Tree Based Methods: Regression Trees

Rebecca C. Steorts, Duke University

STA 325, Chapter 8 ISL

Agenda

- What are tree based methods?
- Regression trees
- Motivation using Hitter's data set
- How to interpret a regression tree
- How to build a regression tree
- Application

Basics of Decision (Predictions) Trees

- The general idea is that we will segment the predictor space into a number of simple regions.
- In order to make a prediction for a given observation, we typically use the mean of the training data in the region to which it belongs.
- Since the set of splitting rules used to segment the predictor space can be summarized by a tree such approaches are called decision tree methods.
- These methods are simple and useful for interpretation.

Basics of Decision Trees

- We want to predict a response or class Y from inputs $X_1, X_2, \ldots X_p$. We do this by growing a binary tree.
- At each internal node in the tree, we apply a test to one of the inputs, say X_i.
- Depending on the outcome of the test, we go to either the left or the right sub-branch of the tree.
- Eventually we come to a leaf node, where we make a prediction.
- This prediction aggregates or averages all the training data points which reach that leaf.

Basics of Decision Trees

- Decision trees can be applied to both regression and classification problems.
- We will first consider regression trees and then move onto classification trees.

Regression Trees

In order to motivate regression trees, we begin with a simple example.

Prediction of baseball player's salary

- Our motivation is to to predict a baseball player's Salary based on Years (the number of years that he has played in the major leagues) and Hits (the number of hits that he made in the previous year).
- We first remove observations that are missing Salary values, and log-transform Salary so that its distribution has more of a typical bell-shape.
- Recall that Salary is measured in thousands of dollars.

Prediction of baseball player's salary

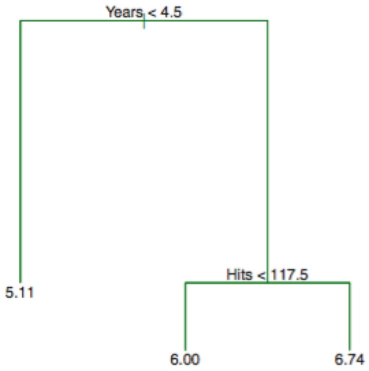
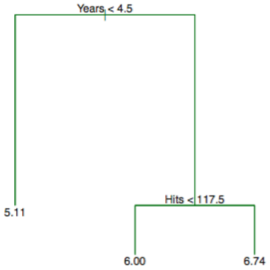


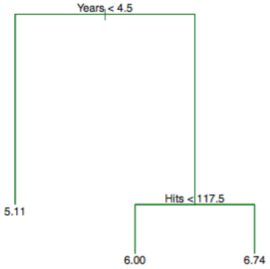
Figure 1: A regression tree for predicting the log salary of a baseball player, based on the number of years that he has played in the major leagues and the number of hits that he made in the previous year. At a given internal node, the label (of the form $X_j < t_k$) indicates the left-hand branch resulting from that split, and the right-hand branch corresponds to $X_j \geq t_k$.

What does the tree mean?



- The tree represents a series of splits starting at the top of the tree.
- The top split assigns observations having $Years < 4.5$ to the left branch.
- The predicted salary for these players is given by the mean response value for the players in the data set with $Years < 4.5$.
- For such players, the mean log salary is 5.107, and so we make a prediction of $e^{5.107}$ thousands of dollars, i.e. $165,174$

What does the tree mean?



- How would you interpret the rest (right branch) of the tree?

Prediction of baseball player's salary

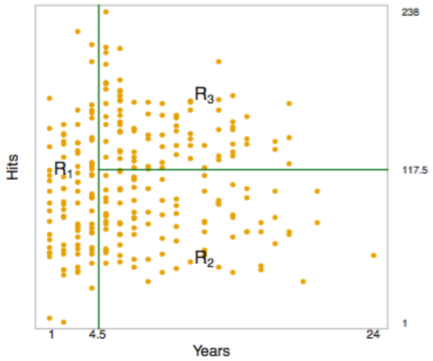


Figure 2: The three-region partition for the Hitters data set from the regression tree illustrated in Figure 2.

What do the regions mean?

