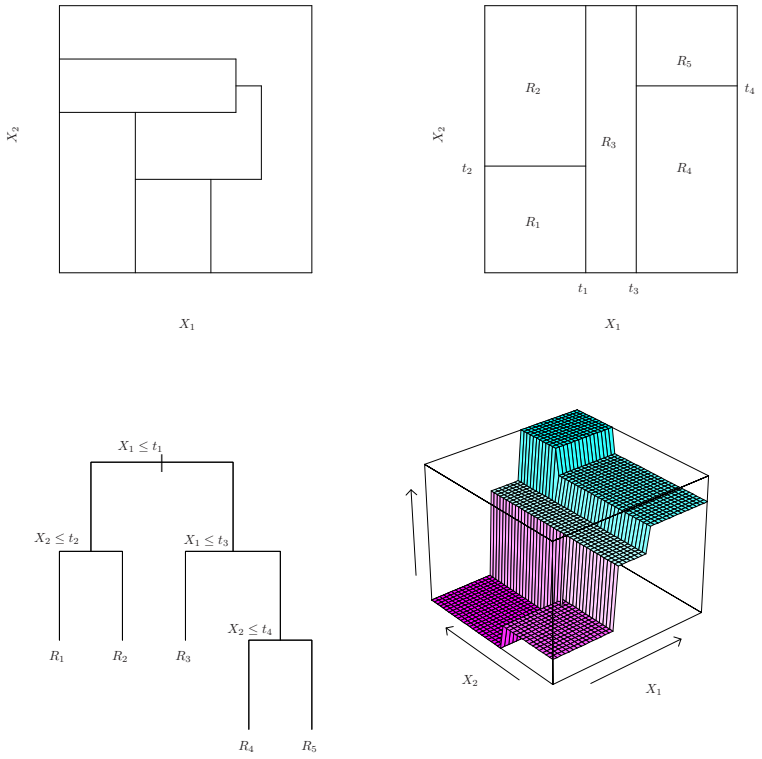
8 Trees

Tree models are very intuitive. But they are often not useful by themselves. They are very good building blocks for random forests and boosting. Tree models for classification are called *classification trees*, and those for quantitative outcomes are called *regression trees*. Thus trees are often referred to as *classification and regression trees* (CART).

8.1 The process of tree building

A tree has a root, a few branches, terminal nodes (leaves), and internal nodes. See Figures 8.1–8.3 for illustrations.



To grow a tree, we use a *recursive binary splitting* algorithm (a greedy algorithm):

- To split a node:
  - Search all features and all possible splits based on a single feature x.
    - If x is quantitative or ordinal, splits are defined by $x < t$ and $x \geq t$ for all possible t;
    - If x is categorical, splits are all possible ways of splitting the categories into 2 groups.
  - Identify the split that yields the most reduction in "impurity" in y.
    - Impurity can be quantified in various ways (see below).
    - A measure of impurity is also called a *splitting index*.
    - In classification, maximum reduction of impurity does not guarantee that the two resulting subsets would have different predictions.
- Stop when every terminal node meets the stopping criteria (see below).
- Every terminal node has a constant prediction value.
  - Continuous y: average y over the observations falling in the node.
  - Categorical y: the majority class for the observations falling in the node.
- Tree pruning (see below) is often necessary if a tree model is the final result of analysis.

We need to specify a measure of impurity and the stopping criteria (see below).

The R `rpart` package is often used for building tree models. It is faster and has more features than the `tree` package. To fit a tree, use `rpart(formula, data=, method=, control=)`. Use `method='class'` for classification trees and `method='anova'` for regression trees.

For example, we now build a tree model to predict `health`, a binary variable, using the `Wage` dataset. First we perform some basic checks on the data. Make sure ordinal variables are coded appropriately. For example, `education` has a natural order in its categories and thus should be reflected in its coding.

