 0.467)$. In practice, we would use cross-validation, or some other validation procedure, to select a reasonable value of the complexity parameter α from this sequence. The next two sections discuss choosing α using k-fold cross-validation.

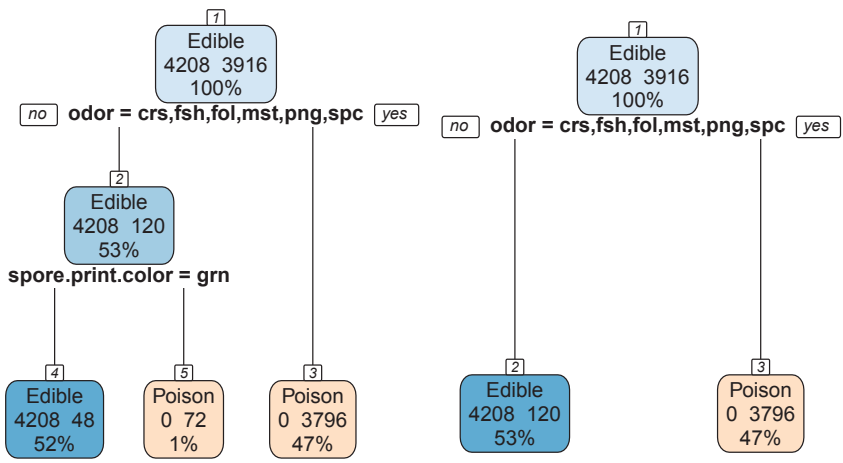

FIGURE 2.17: Optimal subtrees in the sequence with minimum cost-complexity. Since the original tree contains only three splits, there are only two possible subtrees, not counting the tree with zero splits. Here the category names have been truncated to three letters to fit more compactly in the display.

2.5.2 Cross-validation

Once the sequence $\alpha_1, \alpha_2, ..., \alpha_{k-1}$ has been found, we still need to estimate the overall risk/quality of the corresponding sequence of nested subtrees, $R_{\alpha_i}(\mathcal{T})$, for $i = 1, 2, \ldots, k-1$. Breiman et al. [1984, Chap. 11] suggested picking α using a separate validation set or k-fold cross-validation. The latter is more computational, but tends to be preferred since it makes use of all available data, and both tend to lead to similar results. The procedure described in Algorithm 2.1 below follows the implementation in the **rpart** package in R (see the "Introduction to Rpart" vignette):

Algorithm 2.1 K-fold cross-validation for cost-complexity pruning.

---

1) Fit the full model to the learning sample to obtain $\alpha_1, \alpha_2, ..., \alpha_{k-1}$.

2) Define β_i according to

$$\beta_i = \begin{cases} 0 & i = 1 \\ \sqrt{\alpha_{i-1}\alpha_i} & i = 2, 3, \ldots, m-1 \\ \infty & i = m \end{cases}.$$

   Since any value of α in the interval $(\alpha_i, \alpha_{i+1}]$ results in the same subtree, we instead consider the sequence of β_i's, which represent typical values within each range using the geometric midpoint.

3) Divide the data into k groups (or folds), $D_1, D_2, \ldots, D_k$, with approximately k/N observations in each (N being the number of rows in the learning sample). For $i = 1, 2, \ldots, k$, do the following:

   a) Fit the full model to the learning sample, but omit the subset D_i, and find the sequence of optimal subtrees $\mathcal{T}_{\beta_1}, \mathcal{T}_{\beta_2}, \ldots, \mathcal{T}_{\beta_k}$.

   b) Compute the prediction error from each tree on the validation set D_i.

4) For each subtree, aggregate the results by averaging the k out-of-sample prediction errors.

5) Return $\mathcal{T}_\beta$ from the initial sequence of trees based on the full learning sample, where β corresponds to the β_i associated with the smallest prediction error in step 4).

2.5.2.1 The 1-SE rule

When choosing α with k-fold cross-validation, Breiman et al. [1984, Sec. 3.4.3] recommend using the *1-SE rule*, and argue that it is useful in screening out irrelevant features. The 1-SE rule suggests using the most parsimonious tree (i.e., the one with fewest splits) whose cross-validation error is no more than one standard error above the cross-validation error of the best model. This of course requires an estimate of the standard error during cross-validation. A heuristic estimate of the standard error can be found in Breiman et al. [1984, pp. 306–309] or Zhang and Singer [2010, pp. 42–43], but the formula isn't pretty! Applying cost-complexity pruning using cross-validation, with or without the 1-SE rule, would almost surely remove all of the nonsensical splits seen in Figure 2.11. (In fact, this was the case after applying 10-fold cross-validation using the 1-SE rule.)

2.6 Hyperparameters and tuning

There are essentially three hyperparameters associated with CART-like decision trees:

1) the maximum depth or number of splits;
2) the maximum size of any terminal node;
3) the cost-complexity parameter *cp*.

Different software will have different names for these parameters and different default values. Arguably, *cp* is the most flexible and important tuning parameter in CART, and a good strategy is to relax the maximum depth and size of the terminal nodes as much as possible, and use cost-complexity pruning to find an optimal subtree using k-fold cross-validation, or some other validation procedure. In some cases, Chapter 7, for example, trees are intentionally grown to maximal or near maximal depth (in some cases, leaving only a single observation in each terminal node).

2.7 Missing data and surrogate splits

One of the best features of CART is the flexibility with which missing values can be handled. More traditional statistical models, like linear or logistic regression, will often discard any observations with missing values. CART,