# Chapter 5

# Decision Trees

**EXAM PA LEARNING OBJECTIVES**

**7. Topic: Decision Trees**

**Learning Objectives**

The Candidate will be able to construct decision trees for both regression and classification.

a) Understand the basic motivation behind decision trees.

b) Construct regression and classification trees.

c) Use bagging and random forests to improve accuracy.

d) Use boosting to improve accuracy.

e) Select appropriate hyperparameters for decision trees and related techniques.

*Chapter overview:* This chapter introduces the second type of supervised learning technique in Exam PA, namely, *decision trees*. Just like GLMs, decision trees can be applied to tackle both regression and classification problems with both numeric and categorical predictors, but with a fundamentally different approach. While GLMs provide a prediction equation based on a linear combination of the predictors, decision trees divide the feature space (i.e., the set of all combinations of feature values) into a set of non-overlapping and exhaustive regions of relatively homogeneous observations more amenable to prediction. To predict a new case, simply locate the region to which it belongs and use the average value (mean) or the majority class (mode) of the target variable there as the prediction. Because the rules of prediction can be vividly represented in the form of a tree, the resulting predictive model is aptly called a decision "tree."

In Section 5.1, we will look at the basic theory behind decision trees, set up the terminology that will be used throughout the chapter, and learn, at a conceptual level, how a decision tree can be fitted and evaluated. More complex tree models based on ensemble methods are also discussed. Sections 5.2 and 5.3 dive into the practical implementation of the tree-building process, with Section 5.2 presenting a case study that involves a small-scale, two-dimensional dataset and aims to acquaint you with the output generated by R's tree-fitting functions. Then Section 5.3 is a full-length, more sophisticated case study that demonstrates how base and ensemble classification trees can be constructed and evaluated in terms of prediction accuracy, and what considerations go into the tree selection process.

**EXAM NOTE**

As pointed out in Section 3.1, there are only two supervised learning techniques in Exam PA, GLMs and decision trees. While GLMs have been heavily tested in the new exam format (see the June 2019 PA exam and the Hospital Readmissions sample project), decision trees have not. You will see in this chapter that there are not too many interesting things to test about trees in an exam environment, so if the SOA decides to test the construction of decision trees, chances are that they will "marry" trees with GLMs to come up with a comprehensive exam project and ask that you compare the strengths and weaknesses of these two types of predictive models.

# 5.1 Conceptual Foundations of Decision Trees

## 5.1.1 Base Decision Trees

**Decision trees in a nutshell.** As mentioned in the introduction, a decision tree develops a series of classification rules or splits based on the values of the predictor variables. Observations in the same group based on these classification rules are relatively stable (with respect to the behavior of the target variable) and thus much easier to analyze and predict. We will discuss how these splits are constructed very shortly. For now, it is enough to know that to predict or classify a new observation, we drop it down the decision tree and use the classification rules to identify the group to which it belongs. What we use as the prediction depends on the nature of the target variable.[i]

- *Quantitative target variables:* Use the average (mean) of the target variable in that group as the predicted value.
- *Qualitative target variables:* Use the most common class (mode) of the target variable in that group as the predicted class.

Terminology-wise, a decision tree for predicting a quantitative target is called a *regression tree*, whereas a decision tree for predicting a qualitative target is called a *classification tree*. Both types of trees will be covered in this chapter.

A toy example of a decision tree with $p = 2$ quantitative predictors, $X_1$ and $X_2$, is shown in Figure 5.1.1 (a). The top split classifies an observation into the left branch if its $X_1$ value is less than 10; otherwise, it goes to the right and is further assigned depending on whether $X_2 < 8$ or $X_2 \geq 8$. Overall, our predictive model can be represented in the form of a tree and segments the two-dimensional predictor space into three regions, shown in Figure 5.1.1 (b). Conventionally, a decision tree is displayed upside down, with the leaves shown at the bottom and the root (or trunk) shown at the top.

**Example 5.1.1. (SOA Exam SRM Sample Question 33: Predicting three new observations)** The regression tree shown below was produced from a dataset of auto claim payments.

[i]While GLMs assume that the signal function is a linear combination of the predictors, i.e., $f(\mathbf{X}) = \beta_0 + \sum_{j=1}^{p} \beta_j X_j$, decision trees assume the piecewise constant form $f(\mathbf{X}) = \sum_{m=1}^{M} c_m \cdot 1_{\{\mathbf{X} \in R_m\}}$, for some (unknown) constants $c_1, \ldots, c_M$ and non-overlapping regions $R_1, \ldots, R_M$ in the feature space.

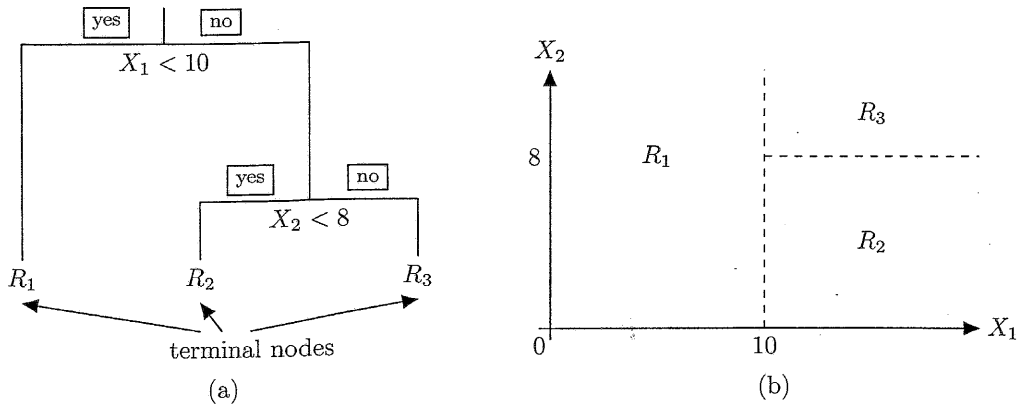
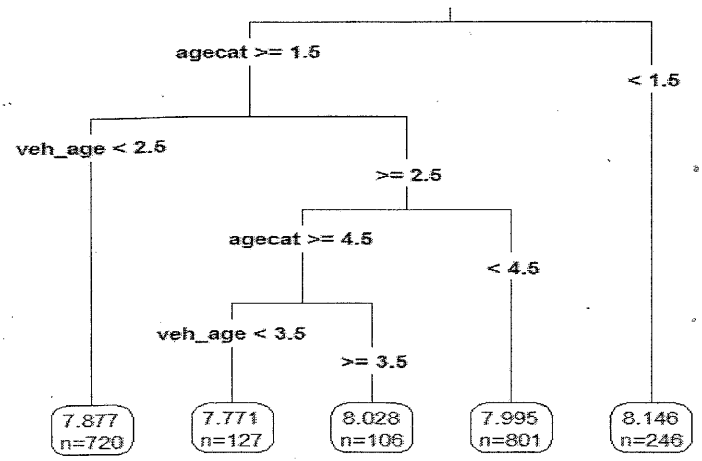



Figure 5.1.1: A toy decision tree with $p = 2$ predictors (left) and the resulting partition of the predictor space (right).

Age Category (1, 2, 3, 4, 5, 6) and Vehicle Age (1, 2, 3, 4) are both predictor variables, and log of claim amount (LCA) is the dependent variable.




Consider three autos I, II, III:

I: An Auto in Age Category 1 and Vehicle Age 4

II: An Auto in Age Category 5 and Vehicle Age 5

III: An Auto in Age Category 5 and Vehicle Age 3

Rank the estimated LCA of Autos I, II, and III.

