Decision Trees

Tree-based methods partition the feature space into a set of rectangles. They can be used for both regression and classification.

Hence the name **CART**, for **classification and regression trees**.

The boundaries of each rectangle are values of the inputs, each of which **splits** the space into two regions.

- With one split on one input, we get two rectangles.
- With two splits on one input, or one on each of two inputs, we get three rectangles.
- For regression trees, the fitted value for any rectangle is the average value of the output for all points in that rectangle.
- Unlike KNN, the locations of the splits depend on the output, not on which input points are nearest.

Multiple trees can be combined in various ways to yield predictors that often perform much better than any single tree.

Regression Trees

To grow a **regression tree**, we use **recursive binary splitting**.

1. Start by splitting the space into two regions based on a single input. Find the predictor and split point that gives the best fit, where the predictor is the mean of the y_i in each region.
2. Next, split one of the regions into two regions, again using the predictor and split point that gives the best fit.
3. Next, do it again. The region we split now could be the one we did not split in step 2, or it could be one just created by the split.
4. Continue until a stopping rule tells us to stop. For example, we might stop if all regions contain less than 5 observations.

Suppose our objective is to minimize a sum of squared residuals:

$$\text{SSR} = \sum_{i=1}^{n} \big(y_i - f(\boldsymbol{x}_i)\big)^2. \tag{1}$$

If the space is divided into M regions, the response at $\boldsymbol{x}_0$ is

$$f(\boldsymbol{x}_0) = \sum_{m=1}^{M} c_m \mathbb{I}(\boldsymbol{x}_0 \in R_m), \tag{2}$$

where R_m is the m^{th} region, and the M coefficients c_m are conditional means, which we have to estimate. $\mathbb{I}(\cdot)$ is the indicator function.

The natural (and probably best) choice for $\hat{c}_m$ is

$$\hat{y}_{\hat{R}_m} = \frac{\sum_{i=1}^{n} y_i \mathbb{I}(\boldsymbol{x}_i \in R_m)}{\sum_{i=1}^{n} \mathbb{I}(\boldsymbol{x}_i \in R_m)} = \frac{1}{n_m} \sum_{\boldsymbol{x}_i \in R_m} y_i, \tag{3}$$

where n_m is the number of points in R_m. This is just the average of the y_i in region m.

There may never be a split for an input that has little impact on y, but there can be many splits for an input that has a big impact.

Initially, we make one split so as to obtain two regions.

If we split according to variable j at the point s, the regions are

$$R_1(j,s) = \{\boldsymbol{x} \,|\, x_j \leq s\} \text{ and } R_2(j,s) = \{\boldsymbol{x} \,|\, x_j > s\}. \tag{4}$$

The estimates $\hat{c}_1$ and $\hat{c}_2$ for regions 1 and 2, respectively, will be

$$\hat{c}_1 = \frac{1}{n_1} \sum_{\boldsymbol{x}_i \in R_1} y_i \quad \text{and} \quad \hat{c}_2 = \frac{1}{n_2} \sum_{\boldsymbol{x}_i \in R_2} y_i. \tag{5}$$

Therefore, we want to choose j and s to minimize

$$\sum_{x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \sum_{x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2. \tag{6}$$

This is not hard to do efficiently, although I suspect the programming is somewhat tricky.

For each variable j, we just search over s to minimize (6). This is cheaper the smaller the number of distinct values of x_j.

Then we choose the variable that, for the optimal choice of s, yields the lowest value of (6).

Next, we split each of the two regions, using the same procedure.

- Sometimes, the second split will involve the same variable as the first one. But in many cases it will involve a different variable.
- Each region has fewer than n points, and the regions get smaller as we make more splits.
- Thus the cost of making a split diminishes with the number of splits we have already made, but there are more splits to consider.
- In principle, we could keep splitting until every terminal node (or leaf) had just one element.
- This is a bad idea! How to stop splitting will be discussed below.