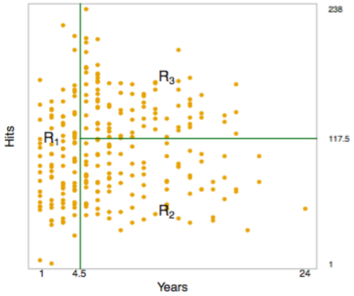


Figure 3: The three-region partition for the Hitters data set from the regression tree illustrated in Figure 2.

We can write these regions as the following:

1. $R_1 = X \mid Years < 4.5$
2. $R_2 = X \mid Years \geq 4.5, Hits < 117.5$
3. $R_3 = X \mid Years \geq 4.5, Hits \geq 117.5.$

Terminology

- In keeping with the tree analogy, the regions R_1, R_2, and R_3 are known as **terminal nodes** or **leaves** of the tree.
- As is the case for Figure 2, decision trees are typically drawn upside down, in the sense that the **leaves are at the bottom of the tree**.
- The points along the tree where the predictor space is split are referred to as **internal nodes**.
- In Figure 2, the two internal nodes are indicated by the text $\mathit{Years} < 4.5$ and $\mathit{Hits} < 117.5$.
- We refer to the segments of the trees that connect the nodes as **branches**.

Interpretation of Figure 2

- Years is the most important factor in determining Salary, and players with less experience earn lower salaries than more experienced players.
- Given that a player is less experienced, the number of hits that he made in the previous year seems to play little role in his salary.
- But among players who have been in the major leagues for five or more years, the number of hits made in the previous year does affect salary, and players who made more hits last year tend to have higher salaries.
- The regression tree shown in Figure 2 is likely an over-simplification of the true relationship between Hits, Years, and Salary, but it's a very nice easy interpretation over more complicated approaches.

How do we build the regression tree?

1. We divide the predictor space—that is, the set of possible values for $X_1, \ldots, X_p$—into J distinct and non-overlapping regions, $R_1, \ldots, R_J$.
2. For every observation that falls into the region R_j, we make the same prediction, which is simply the mean of the response values for the training observations in R_j.

Suppose that in Step 1, we obtain two regions and that the response mean of the training observations in the first region is 10, while the response mean in the second region is 20. Then for a given observation $X = x$, if $x \in R_1$, we will predict a value of 10, and if $x \in R_2$, we will predict a value of 20.

But how do we actually construct the regions?

Constructing the regions

- The regions in theory could have any shape.
- However, we choose to divide the predictor space into high-dimensional rectangles or boxes (for simplicity and ease of interpretation of the resulting predictive model).

Our goal is to find boxes $R_1, \ldots, R_J$ that minimize the RSS given by

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2, \tag{1}$$

where $\hat{y}_{R_j}$ is the mean response for the training observations within the jth box.

Issue with this construction

- Computationally infeasible to consider every possible partition of the feature space into J boxes.
- Thus, we take a top-down, greedy approach called *recursive binary splitting.*
    - Called top-down since it begins at the top of the tree (all observations below to a single region) and then successively splits the predictor space.
    - Each split is indicated via two new branches further down on the tree.
    - It is greedy since at each step of the tree building process, the best split is made at that particular split (rather than looking ahead and picking a split that will lead to a better tree in a future split).

How to build regression trees?

- One alternative is to the build the tree so long as the decrease in RSS due to each split exceeds a threshold (high).
- This results in smaller trees, however, this is problematic since a worthless split early on in the three might be followed by a very good split later on – that is, a split that leads to a large reduction in RSS later on.

How to build regression trees — Pruning

- A better strategy is to grow a very large tree T_o and then *prune* it back to obtain a subtree.
- How to we find the best subtree?
- We want to select a subtree that leads to the lowest test error rate.
- Given a subtree, we can estimate the test error rate using cross-validation (CV).
- Note that estimate the CV for every possible subtree would take a long time since there are many subtrees.
- Thus, we need a way to select a small set of subtrees to consider.

Pruning

Cost complexity pruning or weakest link pruning gives us a way to do just this!

Rather than looking at all possible subtrees, we consider a sequence of trees indexed by a nonnegative tuning parameter α.