In R, a factor is NOT ordered. Internally, scores are assigned to the levels of a factor for the purpose of storage efficiency. Because of this, the levels appear to have an order, but analyses do not treat them as ordered. Thus it is not enough to code an ordinal variable as a factor even if its levels appear to have the correct order. One needs to explicitly convert it to an **ordered factor**.

```
library(ISLR)
names(Wage); dim(Wage); str(Wage)
apply(is.na(Wage), 2, sum)  ## check if there are missing data
table(Wage$health, useNA='always')  ## outcome distribution

levels(Wage$education)  ## check the order of the levels
Wage$education = factor(Wage$education, levels(Wage$education), ordered=T)  ## make it ordered
```

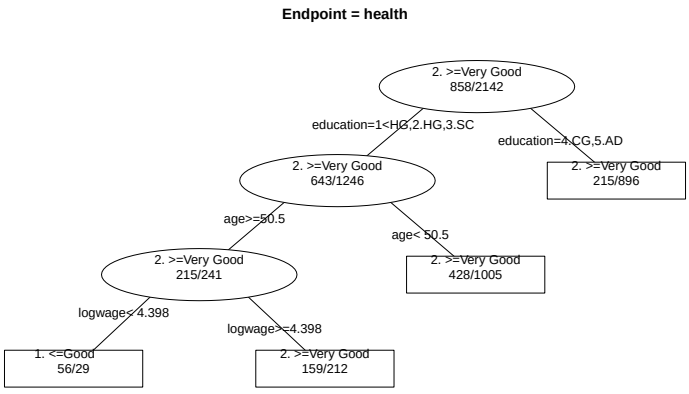
We now build a classification tree, relying on the default control parameters for now.

```
library(rpart)
tree1 = rpart(health ~ ., data=Wage, method='class')
tree1
with(Wage, table(education, health, useNA='always'))
```

To plot the results, we can use any of the following:

```
plot(tree1); text(tree1)
plot(tree1, uniform=T); text(tree1, all=T, use.n=T)
post(tree1, file='')
```

Note that in the function `post()`, the `file=''` argument forces the function to output the result to the screen; otherwise, a .ps file will be generated.



A split may not necessarily improve classification. The first two splits in the tree above do not improve classification at all! By definition, a split always reduces total "impurity" in the terminal nodes, but a reduction in an impurity measure does not guarantee a reduction in misclassification.

Further details of the tree model can be extracted:

```
names(tree1)
tree1$frame           ## the model stored in a data frame
tree1$frame$yval      ## the prediction for all nodes
tree1$frame$yval2     ## a little more details for the nodes
```

The fitted values can be obtained through `predict()`.

```
head(predict(tree1))  ## fitted probabilities; only show the first few using head()
head(predict(tree1, type="vector"))  ## fitted category as numerical levels
head(predict(tree1, type="class"))   ## fitted category in original outcome values

table(Wage$health, predict(tree1, type="class"))  ## confusion matrix
```

If you want to explore further, `tree1$where` contains the information on the leaves all the observations fall into. But the numbering of the nodes can be confusing. The model printout shows nodes as 1,2,4,8,9,5,3; these node numbers are also row names of `tree1$frame` (check `names(tree1$frame)`). But the values in `tree1$where` are 4,5,6,7; they are the row numbers in `tree1$frame`, not node numbers. For example, `where=7` means the 7th row in `tree1$where` (7th node in the printout), which is labelled as node number 3 in the printout.

```
table(tree1$where)  ## distribution of leaf size
all.equal(tree1$frame$yval[tree1$where], predict(tree1, type="vector"),
          check.attributes=F)  ## TRUE
```

Regression trees for continuous outcomes: RSS is often used as a measure of impurity. When splitting a node into two subsets, the total RSS for the node changes from

$$T = \sum_{j \text{ in parent node}} (y_j - \bar{y})^2 \quad \text{to} \quad W = \sum_{j \text{ in node 1}} (y_j - \bar{y}_1)^2 + \sum_{j \text{ in node 2}} (y_j - \bar{y}_2)^2,$$

where $\bar{y}$ is the average outcome for the parent node; $\bar{y}_1$, $\bar{y}_2$ are the average outcomes for the two child nodes. We identify the split with the largest reduction in impurity

$$B = T - W = [n_1(\bar{y}_1 - \bar{y})^2 + n_2(\bar{y}_2 - \bar{y})^2],$$

where n_1 are n_2 are the sizes of the two subsets. In other words, we identify the split that gives the largest between-subset variance.