(A) LCA(I) < LCA(II) < LCA(III)

(B) LCA(I) < LCA(III) < LCA(II)

(C) LCA(II) < LCA(I) < LCA(III)

(D) LCA(II) < LCA(III) < LCA(I)

(E) LCA(III) < LCA(II) < LCA(I)

*Solution.* I: This auto belongs to Age Category 1 and so is classified to the rightmost branch, with a predicted LCA of 8.146.

II: This auto is predicted as follows:

$$\texttt{agecat} \geq 1.5 \quad \rightarrow \quad \texttt{veh\_age} \geq 2.5 \quad \rightarrow \quad \texttt{agecat} \geq 4.5 \quad \rightarrow \quad \texttt{veh\_age} \geq 3.5.$$

The predicted LCA is 8.028.

III: This auto is predicted as follows:

$$\texttt{agecat} \geq 1.5 \quad \rightarrow \quad \texttt{veh\_age} \geq 2.5 \quad \rightarrow \quad \texttt{agecat} \geq 4.5 \quad \rightarrow \quad \texttt{veh\_age} < 3.5.$$

The predicted LCA is 7.771.

In conclusion, we have $\boxed{\text{LCA(III)} < \text{LCA(II)} < \text{LCA(I)}}$. **(Answer: (E))** □

**Some commonly used terms.** The following terms are useful for describing a generic decision tree:

- *Node:* A *node* is a point on a decision tree that corresponds to a subset of the data and often results from one or more binary splits.
- *Root node:* This is the node at the top of a decision tree representing the full dataset. In Figure 5.1.1 (a), the root node is right before the first split by $X_1 < 10$.
- *Terminal node (or leaf):* These are nodes at the bottom of a tree which are not split further. In Figure 5.1.1 (a), there are three terminal nodes, $R_1, R_2$, and $R_3$, defined by

$$\begin{aligned} R_1 &= \{(X_1, X_2) \in \mathbb{R}^2 : X_1 < 10\}, \\ R_2 &= \{(X_1, X_2) \in \mathbb{R}^2 : X_1 \geq 10, X_2 < 8\}, \\ R_3 &= \{(X_1, X_2) \in \mathbb{R}^2 : X_1 \geq 10, X_2 \geq 8\}. \end{aligned}$$

- *Binary tree:* This is a decision tree in which each node has only two children. In Exam PA, we will only use binary trees.
- *Depth:* The number of branches from the tree's root node to the furthest terminal node. It is a measure of the complexity of a decision tree.

**Construction of decision trees: How to make the splits?** To construct a decision tree, we have to think about how the series of binary classification rules, or equivalently, the series of binary splits should be designed. In theory, these binary splits can be constructed using arbitrary configurations of the predictors. In practice, for simplicity it is customary to use binary splits that separate the observations into two groups according to the value of *one and only one* of the predictors. In Figure 5.1.1 (a), for instance, each of the two splits involves either $X_1$ or $X_2$, but not

both. In a two-dimensional feature space, these binary splits partition the coordinate plane into a collection of boxes or rectangles.

How should we go about selecting the predictor to split and the corresponding cutoff level at each step? Remember, our goal is to create terminal nodes that contain similar observations, which are easy to analyze and predict. To do this, in every split we partition the target observations by whichever variable results in the greatest reduction of impurity. Measures of impurity differ depending on whether the decision tree is a regression tree or a classification tree.

- *Regression trees:* Analogous to linear models, the variability of a numeric target variable in a particular node of a decision tree is quantified by the residual sum of squares, or RSS. The lower the RSS, the more homogeneous the target observations. Mathematically, if $R_m$ denotes a certain node (not necessarily a terminal node) of the decision tree, then the RSS of the target variable in $R_m$ is

$$\text{RSS}_m = \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2,$$

where $\hat{y}_{R_m} = \sum_{i \in R_m} y_i / |R_m|$ is the mean of the target variable in $R_m$ ($|R_m|$ is the number of observations in $R_m$) and serves as our prediction for the target variable there.

In terms of the RSS, the first split coming from the root node should be such that the overall RSS after the split

$$\underbrace{\sum_{i:X_{ij}<s} (y_i - \hat{y}_{R_1})^2}_{\text{RSS of obs. in left branch}} + \underbrace{\sum_{i:X_{ij} \geq s} (y_i - \hat{y}_{R_2})^2}_{\text{RSS of obs. in right branch}},$$

where $\hat{y}_{R_1}$ and $\hat{y}_{R_2}$ are the means of the target variable in the two resulting nodes, is the smallest among all possible choices of the predictor $X_j$ and the cutoff level $s$.[ii] The same idea applies to subsequent binary splits along the tree.

- *Classification trees:* Classification trees are more difficult to deal with than regression trees because there are several measures of node impurity. For all of these measures, the larger their values, the more impure or heterogeneous the node is. Here are three common node impurity measures: (let's assume a given node $R_m$ and that the target variable has $K$ distinct classes)

  1. *Entropy:* Let $p_k$ be the proportion of the target observations of the $k$th class in the given tree node. In terms of the $p_k$'s, the *entropy* of the node is defined as

$$E = -\sum_{k=1}^{K} p_k \log_2(p_k);$$

note that it is customary to take the logarithm using base 2. By construction, the value of the entropy increases with the degree of impurity in the node. Its smallest value is 0, attained when one class is completely dominant,[iii] and the largest value is $\log_2 K$, attained when the observations are evenly split among the $K$ classes (i.e., $p_k = 1/K$ for each $k$).

[ii] The discussion here uses numeric predictors, but splits can also be made using categorical predictors, for which the split choices are all possible ways the predictor levels can be partitioned into two subsets.

[iii] By convention, $0 \log_2 0$ is defined to be 0.

2. *Gini:* In a similar spirit to entropy, the *Gini index* is defined as

$$G = \sum_{k=1}^{K} p_k(1 - p_k) = 1 - \sum_{k=1}^{K} p_k^2.$$

   Like entropy, the higher the degree of node impurity, the higher the value of $G$.

3. *Classification error:* Remember that our predicted class label for a categorical target variable is the majority class, i.e., the class with the highest proportion of observations. In other words, $\max_{1 \le k \le K} p_k$ of the observations in the node are correctly classified and the complement $1 - \max_{1 \le k \le K} p_k$ is the *classification error.*

In most cases, the choice of the node impurity measure does not have a dramatic impact on the performance of a classification tree. However, entropy and Gini are differentiable, and thus technically more amenable to numerical optimization.

Given an impurity measure, the best binary split at each step is the split that maximizes the reduction in the impurity measure. The resulting splits are further split recursively (i.e., operating on the results of previous splits) and the splitting process continues until we reach a stopping criterion, e.g., the number of observations in a node falls below some pre-specified threshold or there are no more splits that can contribute to a significant reduction in the node impurity. In Exam SRM, this process is called *recursive binary splitting* (this term is not mentioned in the PA e-learning modules).

**Example 5.1.2. (SOA Exam SRM Sample Question 9: Comparing the classification error rate, Gini index, and cross-entropy of two splits)** A classification tree is being constructed to predict if an insurance policy will lapse. A random sample of 100 policies contains 30 that lapsed. You are considering two splits:

Split 1: One node has 20 observations with 12 lapses and one node has 80 observations with 18 lapses.

Split 2: One node has 10 observations with 8 lapses and one node has 90 observations with 22 lapses.

The total Gini index after a split is the *weighted average* of the Gini index at each node, with the weights proportional to the number of observations in each node.

The total entropy after a split is the *weighted average* of the entropy at each node, with the weights proportional to the number of observations in each node.

Determine which of the following statements is/are true.