Algorithm for Building a Regression Tree

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.
2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.
3. Use K-fold cross-validation to choose α. That is, divide the training observations into K folds. For each $k = 1, \ldots, K$:
    3.1 Repeat Steps 1 and 2 on all but the kth fold of the training data.
    3.2 Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.
    Average the results for each value of α, and pick α to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

Algorithm for Building a Regression Tree (continued)

- $|T|$ indicates the number of terminal nodes of the tree T
- R_m is the rectangle or box corresponding to the mth terminal node
- $\hat{y}_{R_m}$ is the predicted response associated with R_m
- α controls a trade-off between the subtree's complexity and its fit to the training data

For each value of α, there is a corresponding subtree $T \in T_o$ such that

$$\sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T| \tag{2}$$

is as small as possible.

Algorithm for Building a Regression Tree (continued)

This is equivalent to constraining the value of $|T|$, i.e.,

$$\min_{\hat{y}_{R_m}} \{\sum_i (y_i - \hat{y}_{R_m})^2\} \text{ subject to } |T| \leq c_\alpha.$$

Using Lagrange multipliers, we find that

$$\Delta_g = \sum_i (y_i - \hat{y}_{R_m})^2 + \lambda(|T| - c_\alpha) \tag{3}$$

Algorithm for Building a Regression Tree (continued)

We wish to find this $\min_{T,\lambda} \Delta_g$, which is a discrete optimization problem. However, since we're minimizing over T and λ this implies the location of the minimizing T doesn't depend on c_α. But each c_α will imply an optimal value of λ. As far as finding the best tree is concerned, we might as well, just pick a value of λ, and minimize

$$\Delta_{g'} = \sum_i (y_i - \hat{y}_{R_m})^2 + \lambda(|T|) \tag{4}$$

If we declare $\lambda = \alpha$, we have returned (2).

Some insights

- When $\alpha = 0$, then the subtree T will simply equal T_o, because then (2) just measures the training error.
- However, as $\alpha = 0$ increases, there is a price to pay for having a tree with many terminal nodes, and so (2) will be minimized for a smaller sub-tree.
- If you have seen the lasso, (2) is similar to it in the sense the ways the lasso controls the complexity of the linear model.
- As $\alpha = 0$ increases from 0 in (2), branches are pruned from the tree in a nested and predictable way (resulting in the whole sequence of subtrees as a function of $\alpha = 0$ is easy). We can select an α using a validation set of using cross-validation. This process is summarized in the algorithm above.

Application to Hitters data set

Let's return to growing a regression tree for the Hitters dataset.

Recall that we use the Hitters data set to predict a baseball players Salary based on Years (the number of years that he has played in the major leagues) and Hits (the number of hits that he made in the previous year).

There are several R packages for regression trees; the easiest one is called, simply, tree.

Task 1

Remove observations that are missing Hitters or Salary values, and log-transform Salary so that its distribution has more of a typical bell-shape.

Then build a regression tree.

Solution to Task 1

```r
library(ISLR)
library(tree)
attach(Hitters)
# remove NA values
Hitters <- na.omit(Hitters)
Salary <- na.omit(Salary)
# put salary on log scale and fit reg. tree
treefit <- tree(log(Salary) ~ Years + Hits, data=Hitters)
```

Solution to Task 2

Find the summary of the above regression tree and plot the regression tree. Explain your results.

Solution to Task 2

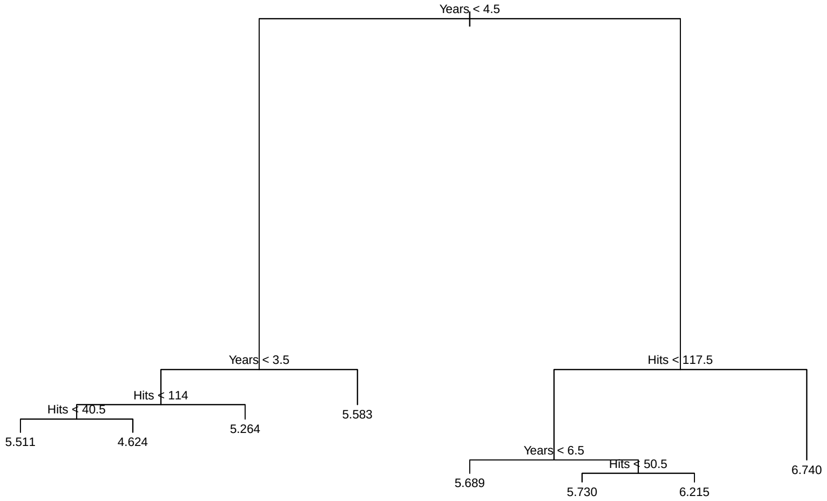
```r
summary(treefit)
```

```
##
## Regression tree:
## tree(formula = log(Salary) ~ Years + Hits, data = Hitte
## Number of terminal nodes:  8
## Residual mean deviance:  0.2708 = 69.06 / 255
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.2400 -0.2980 -0.0365  0.0000  0.3233  2.1520
```

