Tree ~~Modles~~ Models

Vadim Sokolov

Spring 2021

Tree-Based Methods

- Tree models split the predictor space into a number of box (open or close) regions
- The set of splitting rules used to segment the predictor space can be summarized in a tree, we call it a decision-tree.
- Decision trees can be applied to both regression and classification problems.
- We first consider regression problems, and then move on to classification.

Lets start with a quick demo

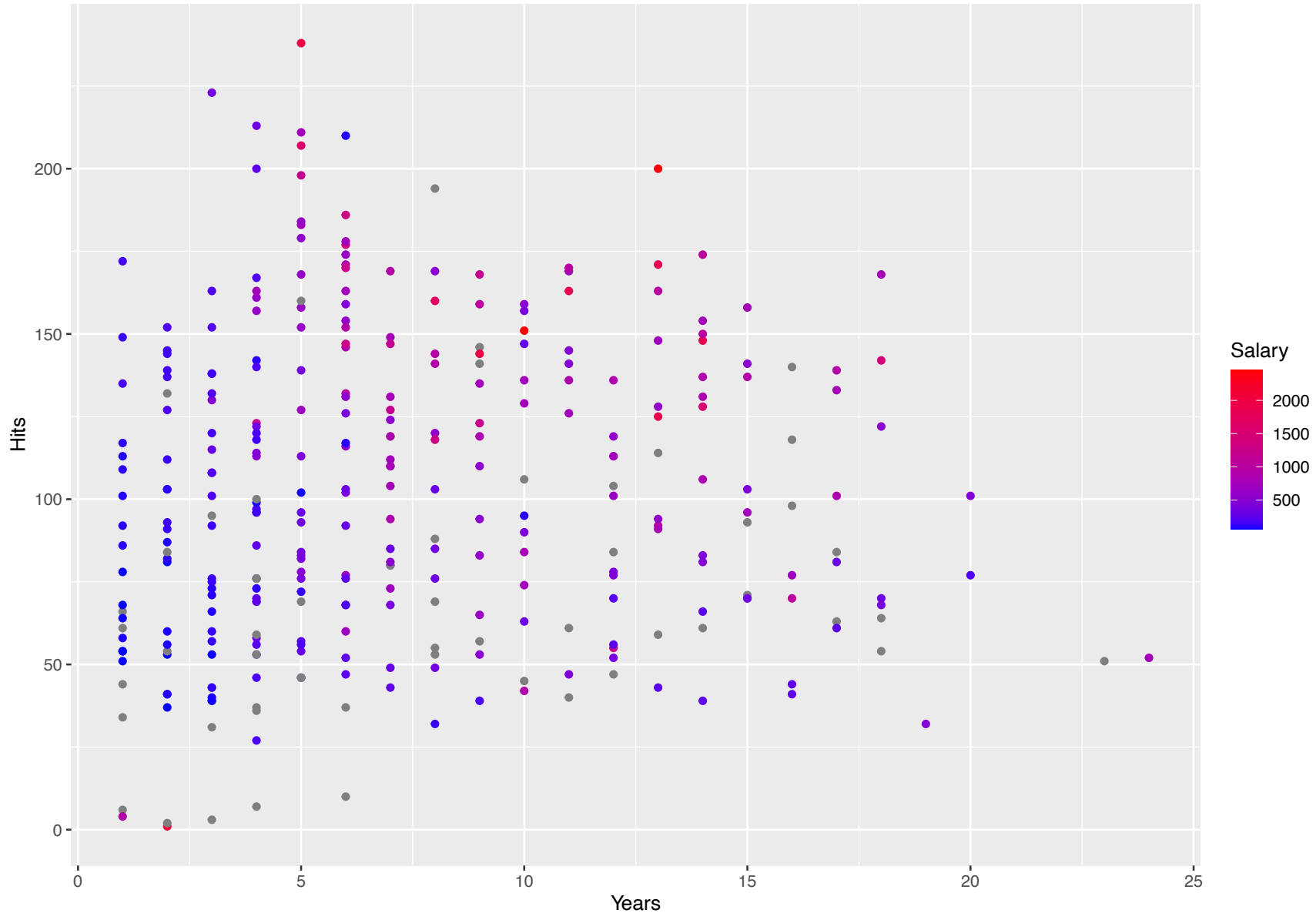
Does experience and performance effect the salary of a baseball player?

```
library(ISLR)
Hitters[1:7,1:7]
```

```
##                   AtBat Hits HmRun Runs RBI Walks Years
## -Andy Allanson      293   66     1   30  29    14     1
## -Alan Ashby         315   81     7   24  38    39    14
## -Alvin Davis        479  130    18   66  72    76     3
## -Andre Dawson       496  141    20   65  78    37    11
## -Andres Galarraga   321   87    10   39  42    30     2
## -Alfredo Griffin    594  169     4   74  51    35    11
## -Al Newman          185   37     1   23   8    21     2
```