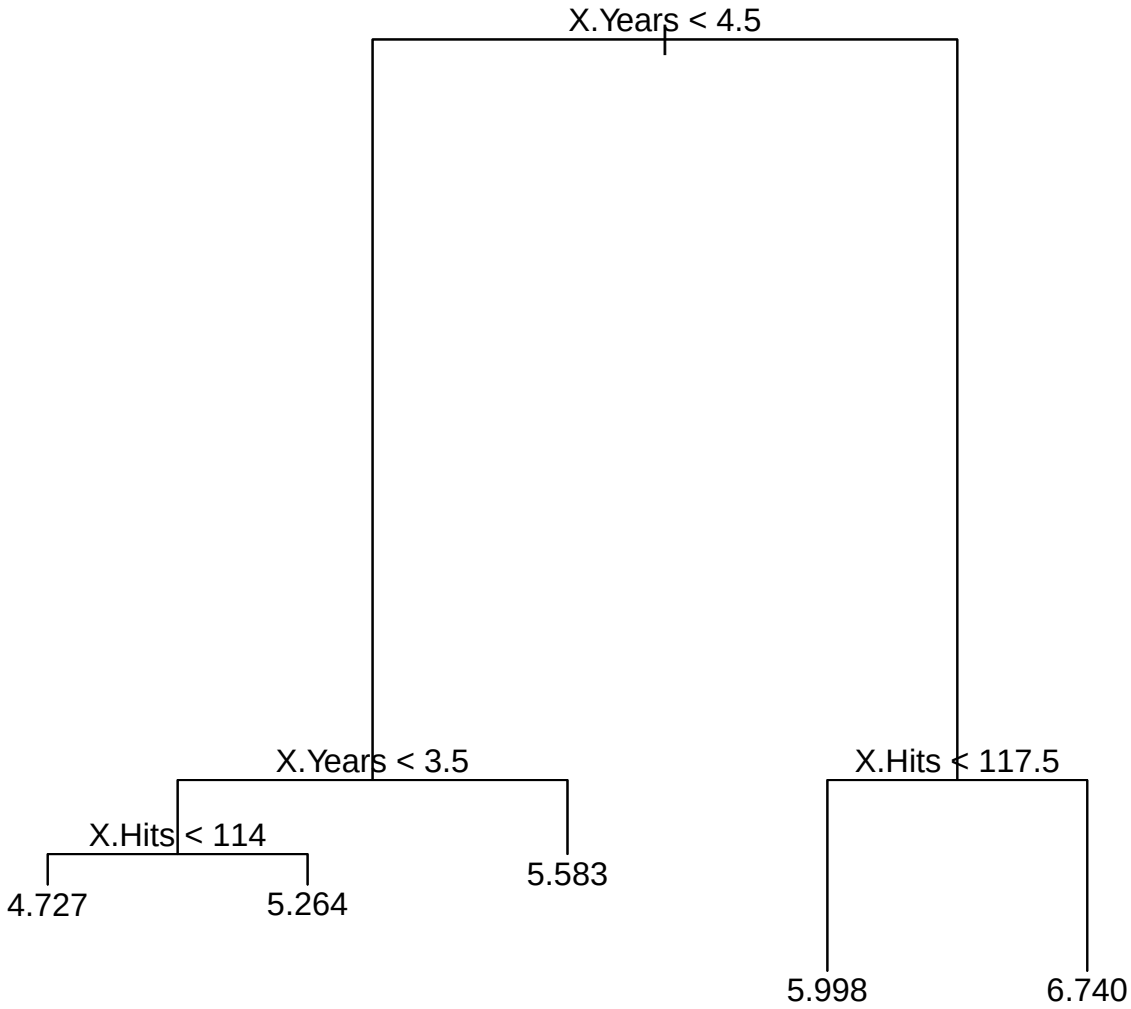
The next few figures concern the salaries of baseball players, in natural logs of thousands of dollars.

The predictors are the number of years they have played in the major leagues, plus several performance measures for the previous year: hits, home runs, runs scored, runs batted in (RBI), and walks.

Note that these figures are not the same as the ones in ISLR/ISLP, partly because I used a larger set of inputs.

- There are 263 observations.
- Figure 15.1 shows a tree that is a bit larger than the one in ISLR/ISLP. There are 5 **terminal nodes**, or **leaves**, versus 3 in Figure 8.1 in the book.
- The three leaves on the left branch replace a single leaf for players with 4.5 or fewer years played. Players in that leaf are predicted to have a log salary of 5.11.

Figure 15.1


X.Years < 4.5
X.Years < 3.5
X.Hits < 117.5
X.Hits < 114
5.583
4.727
5.264
5.998
6.740

The predictor that gets the first split is `Years`. This is probably related to the fact that players become free agents after six years.

- Players with 4 or fewer years of experience get low salaries, which are even lower if they have 3 or fewer years. Note that `Years` is an integer.
- The number of hits matters only for players with 3 or fewer years. For them, having more hits raises salaries from about \$113,000 to about \$193,000.
- Players with 5 or more years of experience get much higher salaries. Weak hitters get about \$403,000, and strong hitters get about \$845,000.

Figure 15.1 is ugly and stupidly designed. We will come back to that point later on.

Figure 15.2 has four panels. The top two show partitions of a binary input (feature) space. The bottom two show other ways to represent the partition shown in the top right.

- The partition in the upper left can never happen with recursive binary splitting.
- The partition in the upper right can happen and seems quite typical. It suggests that the relationship with x_2 is nonlinear.
- There is a step down as x_2 increases for small values of x_1, a step up for large values of x_1, and no step for intermediate values of x_1.
- The tree in the lower left leads to the partition in the upper right.
- The three-dimensional plot in the lower right shows the prediction surface that corresponds to this tree.

Of these plots, only the tree in the lower left could still be used if there were more than two inputs.