I. Split 1 is preferred based on the total Gini index.

II. Split 1 is preferred based on the total entropy.

III. Split 1 is preferred based on having fewer classification errors.

(A) I only

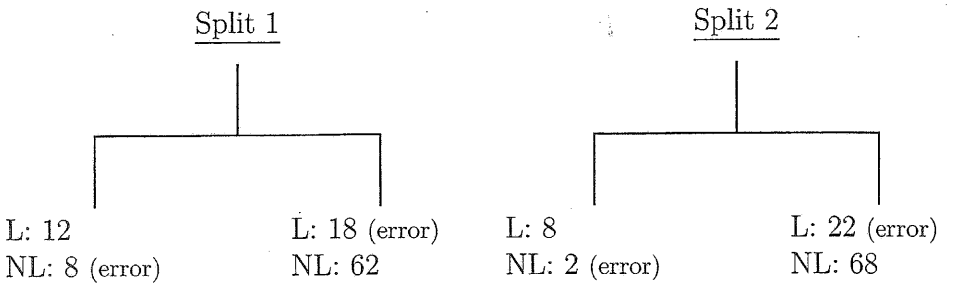
(B) II only

(C) III only

(D) I, II, and III

(E) The correct answer is not given by (A), (B), (C), or (D).

*Solution.* To aid calculations, let us summarize the results of the two splits by the following diagram, where "L" and "NL" stand for "lapse" and "not lapse", respectively.




For example, we are told that the first node of Split 1 has 20 observations with 12 lapses. Accordingly, there must be $20-12=8$ non-lapses. Since "lapse" is the most commonly occurring class in that node, the predicted class for that node is "lapse" and so the 8 non-lapses are misclassified.

I. For Split 1, the estimated probabilities of lapse in the two nodes are $\hat{p}_{1L}=12/(12+8)=0.6$ and $\hat{p}_{2L}=18/(18+62)=0.225$, so the total Gini index, weighted by the two Gini indices of the two nodes, is

$$G=\frac{20}{100}\times 2(0.6)(1-0.6)+\frac{80}{100}\times 2(0.225)(1-0.225)=0.375.$$

For Split 2, we have $\hat{p}_{1L}=0.8$ and $\hat{p}_{2L}=11/45$, so the total Gini index is

$$G=\frac{10}{100}\times 2(0.8)(1-0.8)+\frac{90}{100}\times 2\left(\frac{11}{45}\right)\left(1-\frac{11}{45}\right)=0.3644.$$

The smaller the Gini index, the better the split, so Split 2 is preferred.

II. Using the results above, we can calculate the total entropy of Split 1 as

$$D=-\left[\frac{20}{100}(0.6\log_2 0.6+0.4\log_2 0.4)+\frac{80}{100}(0.225\log_2 0.225+0.775\log_2 0.775)\right]=0.8095$$

and the total entropy of Split 2 as

$$D=-\left[\frac{10}{100}(0.8\log_2 0.8+0.2\log_2 0.2)+\frac{90}{100}\left(\frac{11}{45}\log_2\frac{11}{45}+\frac{34}{45}\log_2\frac{34}{45}\right)\right]=0.7943.$$

Again, Split 2 is preferred.

III. For Split 1, there are $8+18=26$ classification errors; for Split 2, there are $2+22=24$ errors. Split 2 is again preferred for having fewer errors.

All of the three statements are wrong! **(Answer: (E))** □

**Controlling tree complexity by pruning.** An oft-cited downside of decision trees is that they easily suffer from overfitting, meaning that they can be easily made too complex in an attempt to capture signals as well as noise in the training data (recall the discussions in Subsection 3.1.3). In the decision tree setting, the complexity (or size) of a tree is measured by the number of splits it has, or equivalently, the number of terminal nodes. With numerous terminal nodes, a decision tree will be difficult to interpret and may produce predictions with a high variance. One possible reason for the unstable predictions is that the final splits of a large tree are often based on very small number of observations, which are easily affected by noise. This calls for strategies in search of the right level of tree complexity that optimizes the bias-variance trade-off and maximizes predictive performance on independent test data.

One way to control the complexity of a tree is to pre-specify an impurity reduction threshold, starting from the root node and making a split only when the reduction in impurity exceeds this threshold. While this method makes intuitive sense, it overlooks the fact that a "not-so-good split" (one that results in a modest impurity reduction) early on in the tree-building process may be followed by a "good split" (one that results in an impressive impurity reduction), which we will never arrive at following this short-sighted method.

A better way to control the complexity of a tree is *pruning*, whose spirit resembles that of stepwise selection for GLMs. By pruning, we take an overblown tree and retroactively (i.e., starting from the bottom of the tree) remove its splits that do not meet a specified impurity reduction to bring the tree size to a desired level. By pruning back branches of a complex tree that offer little predictive power, we reduce the complexity of the fitted tree, prevent overfitting, and improve predictive accuracy.

**[IMPORTANT!] Pros and cons of decision trees.** As you can see, decision trees and GLMs are radically different predictive models. While GLMs return a compact prediction equation showing how the target mean is related to the linear predictor, with the coefficients indicating both the magnitude and the direction of the effect of the features on the target variable, decision trees produce a set of transparent, easily interpretable classification rules showing how the features are linked to the target variable. Compared to GLMs, decision trees have the following pros and cons: (the list below is not exhaustive!)

Merits.

- *Interpretability:* So long as there are not too many buckets, trees are easy to interpret and explain to non-technical audiences (recall that the client in a PA project is almost always someone unfamiliar with predictive analytics!) because of the if/then nature of the classification rules. For some link functions, the coefficients in a GLM may be difficult to interpret.
- *Nonlinear relationships:* They excel in handling nonlinear relationships and dispense with the need for variable transformations. In fact, any monotone transformations of numeric features will produce the same tree because the split points are based on ranking the feature values.
- *Interactions:* They are good at recognizing interactions between variables. Unlike GLMs, there is no need to identify potential interactions before fitting the tree. This, together with the preceding point, shows that feature generation is less an issue for decision trees.
- *Categorical variables:* Categorical predictors are automatically handled without the need for binarization or selecting a baseline level.

- *Variable selection:* Variables are automatically selected. Variables that do not appear in the tree are filtered out and the most important variables show up at the top of the tree.
- *Robustness:* Being non-parametric and distribution-free in nature (i.e., they do not assume a particular relationship between the target and the features or a particular distribution for the target), they are much less susceptible to model mis-specification issues than GLMs.
- *Missing data:* They can be easily modified to deal with missing data (not much on this point in the PA e-learning modules).

Demerits.

- *Overfitting:* They are more prone to overfitting and produce unstable predictions with a high variance. A small change in the training data can lead to a large change in the fitted tree and in the predictions. One easy way to see why decision trees tend to have large variance is that the splits are made recursively, conditional on the results of preceding splits, so a bad initial split can screw up the rest of the tree!
- *Categorical variables:* They tend to favor categorical features with many levels over those with few levels. For instance, a categorical predictor with four unordered levels $a, b, c, d$, can be split in seven ways: (1) $\{a\}, \{b, c, d\}$; (2) $\{b\}, \{a, c, d\}$; (3) $\{c\}, \{a, b, d\}$; (4) $\{d\}, \{a, b, c\}$; (5) $\{a, b\}, \{c, d\}$; (6) $\{a, c\}, \{b, d\}$; (7) $\{a, d\}, \{b, c\}$. Due to the enormous number of possible combinations, it is easier to choose a split for a multi-level categorical predictor so that the reduction in impurity is large than for a categorical predictor with fewer levels. Combining factor levels prior to fitting a decision tree can help remedy this.
- *Model diagnosis:* There is a lack of model diagnostic tools for decision trees.

**Example 5.1.3. (SOA Exam SRM Sample Question 29 (Adapted): Merits of decision trees)** Determine which of the following considerations may make decision trees preferable to other statistical learning methods.

I. Decision trees are easily interpretable.

II. Decision trees can be displayed graphically.

III. Decision trees are easier to explain to non-technical audiences than linear regression methods.

(A) None

(B) I and II only

(C) I and III only

(D) II and III only

(E) The correct answer is not given by (A), (B), (C), or (D).