Classification trees for categorical outcomes: Measures of impurity for a multinomial distribution $\{p_1, p_2, \cdots, p_K\}$:

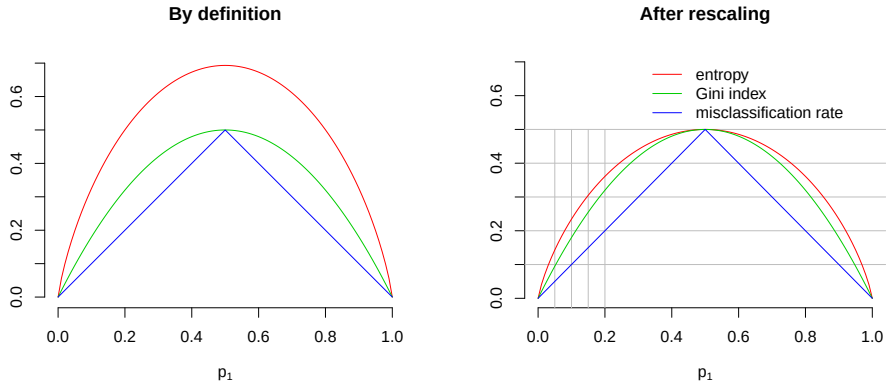
(1) entropy/information, $D = -\sum_k p_k \log(p_k)$;
(2) Gini index, $G = \sum_k p_k(1 - p_k) = 1 - \sum_k p_k^2$;
(3) misclassification rate, $E = 1 - \max(p_k)$.

For example, when a node with size n and Gini index G is split into two subsets with sizes n_1 and n_2 and Gini indices G_1 and G_2, the reduction is

$$nG - n_1G_1 - n_2G_2.$$

We identify the split that has the largest reduction.

Measures (1) and (2) are often used, as they are harder to push to zero than (3) relative to their maximum, as shown below for binary outcomes. The same is true for multinomial distributions.



Note that the best split according to one measure of impurity may not be the best split for another measure of impurity. Different choices of impurity measure can lead to different results.

Stopping rules are used to avoid unnecessary computation. Examples are:

- A minimum size for terminal nodes. Stop splitting a node if its size is smaller than the minimum size.
- A minimum fraction for improvement. Stop splitting a node if the improvement is below the minimum fraction.
- A maximum depth. Stop splitting a node if it is already at the maximum depth level. The root is at depth 0.

Control parameters: For the tree model above, its `control` element contains the control values we used. Check the help page for `rpart.control()` to learn what they mean. (In the `tree` package, `tree.control()`.)

```
tree1$control
```

These are the default values:

- minsplit = 20, minbucket = 7, maxdepth = 30, cp = 0.01 (stopping criteria)
- usesurrogate = 2, surrogatestyle = 0 (surrogate options)
- xval = 10 (number of folds in cross-validation)
- maxcompete = 4, maxsurrogate = 5 (output criteria)

```
tree1$parms
```

For classification trees, some additional parameters are stored in the `parms` element. The parameters and their default values are:

- `prior`: prior probabilities for the classes (default: proportional to the counts in data, which is effectively no prior);
- `loss`: loss matrix for evaluation of performance (default: a matrix for 0–1 loss);
- `split`: impurity measure (default: 'gini'; alternative choice is 'information').

Below is an example of setting some of these control parameters.

```
tree2 = rpart(health ~ ., data=Wage, method='class',
              control = rpart.control(minsplit=10, cp=0.005),
              parms = list(split='information'))
```

8.2 Properties of tree models

- A tree is a multi-dimensional piecewise constant model on a partition of the feature space.
  - The terminal nodes form a partition of the whole feature space. Every node is a multi-dimensional cube.
  - DF of a tree = #terminal nodes.
- Trees are inefficient with respect to the number of parameters (i.e., the DF).
- Trees are built in a greedy way. There is no guarantee that the partition is optimal.
- Trees are easy to explain, but not necessarily "easy to interpret". When the tree depth increases, it becomes difficult to interpret the nodes.
- Trees are scale-independent with respect to features.
  - They are very fast to grow.
  - They can handle continuous, ordered categorical, and categorical variables well.
- Tree models are very flexible but tend to have a high variance.
- The tree method is a "local" estimation method, with "locality" determined by data (may depend on X only or on both X and y, depending on the stopping criteria).
- Trees are very good base models for ensemble methods (random forests, boosted trees).
  - In random forests, individual trees are often grown to a high depth.
  - In boosting, individual trees are often very shallow.

8.3 Complexity parameter and cost complexity pruning

Complexity parameter (CP/cp): It is a threshold for per-split gain in classification rate (for classification trees) or in R^2 (for regression trees). It is expressed as a fraction. If the per-split gain is below this threshold, do not split. (Some other factors also contribute to the computation of cp. This is why the computed cp for some nodes is smaller than the per-split gain.)