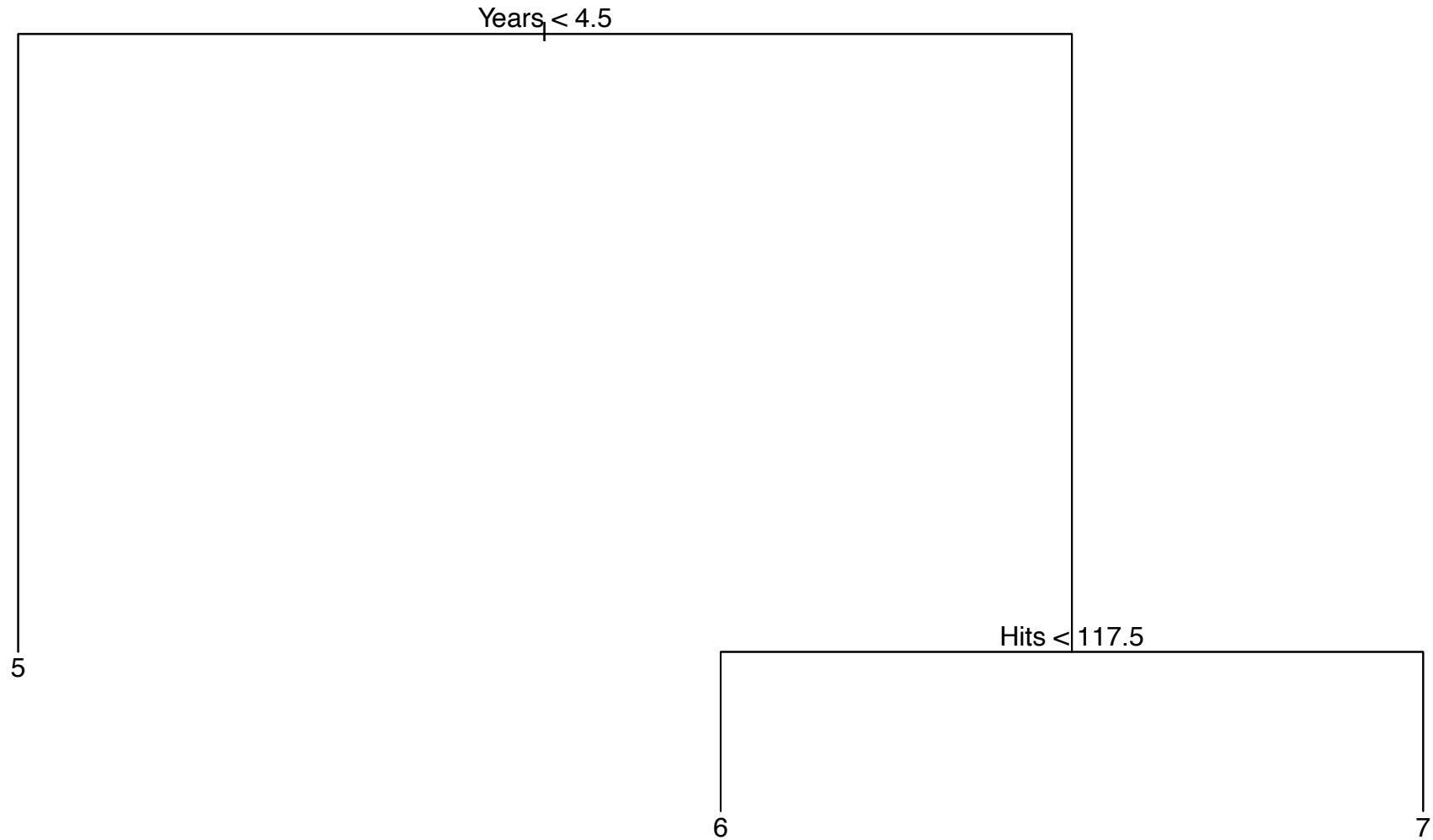
Lets plot the data

```
qplot(Years,Hits,data=Hitters, colour = Salary) +
  scale_color_gradient(low="blue", high="red")
```