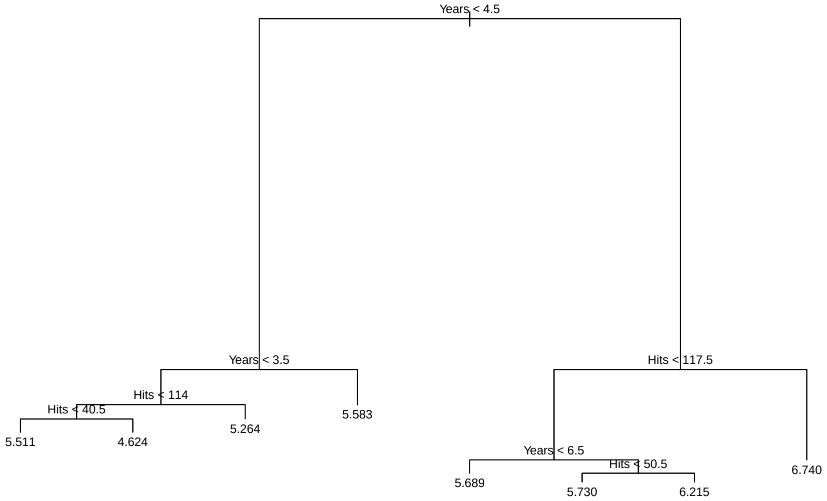
- There are 8 terminal nodes or leaves of the tree.
- Here “deviance” is just mean squared error; this gives us an RMS error of 0.27.

Solution to Task 2

```r
plot(treefit)
text(treefit,cex=0.75)
```

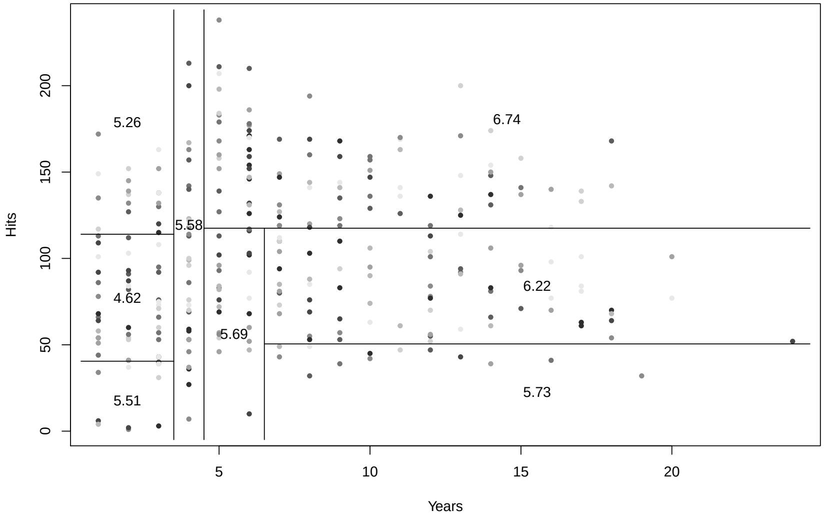


Solution to Task 2



Regression tree for predicting log salary from hits and years played. At each internal node, we ask the associated question, and go to the left child if the answer is "yes", to the right child if the answer is "no". Note that leaves are labeled with log salary; the plotting function isn't flexible enough, unfortunately, to apply transformations to the labels.