The information is stored in the `cptable` element. There are also functions `printcp()` and `plotcp()`.

```
tree1$cptable
printcp(tree1)
plotcp(tree1)
```

The CP table shows all the steps of gain in classification. The `rel error` column is the classification error relative to the baseline (the root). It decreases as the tree grows. All columns except `nsplit` are in fractions. We convert the numbers back to the scale of count to reveal what they are.

```
cptable2 = function(cptable, E0) {
  cpt2 = cptable * E0; cpt2[,2] = cptable[,2]; cpt2
}
cptable2(tree1$cptable, 858)  ## 858 is the error count for the root
table(Wage$health, predict(tree1, type="class"))  ## 831 errors in this model
```

Let us relax the cp threshold to allow the tree to be grown further.

```
tree3 = rpart(health ~ ., data=Wage, method='class', control=rpart.control(cp=0.001))
plot(tree3)

cptable2(tree3$cptable, 858)
table(Wage$health, predict(tree3, type="vector"))
```

Cost complexity pruning: An overly grown tree can overfit the data. Consider minimizing

$$R(T) + \alpha|T|, \tag{8.4}$$

overall all subtrees of a fully grown tree, where $R(T)$ is the total "risk" for subtree T, $|T|$ is the number of terminal nodes in T, and $\alpha > 0$ is the "cost" of adding a parameter to the model. For regression trees, $R(T)$ is the total within-node RSS over all leaves. For classification trees, $R(T)$ is often the total classification error over all leaves. The subtree that minimizes (8.4) is the "optimal" subtree at cost α.

We often do not know what α to use. We can use cross-validation over a set of α values to identify the "optimal" alpha and its corresponding "optimal" tree. Given a datasest, because there are a finite number of tree models, there are a finite number of α values to evaluate. In `rpart`, a cross-validation is done for every step of gain in classification. The results are in the last two columns of the CP table. The information can be plotted with `plotcp()`.

```
tree3 = rpart(health ~ ., data=Wage, method='class', control=rpart.control(cp=0.001))
plotcp(tree3)
```

Note that from one run to another, the cross-validation results change due to the randomness in partitioning, and our conclusion on where the "best" fit is may change. The trees are always the same because the tree growing process is deterministic.

In the plot, the values at the x-axis are not those in the `CP` column. They are the geometric averages of every value in the `CP` column with the next value in the column. These geometric averages were used as the "cost" in (8.4) in cross-validation. We now compute these geometric averages.

```
cptablebeta = function(cptable) {
  sqrt(cptable[-1,1] * cptable[-dim(cptable)[1],1])
}
cptablebeta(tree3$cptable)
```

Pruning: In `rpart`, this is done with `prune()`, in which the argument `cp=` specifies where to prune a tree to. For example, we may prune to the `CP` value that has the smallest `xerror` in `cptable`.

```
cpminxerror = function(cptable) {
  cptable[which.min(cptable[,"xerror"]), "CP"]
}
tree3prune = prune(tree3, cp = cpminxerror(tree3$cptable))
tree3prune$cptable
```

Knowing the result changes from one run to another, we may just specify a cp value.

```
tree3prune2 = prune(tree3, cp = 0.002)
tree3prune2$cptable
```

Because tree growing is very greedy, it is recommended that we always over-grow a tree and prune it back.

8.4 Missing data and surrogate splits

In `rpart`, if an observation has missing value on the outcome, it is not used. Observations with values for the outcome and at least one feature will be used.

Selection of the splitting feature for a node: If a feature has missing data in the node, the reduction of impurity for a split of the feature is first calculated using the obversations that are not missing for the feature, and then rescaled so that the value is comparable across all features.

Surrogate splits: Once a splitting variable and a split point for it have been decided, for the observations missing that variable, one needs to decide where they go. Surrogate variables can be used to help make the decision.

Without surrogate variables, an option is the *blind rule* of "go with the majority", which is to assign those observations with missing values for the chosen splitting variable to the subset where the majority observations go.

Identification of surrogates for the chosen split for a node: Once a splitting variable x and a split point t have been decided for the node, treat $x < t$ vs. $x \geq t$ as a binary outcome, and consider splits of other features to predict this new outcome. The variables whose best split performs better than the "blind rule" are the surrogate variables, and those best splits are surrogate splits. Here, "better" means having a lower misclassification rate. The surrogate splits are then ranked. (To avoid artifacts, splits with one of the subsets containing only one observation are ignored.)

