Lab 03: Decision Trees

PSTAT 131/231, Spring 2021

Learning Objectives

- Get familier with basic R commands
- Know how to split the data to a training and a test set
- Cross-validation
- Fit decision tree models using package `tree` and `base` - `tree()` and `summary()` - `predict()` and `table()` - `cv.tree()` and `prune.tree()`
- Be able to visualize the trees

---

1. Install packages and import dataset

We are going to use the dataset `Carseats` in the package `ISLR` and various tree-fitting functions in `tree`. `Carseats` is a simulated data set containing sales of child car seats at 400 different stores on 11 features. The features include: `Sales`, `CompPrice`, `Income`, `Advertising`, `Population`, `Price`, `ShelveLoc`, `Age`, `Education`, `Urban` and `US`. Among all the variables, `ShelveLoc`, `Urban` and `US` are categorical and the rest are continuous.

Notice that originally `Sales` is a continuous variable. Now we create a new binary variable `High` using `Sales`:

$$\text{High} = \begin{cases} \text{No}, & \text{if Sales} \leq \text{median(Sales)} \\ \text{Yes}, & \text{if Sales} > \text{median(Sales)} \end{cases}$$

Our goal is to investigate how other features (`CompPrice`, `Income`, `Advertising`, `Population`, `Price`, `ShelveLoc`, `Age`, `Education`, `Urban` and `US`) influence whether the unit sales at each location is high or not. In other words, we look for the relationship between the binary response `High` and all variables but `Sales`.

Using the following codes, the data can be read into R:

```
##install.packages("ISLR")
##install.packages("tree")
##install.packages('maptree')

# Load libraries
library(ISLR)
library(tree)
library(maptree)

# Utility library
library(dplyr)

# See description of data
# ?Carseats
```

Using `mutate()` and `ifelse()` to create the binary response variable `High`, then check the structure of resulting data frame with the following codes:

```
# Create data frame with the oringinal eleven variables and High
Carseats = Carseats %>%
    mutate(High=as.factor(ifelse(Sales <= median(Sales), "No", "Yes")))

# Check the structure of above data frame we just created
glimpse(Carseats)
```

```
## Registered S3 method overwritten by 'cli':
##   method     from
##   print.tree tree
```

```
## Rows: 400
## Columns: 12
## $ Sales       <dbl> 9.50, 11.22, 10.06, 7.40, 4.15, 10.81, 6.63, 11.85, 6.54, ~
## $ CompPrice   <dbl> 138, 111, 113, 117, 141, 124, 115, 136, 132, 132, 121, 117~
## $ Income      <dbl> 73, 48, 35, 100, 64, 113, 105, 81, 110, 113, 78, 94, 35, 2~
## $ Advertising <dbl> 11, 16, 10, 4, 3, 13, 0, 15, 0, 0, 9, 4, 2, 11, 11, 5, 0, ~
## $ Population  <dbl> 276, 260, 269, 466, 340, 501, 45, 425, 108, 131, 150, 503,~
## $ Price       <dbl> 120, 83, 80, 97, 128, 72, 108, 120, 124, 124, 100, 94, 136~
## $ ShelveLoc   <fct> Bad, Good, Medium, Medium, Bad, Bad, Medium, Good, Medium,~
## $ Age         <dbl> 42, 65, 59, 55, 38, 78, 71, 67, 76, 76, 26, 50, 62, 53, 52~
## $ Education   <dbl> 17, 10, 12, 14, 13, 16, 15, 10, 10, 17, 10, 13, 18, 18, 18~
## $ Urban       <fct> Yes, Yes, Yes, Yes, Yes, No, Yes, Yes, No, No, No, Yes, Ye~
## $ US          <fct> Yes, Yes, Yes, Yes, No, Yes, No, Yes, No, Yes, Yes, Yes, N~
## $ High        <fct> Yes, Yes, Yes, No, No, Yes, No, Yes, No, No, Yes, Yes, No,~
```