*Solution.* All of the three statements are correct, as discussed above. **(Answer: (E))** □

### 5.1.2 Ensemble Trees: Random Forests

**Motivation behind ensemble methods.** Despite its transparency and easy interpretability, a decision tree, when used alone, is sensitive to noise and tends to overfit (even with pruning). With small perturbations in the training data, the fitted decision tree can be vastly different and the resulting predictions can be highly unstable. *Ensemble methods*, the subject of this subsection, allow us to hedge against overfitting and substantially improve predictive performance in many cases. These general-purpose methods produce *multiple* base models, instead of relying on a single model, and combine the results of these models to make predictions. This has the effect of enhancing the model's ability to capture complex relationships in the data (due to the use of multiple base models each working on different parts of the complex relationship) and to reduce the variability of the model's predictions (because the variance of an average is smaller than that of one component). Overall, the predictions are often far superior to those of a single base model in terms of bias and/or variance.

In Exam PA, we will study two ensemble tree methods (i.e., each base model is a decision tree): random forests and boosting. (You may have studied three methods in Exam SRM, but bagging is a special case of random forests.) Their workings may not be easy to understand on first encounter, so to ease our expositions, we will use a small amount of notation.

**Basic ideas.** *Random forests* entail generating multiple bootstrapped[iv] samples of the training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ (each $\mathbf{x}_i$ is the vector of predictor values associated with $y_i$) and fitting base trees in parallel independently on each of the bootstrapped training samples. By the term "bootstrapped," we mean that these samples are generated by sampling the training observations *with replacement* (i.e., a training observation may appear in a bootstrapped sample more than once). Specifically, let's say $B$ such bootstrapped samples are produced and denoted by

$$\{(\mathbf{x}_i^{*1}, y_{i^*}^{*1})\}_{i=1}^n, \qquad \{(\mathbf{x}_i^{*2}, y_i^{*2})\}_{i=1}^n \qquad , \quad \cdots \quad , \qquad \{(\mathbf{x}_i^{*B}, y_i^{*B})\}_{i=1}^n,$$

all having the same size $n$ as the original training set. Each $(\mathbf{x}_i^{*b}, y_i^{*b})$ for $i = 1, 2, \ldots, n$ and $b = 1, 2, \ldots, B$ is independently and randomly sampled from the original training observations $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$. We then train a base, *unpruned* decision tree on each of the $B$ bootstrapped samples separately (with a "trick"; see below), denoted by $\hat{f}^{*b}$ for $b = 1, 2, \ldots, B$. The results from these $B$ base trees are then combined to form an overall prediction.

- *Quantitative target variables:* If the target variable is quantitative, then the overall prediction is the average of the $B$ base predictions:

$$\boxed{\hat{f}_{\text{rf}}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(\mathbf{x}),}$$

  where $\mathbf{x}$ is the set of predictor values of interest.

- *Qualitative target variables:* If the target variable is qualitative, then $\hat{f}^{*b}(\mathbf{x})$ gives the class predicted by the $b$th bootstrapped tree at the predictor values of interest. To get a single predicted class label across all $B$ trees, we take a "majority vote" and pick the predicted class as the one that is the most commonly occurring among the $B$ predictions.

A schematic diagram of how random forests work is given in Figure 5.1.2. Because the resulting tree-based predictive model is made up of numerous trees constructed on random subsets of the data, it is aptly designated as a random forest (remember, a "forest" has many many trees!).

[iv] You may have learned bootstrap if you took Exam C, the predecessor of Exam STAM.

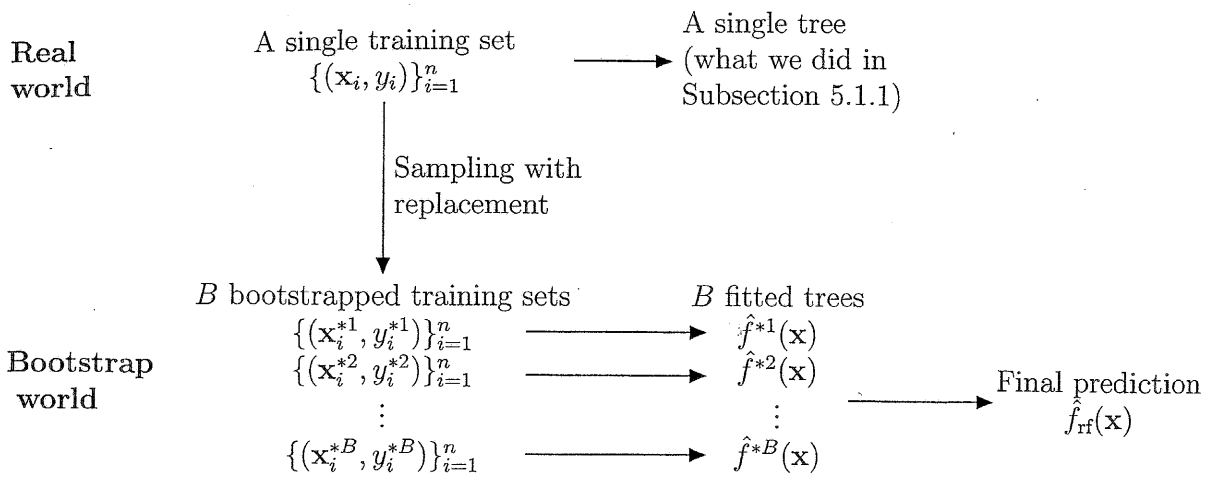



Figure 5.1.2: A schematic diagram of the mechanics of random forests.

**Randomization at each split.** A distinguishing characteristic of random forests is that when the $B$ base decision trees are grown:

> At each split a *random sample of $m$ predictors* is chosen as the split candidates out of the $p$ available features. Among these $m$ predictors, the one that contributes the greatest reduction in impurity is used to construct the split. A new random sample of $m$ predictors is made at every split, so a predictor that is sampled in one split can still be sampled and used in another split.

The typical choice of $m$[v] is $\sqrt{p}$ for a classification problem and $p/3$ for a regression problem, but it can be tuned by cross-validation as part of the model-building process.

At first sight, the idea of forcing each split to consider only a small subset of the full set of predictors may seem weird, but there is a sound rationale behind this move. If there is a very strong predictor among the $p$ predictors, then most of the base trees will make use of this strong predictor in the first split and, as a result, become highly correlated with one another. Random forests *decorrelate* the base trees by making them use more diverse features for different splits and therefore lead to greater variance reduction when the base predictions are averaged (or when the majority vote is taken).

**Pros and cons of random forests (relative to a single tree).**

Merits. Random forests, as an ensemble method, are usually much more robust than a single decision tree. Although the $B$ base trees are unpruned and therefore have low bias and high variance, averaging the results of these trees contributes substantial variance reduction, especially when the number of trees $B$ is large (however, bias is generally not reduced by random forests), and produces much more precise predictions.

Demerits. On the downside, random forests are inferior to base trees in two respects:

1. *Interpretability:* Random forests, which consist of potentially hundreds or thousands of decision trees, cannot be visualized in a tree-like diagram and are much

[v]Bagging is a special case of random forests with all of the $p$ predictors considered in each split, i.e., $m = p$.

less interpretable than a single decision tree. It is difficult to see how the predictions depend on each feature and the whole process may sound like a "black box" to laymen (e.g., PA project clients). In brief, random forests improve prediction accuracy at the expense of interpretability.

2. *Computational power:* It takes a considerably longer time to implement a random forest compared to a base decision tree due to the computational burden that comes with fitting multiple base trees. (This is exactly the reason why the exam will probably not test the construction of a random forest!)