By default, `usesurrogate=2`, which corresponds to the following steps for an observation that has missing value for the splitting variable: It is classified according to the best surrogate. If that is missing, the next best surrogate will be used, etc. The last choice is the "blind rule".

8.5 Evaluation of tree models

Importance of a variable: Every split involves a splitting variable and an improvement in the splitting index. The importance of a variable is defined as the total improvement attributable to the variable. In `rpart`, it is calculated with the additional contribution if the variable is a surrogate at some nodes. This makes sense. For example, in the `Wage` dataset, `wage` and `logwage` are redundant for tree building because they have exactly the same order. Even though only one feature is chosen as the splitting variable at every step, these two features should have the same importance.

```
with(Wage, all.equal(rank(wage), rank(logwage)))  ## TRUE
```

The `variable.importance` element contains the absolute importance.

```
tree1$variable.importance
tree3$variable.importance
tree3prune$variable.importance
```

Note that the order of the features according to their importance can change from a tree to another. The absolute importance is not comparable across trees of different depths. To compare them across tree models, we compute the relative importance by rescaling the numbers so that the total is 100. The variable importance shown by `summary()` are relative importance.

```
X = matrix(0, dim(Wage)[2], 3)
rownames(X) = names(Wage); colnames(X) = c("tree1", "tree3prune", "tree3")
X[names(tree1$variable.importance), 1] = tree1$variable.importance
X[names(tree3prune$variable.importance), 2] = tree3prune$variable.importance
X[names(tree3$variable.importance), 3] = tree3$variable.importance
round(X * rep(1, dim(X)[1]) %o% (100/apply(X,2,sum)))
```

The function `summary()` outputs **detailed information** for each step during the tree building process.

```
summary(tree1)
summary(tree1, cp=.02)  ## This is to trim the summary, not to prune the tree.
```

It outputs the competing variables and their best splits for every node. The number of competing variables to output is controled by `maxcompete` in `rpart.control()`. The number of surrogate splits to output is controled by `maxsurrogate` in `rpart.control()`.

Out of curiosity, I want to check some numbers. First I need a function to compute the Gini index and entropy.

```
myfun = function(k1, k2) {
    kk = k1+k2; p1 = k1/kk; p2 = 1-p1
    list(gini = kk * (1-p1^2-p2^2), info = -kk * (p1*log(p1)+p2*log(p2)))
}

## improvement of Gini for the split of the root node
myfun(858, 2142)$gini - myfun(643, 1246)$gini - myfun(215, 896)$gini  ## improve=30.1811
## second choice
with(Wage, table(logwage<4.898724, health))
myfun(858, 2142)$gini - myfun(750, 1602)$gini - myfun(108, 540)$gini  ## improve=23.54033
## first surrogate split
with(Wage, table(as.numeric(education) %in% 1:3, logwage < 4.827689))
(589+1610)/3000    ## agree=0.733
```

8.6 An example of regression tree

For the analysis behind ISLR Figure 8.4, the authors say they used 9 features to build a tree. I assume they used `AtBat`, `Hits`, `HmRun`, `Runs`, `RBI`, `Walks`, `Years`, `PutOuts`, `Assists` to predict `Salary` on the log scale. The data were randomly split into training and test sets. I cannot replicate that without knowing the split. Below I use all the data to build tree models.

```
library(rpart)
library(ISLR)
Hitters2 = Hitters[,c(1:7,16,17,19)]
Hitters2 = na.omit(Hitters2)
names(Hitters2)
```

```
tree4 = rpart(log(Salary) ~ ., data=Hitters2, method='anova')
plot(tree4); text(tree4)
table(tree4$where)  ## distribution of leaf size
tree4$frame
```

Plot the observed vs. fitted values.

```
plot(log(Hitters2$Salary), predict(tree4)); abline(0,1)
sort(tree4$frame$yval[tree4$frame$var == "<leaf>"])  ## fitted values at the leaves
```

CP table and associated plots.

```
printcp(tree4)
var(log(Hitters2$Salary))*262  ## total RSS = 207.15
plotcp(tree4)
rsq.rpart(tree4)  ## for regression trees
```

To prune the tree.

```
tree4$cptable
prune(tree4, cp = tree4$cptable[which.min(tree4$cptable[,"xerror"]),"CP"])
```

8.7 The R tree package

The function `tree()` in the `tree` package has different default splitting criterion and stopping rules. For classification trees the default impurity measure is `deviance`, which is equivalent to entropy. It does not deal with missing data as well as in `rpart()`. It does not output as much information as `rpart()`.