2. A decision tree trained with the entire dataset

Based on the data frame `Carseats` with `High`, we will build a classification tree model, in which `High` will be the response (dependent variable), and the rest 10 features, excluding `Sales`, will be the explanatory variables (independent variables). The classification tree model can be built with function `tree()` in the package `tree`. (Yeah, they share the same name! :])

Fit, summarize and Visualize the tree

- **`tree()`** can be used to fit both classification and regression tree models. A regression tree is very similar to a classification tree, except that it is used to predict a quantitative response rather than a qualitative one. In this lab, we will focus on classification trees. We put the response variable on the left of tilde, explanatory variables on the right of tilde; the dot is merely an economical way to represent "everything else but High".[1]

  ```
  tree.carseats = tree(High ~.-Sales, data = Carseats)
  ```

- **`summary()`** is a generic function used to produce result summaries of various model fitting functions. When we call the `summary` of a tree, we will have the following reported:

  - *Classification tree*: displays the model and the dataset

[1]Note: The reason why we have to exclude `Sales` from the explanatory variables is that the response (`High`) is derived from it.

- *Variables . . . construction*: variables that are truly useful to construct the tree
- *Number . . . nodes*: the number of leaf node, which is a node that has no child nodes. Let's denote this quantity as T_0 for further reference
- *Residual mean deviance*: is simply the deviance divided by $n - T_0$, which in this case is $400 - 23 = 377$
- *Misclassification error rate*: is the number of wrong predictions divided by the number of total predictions

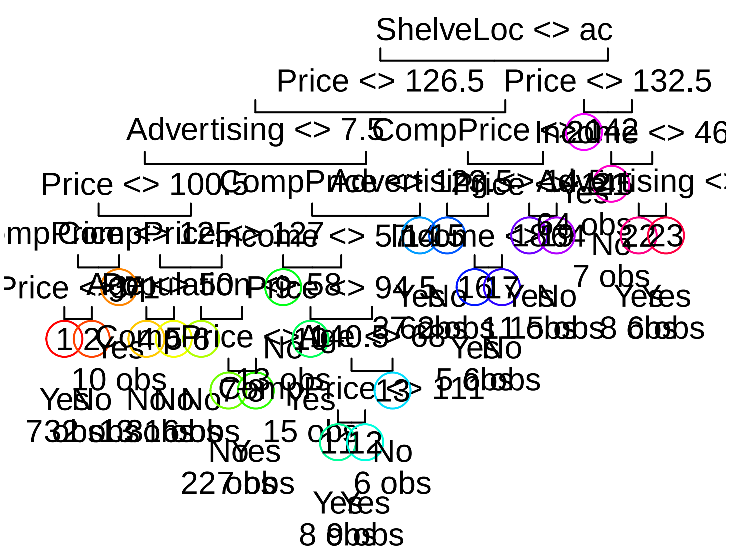
```
summary(tree.carseats)
```

```
##
## Classification tree:
## tree(formula = High ~ . - Sales, data = Carseats)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Advertising" "CompPrice"   "Age"
## [6] "Population"  "Income"
## Number of terminal nodes:  23
## Residual mean deviance:  0.4945 = 186.4 / 377
## Misclassification error rate: 0.115 = 46 / 400
```

R displays the split criterion, the number of observations in that branch, the deviance, the overall prediction for the branch (Yes or No), and the fraction of observations in that branch that take on values of Yes and No. Branches that lead to terminal nodes are indicated using asterisks.