### 5.1.3 Ensemble Trees: Boosting

**Idea.** *Boosting* adopts a different approach to ensemble learning than random forests and works on the principle of sequential learning. Instead of making independent bootstrapped samples and averaging the predictions based on different bootstrapped trees, boosting builds a sequence of interdependent trees using information from previously grown trees. In each iteration of the algorithm, we fit a tree to the *residuals* of the preceding tree (using the same set of predictors) and a *scaled-down* version of the current tree's predictions is subtracted from the preceding tree's residuals to form the new residuals. The whole process is repeated, with the effect being that each tree will focus on predicting observations that the previous tree predicted poorly. This highlights one key difference between boosting and random forests:

> While random forests address model *variance*, boosting focuses on model *bias*, i.e., the ability of the model to capture signal in the data.

The overall prediction of the boosted tree is the sum of the scaled-down prediction of each model. The use of scale-down predictions is meant to slow down the learning process. It is generally true in statistical learning that methods that learn slowly tend to do well.

**Pros and cons of boosting (relative to base trees and random forests).** Here we are not going to present mathematical formulas for boosting since they add little value to the discussions (if you are interested, please read Subsection 8.2.3 of ISLR for the precise algorithm). We will content ourselves with a qualitative analysis of the properties of boosting.

- *Boosted trees vs. random forests:* By design, boosted models often perform better in terms of prediction accuracy than random forests due to their emphasis on bias reduction, but they are also more vulnerable to overfitting. In common with all ensemble models, boosting requires that model fitting be performed many times, which entails significant computational cost.
- *Boosted trees vs. base trees:* Just like random forests, the gain in prediction accuracy as a result of the use of boosting comes with the loss of interpretability and computational efficiency.

## 5.2 Mini-Case Study: A Toy Decision Tree

In this preparatory section we construct a toy decision tree on a small-scale dataset taken from a sample question[vi] of the Modern Actuarial Statistics II Exam of the Casualty Actuarial Society

[vi] See Question 12 in `https://www.casact.org/admissions/syllabus/MASII_Sample_Questions.pdf`.

and displayed in Table 5.1. The small number of observations makes it possible for us to perform calculations by hand and replicate the R output that is inadequately explained in the PA e-learning modules and commonly misunderstood by many users (partly due to the rather confusing names adopted by the package we will use).

| $X_1$ | $X_2$ | $Y$ |
|---|---|---|
| 1 | 0 | 1.2 |
| 2 | 1 | 2.1 |
| 3 | 2 | 1.5 |
| 4 | 1 | 3.0 |
| 2 | 2 | 2.0 |
| 1 | 1 | 1.6 |

Table 5.1: The toy dataset used in Section 5.2.

After completing this mini-case study, you should be able to:

- Fit a regression tree using the `rpart()` function from the `rpart` package.
- Understand how the `control` parameters of the `rpart()` function control tree complexity.
- Produce a graphical representation of a fitted decision tree using the `rpart.plot()` function.
- Interpret the output for a decision tree fitted by `rpart()`.
- Prune a regression tree using the `prune()` function.

To begin with, run CHUNK 1 to set up the dataset `dat` carrying the numeric target variable $Y$ and the two numeric predictors $X_1$ and $X_2$.

```
# CHUNK 1
X1 <- c(1, 2, 3, 4, 2, 1)
X2 <- c(0, 1, 2, 1, 2, 1)
Y <- c(1.2, 2.1, 1.5, 3.0, 2.0, 1.6)
dat <- data.frame(X1, X2, Y)
```

**Main function: `rpart()` and its control parameters.** In R, there are several packages for fitting decision trees, each with its implementation subtleties. Although the underlying concepts are essentially the same, different packages may use the same term to mean different quantities. The package we will use for Exam PA[vii] is the `rpart` (short for "recursive partitioning") package, where the main function is the function of the same name, `rpart()`. Just like the `lm()` and `glm()` functions, the `rpart()` function takes a `formula` argument specifying the target variable and the predictors and a `data` argument specifying the data frame hosting all of the necessary variables. The `method` argument determines whether a regression tree (with `method = "anova"`) or a classification tree (with `method = "class"`) is to be grown. If this argument is left unspecified,

[vii] ISLR uses the `tree` package for fitting decision trees (see Section 8.3 there). We will not use this package in Exam PA.

then R will examine the nature of the target variable and make an intelligent guess on whether a regression or classification tree should be built, which may not be desirable, especially when the target variable is a categorical variable coded by numeric labels. Perhaps the most important argument is the `control` argument, which uses the `rpart.control()` function to specify a list of parameters "controlling" when the partition stops, or equivalently, the complexity of the tree to be built. Here are the most commonly used parameters:

- `minsplit`: This is the minimum number of observations that must exist in a node in order for a split to be attempted. For example, if `minsplit` is set to 10, then a node with fewer than 10 observations will not be considered for further splitting. Everything else equal, the lower the value of `minsplit` (the lowest value is 1), the larger and more complex the fitted tree.

- `minbucket`: Short for "minimum bucket size," this is the minimum number of observations in any terminal node ("bucket"). If a split generates a node that has fewer than this number of observations, then the split will not be made. Same as `minsplit`, the lower the value of `minbucket` (the lowest value is 1), the larger and more complex the fitted tree.

  Comparing `minsplit` and `minbucket`, we can see that the former refers to the minimum number of observations *before* splitting and the latter refers to the the minimum number of observations *after* splitting. In most situations, either `minsplit` or `minbucket` needs to be specified as they function similarly to limit decision tree complexity. According to the documentation of `rpart.control()`, when only one of `minbucket` or `minsplit` is specified, either `minsplit` is set to `minbucket*3` or `minbucket` is set to `minsplit/3`.

- `cp`:[viii] Short for "complexity parameter," this is a value between 0 and 1 (inclusive) specifying the minimum amount of impurity reduction required for a split to be made. If a split does not bring sufficient improvement in the model fit in the sense that *strictly* decreases the impurity metric for a tree (which is the overall $R^2$ for a numeric target and Gini for a categorical target) by `cp`, then this split will not be performed. The default value of `cp` is 0.01. The higher the value of `cp`, the fewer the number of splits to be made and the *less* complex the fitted tree. In the two extreme cases, setting `cp` to 0 leads to the most complex tree (subject to the constraints on other parameters such as `minbucket` and `maxdepth`) and setting `cp` to 1 prohibits any splits.

  Later we will prune a fitted decision tree using the results returned by the `rpart()` function to reduce its complexity. You may then wonder: Why do we have to pre-specify a complexity parameter? Why do we not grow a very complex tree and prune later? The role played by the `cp` parameter is to "pre-prune" splits that are obviously not worthwhile to pursue (which will likely be pruned off by cross-validation when conducted) and thus to save computational effort in the case of large datasets. We should be careful not to prescribe an inappropriately large `cp` to avoid "over-pruning." A reasonably small value such as `cp = 0.001` will do this screening well.

- `maxdepth`: This is the maximum depth of the tree, or the maximum number of branches from the tree's root node to the furthest terminal node. The higher the value of `maxdepth`, the more complex the fitted tree. The default value of `maxdepth` is 30.

[viii] An interesting thread on Actuarial Outpost about how cross-validation is done in `rpart()` and what the `cp` parameter really means can be found here.

Very often, the control parameters above are hyperparameters that are tuned as part of the tree construction process in search of the right level of tree complexity.

The two parameters below are not related to tree complexity, but affect how tree splits are evaluated.

- `xval`: This is the number of folds used when doing cross-validation. The default value is 10, i.e., 10-fold cross-validation is performed. Although this parameter has no effect on the complexity of the fitted tree, it affects the performance, assessed by cross-validation, of the decision trees automatically fitted when the `rpart()` function is called.
- `parms`: This argument is limited to categorical target variables and describes the parameters that guide how the splits are performed, using Gini or information gain as the node impurity measure.

