8.3 Lab: Decision Trees

8.3.1 Fitting Classification Trees

The `tree` library is used to construct classification and regression trees.

```
> library(tree)
```

We first use classification trees to analyze the `Carseats` data set. In these data, `Sales` is a continuous variable, and so we begin by recoding it as a binary 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. ifelse()

```
> library(ISLR2)
> attach(Carseats)
> High <- factor(ifelse(Sales <= 8, "No", "Yes"))
```

Finally, we use the `data.frame()` function to merge `High` with the rest of the `Carseats` data.

```
> Carseats <- data.frame(Carseats, High)
```

We now 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()

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

The `summary()` function lists the variables that are used as internal nodes in the tree, the number of terminal nodes, and the (training) error rate.

```
> summary(tree.carseats)
Classification tree:
tree(formula = High ~ . - Sales, data = Carseats)
Variables actually used in tree construction:
[1] "ShelveLoc"   "Price"       "Income"      "CompPrice"
[5] "Population"  "Advertising" "Age"         "US"
Number of terminal nodes:  27
Residual mean deviance:  0.4575 = 170.7 / 373
Misclassification error rate: 0.09 = 36 / 400
```

We see that the training error rate is 9%. For classification trees, the deviance reported in the output of `summary()` is given by

$$-2\sum_{m}\sum_{k} n_{mk}\log\hat{p}_{mk},$$

where n_{mk} is the number of observations in the mth terminal node that belong to the kth class. This is closely related to the entropy, defined in (8.7). A small deviance indicates a tree that provides a good fit to the (training) data. The *residual mean deviance* reported is simply the deviance divided by $n - |T_0|$, which in this case is $400 - 27 = 373$.

One of the most attractive properties of trees is that they can be graphically displayed. We use the `plot()` function to display the tree structure, and the `text()` function to display the node labels. The argument `pretty = 0` instructs `R` to include the category names for any qualitative predictors, rather than simply displaying a letter for each category.

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

The most important indicator of `Sales` appears to be shelving location, since the first branch differentiates `Good` locations from `Bad` and `Medium` locations.

If we just type the name of the tree object, `R` prints output corresponding to each branch of the tree. `R` displays the split criterion (e.g. `Price < 92.5`), 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.

```
> tree.carseats
node), split, n, deviance, yval, (yprob)
      * denotes terminal node
 1) root 400 541.5 No ( 0.590 0.410 )
   2) ShelveLoc: Bad,Medium 315 390.6 No ( 0.689 0.311 )
     4) Price < 92.5 46  56.53 Yes ( 0.304 0.696 )
       8) Income < 57 10  12.22 No ( 0.700 0.300 )
```

In order to properly evaluate the performance of a classification tree on these data, we must estimate the test error rather than simply computing the training error. 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
```

(If you re-run the `predict()` function then you might get slightly different results, due to "ties": for instance, this can happen when the training observations corresponding to a terminal node are evenly split between `Yes` and `No` response values.)

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; cost complexity pruning is used in order to select a sequence of trees for consideration. 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)).

`cv.tree()`

```
> 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 plot the error rate as a function of both `size` and `k`.

```
> par(mfrow = c(1, 2))
> plot(cv.carseats$size, cv.carseats$dev, type = "b")
> plot(cv.carseats$k, cv.carseats$dev, type = "b")
```

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

`prune.misclass()`

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

How well does this pruned tree perform on the test data set? Once again, we apply the `predict()` function.

```
> 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
```

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.

If we increase the value of `best`, we obtain a larger pruned tree with lower classification accuracy:

```
> prune.carseats <- prune.misclass(tree.carseats, best = 14)
> plot(prune.carseats)
> text(prune.carseats, pretty = 0)
> tree.pred <- predict(prune.carseats, Carseats.test,
    type = "class")
> table(tree.pred, High.test)
         High.test
tree.pred  No Yes
      No  102  31
      Yes  15  52
> (102 + 52) / 200
[1] 0.77
```

8.3.2 Fitting 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.

```
> 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:  10.4 = 2550 / 246
Distribution of residuals:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-10.200  -1.780  -0.177   0.000   1.920  16.600
```

Notice that the output of `summary()` indicates that only four of the variables have been used in constructing the tree. In the context of a regression tree, the deviance is simply the sum of squared errors for the tree. We now plot the tree.

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

The variable `lstat` measures the percentage of individuals with lower socioeconomic status, while the variable `rm` corresponds to the average number of rooms. The tree indicates that larger values of `rm`, or lower values of `lstat`, correspond to more expensive houses. For example, the tree predicts a median house price of $45,400 for homes in census tracts in which `rm >= 7.553`.

It is worth noting that we could have fit a much bigger tree, by passing `control = tree.control(nobs = length(train), mindev = 0)` into the `tree()` function.

Now we use the `cv.tree()` function to see whether pruning the tree will improve performance.

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

In this case, the most complex tree under consideration is selected by cross-validation. However, if we wish to prune the tree, we could do so as follows, using the `prune.tree()` function:

`prune.tree()`

```
> 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.

```
> yhat <- predict(tree.boston, newdata = Boston[-train, ])
> boston.test <- Boston[-train, "medv"]
> plot(yhat, boston.test)
> abline(0, 1)
> mean((yhat - boston.test)^2)
[1] 35.29
```

In other words, the test set MSE associated with the regression tree is 35.29. The square root of the MSE is therefore around 5.941, indicating that this model leads to test predictions that are (on average) within approximately $5,941 of the true median home value for the census tract.

8.3.3 Bagging and Random Forests

Here we apply bagging and random forests to the `Boston` data, using the `randomForest` package in `R`. The exact results obtained in this section may depend on the version of `R` and the version of the `randomForest` package installed on your computer. Recall that bagging is simply a special case of a random forest with $m = p$. Therefore, the `randomForest()` function can be used to perform both random forests and bagging. We perform bagging as follows:

`randomForest()`

```
> library(randomForest)
> 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: 11.40
                    % Var explained: 85.17
```

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. How well does this bagged model perform on the test set?

```
> yhat.bag <- predict(bag.boston, newdata = Boston[-train, ])
> plot(yhat.bag, boston.test)
> abline(0, 1)
> mean((yhat.bag - boston.test)^2)
[1] 23.42
```

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. We could change the number of trees grown by `randomForest()` using the `ntree` argument:

```
> bag.boston <- randomForest(medv ~ ., data = Boston,
    subset = train, mtry = 12, ntree = 25)
> yhat.bag <- predict(bag.boston, newdata = Boston[-train, ])
> mean((yhat.bag - boston.test)^2)
[1] 25.75
```

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`.

```
> 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] 20.07
```

The test set MSE is 20.07; this indicates that random forests yielded an improvement over bagging in this case.

Using the `importance()` function, we can view the importance of each variable.

`importance()`

```
> importance(rf.boston)
        %IncMSE IncNodePurity
crim     19.436       1070.42
```

```
zn         3.092         82.19
indus      6.141        590.10
chas       1.370         36.70
nox       13.263        859.97
rm        35.095       8270.34
age       15.145        634.31
dis        9.164        684.88
rad        4.794         83.19
tax        4.411        292.21
ptratio    8.613        902.20
lstat     28.725       5813.05
```

Two measures of variable importance are reported. The first is based upon the mean decrease of accuracy in predictions on the out of bag samples when a given variable is permuted. The second is a measure of the total decrease in node impurity that results from splits over that variable, averaged over all trees (this was plotted in Figure 8.9). In the case of regression trees, the node impurity is measured by the training RSS, and for classification trees by the deviance. Plots of these importance measures can be produced using the `varImpPlot()` function.

`varImpPlot()`

```
> varImpPlot(rf.boston)
```

The results indicate that across all of the trees considered in the random forest, the wealth of the community (`lstat`) and the house size (`rm`) are by far the two most important variables.

8.3.4 Boosting

Here we use the `gbm` package, and within it the `gbm()` function, to fit boosted regression trees to the `Boston` data set. We run `gbm()` with the option `distribution = "gaussian"` since this is a regression problem; if it were a binary classification problem, we would use `distribution = "bernoulli"`. The argument `n.trees = 5000` indicates that we want 5000 trees, and the option `interaction.depth = 4` limits the depth of each tree.

`gbm()`

```
> library(gbm)
> set.seed(1)
> boost.boston <- gbm(medv ~ ., data = Boston[train, ],
    distribution = "gaussian", n.trees = 5000,
    interaction.depth = 4)
```

The `summary()` function produces a relative influence plot and also outputs the relative influence statistics.

```
> summary(boost.boston)
            var rel.inf
rm           rm  44.482
lstat     lstat  32.703
crim       crim   4.851
dis         dis   4.487
nox         nox   3.752
```

```
age          age   3.198
ptratio ptratio   2.814
tax          tax   1.544
indus      indus   1.034
rad          rad   0.876
zn            zn   0.162
chas        chas   0.097
```

We see that `lstat` and `rm` are by far the most important variables. We can also produce *partial dependence plots* for these two variables. These plots illustrate the marginal effect of the selected variables on the response after *integrating* out the other variables. In this case, as we might expect, median house prices are increasing with `rm` and decreasing with `lstat`.

partial dependence plot

```
> plot(boost.boston, i = "rm")
> plot(boost.boston, i = "lstat")
```

We now use the boosted model to predict `medv` on the test set:

```
> yhat.boost <- predict(boost.boston,
    newdata = Boston[-train, ], n.trees = 5000)
> mean((yhat.boost - boston.test)^2)
[1] 18.39
```

The test MSE obtained is 18.39: this is superior to the test MSE of random forests and bagging. If we want to, we can perform boosting with a different value of the shrinkage parameter λ in (8.10). The default value is 0.001, but this is easily modified. Here we take $\lambda = 0.2$.

```
> boost.boston <- gbm(medv ~ ., data = Boston[train, ],
    distribution = "gaussian", n.trees = 5000,
    interaction.depth = 4, shrinkage = 0.2, verbose = F)
> yhat.boost <- predict(boost.boston,
    newdata = Boston[-train, ], n.trees = 5000)
> mean((yhat.boost - boston.test)^2)
[1] 16.55
```

In this case, using $\lambda = 0.2$ leads to a lower test MSE than $\lambda = 0.001$.

8.3.5 Bayesian Additive Regression Trees

In this section we use the `BART` package, and within it the `gbart()` function, to fit a Bayesian additive regression tree model to the `Boston` housing data set. The `gbart()` function is designed for quantitative outcome variables. For binary outcomes, `lbart()` and `pbart()` are available.

`gbart()`

`lbart()`
`pbart()`

To run the `gbart()` function, we must first create matrices of predictors for the training and test data. We run BART with default settings.

```
> library(BART)
> x <- Boston[, 1:12]
> y <- Boston[, "medv"]
> xtrain <- x[train, ]
> ytrain <- y[train]
```

```
> xtest <- x[-train, ]
> ytest <- y[-train]
> set.seed(1)
> bartfit <- gbart(xtrain, ytrain, x.test = xtest)
```

Next we compute the test error.

```
> yhat.bart <- bartfit$yhat.test.mean
> mean((ytest - yhat.bart)^2)
[1] 15.95
```

On this data set, the test error of BART is lower than the test error of random forests and boosting.

Now we can check how many times each variable appeared in the collection of trees.

```
> ord <- order(bartfit$varcount.mean, decreasing = T)
> bartfit$varcount.mean[ord]
    nox   lstat     tax     rad      rm   indus    chas ptratio
  22.95   21.33   21.25   20.78   19.89   19.82   19.05   18.98
    age      zn     dis    crim
  18.27   15.95   14.46   11.01
```

8.4 Exercises

Conceptual

1. Draw an example (of your own invention) of a partition of two-dimensional feature space that could result from recursive binary splitting. Your example should contain at least six regions. Draw a decision tree corresponding to this partition. Be sure to label all aspects of your figures, including the regions $R_1, R_2, \ldots$, the cutpoints $t_1, t_2, \ldots$, and so forth.

   Hint: Your result should look something like Figures 8.1 and 8.2.

2. It is mentioned in Section 8.2.3 that boosting using depth-one trees (or *stumps*) leads to an *additive* model: that is, a model of the form

$$f(X) = \sum_{j=1}^{p} f_j(X_j).$$

   Explain why this is the case. You can begin with (8.12) in Algorithm 8.2.

3. Consider the Gini index, classification error, and entropy in a simple classification setting with two classes. Create a single plot that displays each of these quantities as a function of $\hat{p}_{m1}$. The x-axis should