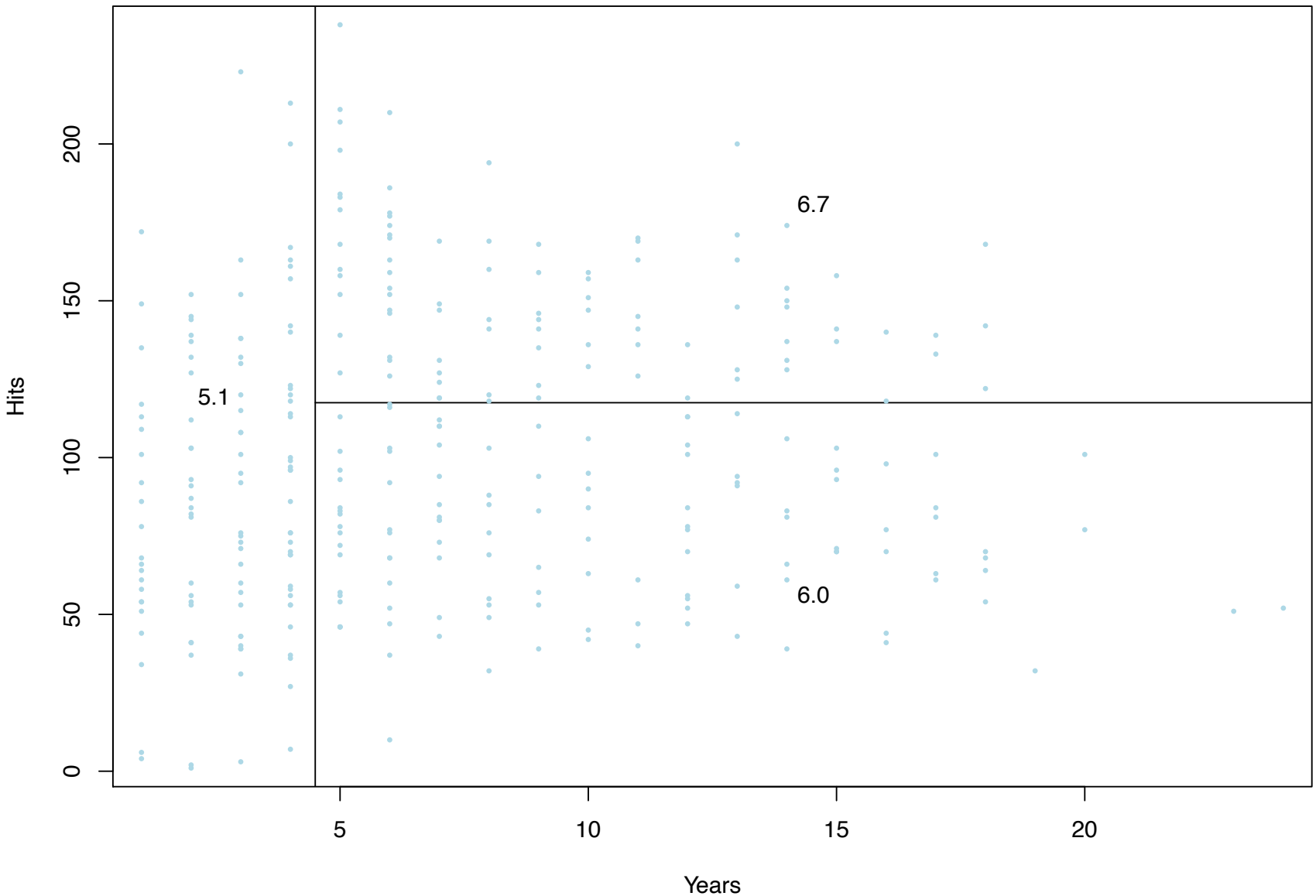
Tree Model

```
library(tree)
tree.hit = tree(log(Hitters$Salary)~Years+Hits,Hitters)
prune.hit=prune.tree(tree.hit,best=3)
plot(prune.hit)
text(prune.hit)
```