**Fitting and visualizing a decision tree.** In CHUNK 2, we first load the `rpart` and `rpart.plot` (for plotting `rpart` objects) packages and fit a regression tree for $Y$ using $X_1$ and $X_2$ as predictors and save it as an `rpart` object named `dt` (meaning a "decision tree"). Because the sample size is so small and this case study is for illustration purposes, we have deliberately set the control parameters such that the most complex tree will be constructed. With `minsplit` and `minbucket` both set to 1 and a zero `cp`, each node only has to contain at least one observation and a split will be made as long as it results in some improvement in node impurity. Finally, we pass `dt` to the `rpart.plot()` function to give a graphical representation of the regression tree just grown.

In Figure 5.2.1, the top number in each node of the tree indicates the fitted target value and the bottom number represents the proportion of observations lying in that node. For instance, at the top of the tree the root node has all of the $n = 6$ observations (hence 100% of the observations) and a fitted value of 1.9, which is simply the target mean of the 6 observations:

$$\bar{y} = \frac{1.2 + 2.1 + 1.5 + 3.0 + 2.0 + 1.6}{6} = 1.9.$$

The first split uses $X_1 < 4$ as the criterion,[ix] meaning that out of the two predictors $X_1$ and $X_2$ and among all possible cutoff points, partitioning the data using $X_1$ and using 4 as the cutoff at this stage leads to the greatest reduction in tree impurity measured by RSS. Observations satisfying $X_1 < 4$ (i.e., all except observation 4) are classified to the left branch while the one that does not (i.e., observation 4) is assigned to the right branch. The partitions continue with the left branch, which still contains five observations, until no more node can be split to reduce the overall RSS of the model. In this simple case, each of the six observations occupies a separate terminal node with a fitted value equal to the observed value and the RSS being exactly zero. We have a perfect fit, which will be the case if the control parameters impose no restrictions on the complexity of the fitted tree.[x]

**Using a decision tree to detect interaction.** From Figure 5.2.1, it appears that there is some interaction between $X_1$ and $X_2$. Recall from earlier chapters that an interaction exists if the relationship between one predictor and the target variable depends on the value of another predictor.

[ix] The convention of `rpart.plot()` is that observations satisfying the splitting criterion are sent to the left branch and those that do not to the right branch.

[x] Strictly speaking, this also requires that target observations with the same set of predictor values, if any, be identical in value. If we change the third set of predictor values from $(3, 2)$ to $(2, 2)$, then $y_3 \neq y_5$ even though they have the same set of predictor values, and we have a close to perfect fit, but not an exactly perfect fit.

```
# CHUNK 2
# Uncomment the next two lines the first time you use these two packages
# install.packages("rpart")
# install.packages("rpart.plot")
library(rpart)
library(rpart.plot)
dt <- rpart(Y ~ ., data = dat, control = rpart.control(minsplit = 1,
                                                       minbucket = 1,
                                                       cp = 0,
                                                       xval = 6))

rpart.plot(dt)
```

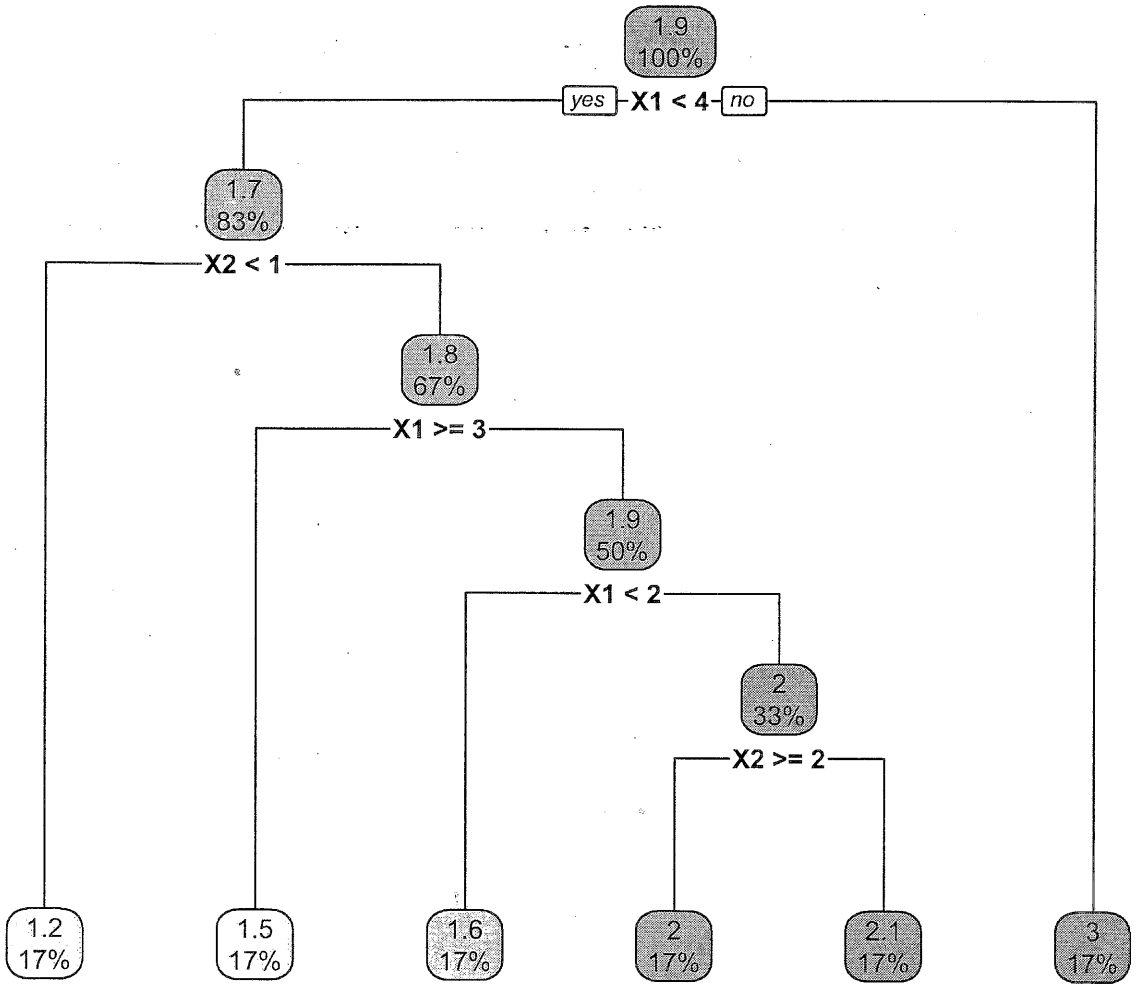



Figure 5.2.1: The fully grown regression tree for $Y$ with $X_1$ and $X_2$ as predictors.

In the case of dt, $X_2$ has a significant effect on $Y$ only when $X_1 < 4$. The left branch of the root node is followed by a split using $X_2$ while the same split cannot be found in the right branch. If $X_1 \geq 4$, then the fitted tree says that a further split by $X_2$ is no longer useful (of course, we know that the right branch contains only one observation and so further splitting is impossible, but the considerations here apply to larger datasets). In contrast, if there were no interactions between $X_1$ and $X_2$, then we would expect the same set of splits that appear in the left branch to be used in the right branch as well. In fact, if we fit an ordinary least squares linear model regressing $Y$ on $X_1$ and $X_2$ with interaction, then the model summary confirms that the interaction is statistically significant.

```
# CHUNK 3
ols <- lm(Y ~ X1 * X2, data = dat)
summary(ols)

##
## Call:
## lm(formula = Y ~ X1 * X2, data = dat)
##
## Residuals:
##        1        2        3        4        5        6
##  0.053226 -0.024194 -0.053226  0.008065  0.106452 -0.090323
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.06129    0.18810  -0.326  0.77547
## X1           1.20806    0.10892  11.092  0.00803 **
## X2           1.31774    0.20351   6.475  0.02303 *
## X1:X2       -0.77419    0.09995  -7.746  0.01626 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1136 on 2 degrees of freedom
## Multiple R-squared:  0.9871,Adjusted R-squared:  0.9677
## F-statistic:    51 on 3 and 2 DF,  p-value: 0.01929
```

**EXAM NOTE**

The December 2018 PA exam expects candidates to recognize the existence of interaction between predictor variables from a fitted decision tree, then incorporate the interaction term into a GLM, although there is hardly any mention of interaction in the Rmd sections of Module 7 of the PA e-learning modules.