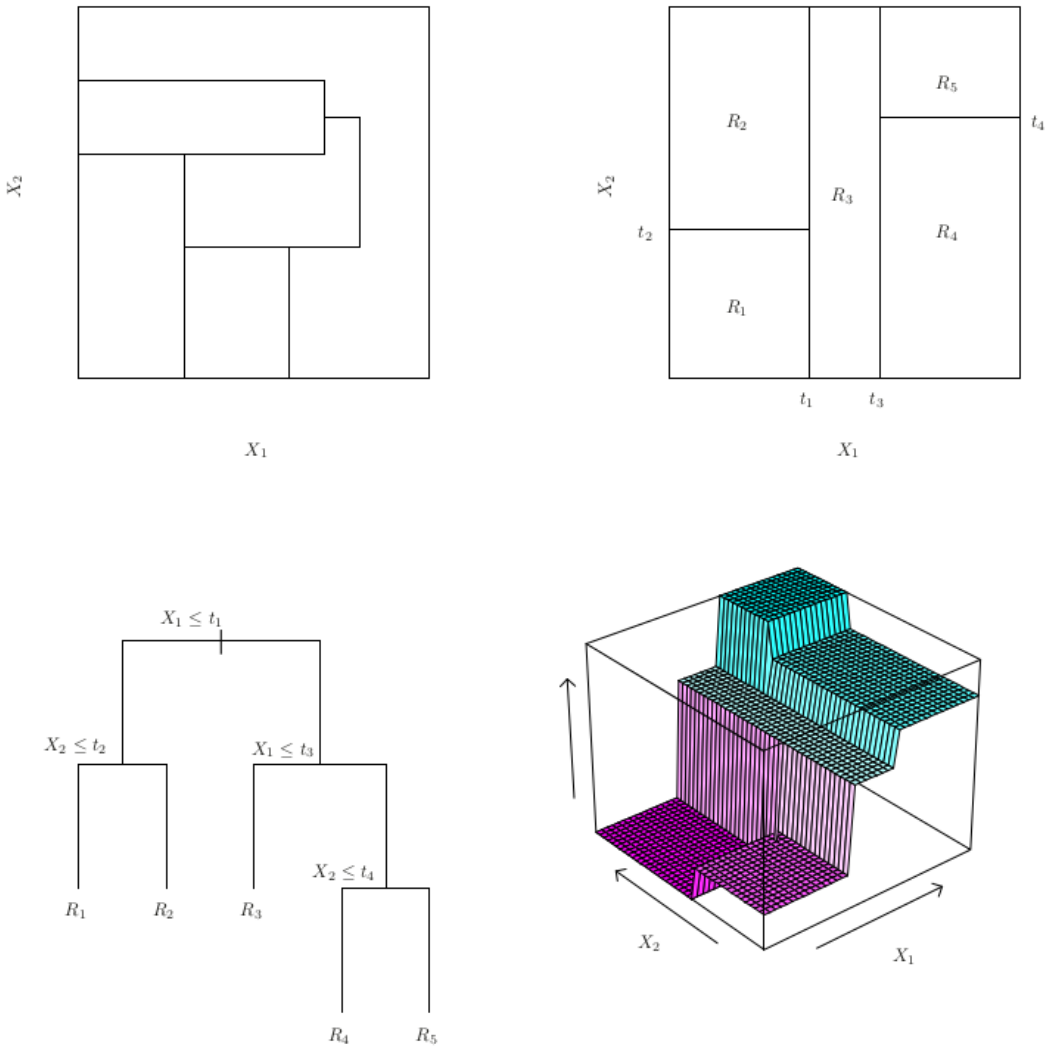


FIGURE 8.3. Top Left: *A partition of two-dimensional feature space that could not result from recursive binary splitting.* Top Right: *The output of recursive binary splitting on a two-dimensional example.* Bottom Left: *A tree corresponding to the partition in the top right panel.* Bottom Right: *A perspective plot of the prediction surface corresponding to that tree.*

How do we decide the number of splits?

One possibility would be to stop as soon as the best proposed split has a sufficiently small effect on the fit. But this is too short-sighted.

The preferred strategy is to grow a very large tree, say T_0, stopping when every region is very small (say, less than 5 points).

The large tree is almost certainly too large, so we need to **prune** it.

This is often done using **cost-complexity pruning**, or **weakest-link pruning**. We want to penalize **node impurity**.

For regression, a measure of node impurity is

$$Q_m(T) = \frac{1}{n_m} \sum_{\boldsymbol{x}_i \in R_m} (y_i - \hat{c}_m)^2, \tag{7}$$

where T denotes some subtree of T_0, with terminal nodes (leaves) indexed by m.

Then define the **cost-complexity criterion**

$$C_\alpha(T) = \sum_{m=1}^{|T|} n_m Q_m(T) + \alpha|T|. \tag{8}$$

Here $|T|$ is the number of nodes in the tree T, and α is a tuning parameter.

$C_\alpha(T)$ is the sum over all terminal nodes of the squared error losses, plus a penalty term. The n_m factors in (7) and (8) cancel out.

For each α, we find the subtree that minimizes (8) by undoing some of the splits that we made previously.

Weakest-link pruning collapses the split that causes the the smallest reduction in $\sum_{m=1}^{|T|} n_m Q_m(T)$. Doing so may or may not cause $C_\alpha(T)$ to decline. The second term gets smaller but the first one gets larger.

When $\alpha = 0$, there is no penalty term in (8). Thus the chosen subtree will just be T_0.

- As α increases, more branches get pruned from the tree.
- They do so in a nested and predictable fashion. This make it easy to obtain the sequence of subtrees $T(\alpha)$.
- As α increases, the number of nodes declines, although not continuously, since it is an integer.
- The value of α is usually chosen by K-fold cross-validation, with K normally 5 or 10.
- This yields the tuning parameter $\hat{\alpha}$ and the associated tree $T(\hat{\alpha})$.
- Note that α plays essentially the same role as the tuning parameter for lasso. The objective function even looks similar, although the meaning of $|T|$ is quite different from the meaning of $|\beta_j|$.