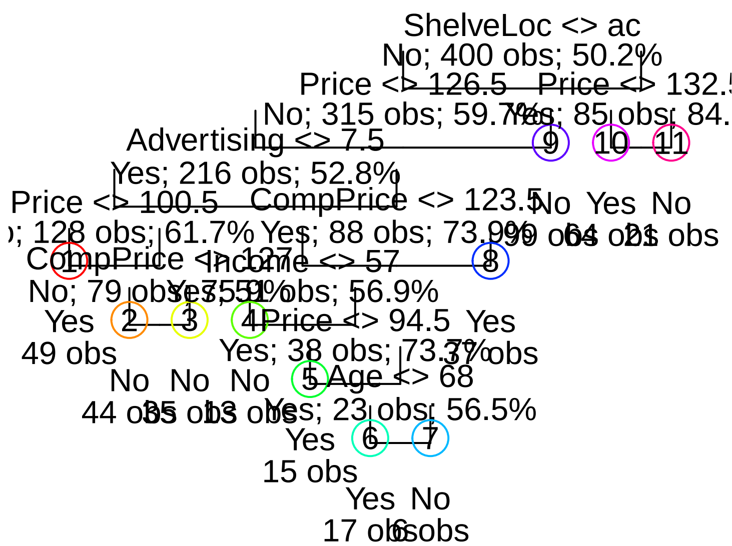
- **`draw.tree()`** in the `maptree` package is helpful for visualizing the structure

```
draw.tree(tree.carseats, nodeinfo=FALSE)
```




```
draw.tree(prune.tree(tree.carseats, best=10), nodeinfo=TRUE)
title("Classification Tree")
```

Classification Tree




```r
# plot(tree.carseats)
# text(tree.carseats, pretty = 0, cex = .8, col = "red")
```

3. A decision tree trained with training/test split

In order to properly evaluate the performance of a classification tree, we should estimate the **test error rate** rather than simply compute the training error rate. Therefore we split all observations into a **training set** and a **test set**, build the tree using the training set, and evaluate the model's performance on the test set.

(a). Split the data into a training set and a test set

We sample 75% of observations as the training set and the rest 25% as the test set.

```r
# Set random seed for results being reproducible
set.seed(3)
# Get dimension of dataset
dim(Carseats)
```

```
## [1] 400  12
```

```r
# Sample 75% of observations as the training set
train = sample(1:nrow(Carseats), 0.75*dim(Carseats)[1])
# The rest 25% as the test set
Carseats.test = Carseats[-train,]

# For later convenience in coding, we create High.test, which is the true labels of the
# test cases
High.test = Carseats.test$High
```

(b). Fit the tree on training set and compute test error rate

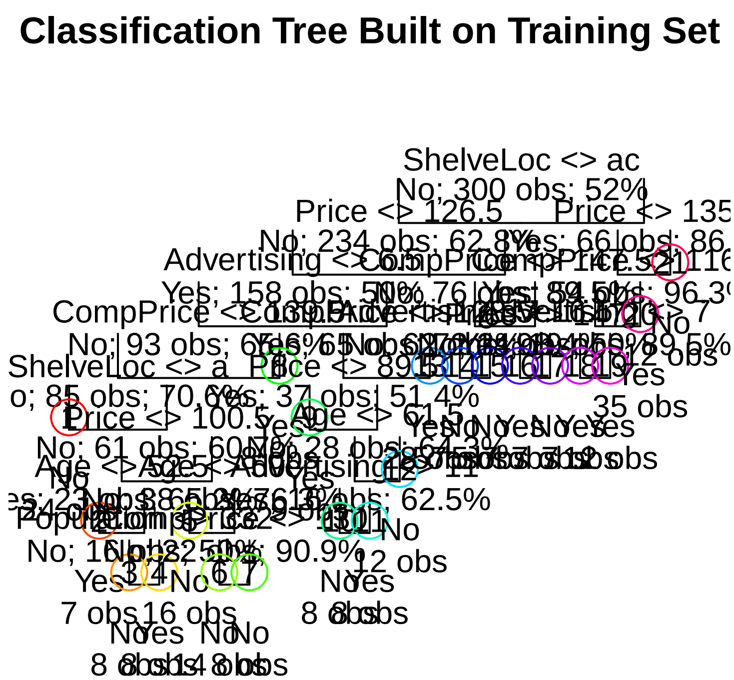
- `tree()` can be used to grow the tree as we discussed in the previous section.
- `predict()` is helpful to predict the response (`High`) on the test set. In the case of a classification tree, specifying `type="class"` instructs R to return the actual class predictions instead of probabilities.