**A closer look at the tree splits.** If we type the name of an rpart object, we will get a condensed printout of how all of the tree splits are performed. This is done in CHUNK 4 for dt. Although the output is not as aesthetically appealing as the displayed tree in Figure 5.2.1, we can get more detailed information about different splits.

```
# CHUNK 4
dt

## n= 6
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 6 2.000 1.90
##    2) X1< 3.5 5 0.548 1.68
##      4) X2< 0.5 1 0.000 1.20 *
##      5) X2>=0.5 4 0.260 1.80
##       10) X1>=2.5 1 0.000 1.50 *
##       11) X1< 2.5 3 0.140 1.90
##         22) X1< 1.5 1 0.000 1.60 *
##         23) X1>=1.5 2 0.005 2.05
##           46) X2>=1.5 1 0.000 2.00 *
##           47) X2< 1.5 1 0.000 2.10 *
##    3) X1>=3.5 1 0.000 3.00 *
```

For each node, we can see the splitting criterion (the `split` column), the number of observations (the `n` column), the residual sum of squares (the `deviance` column; recall that we are dealing with a numeric target variable), and the fitted target value (the `yval` column). The tree starts with the root node, numbered the first node, where all the 6 observations are hosted, with a residual sum of squares of TSS $= \sum_{i=1}^{6}(y_i - \bar{y})^2 = 2$ (which is also the total sum of squares in the terminology of linear models) and a fitted target value of 1.9, which is the sample target mean $\bar{y} = \sum_{i=1}^{6} y_i/6$. The root node is then split into child nodes 2 and 3, with observations satisfying $X_1 < 3.5$ sent to the left branch, where the predicted target value is $1.68\,(= (1.2 + 2.1 + 1.5 + 2.0 + 1.6)/5)$, and those satisfying $X_1 \geq 3.5$ (in fact, only observation 4) sent to the right branch, where the predicted target value is 3. To facilitate identification and inspection, child nodes in a decision tree are indented and numbered in this format: The two child nodes of node $x$ are numbered $2x$ and $2x + 1$, so, for example, the two child nodes of node 2 are node $4\,(= 2 \times 2)$ and node $5\,(= 2 \times 2 + 1)$. When the splitting stops, we have reached a terminal node, which is signified by an asterisk (`*`) at the end of the line.

**Pruning a decision tree.** A powerful feature of the `rpart()` function is that in addition to fitting the decision tree with the indicated complexity parameter, it automatically fits a collection of decision trees corresponding to every value of the complexity parameter *greater than* the value of `cp` used (hence simpler than the fitted tree) and evaluates their predictive performance using internal cross-validation. This allows us to evaluate the model at a wide range of complexity and choose the best one. To access these results, we take into account the fact that an `rpart` object such as `dt` is a list (that is why we had to learn lists back in Subsection 1.2.4!) and look at its `cptable` component in CHUNK 5.

```
# CHUNK 5
dt$cptable

##         CP nsplit rel error   xerror     xstd
```

```
##  1 0.72600      0    1.0000 1.440000 0.745794
##  2 0.14400      1    0.2740 2.399012 1.059821
##  3 0.06375      2    0.1300 2.220556 1.004330
##  4 0.00250      4    0.0025 2.420000 1.252211
##  5 0.00000      5    0.0000 2.420000 1.252211
```

The `cptable` is a matrix (recall that matrices are discussed in Subsection 1.2.2) that shows, for each cutoff value of the complexity parameter (the `CP` column), the number of splits of the tree constructed (the `nsplit` column), relative error (the `rel error` column), cross-validation error (the `xerror` column), and standard error of the cross-validation error (the `xstd` column). The number of splits is a measure of tree complexity; the total number of terminal nodes is the number of splits plus one. As we expect, the lower the value of the complexity parameter, the larger the number of splits. The last three columns of the `cptable` are not adequately explained in the PA e-learning modules and are described below:

- `rel error`: For numeric target variables, this is $1 - R^2 = \text{RSS}/\text{TSS}$, which is a *scaled* version of the training error (but not exactly the training error!). The fact that the training error is relative to the total sum of squares explains why it is termed the "relative" error.

  For instance, the root node has a relative error of $\text{TSS}/\text{TSS} = 1$, which is the first value in the `rel error` column. The first split, when made, assigns all observations except observation 4 to the left branch (node 2) and observation 4 to the right branch (node 3):

| Node 2 ($X_1 < 3.5$) | | | Node 3 ($X_1 \geq 3.5$) | | |
|---|---|---|---|---|---|
| Obs. # | Observed Value | Fitted Value | Obs. # | Observed Value | Fitted Value |
| 1 | 1.2 | | | | |
| 2 | 2.1 | | | | |
| 3 | 1.5 | 1.68 | 4 | 3.0 | 3.0 |
| 5 | 2.0 | | | | |
| 6 | 1.6 | | | | |
| RSS = 0.548 | | | RSS = 0 | | |

  With a fitted target value of 1.68, the RSS in the left branch is

$$\sum_{i \neq 4} (y_i - 1.68)^2 = (1.2 - 1.68)^2 + (2.1 - 1.68)^2 + \cdots + (1.6 - 1.68)^2 = 0.548.$$

  The RSS in the right branch, with only observation 4 there, is $(3.0 - 3.0)^2 = 0$, so the overall relative error is $(0.548 + 0)/2 = 0.2740$, which is the second value in the `rel error` column. The first split therefore improves the relative error by $1 - 0.2740 = 0.72600$, which is the first value in the `CP` column. We can say that for any complexity parameter `cp` $\in [0.72600, 1]$ (note that both boundaries of the interval are closed), the first split will not be made[xi] as the reduction in the relative error fails to *strictly* exceed the `cp` threshold and the fitted tree consists only of the root node containing all of the 6 observations.

[xi] Note that when `cp = 0.72600`, the first split still will not be made because it fails to improve the relative error *strictly* by `cp`.

**Example 5.2.1. (To verify the second cp threshold by hand)** Consider CHUNK 5 again. Explain how the second value in the CP column, 0.14400, is obtained. For what range of values of the complexity parameter will the fitted tree have one split or two splits?

*Solution.* We can adapt the analysis above to node 2, which results from the root node as a result of the criterion $X_1 < 3.5$. The next split will produce Node 4 ($X_2 < 0.5$) and Node 5 ($X_2 \geq 0.5$), which contain observation 1 and observations 2, 3, 5, and 6, respectively. The following table shows how the RSS within each node is computed:

| Node 4 ($X_1 < 3.5, X_2 < 0.5$) | | | Node 5 ($X_1 < 3.5, X_2 \geq 0.5$) | | |
|---|---|---|---|---|---|
| Obs. # | Observed Value | Fitted Value | Obs. # | Observed Value | Fitted Value |
| 1 | 1.2 | 1.2 | 2<br>3<br>5<br>6 | 2.1<br>1.5<br>2.0<br>1.6 | 1.8 |
| RSS = 0 | | | RSS = 0.26 | | |

The sum of the two RSSs is 0.26, which is lower than the RSS of Node 2 by $0.548 - 0.26 = 0.288$. The split using $X_2 < 0.5$ as the criterion thus decreases the relative error of the tree by $0.288/2 = 0.144$, which is the second value in the CP column. We can say that:

- If the complexity parameter satisfies cp $\in [0.144, 0.726)$, then only the first split using $X_1 < 3.5$ will be made, because the second split fails to decrease the relative error strictly by cp.
- If the complexity parameter satisfies cp $\in [0.06375, 0.144)$ (where 0.06375 will be verified below), then both the first (using $X_1 < 3.5$) and second splits (using $X_2 < 0.5$) meet the threshold requirement and will be made.

