Tree-based Methods

Classification Trees

The `tree` library is used to construct classification and regression trees. We analyze the `Carseats` data set.

```
library(tree)
library(ISLR2)
attach(Carseats)
```

We transform the continuous variable `Sales` into a categorical variable. We use the `ifelse()` function to create a variable, called `High`, which takes on a value of `Yes` if the `Sales` variable exceeds 8, and takes on a value of `No` otherwise. We use the `data.frame()` function to merge `High` with the rest of the `Carseats` data.

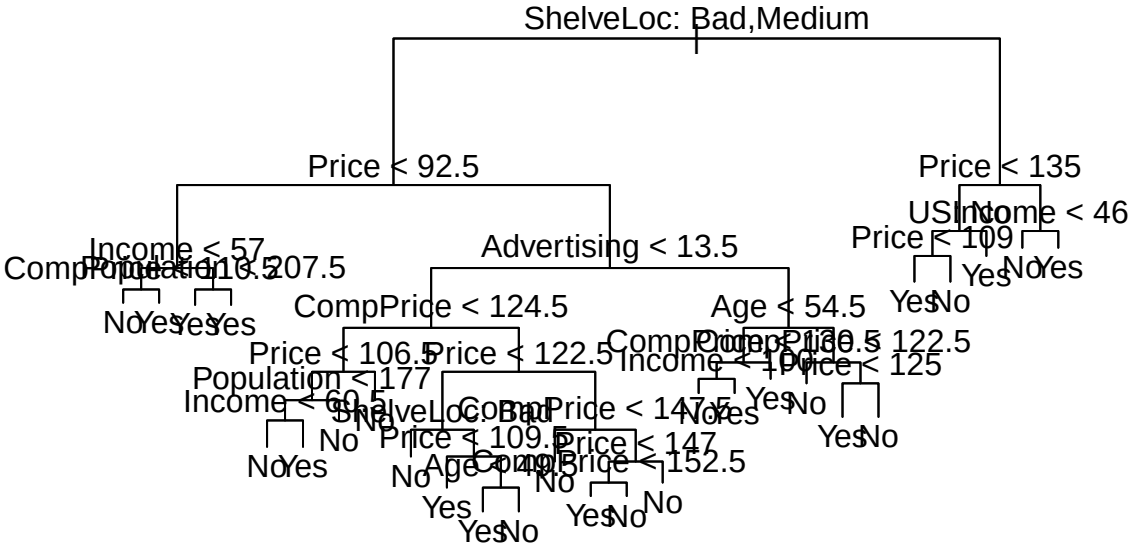
```
High <- factor(ifelse(Sales <= 8, "No", "Yes"))
Carseats <- data.frame(Carseats, High)
```

We use the `tree()` function to fit a classification tree in order to predict `High` using all variables but `Sales`. The syntax of the `tree()` function is quite similar to that of the `lm()` function.

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

Trees can be graphically displayed. We use the `plot()` function to display the tree structure, and the `text()` function to display the node labels.

```
plot(tree.carseats)
text(tree.carseats, pretty = 0)
```



We split the observations into a training set and a test set, build the tree using the training set, and evaluate its performance on the test data. The `predict()` function can be used for this purpose. In the case of a classification tree, the argument `type = "class"` instructs `R` to return the actual class prediction. This approach leads to correct predictions for around 77 % of the locations in the test data set.

```
set.seed(2)
train <- sample(1:nrow(Carseats), 200)
Carseats.test <- Carseats[-train, ]
High.test <- High[-train]
tree.carseats <- tree(High ~ . - Sales, Carseats, subset = train)
tree.pred <- predict(tree.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred  No Yes
##       No  104  33
##       Yes  13  50
```

```
(104 + 50) / 200
```

```
## [1] 0.77
```

Next, we consider whether pruning the tree might lead to improved results. The function `cv.tree()` performs cross-validation in order to determine the optimal level of tree complexity. We use the argument `FUN = prune.misclass` in order to indicate that we want the classification error rate to guide the cross-validation and pruning process, rather than the default for the `cv.tree()` function, which is deviance. The `cv.tree()` function reports the number of terminal nodes of each tree considered (`size`) as well as the corresponding error rate and the value of the cost-complexity parameter used (`k`, which corresponds to α in (8.4)).

```
set.seed(7)
cv.carseats <- cv.tree(tree.carseats, FUN = prune.misclass)
names(cv.carseats)
```