  As discussed earlier, we build the model on the training set and predict the labels for `High` on the test set:

```
# Fit model on training set
tree.carseats = tree(High~.-Sales, data = Carseats, subset = train)

# Plot the tree

draw.tree(tree.carseats, nodeinfo=TRUE)
title("Classification Tree Built on Training Set")
```




```
# plot(tree.carseats)
# text(tree.carseats, pretty = 0, cex = .8, col = "red")
# title("Classification Tree Built on Training Set")

# Predict on test set
tree.pred = predict(tree.carseats, Carseats.test, type="class")
tree.pred
```

```
##   [1] No  Yes Yes No  Yes Yes No  Yes No  No  Yes No  Yes No  No  No  No  No
##  [19] Yes Yes No  No  Yes No  Yes Yes No  No  No  No  No  Yes No  No  No  No
##  [37] Yes Yes Yes Yes No  No  No  No  Yes Yes Yes No  No  No  No  No  No  Yes
##  [55] Yes No  Yes No  Yes No  Yes No  Yes No  No  Yes Yes Yes No  Yes Yes No
##  [73] No  No  No  No  No  No  No  No  Yes No  Yes No  No  Yes Yes No  Yes No
##  [91] No  Yes No  No  Yes Yes No  No  Yes Yes
## Levels: No Yes
```

- To calculate the test error rate, we can construct a confusion matrix and use the counter diagonal sum divided by the total counts.

```
# Obtain confusion matrix
error = table(tree.pred, High.test)
error
```

```
##          High.test
## tree.pred No Yes
##       No  39  20
##       Yes  6  35
```

```
# Test accuracy rate
sum(diag(error))/sum(error)
```

```
## [1] 0.74
```

```
# Test error rate (Classification Error)
1-sum(diag(error))/sum(error)
```

```
## [1] 0.26
```

This approach leads to correct predictions for 74% of the locations in the test set. In other words, the test error rate is 26%.

4. Prune the tree using `prune.tree()`/`cv.tree()` and `prune.misclass()`

Next, we consider whether pruning the tree might lead to a lower test error. To do so, primarily we have to decide what the best size of the tree should be, then we can trim the tree to this pre-determined size.

(a). Determine the best size

By 'best' size, for example, if we use classification error rate to guide the pruning process, we mean the number of terminal nodes which corresponds to the **smallest** classification error. There are other goodness-of-fit measures available, such as deviance, the 'best' size in this case is the number of leaf nodes which gives the smallest deviance. We have two ways to determine the best size of the tree: either use **`prune.tree()`** or **`cv.tree()`**, which are both from package `tree`.

- **`prune.tree()`** does a cost-complexity pruning of a tree object. The argument `method` is the scoring measure used to trim the tree. The argument `k` is user-specified cost-complexity parameter, and `best` instructs R to return a tree exactly of this size. Larger the cost-complexity `k`, smaller the tree, although cost-complexity `k` does not correspond to tree size in any exact way. (`k` is similar to parameter α in equation 8.4 in ISLR). `prune.tree()` yields several results such as sizes of the trees, complexity parameters and guiding method of the pruning.

```
prune = prune.tree(tree.carseats, k = 0:20, method = "misclass")
# Best size
best.prune = prune$size[which.min(prune$dev)]
best.prune
```

```
## [1] 16
```

Note: we specified misclassification error as the scoring method, so `$dev` is not deviance but actually misclassification error. Also, we didn't specified `newdata` option in `prune.tree`, so `$dev` is computed on the training data. From the output, the 'best' size is 16 since this number of terminal nodes corresponds to the smallest misclassification error.