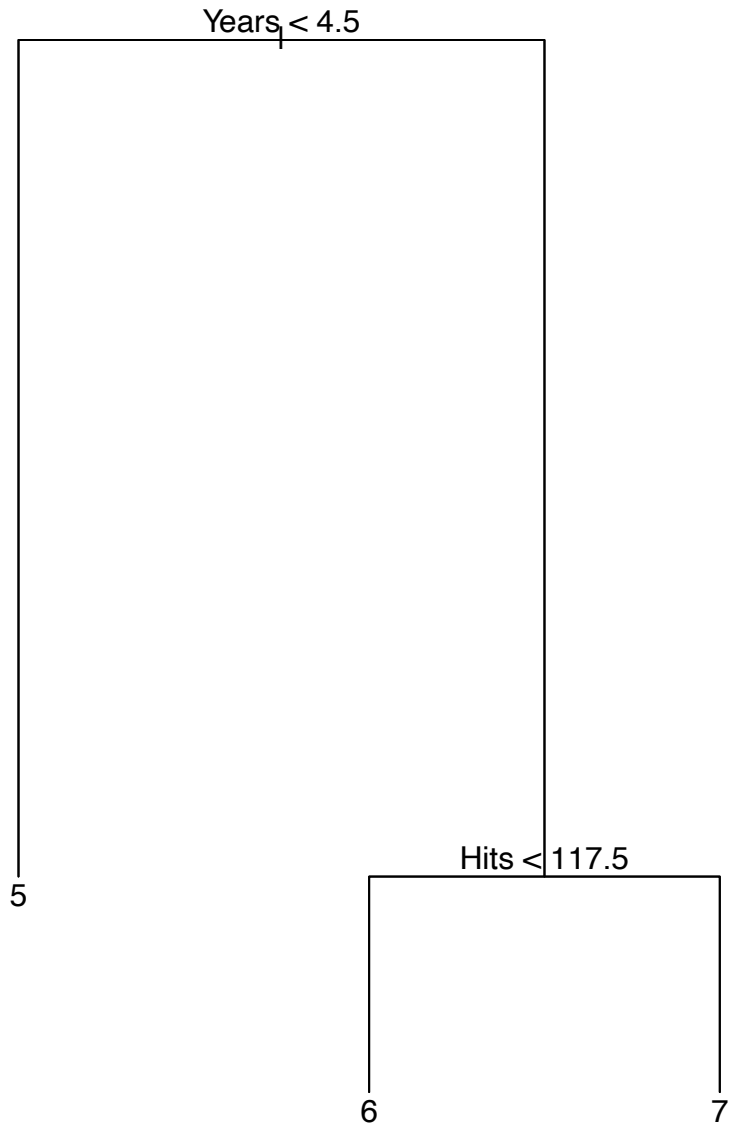


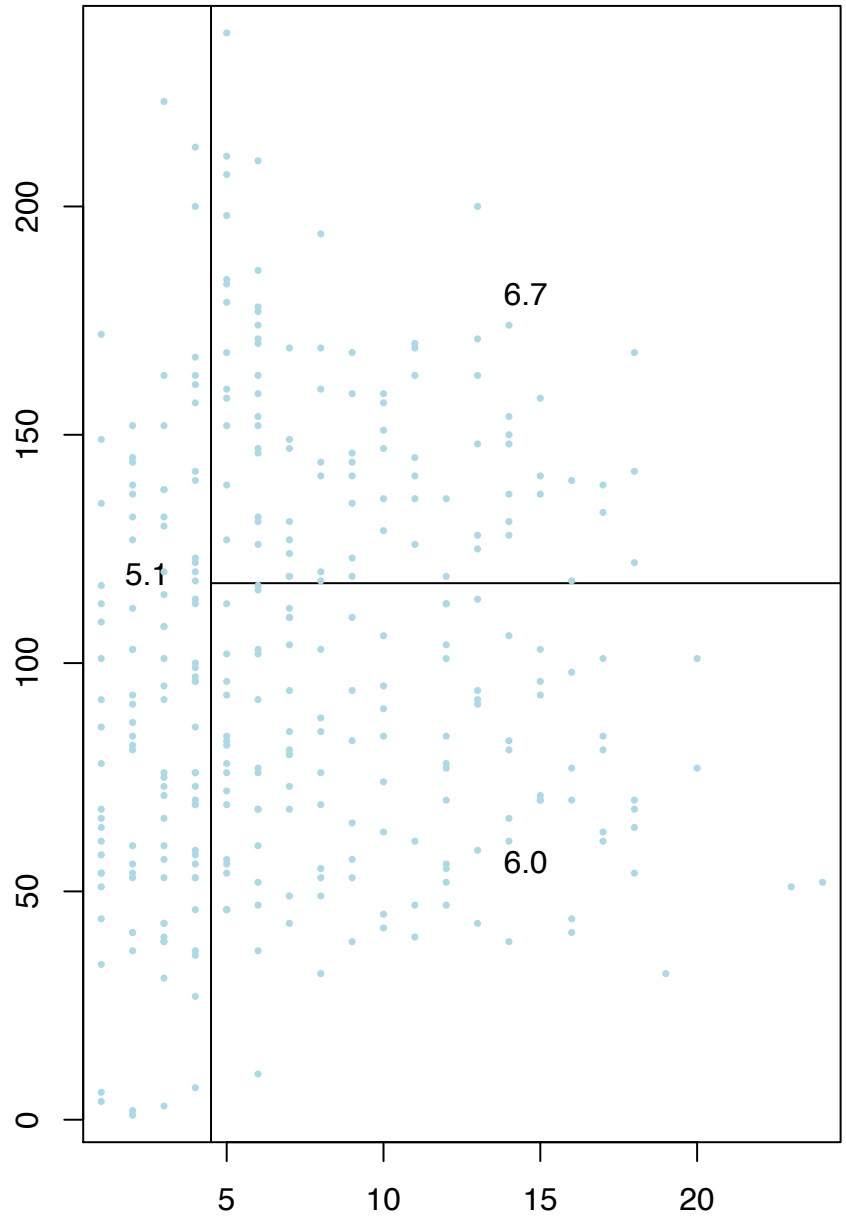
Each Terminal Node is a Region

```
partition.tree(prune.hit, label = "yval")
lines(Hitters$Years, Hitters$Hits, type='p', pch=16, cex=0.5,col="lightblue")
```