Figure 8.5 from ISLR2

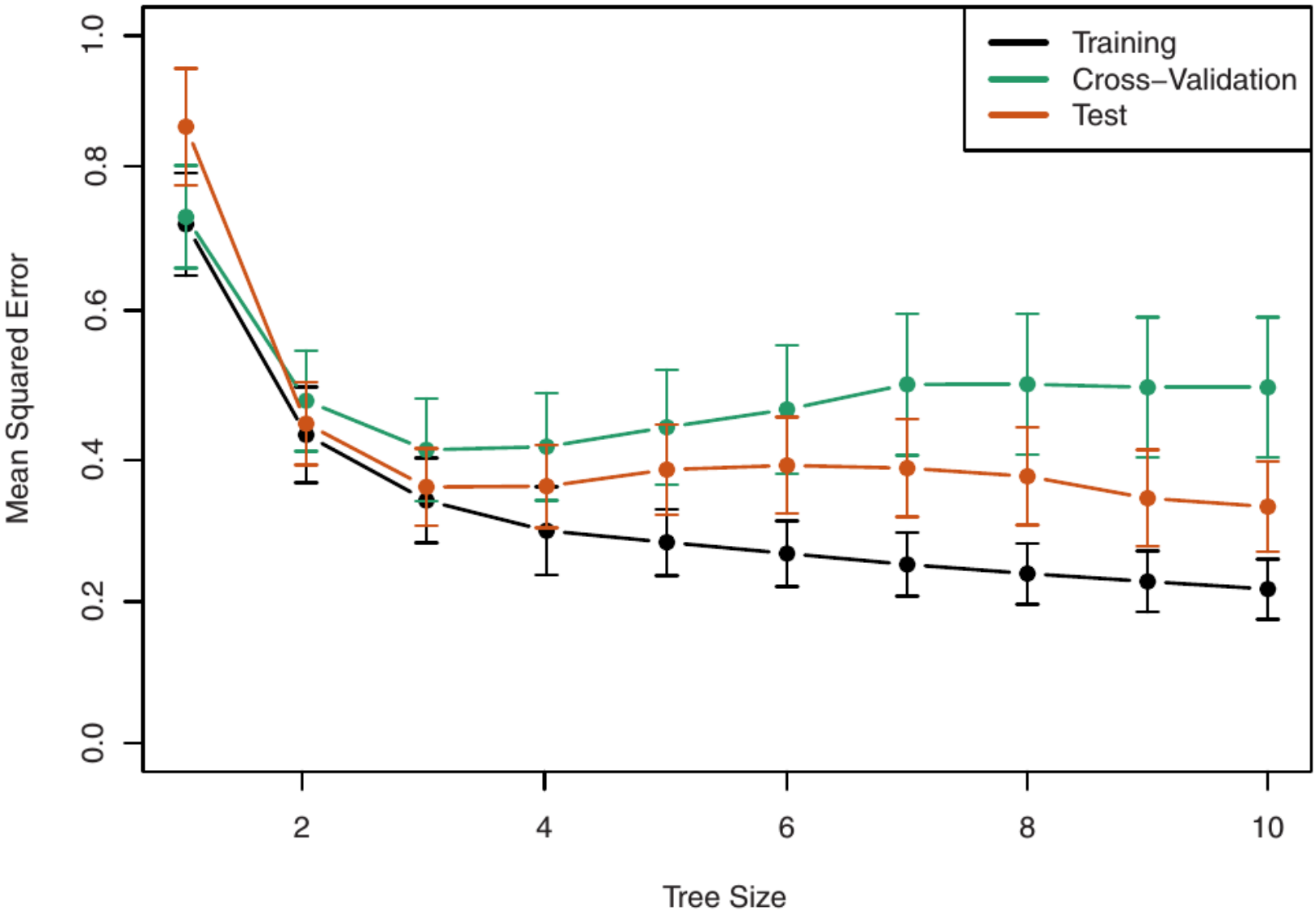


Figure 8.5 from ISLR shows MSE as a function of tree size for the hitters data.

- As must be the case, the training error always diminishes as the tree gets larger.
- In contrast, the cross-validation error has a minimum at only 3 nodes. That yields the simple model mentioned, but not shown, above.
- The test error also has a (local) minimum at 3 nodes, but its global minimum seems to be at 10.
- These results seem strange, but this is a pretty small sample!

The next two figures show a somewhat less parsimonious tree created by the `rpart` package, using two differents sets of graphing options.

Unfortunately, `rpart`'s plotting routines can only graph trees that `rpart` has constructed.