- `cv.tree()` performs k-fold Cross-validation in order to determine the optimal level of tree complexity; cost-complexity pruning is used in order to select a sequence of trees for consideration. The argument `FUN=prune.misclass` is to indicate that misclassification error should guide the Cross-validation and pruning process, rather than the default deviance in the `cv.tree()` function. `K=10` instructs R to use a 10-fold Cross-validation in order to find the best size. The `cv.tree()` function reports the number of terminal nodes of each tree considered, as well as the corresponding error rate and the value of the cost-complexity parameter `k` used.

```
# Set random seed
set.seed(3)

# K-Fold cross validation
cv = cv.tree(tree.carseats, FUN=prune.misclass, K=10)
# Print out cv
cv
```

```
## $size
##  [1] 21 16 14 12 10  8  5  4  2  1
##
## $dev
##  [1]  60  60  78  78  75  76  78  74  97 144
##
## $k
##  [1] -Inf  0.0  2.0  2.5  3.0  3.5  4.0  6.0 14.5 48.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
# Best size
best.cv = cv$size[which.min(cv$dev)]
best.cv
```

```
## [1] 21
```

```
# Get names of entries in cv
names(cv)
```

```
## [1] "size"   "dev"    "k"      "method"
```

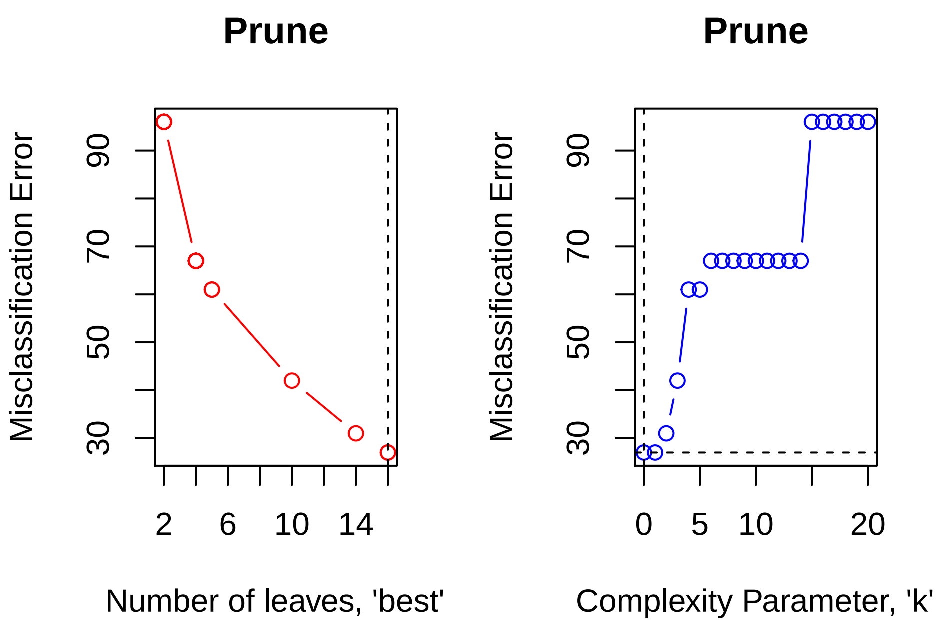
```
# Get classes in cv, produce the same result
class(cv)
```

```
## [1] "prune"         "tree.sequence"
```

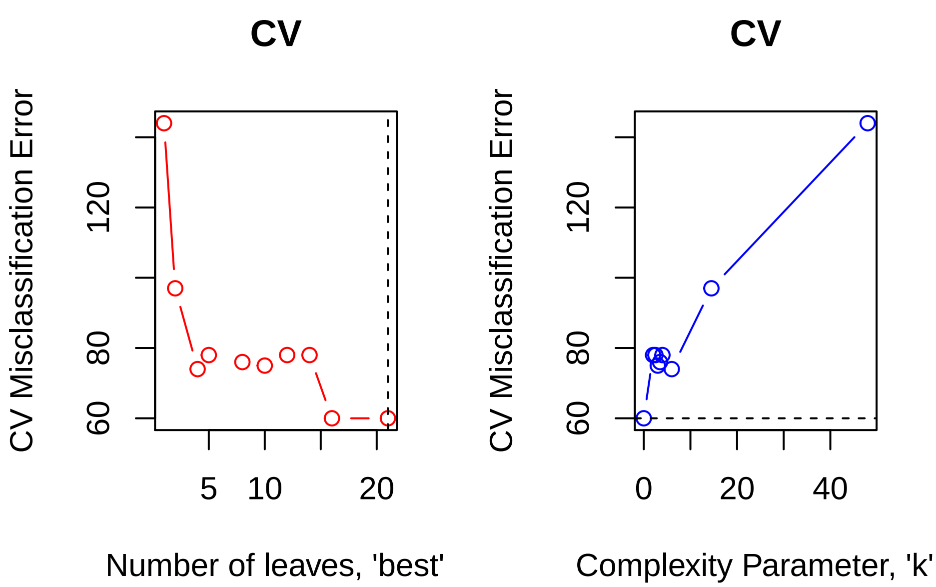
Note again, despite the name, `$dev` is the Cross-validation error instead of deviance. The tree with 21 terminal nodes results in the lowest error.