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

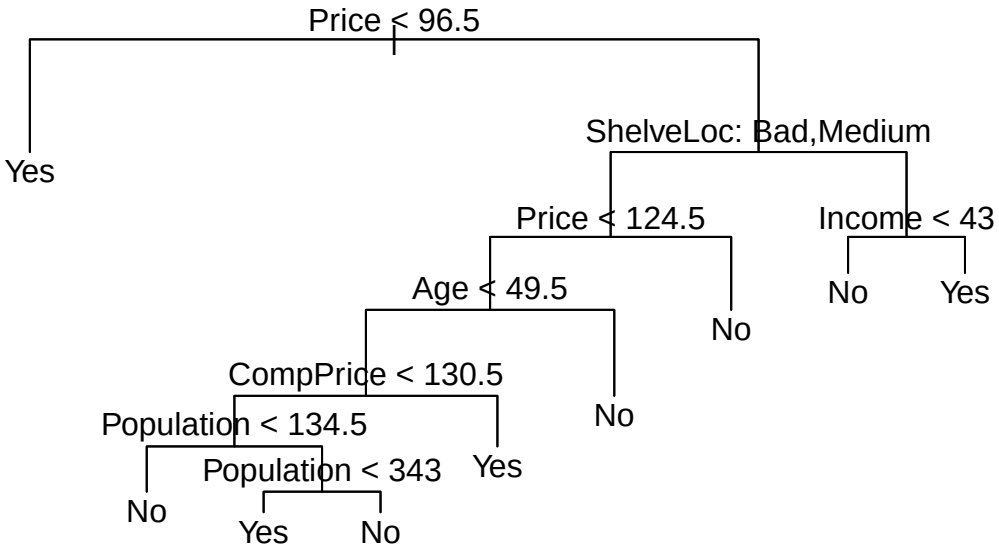
```
cv.carseats
```

```
## $size
## [1] 21 19 14  9  8  5  3  2  1
##
## $dev
## [1] 75 75 75 74 82 83 83 85 82
##
## $k
## [1] -Inf  0.0  1.0  1.4  2.0  3.0  4.0  9.0 18.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Despite its name, `dev` corresponds to the number of cross-validation errors. The tree with 9 terminal nodes results in only 74 cross-validation errors.

We now apply the `prune.misclass()` function in order to prune the tree to obtain the nine-node tree.

```
prune.carseats <- prune.misclass(tree.carseats, best = 9)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```



Check how the pruned tree performs on the test data. Now 77.5 % of the test observations are correctly classified, so not only has the pruning process produced a more interpretable tree, but it has also slightly improved the classification accuracy.

```
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred No Yes
##       No  97  25
##       Yes 20  58
```

```
(97 + 58) / 200
```

```
## [1] 0.775
```

Regression Trees

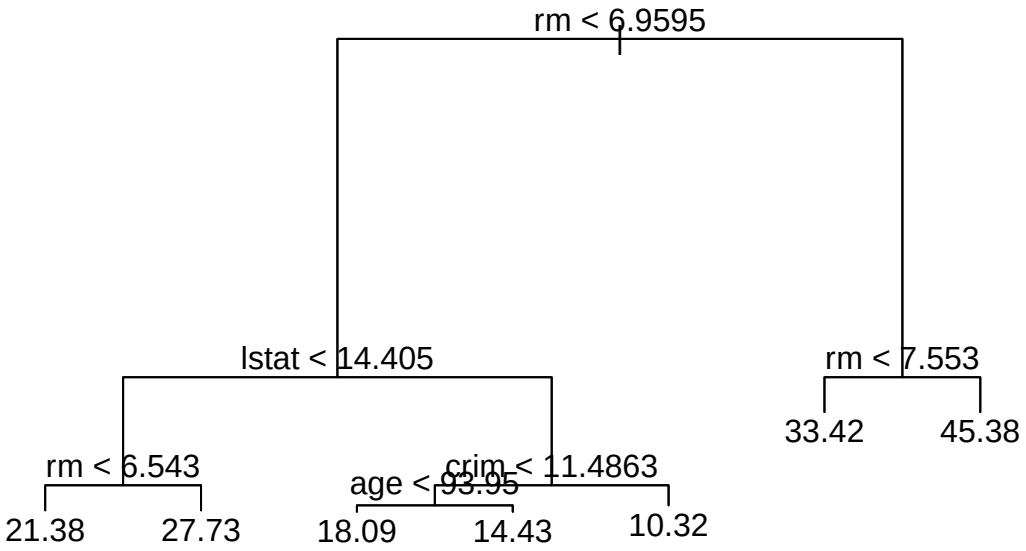
Here we fit a regression tree to the `Boston` data set. First, we create a training set, and fit the tree to the training data. Notice that the output of `summary()` indicates that only four of the variables have been used in constructing the tree.

```
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston) / 2)
tree.boston <- tree(medv ~ ., Boston, subset = train)
summary(tree.boston)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "crim"  "age"
## Number of terminal nodes:  7
## Residual mean deviance:  10.38 = 2555 / 246
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -10.1800  -1.7770  -0.1775   0.0000   1.9230  16.5800
```