□

If you repeat the analysis above and investigate the remaining splits, you will see that the third ($X_1 \geq 2.5$), fourth ($X_1 < 1.5$), and fifth splits ($X_2 \geq 1.5$) decrease the RSS of the tree by 0.12, 0.135, and 0.005, and so decrease the relative error by 0.06, 0.0675, and 0.0025, respectively. Because the fourth split contributes a greater drop in the relative error, this is a clear demonstration that an even better split can occur after the split which is currently the best. It seems that `rpart()` is able to detect this and takes the average of 0.06 and 0.0675, which is 0.06375, as the next cutoff for the complexity parameter. This explains why the number of splits in the `nsplit` column jumps from 2 to 4 without going through 3 splits. If we a priori restrict the `maxdepth` parameter to 3, then the most the tree can get is the third split. In this case, the third cutoff for the complexity parameter is indeed 0.060, as shown in CHUNK 6, and the resulting tree is exhibited in Figure 5.2.2.

```
# CHUNK 6
dt.new <- rpart(Y ~ ., data = dat,
                control = rpart.control(minsplit = 1,
                                        minbucket = 1,
```

```
                                    maxdepth = 3,
                                    cp = 0,
                                    xval = 6))
dt.new$cptable

##       CP nsplit rel error   xerror     xstd
## 1 0.726      0     1.000 1.440000 0.745794
## 2 0.144      1     0.274 2.399012 1.059821
## 3 0.060      2     0.130 2.220556 1.004330
## 4 0.000      3     0.070 2.466250 1.237930

rpart.plot(dt.new)
```

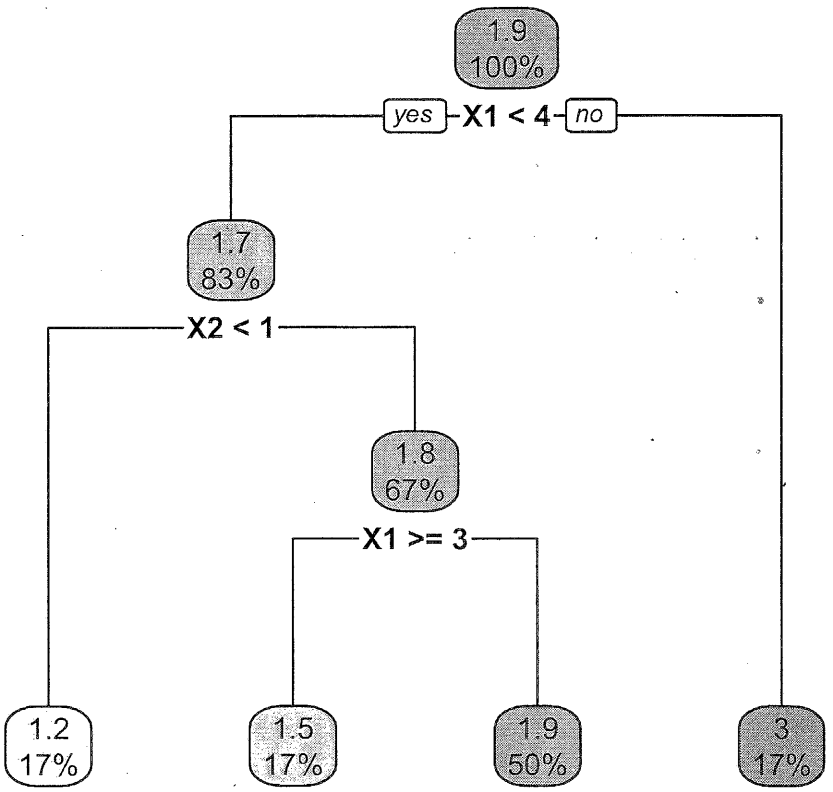



Figure 5.2.2: The fully grown 3-split regression tree for $Y$ with $X_1$ and $X_2$ as predictors.

Overall, we can see that as the `cp` parameter decreases, so does the relative error, consistent with the fact that training error decreases with model complexity. In the limit when `cp = 0`, five splits are performed and each of the 6 observations belongs to a separate terminal node, resulting in a zero relative error.

- `xerror`: As pointed out above, `rpart()` performs internal cross-validation to calculate the cross-validation error of all decision trees fitted by a complexity parameter greater than or

equal to the specified `cp` value. By default, 10-fold cross validation is conducted, though the number of folds used can be changed by the `xval` parameter. Because there are only six observations in our sample, we have set the `xval` parameter to the sample size (6) to eliminate the training/validation split variability. Each of the six observations is left out in turn, a decision tree is fitted to the remaining five observations, and a predicted value is generated for the held-out observation. The six prediction errors are squared, summed, and divided by the total sum of squares (not the sample size!) to get `xerror`. The division by the total sum of squares parallels the way the relative error is calculated and makes `xerror` different from the usual cross-validation error.

For instance, the following table shows how the `xerror` for the tree with only the root node (i.e., `nsplit = 0`) is computed:

| Held-Out Obs. | Value of Held-Out Obs. | Predicted Value ( = Average of Other Five $y_i$'s) | Squared Prediction Error |
|---|---|---|---|
| 1 | 1.2 | 2.04 | 0.7056 |
| 2 | 2.1 | 1.86 | 0.0576 |
| 3 | 1.5 | 1.98 | 0.2304 |
| 4 | 3.0 | 1.68 | 1.7424 |
| 5 | 2.0 | 1.88 | 0.0144 |
| 6 | 1.6 | 1.96 | 0.1296 |

Then the `xerror` is

$$\frac{0.7056 + 0.0576 + 0.2304 + 1.7424 + 0.0144 + 0.1296}{2} = 1.44,$$

which is the first value in the `xerror` column.

- `xstd`: Besides the scaled cross-validation error, `cptable` also has the standard error of the cross-validation error for each fitted decision tree. Very often, only the standard error associated with the smallest cross-validation error is of interest, so we will not devote too much attention to `xstd`.

While the relative error decreases as we lower the complexity parameter (thus the tree becomes more complex), the (scaled) cross-validation error follows a different pattern which is somewhat irregular due to the small sample size. We can see that although the five-split tree has a zero relative error, it has the highest cross-validation error, showing that it is seriously overfitted. To ensure that the decision tree has the right level of complexity, we prune it by setting the complexity parameter to an appropriate value rather than 0. One natural choice is the value that gives rise to the smallest cross-validation error. Another commonly used choice will be introduced in Section 5.3. In CHUNK 7, we first extract the optimal complexity parameter and use the `prune()` function to prune the tree to the corresponding level of complexity.

To understand how the optimal complexity parameter is extracted, first realize that `dt$cptable` is a matrix, whose elements can be extracted by the square bracket (`[, ]`) operator (recall what we learned in Subsection 1.2.2). The command `dt$cptable[, "xerror"]` extracts the column of cross-validation errors and `which.min(dt$cptable[, "xerror"])` returns the row corresponding to the minimum cross-validation error. Finally, `dt$cptable[which.min(dt$cptable[, "xerror"]), "CP"]` uses this row to select along the column of complexity parameters and produces the optimal `cp`. This optimal value is then fed into the `cp` argument of the `prune()` function to prune back