(b). Error vs. Best Size plot and Error vs. Complexity plot

- On the basis of `prune.tree()` result:



- Based on `cv.tree()` result:



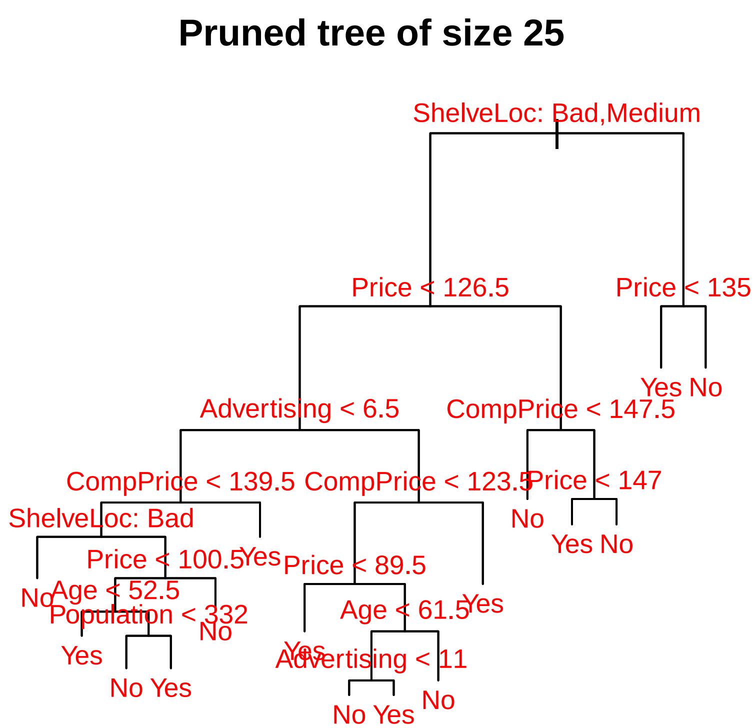
(c) Prune the tree and visualize it

- **`prune.misclass`** is used to prune a tree in order to have a tree with targeted best number of terminal nodes.

  First, let's "trim" the original tree, `tree.carseats`, to have 16 nodes. (16 was determined from `prune.tree()`.)

```r
# Prune tree.carseats
pt.prune = prune.misclass (tree.carseats, best=best.prune)

# Plot pruned tree
plot(pt.prune)
text(pt.prune, pretty=0, col = "red", cex = .8)
title("Pruned tree of size 25")
```