Side-by-Side





Prediction via Stratification of the Feature Space

We now discuss the process of building a regression tree. Roughly speaking, there are two steps.

1. We divide the predictor space-that is, the set of possible values for $x_1, x_2, \ldots, x_p$ - into J distinct and non-overlapping boxes, $R_1, R_2, ..., R_J$.
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.

$$f(x) = \sum_{j=1}^{J} \bar{y}_j I(x \in R_j)$$

$$\bar{y}_j = \text{Average}(y_i \mid x_i \in R_j)$$

Model Fitting

Thus the goal is to find regions that lead to minima of the Residual Sum of Squares (RSS)

$$\mathrm{RSS} = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \bar{y}_j)^2 \rightarrow \text{minimize}$$

Unfortunately, it is computationally infeasible (NP-hard problem) to consider every possible partition of the feature space into J boxes.

CART Algoritms

We can find a good approximate solution, using top-down approach

- All observations belong to a single region
- Successively splits the predictor space
- Each split creates two new brances
- It is a greedy (myopic) approach
- At each iteatoin we decide on: which variable j to split and split point s.

$$R_1(j,s) = \{x \mid x_j < s\} \text{ and } R_2(j,s) = \{x \mid x_j \geq s\},$$

thus, we seek to minimize (in case of regression tree)

$$\min_{j,s} \left[\sum_{i:x_i \in R_1} (y_i - \bar{y}_1)^2 + \sum_{i:x_i \in R_2} (y_i - \bar{y}_2)^2 \right]$$

Tree Prooning

- At one extreme end, we can have n regions, one for each observaiton
- At the other end, we can have one big region for the entire input space and then every prediction
- Both models can be used but usually the best one is in the middle.
- The number modesl is in between
- Number of regions (branches) controls the complexity of the model. We need to find a good size on the variance-bias scale
- A smaller tree with fewer splits (that is, fewer regions $R_1, ..., R_J$) might lead to lower variance and better interpretation at the cost of a little bias

Tree Prooning

How do we build a tree with "reasonable" number of branches?

- Keep building the tree until RSS stagnates
- Too short-sighted since a seemingly worthless split early on in the tree might be followed by a very good split. Can see large drop in RSS later in later iterations
- A better strategy is to grow a very large tree T_0, and then prune it back in order to obtain a subtree
- We can choose the size of the subtree using cross-validaiton.
- However there are exponential number of subtrees!

Tree Prooning

- Instead of condsidering all possible sub-trees, we will do cost complexity pruning - also known as weakest link pruning - gives
- We consider a sequence of trees indexed by a nonnegative tuning parameter α.

For each value of α there corresponds a subtree $T \subset T_0$ such that minimizes