The `rpart` package is recommended because it has richer features and it treats ordered factors correctly. For example, for an ordered factor with levels A < B < C < D, the only splits considered by `rpart()` are A-BCD, AB-CD, ABC-D. But `tree()` also considers other dichotomizations such as B-ACD.

The model we built above can be generated with the `tree()` function. I need to specify the control parameters because the default values are different.

```
library(tree)
tree4b = tree(log(Salary) ~ ., data=Hitters2, split="deviance",
              control=tree.control(nobs=263, mincut=7, minsize=20))
plot(tree4b); text(tree4b)

tree4b$frame
all.equal(tree4b$frame[,1:4], tree4$frame[,c(1,3:5)], check.attributes=F)  ## TRUE
```

The tree can be pruned with `prune.tree()`.

```
cv.tree(tree4b)  ## default is 10-fold CV
prune.tree(tree4b)
```

8.8 Another example of classification tree

I try to repeat the analysis shown in ISLR Figure 8.6. I cannot generate the same result not knowing the parameters used. `rpart()` and `tree()` also give different results even after removing all missing data records.

```
Heart = read.csv("Heart.csv", row.names=1)  ## first column is row name, not a variable
names(Heart); dim(Heart)  ## 303, 14
apply(is.na(Heart), 2, sum)
sum(apply(is.na(Heart), 1, sum))
table(Heart$AHD, useNA="ifany")  ## outcome distribution
class(Heart$AHD)  ## factor
Heart = na.omit(Heart); dim(Heart)
```

```
library(tree)
library(rpart)

tree5 = rpart(AHD ~ ., data=Heart, method='class', control=rpart.control(cp=0.001, minsplit = 5))
plot(tree5, uniform=T); text(tree5)

tree6a = tree(AHD ~ ., data=Heart)
tree6b = tree(AHD ~ ., data=Heart, split='gini',
              control=tree.control(nobs=297, mincut=7, minsize=20))
plot(tree6a); text(tree6a)
plot(tree6b); text(tree6b)
```

The `Heart` dataset has a few ordered categorical predictors. Thallium stress test result has three levels – normal, reversible defect, fixed defect – with increasing severity. The `Thal` variable does not reflect that and the levels are alphabetical (fixed, normal, reversable). Chest pain (`ChestPain`) has four categories – asymptomatic, nonanginal, nontypical, typical. Their alphabetical order happens to be the order of increasing severity.

```
Heart2 = Heart
levels(Heart2$Thal)
class(Heart2$Thal)
```

```
Heart2$Thal = factor(Heart2$Thal, c('normal', 'reversable', 'fixed'), ordered=T)
## The line below won't make it an ordered factor.  It just changes the internal coding.
# Heart2$Thal = factor(Heart2$Thal, c('normal', 'reversable', 'fixed'))

levels(Heart2$ChestPain)
class(Heart2$ChestPain)
Heart2$ChestPain = factor(Heart2$ChestPain, levels(Heart$ChestPain), ordered=T)
```

`rpart()` gives a different result now.

```
tree5b = rpart(AHD ~ ., data=Heart2, method='class', control=rpart.control(cp=0.001, minsplit = 5))
plot(tree5, uniform=T); text(tree5)    ## there was a split "ChestPain=b"
plot(tree5b, uniform=T); text(tree5b)  ## it is now "ChestPain=ab"
```

But `tree()` gives the same result as before.

```
tree6c = tree(AHD ~ ., data=Heart2)
all.equal(tree6a$frame[,1:4], tree6c$frame[,1:4], check.attributes=F)  ## TRUE
plot(tree6c); text(tree6c)  ## there was a split "ChestPain:bc"
plot(tree6a); text(tree6a)  ## it is still there
```

Notes on other R packages related to trees:

- `rpart.plot` gives nicer plots for trees built with `rpart`.
- `caret` (document) provides a unified interface (and a wrapper) for using `rpart` and many other R packages.
- `rpartScore` is for ordered categorical outcomes.
- `party` privides functions to build "conditional inference trees" (ctree) and to fit random forest models with conditional inference trees as base models.

8.9 Assignment

1. **Homework**: ISLR Chapter 8 Exercises 9