Task 3



How would you reproduce the above plot? (Hint: think about breaking salary up into quantiles. Reproducing the plot and giving the caption is the goal of Task 3.)

Cross-Validation and Pruning

The tree package contains functions prune.tree and cv.tree for pruning trees by cross-validation.

The function prune.tree takes a tree you fit by tree, and evaluates the error of the tree and various prunings of the tree, all the way down to the stump.

The evaluation can be done either on new data, if supplied, or on the training data (the default).

If you ask it for a particular size of tree, it gives you the best pruning of that size.

If you don't ask it for the best tree, it gives an object which shows the number of leaves in the pruned trees, and the error of each one.

This object can be plotted.

Pruning your tree

The prune.tree has an optional method argument.

The default is method="deviance", which fits by minimizing the mean squared error (for continuous responses) or the negative log likelihood (for discrete responses).

The function cv.tree does k-fold cross-validation (default is 10).

It requires as an argument a fitted tree, and a function which will take that tree and new data. By default, this function is prune.tree.

Generic code for pruning a tree

```
my.tree = tree(y ~ x1 + x2, data=my.data) # Fits tree
prune.tree(my.tree,best=5) # Returns best pruned tree
prune.tree(my.tree,best=5,newdata=test.set)
my.tree.seq = prune.tree(my.tree) # Sequence of pruned
# tree sizes/errors
plot(my.tree.seq) # Plots size vs. error
my.tree.seq$dev # Vector of error
# rates for prunings, in order
opt.trees = which(my.tree.seq$dev == min(my.tree.seq$dev))
# Positions of
  # optimal (with respect to error) trees
min(my.tree.seq$size[opt.trees])
# Size of smallest optimal tree
```

5-fold CV on Hitters Data Set

Let's create a training and test data set, fit a new tree on just the training data, and then evaluate how well the tree does on the held out training data.

Specifically, we will use 5-fold CV for evaluation.

Training and Test Set

```
fold <- floor(runif(nrow(Hitters),1,11))
Hitters$fold <- fold
## the test set is just the first fold
test.set <- Hitters[Hitters$fold == 1,]
##exclude the first fold from the data here
train.set <- Hitters[Hitters$fold != 1,]
my.tree <- tree(log(Salary) ~ Years
                + Hits,data=train.set, mindev=0.001)
```

Prune Tree on Training Data

```r
# Return best pruned tree with 5 leaves,
# evaluating error on training data
prune.tree(my.tree, best=5)
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 235 189.200 5.948
##   2) Years < 4.5 84  40.260 5.144
##     4) Years < 3.5 57  22.220 4.916
##       8) Hits < 114 38  16.700 4.742 *
##       9) Hits > 114 19   2.069 5.264 *
##     5) Years > 3.5 27   8.854 5.624 *
##   3) Years > 4.5 151  64.340 6.395
##     6) Hits < 117.5 75  24.130 6.017 *
##     7) Hits > 117.5 76  18.890 6.769 *
```

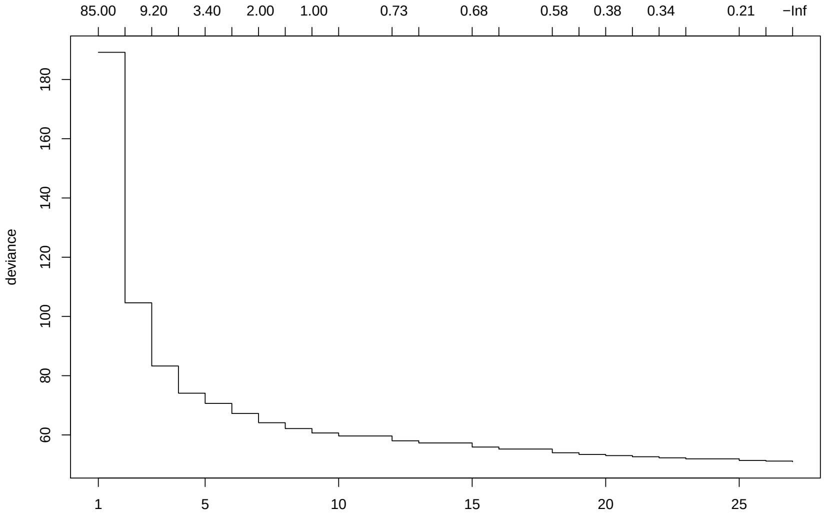
Prune Tree on Test Data

```r
# Ditto, but evaluates on test.set
prune.tree(my.tree,best=5,newdata=test.set)
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 235 189.200 5.948
##   2) Years < 4.5 84  40.260 5.144
##     4) Years < 3.5 57  22.220 4.916
##       8) Hits < 114 38  16.700 4.742 *
##       9) Hits > 114 19   2.069 5.264 *
##     5) Years > 3.5 27   8.854 5.624 *
##   3) Years > 4.5 151  64.340 6.395
##     6) Hits < 117.5 75  24.130 6.017 *
##     7) Hits > 117.5 76  18.890 6.769 *
```