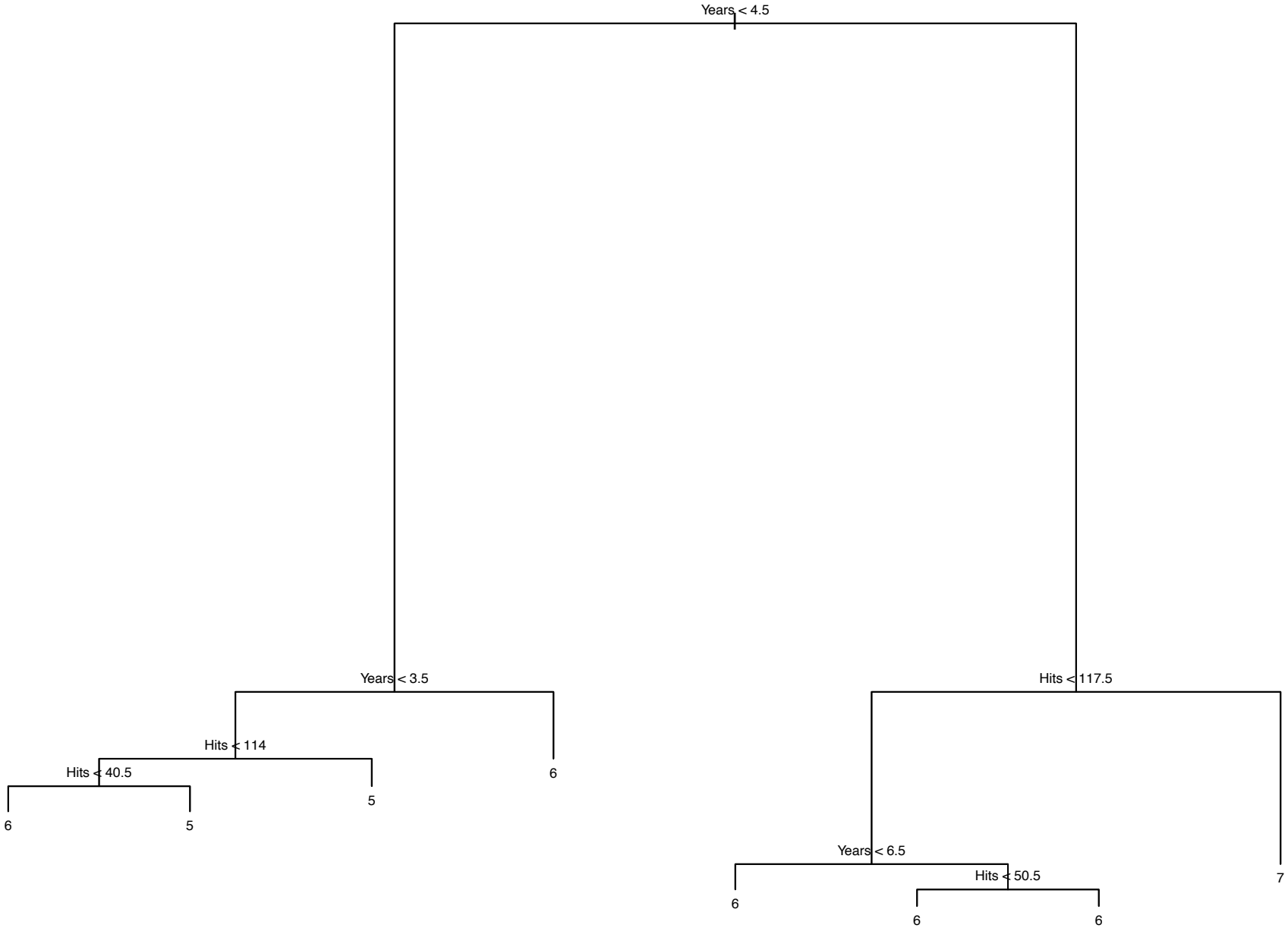
$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \bar{y}_m)^2 + \alpha |T|$$

Choosing the best subtree

- The tuning parameter α controls a trade-off between the subtree's complexity and its fit to the training data.
- As we increase α from zero, branches get pruned from the tree in a nested and predictable fashion, so obtaining the whole sequence of subtrees as a function of α is easy.
- We select an optimal value $\hat{\alpha}$ using cross-validation.
- We then return to the full data set and obtain the subtree corresponding to $\hat{\alpha}$.