We plot the tree.

```
plot(tree.boston)
text(tree.boston, pretty = 0)
```

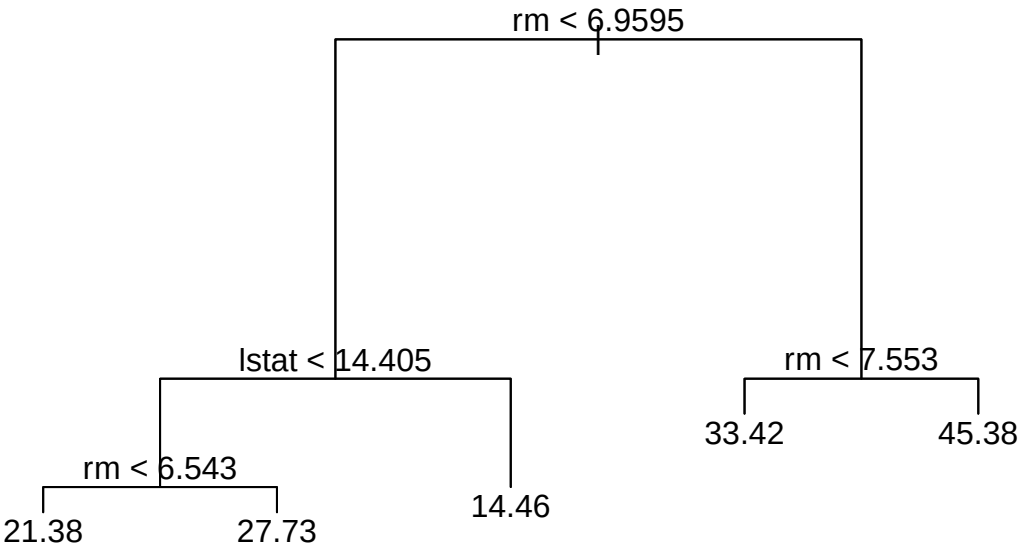


Now we use the `cv.tree()` function to see whether pruning the tree will improve performance. In this case, the most complex tree under consideration is selected by cross-validation.

```
cv.boston <- cv.tree(tree.boston)
plot(cv.boston$size, cv.boston$dev, type = "b")
```

However, if we wish to prune the tree, we could do so as follows, using the `prune.tree()` function:

```
prune.boston <- prune.tree(tree.boston, best = 5)
plot(prune.boston)
text(prune.boston, pretty = 0)
```



In keeping with the cross-validation results, we use the unpruned tree to make predictions on the test set. The test set MSE associated with the regression tree is 35.29.

```
yhat <- predict(tree.boston, newdata = Boston[-train, ])
boston.test <- Boston[-train, "medv"]
mean((yhat - boston.test)^2)
```

```
## [1] 35.28688
```

Bagging and Random Forests

Here we apply bagging and random forests to the `Boston` data, using the `randomForest` package in `R`. Bagging is simply a special case of a random forest. The argument `mtry = 12` indicates that all 12 predictors should be considered for each split of the tree. In other words, that bagging should be done.

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
set.seed(1)
bag.boston <- randomForest(medv ~ ., data = Boston, subset = train, mtry = 12, importance = TRUE)
bag.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 12, importance = TRUE,      subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 12
##
##           Mean of squared residuals: 11.40162
##                     % Var explained: 85.17
```

How well does this bagged model perform on the test set? The test set MSE associated with the bagged regression tree is 23.42, about two-thirds of that obtained using an optimally-pruned single tree.

```
yhat.bag <- predict(bag.boston, newdata = Boston[-train, ])
mean((yhat.bag - boston.test)^2)
```

```
## [1] 23.41916
```

Growing a random forest proceeds in exactly the same way, except that we use a smaller value of the `mtry` argument. By default, `randomForest()` uses $p/3$ variables when building a random forest of regression trees, and $\sqrt{p}$ variables when building a random forest of classification trees. Here we use `mtry = 6`. The test set MSE is 20.07; this indicates that random forests yielded an improvement over bagging in this case.

```
set.seed(1)
rf.boston <- randomForest(medv ~ ., data = Boston, subset = train, mtry = 6, importance = TRUE)
yhat.rf <- predict(rf.boston, newdata = Boston[-train, ])
mean((yhat.rf - boston.test)^2)
```

```
## [1] 20.06644
```