Figure 15.3

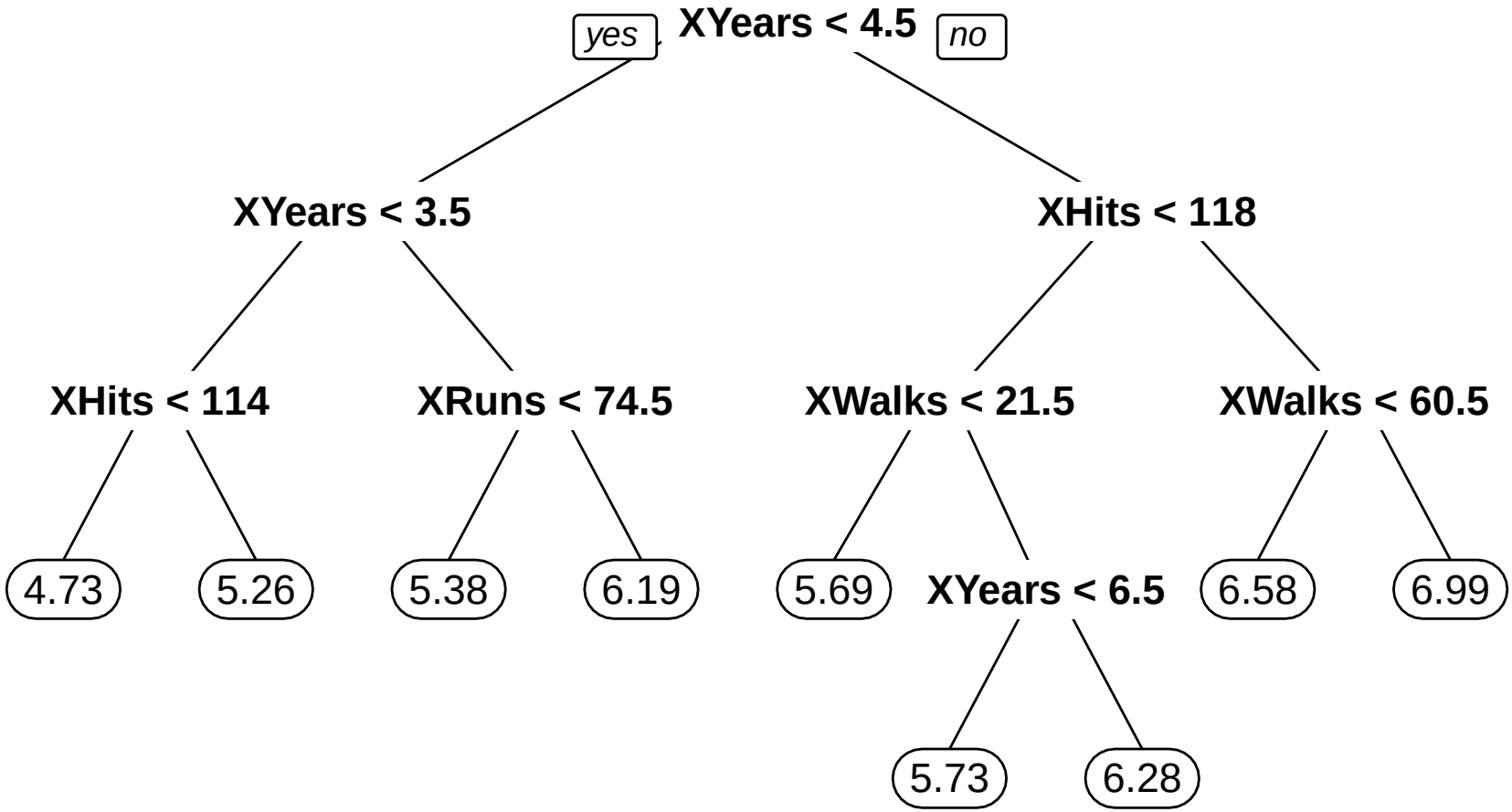
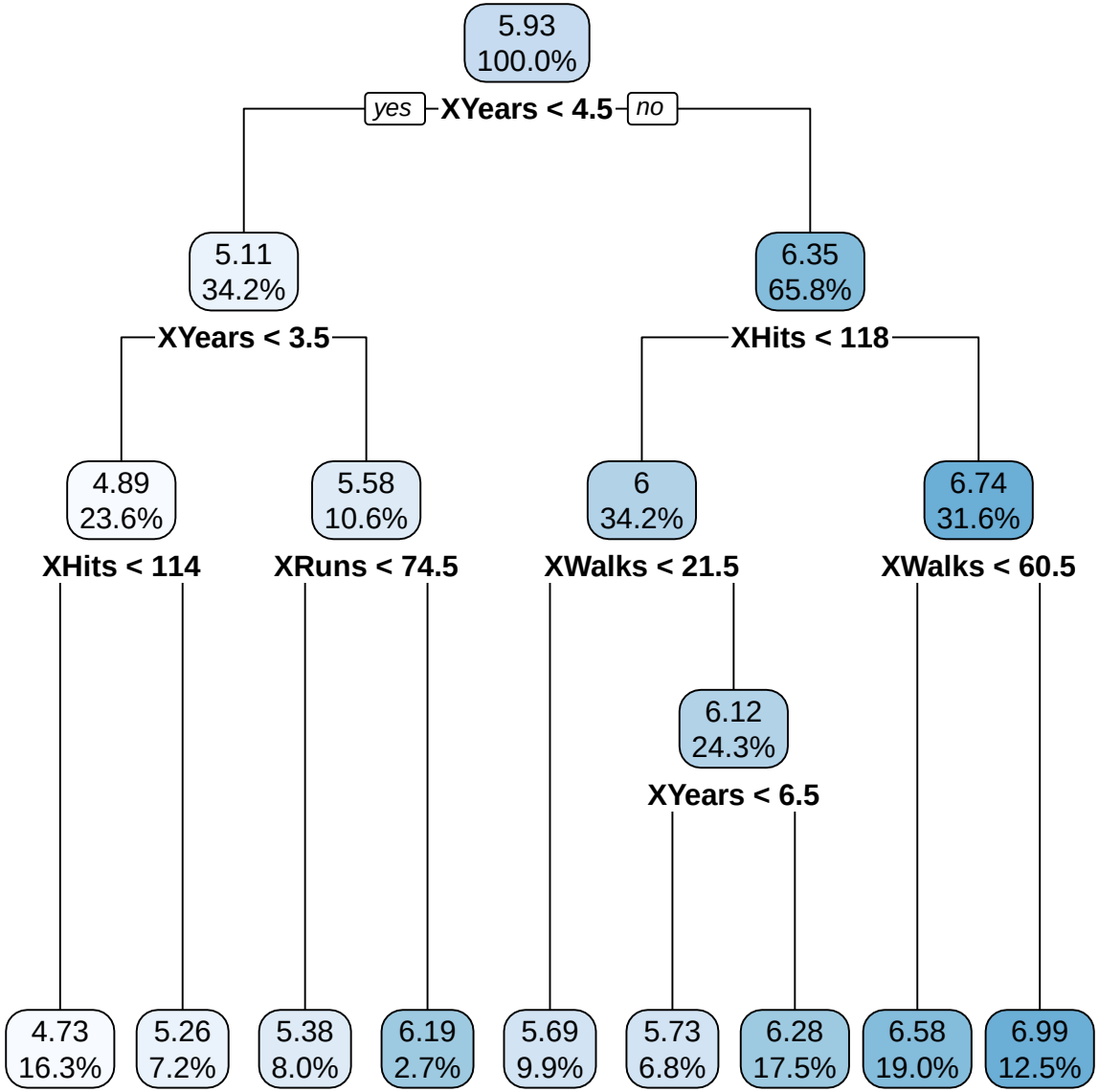

yes
XYears < 4.5
no
XYears < 3.5
XHits < 118
XHits < 114
XRuns < 74.5
XWalks < 21.5
XWalks < 60.5
4.73
5.26
5.38
6.19
5.69
XYears < 6.5
6.58
6.99
5.73
6.28