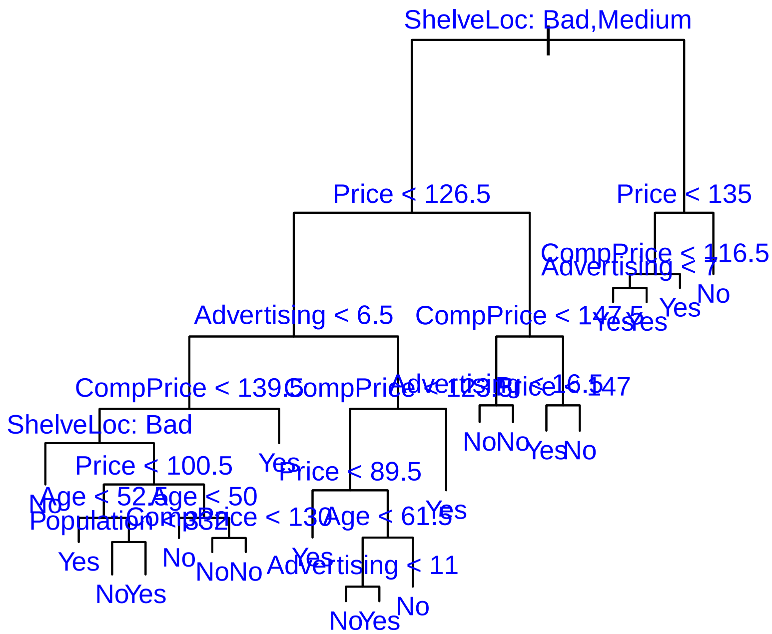


Second, let's trim `tree.carseats` to have 21 nodes. This number was determined by `cv.tree()`.

```r
# Prune tree.carseats
pt.cv = prune.misclass (tree.carseats, best=best.cv)

# Plot pruned tree
plot(pt.cv)
text(pt.cv, pretty=0, col = "blue", cex = .8)
title("Pruned tree of size 11")
```

Pruned tree of size 11



(d) Calculate respective test error rate for model `pt.prune` and `pt.cv`

Recall that in (3b), we built `tree.carseats` on the training set and obtained the test error rate as 21%. In (4a) and (4c), we trimmed the tree in two ways and got two tree models: `pt.prune` and `pt.cv`, thus we want to see if the two trimmed tree are better than `tree.carseats`, judged by the test misclassification error rate. Let's predict the labels for `High` on test set for two models and construct confusion matrices.

- Tree `pt.prune`

```
# Predict on test set
pred.pt.prune = predict(pt.prune, Carseats.test, type="class")
# Obtain confusion matrix
err.pt.prune = table(pred.pt.prune, High.test)
err.pt.prune
```

```
##              High.test
## pred.pt.prune No Yes
##           No  39  20
##           Yes  6  35
```

```
# Test accuracy rate
sum(diag(err.pt.prune))/sum(err.pt.prune)
```

```
## [1] 0.74
```

```
# Test error rate (Classification Error)
1-sum(diag(err.pt.prune))/sum(err.pt.prune)
```

```
## [1] 0.26
```

The test error rate for `pt.prune` is 0.26, which is the same as the result in (3b). This is not surprising because `pt.prune` is exactly the same as `tree.carseats`. To verify it, you can compare the two trees visually or notice that the number of terminal nodes in `pt.prune` and `tree.carseats` are both 25, indicating the trees grown are identical.

- Tree `pt.cv`

```
# Predict on test set
pred.pt.cv = predict(pt.cv, Carseats.test, type="class")
# Obtain confusion matrix
err.pt.cv = table(pred.pt.cv, High.test)
err.pt.cv
```

```
##           High.test
## pred.pt.cv No Yes
##        No  39  20
##        Yes  6  35
```

```
# Test accuracy rate
sum(diag(err.pt.cv))/sum(err.pt.cv)
```

```
## [1] 0.74
```

```
# Test error rate (Classification Error)
1-sum(diag(err.pt.cv))/sum(err.pt.cv)
```

```
## [1] 0.26
```

The test error rate for `pt.cv` is 0.26, which is really close to the result in (3a). Since this tree is simpler (as shown in 4c) without much loss of accuracy, therefore we think `pt.cv` is the best among all trees we grew.

Your turn

Using the original tree `tree.carseats`, perform 5-fold Cross-validation to determine the best size of the tree:

```
# Codes start here
```

Calculate the test error rate:

```
# Codes start here:

# Test set is Carseats.test
```

Credit: Adopted from *An Introduction to Statistical Learning* by *Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani*