Unit 4 Lecture 1: Decision Trees

October 27, 2022

Today, we will be using the `rpart` package to fit regression and classification trees (and the `rpart.plot` package to plot them).

First, let's load some libraries:

```
library(rpart)        # for fitting decision trees
library(rpart.plot)   # for plotting decision trees
library(tidyverse)    # for everything else
```

Regression trees

We will be using the `Hitters` data from the `ISLR2` package. Let's take a look:

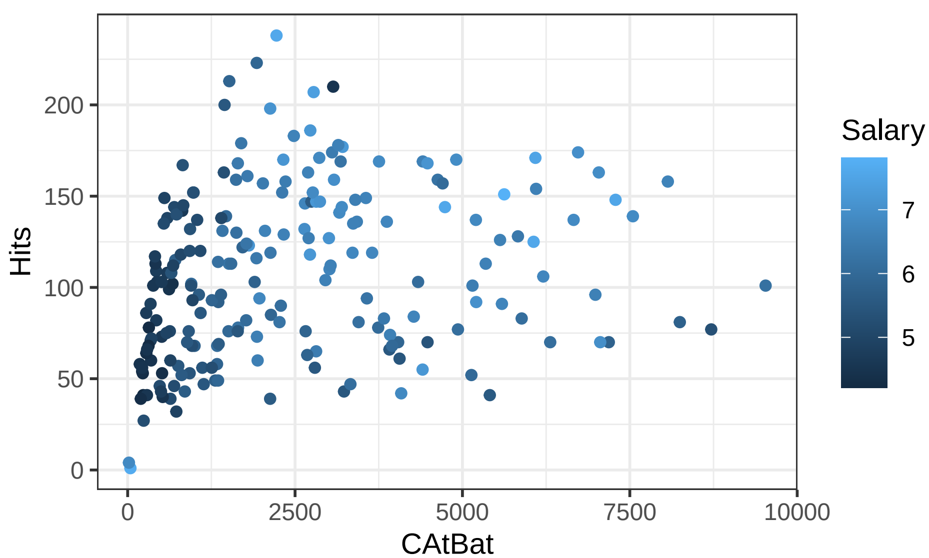
```
hitters_data <- read_csv("hitters-data.csv")
```

Let's split into train/test as usual:

```
set.seed(1) # set seed for reproducibility
train_samples <- sample(1:nrow(hitters_data), round(0.8 * nrow(hitters_data)))
hitters_train <- hitters_data %>% filter(row_number() %in% train_samples)
hitters_test <- hitters_data %>% filter(!(row_number() %in% train_samples))
```

Before actually building the tree, let's look at how `Salary` depends on a couple important predictors: `CAtBat` and `Hits`:

```
hitters_train %>%
  ggplot(aes(x = CAtBat, y = Hits, colour = Salary)) +
  geom_point()
```



By eye, what split point on what feature would make sense to separate players with high salaries from players with low salaries?

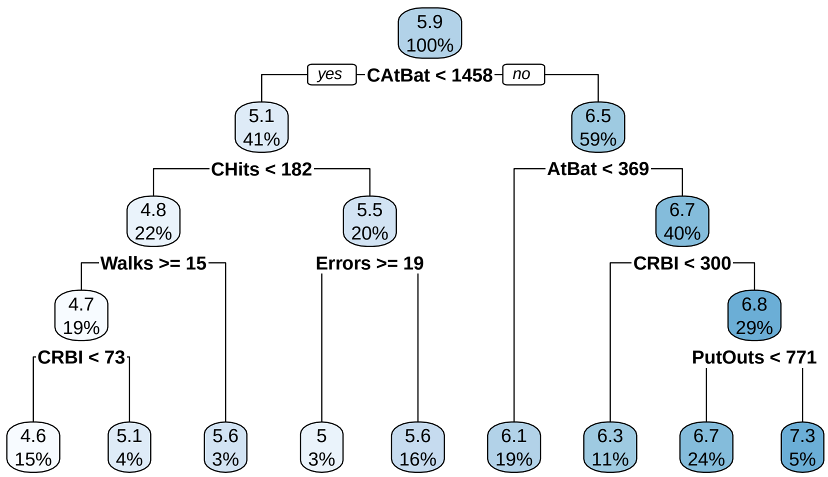
Fitting and plotting a regression tree

Next, let's actually run the regression tree. The syntax is essentially the same as `lm`, so we get to use the nice formula notation again:

```
tree_fit <- rpart(Salary ~ ., data = hitters_train)
```

We can plot the resulting tree using `rpart.plot`:

```
rpart.plot(tree_fit)
```



Does the first split point match what we predicted above?

We can get a text summary of the tree as follows:

```
tree_fit
```

```
## n= 210
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 210 160.2491000 5.915267
##    2) CAtBat< 1458 87  31.6754900 5.132687
##      4) CHits< 182 46  16.9359300 4.810335
##        8) Walks>=14.5 39   3.5486600 4.675338
##         16) CRBI< 72.5 31   1.7413860 4.571094 *
##         17) CRBI>=72.5 8   0.1650204 5.079285 *
##        9) Walks< 14.5 7   8.7166710 5.562462 *
##      5) CHits>=182 41   4.5968600 5.494350
##       10) Errors>=18.5 7   0.1801028 5.022313 *
##       11) Errors< 18.5 34   2.5359020 5.591534 *
##    3) CAtBat>=1458 123  37.6052300 6.468799
##      6) AtBat< 369 39   7.9199380 6.056463 *
##      7) AtBat>=369 84  19.9758800 6.660241
##       14) CRBI< 300 24   5.0468900 6.258952 *
##       15) CRBI>=300 60   9.5182870 6.820756
##         30) PutOuts< 771 50   6.1657560 6.730722 *
##         31) PutOuts>=771 10   0.9207013 7.270926 *
```

The tree fit object has several other useful fields, including `variable.importance`:

```
tree_fit$variable.importance
```

```
##      CAtBat      CRuns      CHits       CRBI     CWalks      Years
## 105.4972507 103.1909930 100.7612160  89.5112474  88.4443594  66.9324667
##       AtBat       Hits      Walks       Runs        RBI    PutOuts
##  13.1994577  11.3824932   9.1518716   8.4746301   5.9750242   3.9255866
##      CHmRun     Errors      HmRun    Assists
##   2.6045311   1.8808557   0.8211271   0.8060810
```

Controlling the complexity of the fit

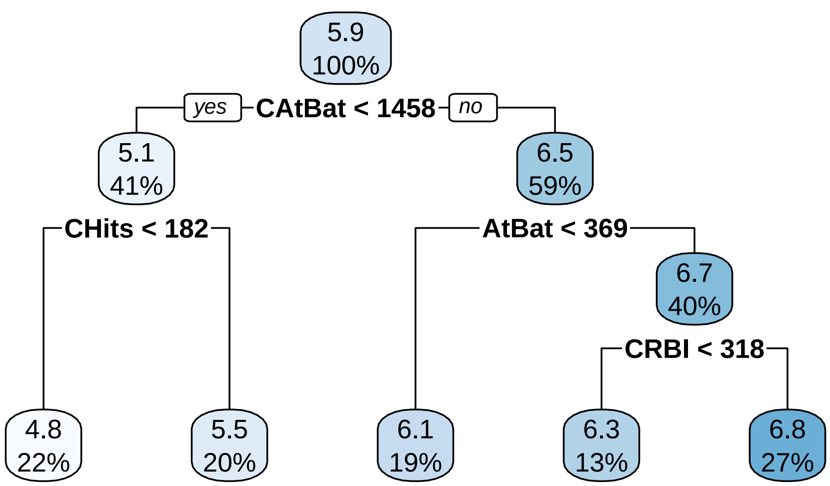
The `control` argument of `rpart` can be specified to control how far down the tree is fit. In particular, the default for `control` is

```
# this code is not meant to be run
control <- rpart.control(minsplit = 20, minbucket = round(minsplit / 3))
```

Here, `minsplit` is the minimum number of observations that must exist in a node in order for a split to be attempted, and `minbucket` is the minimum number of observations in any terminal (i.e. leaf) node. The larger these numbers, the fewer nodes there will be in the tree.

Let's see what happens when we crank `minsplit` up to 80:

```
tree_fit_2 <- rpart(Salary ~ .,
  control = rpart.control(minsplit = 80),
  data = hitters_train
)
rpart.plot(tree_fit_2)
```



Making predictions and evaluating test error

As usual, we evaluate the performance of decision trees based on their test error. We can use the `predict` function to make predictions on our held-out test set for the two trees fitted above:

```
pred_1 <- predict(tree_fit, newdata = hitters_test)
pred_2 <- predict(tree_fit_2, newdata = hitters_test)
results <- tibble(Y = hitters_test$Salary, Y_hat_1 = pred_1, Y_hat_2 = pred_2)
results
```

```
## # A tibble: 53 x 3
```

```
##         Y Y_hat_1 Y_hat_2
##     <dbl>   <dbl>   <dbl>
##  1  6.21    6.73    6.84
##  2  4.52    4.57    4.81
##  3  4.25    4.57    4.81
##  4  4.32    5.56    4.81
##  5  6.24    6.06    6.06
##  6  4.61    4.57    4.81
##  7  6.66    7.27    6.84
##  8  6.77    6.73    6.84
##  9  5.62    6.06    6.06
## 10  6.75    6.73    6.84
## # ... with 43 more rows
```

We can then extract the RMSE of the two methods using `summarise`, as usual:

```
results %>% summarise(
  RMSE_1 = sqrt(mean((Y - Y_hat_1)^2)),
  RMSE_2 = sqrt(mean((Y - Y_hat_2)^2))
)
```

```
## # A tibble: 1 x 2
##   RMSE_1 RMSE_2
##    <dbl>  <dbl>
## 1  0.598  0.504
```

Which method performs better? Why might this be the case?

Classification trees

To illustrate classification trees, let's use the `Heart` data:

```
heart_data <- read_csv("heart-data.csv")
```

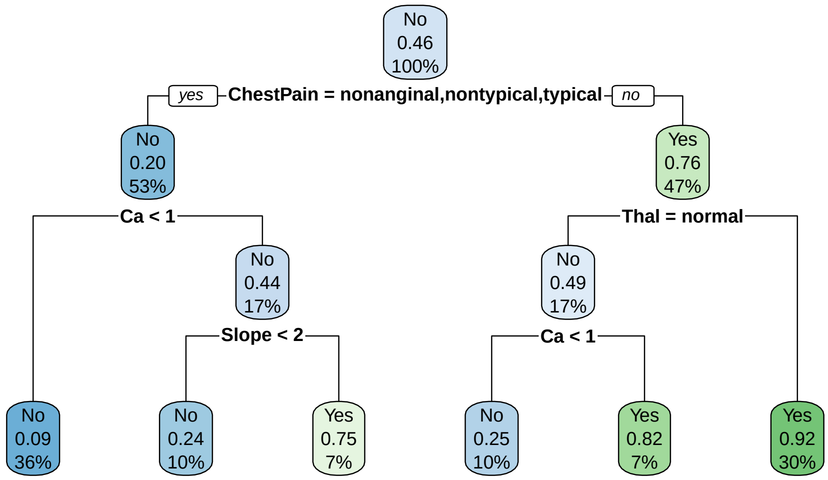
Again, let's split into train and test:

```
set.seed(1) # set seed for reproducibility
train_samples <- sample(1:nrow(heart_data), round(0.8 * nrow(heart_data)))
heart_train <- heart_data %>% filter(row_number() %in% train_samples)
heart_test <- heart_data %>% filter(!(row_number() %in% train_samples))
```

Now, we can fit a classification tree as follows:

```
tree_fit <- rpart(AHD ~ .,
  method = "class",                # classification
  parms = list(split = "gini"),    # Gini index for splitting
  data = heart_train
)

rpart.plot(tree_fit)
```



To make predictions, we can use `predict` as before:

```
pred <- predict(tree_fit, newdata = heart_test)
pred %>% head()
```

```
##            No        Yes
## 1 0.08333333 0.91666667
## 2 0.90909091 0.09090909
## 3 0.17647059 0.82352941
## 4 0.75000000 0.25000000
## 5 0.08333333 0.91666667
## 6 0.08333333 0.91666667
```

Note that by default, `predict` gives fitted probabilities for each class. We can either manually threshold these at 0.5 (or another value), or we can specify `type = "class"` to get the class predictions directly:

```
pred <- predict(tree_fit, newdata = heart_test, type = "class")
pred
```

```
##   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
## Yes  No Yes  No Yes Yes  No  No  No Yes Yes  No Yes  No Yes Yes Yes Yes Yes  No
##  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40
##  No Yes  No  No  No  No  No  No Yes Yes  No  No  No Yes  No  No Yes Yes Yes  No
##  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
##  No  No  No Yes  No  No Yes  No Yes  No  No  No  No  No Yes Yes  No  No  No  No
##  61
## Yes
## Levels: No Yes
```

We can then get the test misclassification error or the confusion matrix as usual:

```
# misclassification error
mean(pred != heart_test$AHD)
```

```
## [1] 0.1967213
```

```
# confusion matrix
table(pred, truth = heart_test$AHD)
```

```
##      truth
## pred  No Yes
##   No  29   7
```

```
##    Yes  5  20
```