Figure 15.4


5.93
100.0%
yes
XYears < 4.5
no
5.11
34.2%
6.35
65.8%
XYears < 3.5
XHits < 118
4.89
23.6%
5.58
10.6%
6
34.2%
6.74
31.6%
XHits < 114
XRuns < 74.5
XWalks < 21.5
XWalks < 60.5
6.12
24.3%
XYears < 6.5
4.73
16.3%
5.26
7.2%
5.38
8.0%
6.19
2.7%
5.69
9.9%
5.73
6.8%
6.28
17.5%
6.58
19.0%
6.99
12.5%

The Mileage Data

I created a regression tree for the mileage data. The output was gallons per mile (`gpm`) and the inputs were horsepower (`hp`) and weight (`wght`).

The tree had six nodes, all of which survived pruning based on cross-validation. Figure 15.5 shows the 6-node tree for mileage.

- The first split is based on weight.
- Then the light and heavy cars are both split based on horsepower.
- Then the heavy and powerful cars are again split based on weight.
- Finally, the really heavy cars are split on horsepower.

Even though the six-node tree is very simple, it works quite well.

A linear regression of `gpm` on `hp` and `wght` has a residual standard error of 0.007173.

The 6-node tree has a residual standard error of 0.007199, which is almost identical. See Figure 15.6.

Figure 15.5 — Tree for Mileage

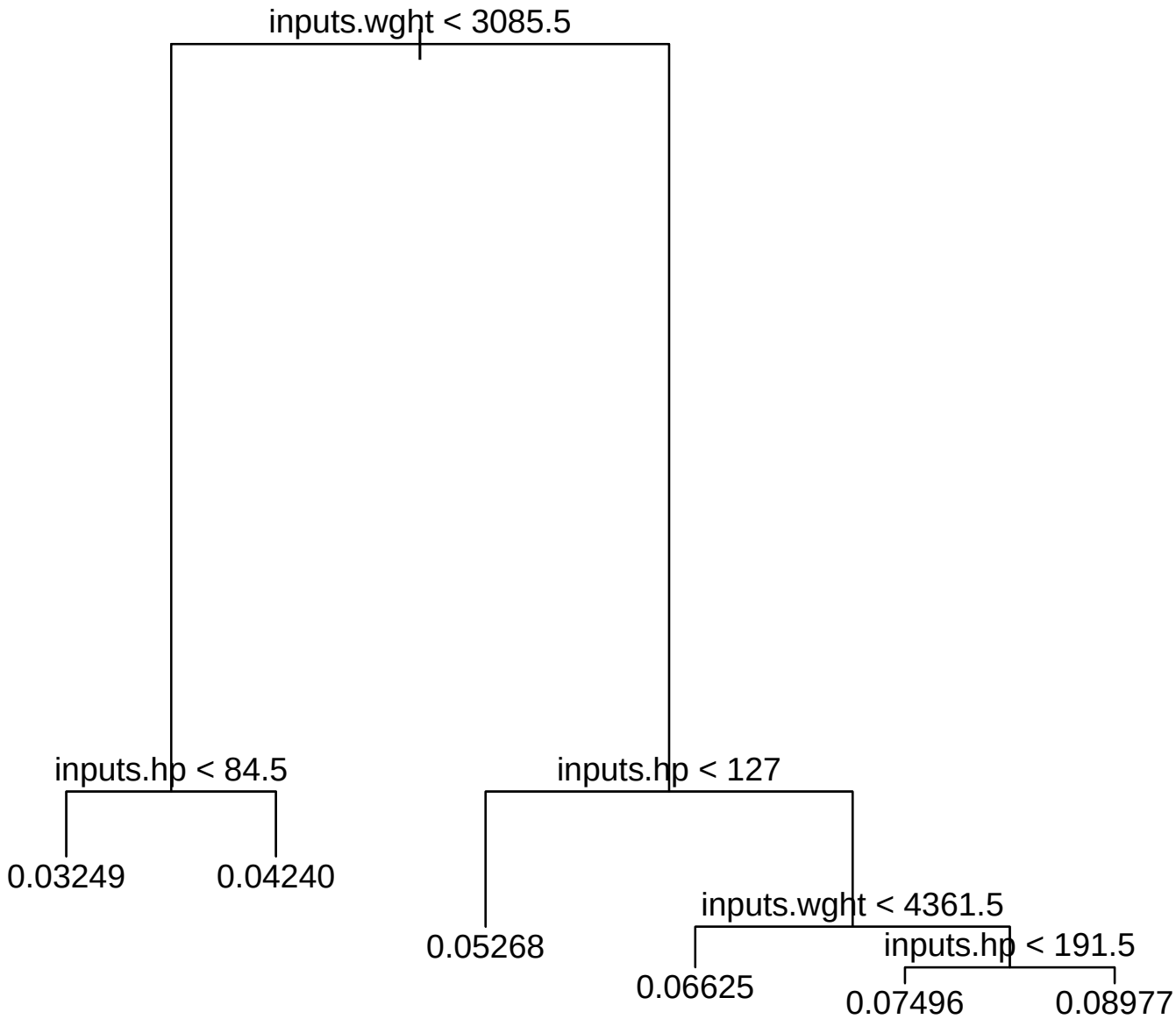
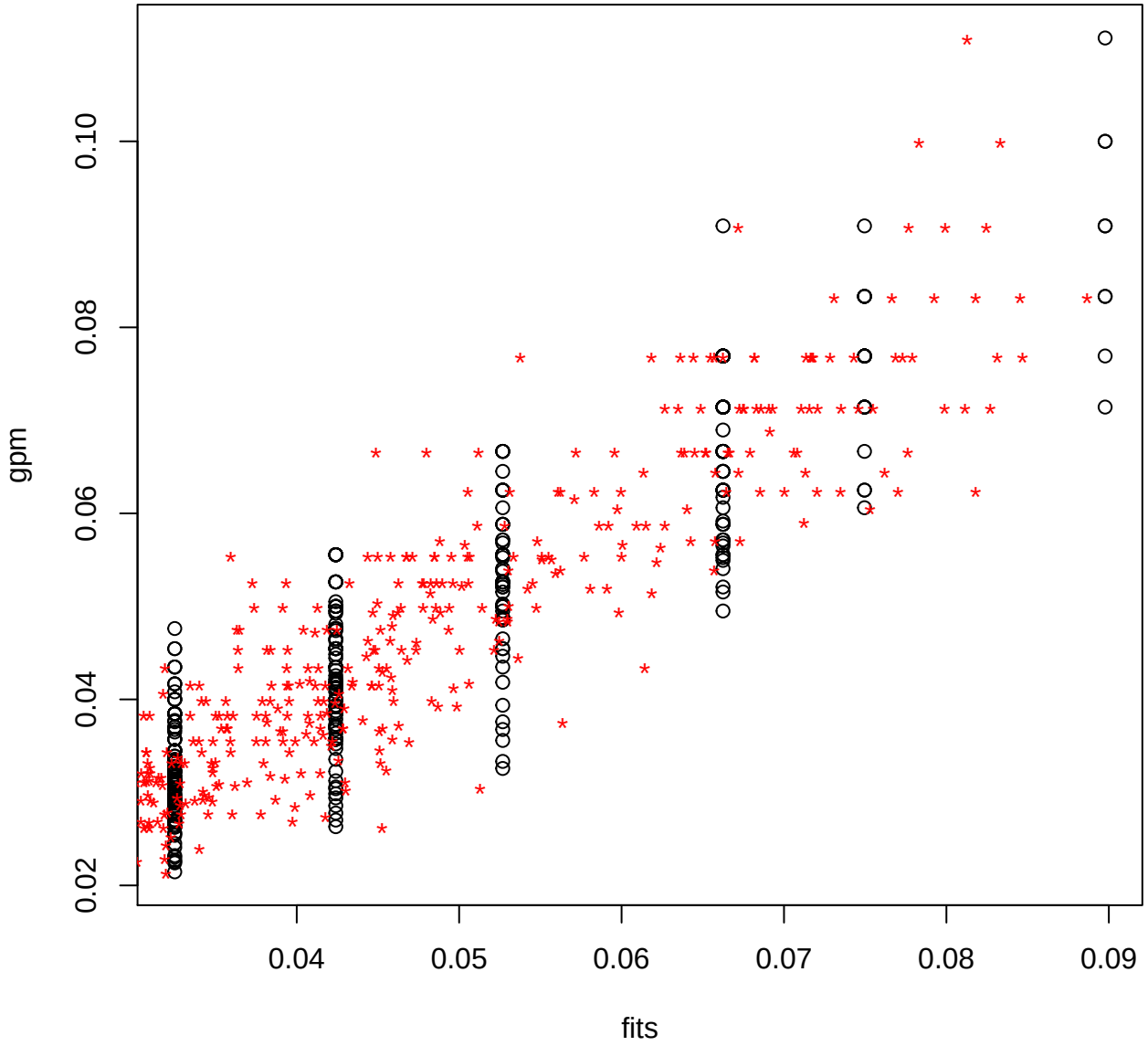