Prune Tree on Test Data

```r
# Sequence of pruned tree sizes/errors
my.tree.seq = prune.tree(my.tree)
plot(my.tree.seq) # error versus plot size
```



Prune Tree on Test Data

```
# Vector of error rates
#for prunings, in order
my.tree.seq$dev
```

```
##  [1]  50.96518  51.16487  51.37464  51.92452  52.26719  52.63
##  [8]  53.39839  53.97898  55.23122  55.90812  57.28763  58.01
## [15]  60.66070  62.14591  64.11904  67.23910  70.63947  74.08
## [22] 104.59653 189.17362
```

Prune Tree on Test Data

```r
# Positions of
  # optimal (with respect to error) trees
opt.trees = which(my.tree.seq$dev == min(my.tree.seq$dev))
# Size of smallest optimal tree
(best.leaves = min(my.tree.seq$size[opt.trees]))
```

```
## [1] 27
```

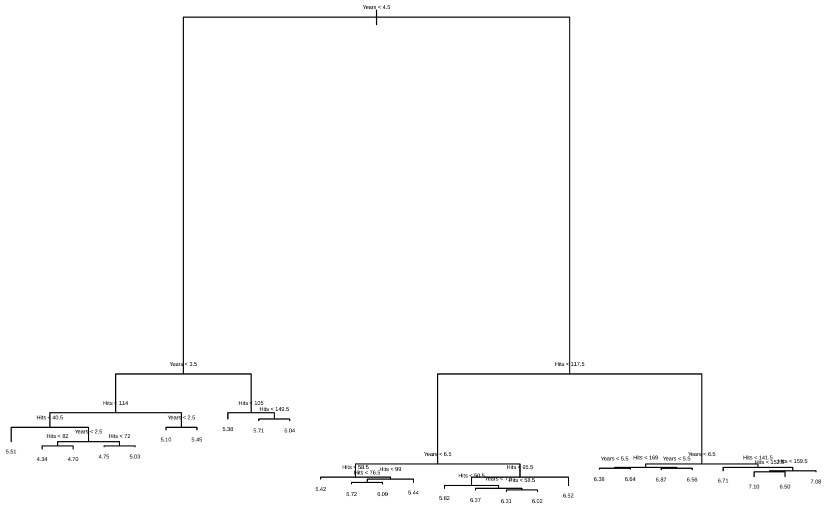
```r
my.tree.pruned = prune.tree(my.tree,best=best.leaves)
```

Task 4

Now plot the pruned tree and also the corresponding partition of regions for this tree. Interpret the pruned tree and the partition of the regions for the tree.

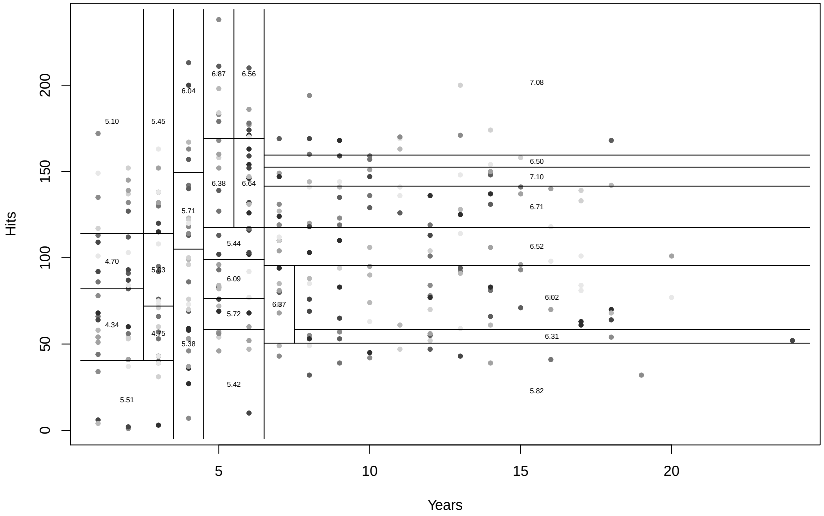
Solution to Task 4

```r
plot(my.tree.pruned)
text(my.tree.pruned,cex=0.3,digits=3)
```