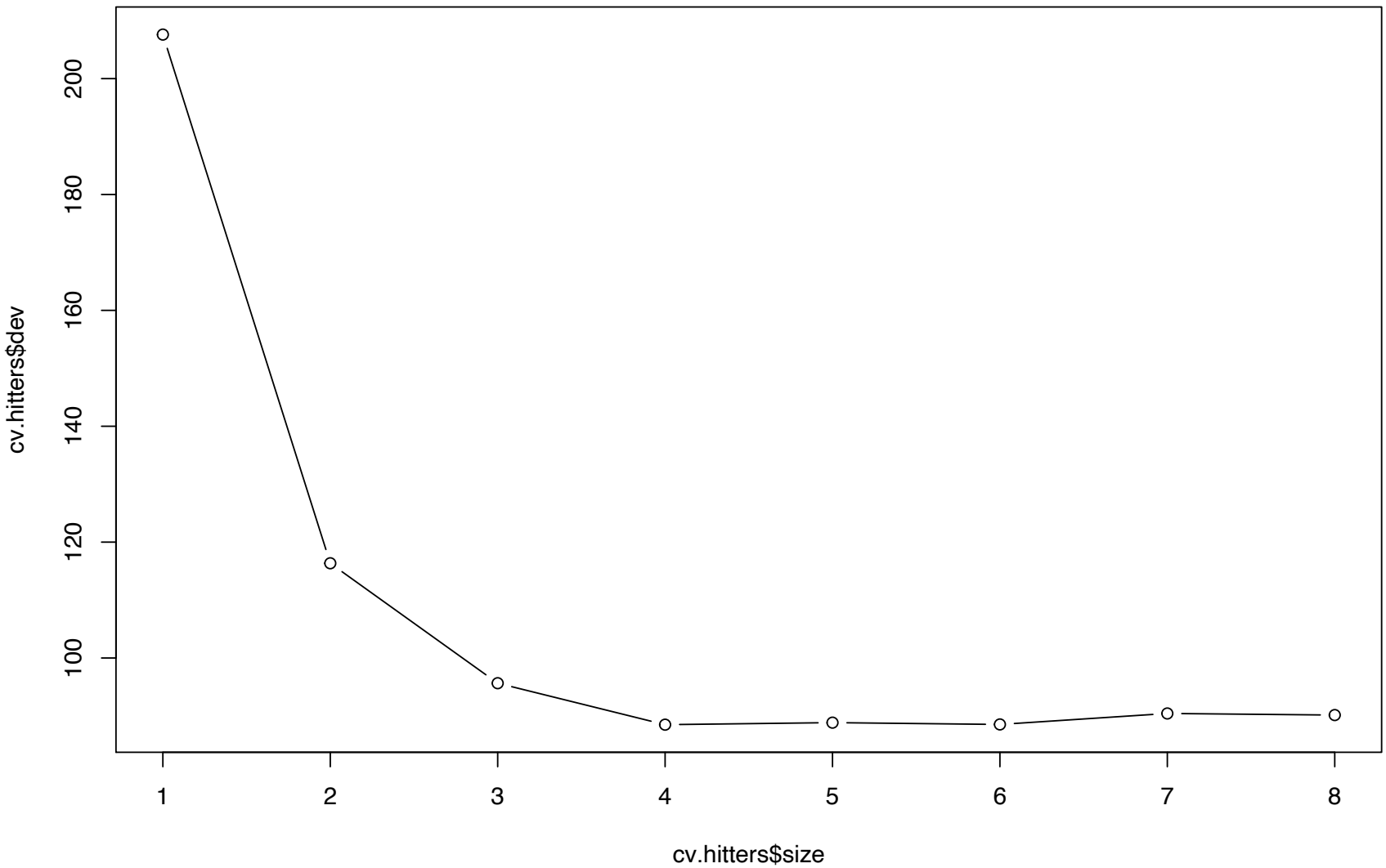
Back to Baseball Example

```
plot(tree.hit); text(tree.hit, cex=0.6)
```



Let's find the best tree

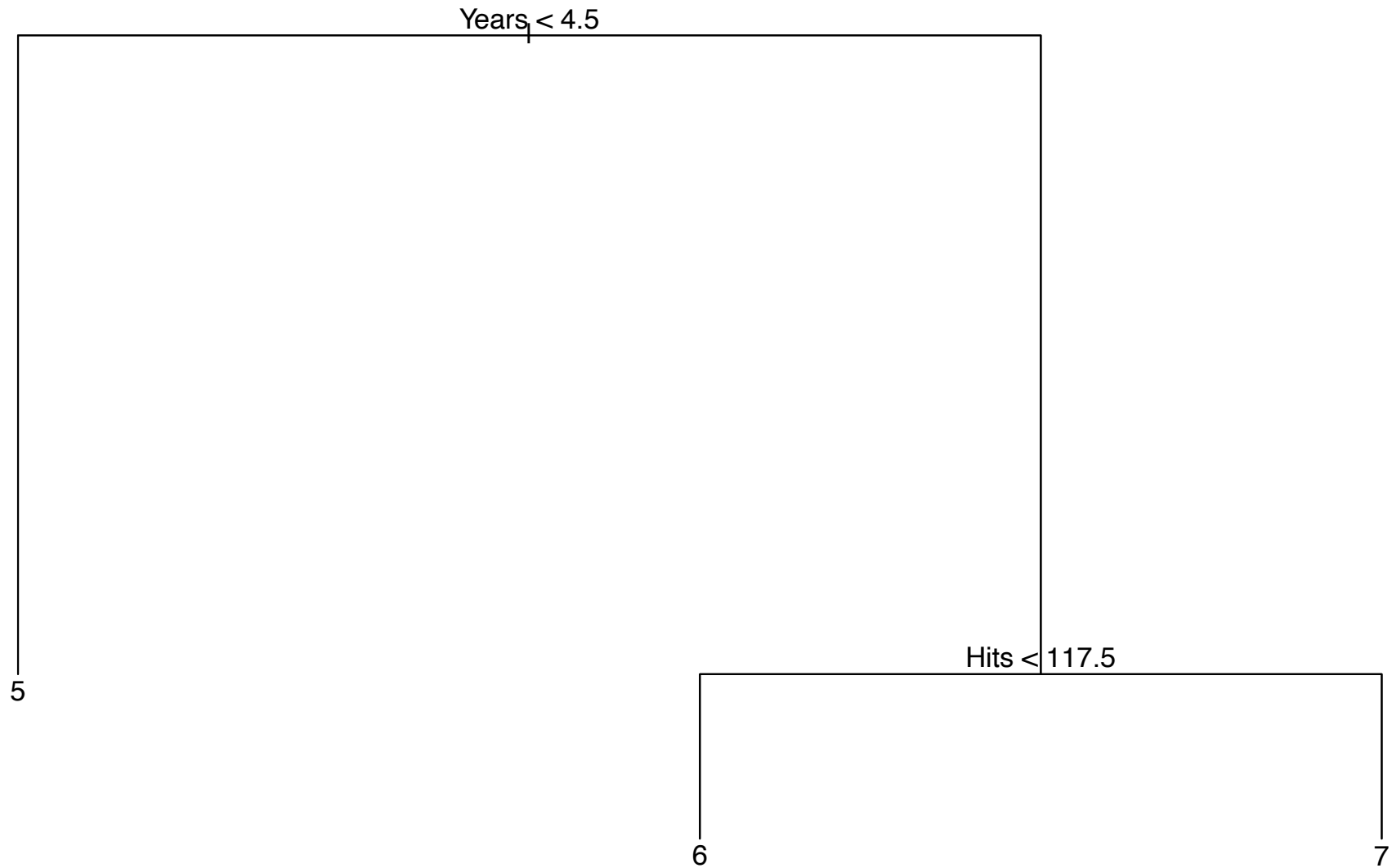
```
cv.hitters=cv.tree(tree.hit)
plot(cv.hitters$size,cv.hitters$dev,type="b")
```