Figure 15.6 — Two Fits vs. Actual for Mileage



With the standard `tree` package, you seem to be stuck with ugly trees that waste an enormous amount of vertical space.

Instead, you can use the `rpart` package and its associated `rpart.plot` package, which contains the `prp` command.

Unfortunately, it seems that `rpart.plot` can only plot trees created using `rpart`.

However, `tree` and `rpart` should be able to produce the same trees.

The following commands produce the output shown in Figure 15.7:

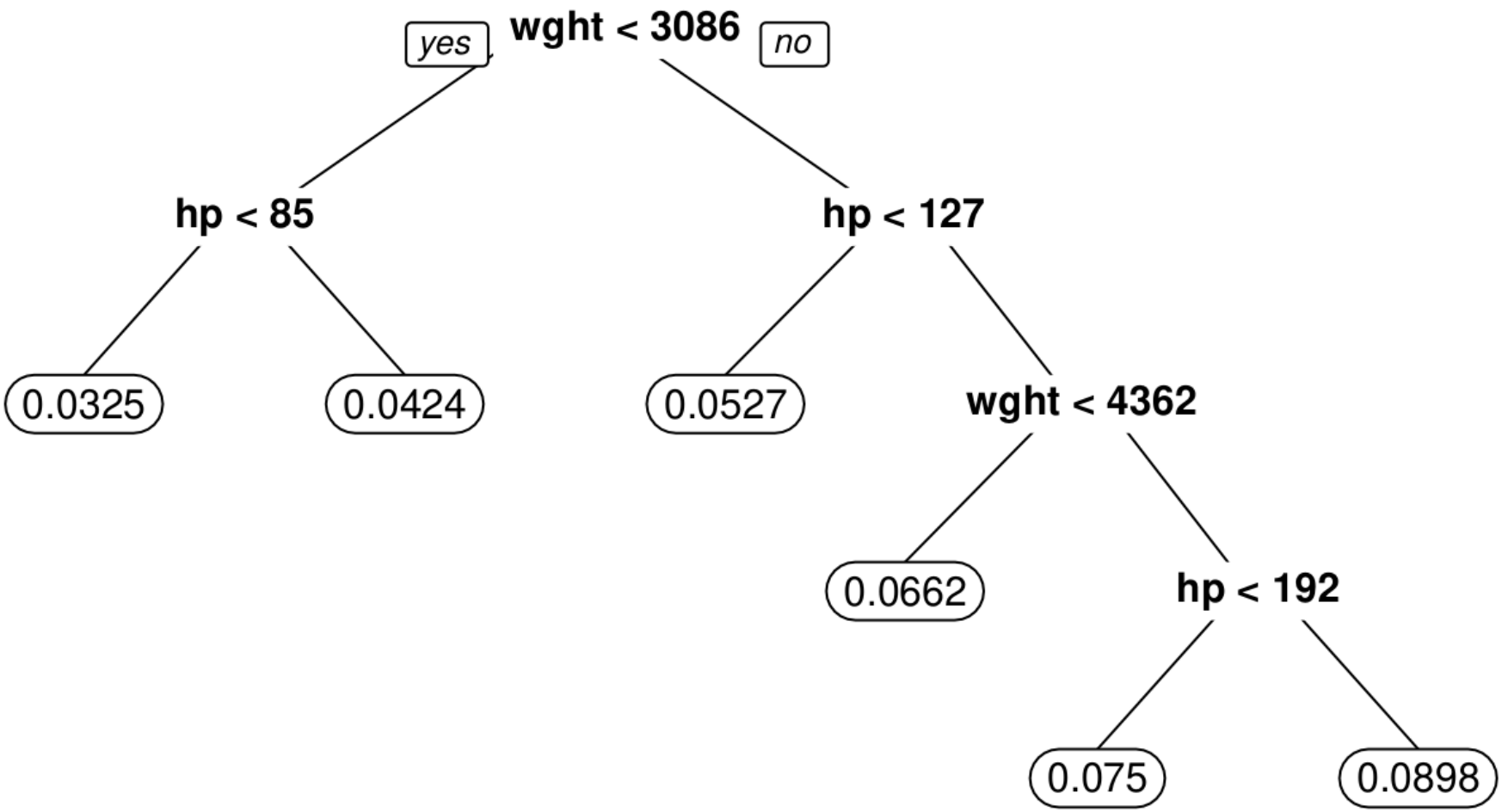
```
tree.rpart <- rpart(gpm ~ hp + wght)
prp(tree.rpart, digits=3)
```

To get a tree like the one in Figure 15.4, use

```
rpart.plot(tree.rpart, digits=3)
```

The `prp` command accepts many options, while `rpart.plot` is a front-end that is easier to use.

Figure 15.7 — Tree for Mileage Using rpart.plot



Classification Trees

Building a **classification tree** is similar to building a regression tree.

When there are just two classes, we can pretend that a 0-1 output is continuous and simply build a regression tree.

For more than two classes, however, we need to change what we are trying to minimize.

Suppose that

$$\hat{p}_{mk} = \frac{1}{n_m} \sum_{x_i \in R_m} \mathbb{I}(y_i = k) \tag{9}$$

denotes the proportion of class k observations in node m.

We can assign node m to class k if $\hat{p}_{mk}$ is higher for k than for any other class. The class with the highest proportion of the observations in node m is denoted $k(m)$.

Ideally, every output in each node would belong to the same class.

That is usually impossible to attain. To the extent possible, we try to avoid **node impurity**, which can be measured in several ways.

Three measures are discussed on the next slide.

We can use any of them when making the splits. In each case, we add them up over all $m = 1, \ldots, M$ to obtain the objective function that we are minimizing.

It is particularly natural to base a split on a qualitative variable.

- One branch corresponds to 0, and one branch corresponds to 1.
- There is no need to search for the optimal split point.

It is also easy to split on the basis of categorical variables that take on just a few values.

- Unlike with regression models, we do not need to convert a categorical variable into several dummies using `factor`, although order may matter if we do not.
- If there are C categories, then there are $C - 1$ split points.

Three measures of node impurity:

1. The **misclassification error** is

$$\frac{1}{n_m} \sum_{i \in R_m} \mathbb{I}\big((y_i \neq k(m)\big). \tag{10}$$

This is not differentiable and is not sensitive to the values of $\hat{p}_{mk}$ except at points where $k(m)$ changes.

2. The **Gini index** is

$$\sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}). \tag{11}$$

The variance of a 0-1 response with probability $\hat{p}_{mk}$ is $\hat{p}_{mk}(1 - \hat{p}_{mk})$. Summing this over all classes gives the Gini index.

3. The **entropy** or **deviance** is

$$-\sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk}). \tag{12}$$

Since $0 \leq \hat{p}_{mk} \leq 1$, it follows that $D \geq 0$.

Trees and linear models are very different.

- Trees completely dispense with the additivity assumption.
- Trees, at least small ones, are very easy to explain and to display graphically.
- Trees may work well for decisions that people make based on a **lexicographic ordering**.
- Trees can easily handle qualitative predictors without the need to create dummy variables, although results may depend on how the predictors are coded.
- Trees can handle a very large number of inputs, perhaps even $p >> n$, but the final tree will not use most of them.
- Trees typically have less predictive accuracy than approaches based on regression, such as GAMs.
- Trees can be very non-robust. A small change in the data can cause a large change in the estimated tree.