Solution to Task 4

```
plot(Years,Hits,col=grey(10:2/11)[cut.salary],pch=20, xlab=
partition.tree(my.tree.pruned,ordvars=c("Years","Hits"), ad
```



Test

```r
my.tree.cv = cv.tree(my.tree)
cv.tree(my.tree,best=5)
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 235 189.200 5.948
##   2) Years < 4.5 84  40.260 5.144
##     4) Years < 3.5 57  22.220 4.916
##       8) Hits < 114 38  16.700 4.742 *
##       9) Hits > 114 19   2.069 5.264 *
##     5) Years > 3.5 27   8.854 5.624 *
##   3) Years > 4.5 151  64.340 6.395
##     6) Hits < 117.5 75  24.130 6.017 *
##     7) Hits > 117.5 76  18.890 6.769 *
```

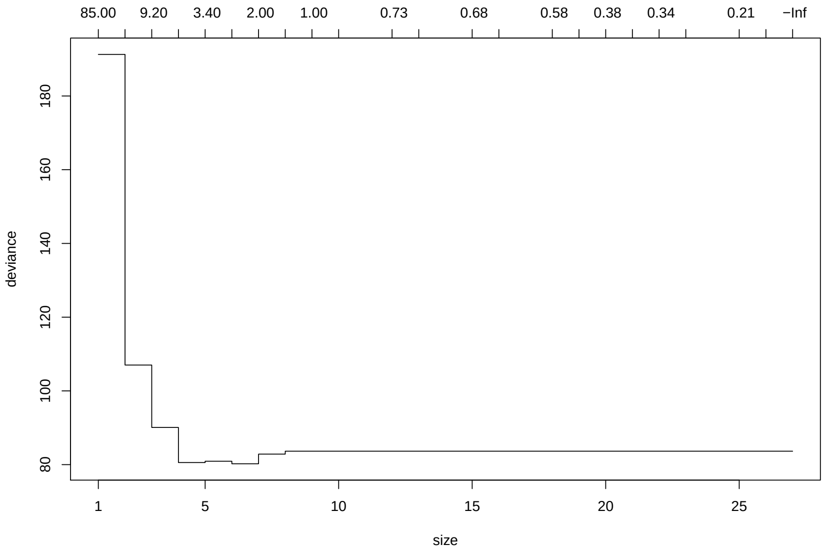
Test

```r
cv.tree(my.tree)
```

```
## $size
##  [1] 27 26 25 23 22 21 20 19 18 16 15 13 12 10  9  8  7  6  5  4  3  2  1
##
## $dev
##  [1]  79.08460  79.08460  79.08460  79.08460  79.08460  79.08460  79.08460
##  [8]  79.08460  79.08460  79.08460  79.08460  79.08460  79.08460  79.08460
## [15]  79.08460  79.08460  78.44413  81.69170  81.98365  81.95965  87.98783
## [22] 107.98663 190.43038
##
## $k
##  [1]       -Inf  0.1996820  0.2097749  0.2749388  0.3426721  0.3667108
##  [7]  0.3814724  0.3830205  0.5805901  0.6261173  0.6769023  0.6897523
## [13]  0.7270663  0.8134646  1.0190775  1.4852158  1.9731213  3.1200601
## [19]  3.4003728  3.4497897  9.1825804 21.3246890 84.5770933
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Test

```
plot(cv.tree(my.tree))
```


−Inf
deviance
size