Size of 3 seems good!

Let's find the best tree

```
prune.hit=prune.tree(tree.hit,best=3)
plot(prune.hit)
text(prune.hit)
```



Classification Trees

- A classification tree is very similar to a regression tree
- For prediciton, we use "majority vote": pick the most commonly occurring class in the region
- The task of growing a classification tree is quite similar to the task of growing a regression tree: recursive binary splitting
- Instead of RSS use classification error rate: the fraction of the observations in that region that do not belong to the most common class.

Some notatations

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$$

which is proportion of observations of class k in region m.

The classificaiton then done as follows

$$p_m = \max_k p_{mk}, \quad E_m = 1 - p_m$$

i.e the most frequent obsevation in region m

Then classification is done as follows

$$P(y = k) = \sum_{j=1}^{J} p_j I(x \in R_j)$$

Gini Index and Cross-Entropy

- I have 400 obsevations in each class (400,400)
- I create a tree with two region: (300,100) and (100,300)
- Say I have another tree: (200,400) and (200,0)
- In both cases misclassification rate is 0.25

Gini Index and Cross-Entropy

The Gini index:

$$G_m = \sum_{k=1}^{K} p_{mk}(1 - p_{mk})$$

- A variance across the K classes. - Takes on a small value if all of the p_{mk}'s are close to zero or one

An alternative to the Gini index is cross-entropy (a.k.a deviance), given by

$$D_m = -\sum_{k=1}^{K} p_{mk} \log p_{mk}$$

Near zero if the $p_m k$'s are all near zero or near one.

Gini index and the cross-entropy led to similar results.

Bostong Housing Example

```
library(MASS); data(Boston); attach(Boston)
head(Boston)
```

```
##     crim zn indus chas  nox  rm age dis rad tax ptratio black lstat medv
## 1 0.0063 18   2.3    0 0.54 6.6  65 4.1   1 296      15   397   5.0   24
## 2 0.0273  0   7.1    0 0.47 6.4  79 5.0   2 242      18   397   9.1   22
## 3 0.0273  0   7.1    0 0.47 7.2  61 5.0   2 242      18   393   4.0   35
## 4 0.0324  0   2.2    0 0.46 7.0  46 6.1   3 222      19   395   2.9   33
## 5 0.0691  0   2.2    0 0.46 7.1  54 6.1   3 222      19   397   5.3   36
## 6 0.0299  0   2.2    0 0.46 6.4  59 6.1   3 222      19   394   5.2   29
```

Bostong Housing

First we build a big tree

```
temp = tree(medv~lstat,data=Boston,mindev=.0001)
length(unique(temp$where)) # first big tree size
```

```
## [1] 73
```

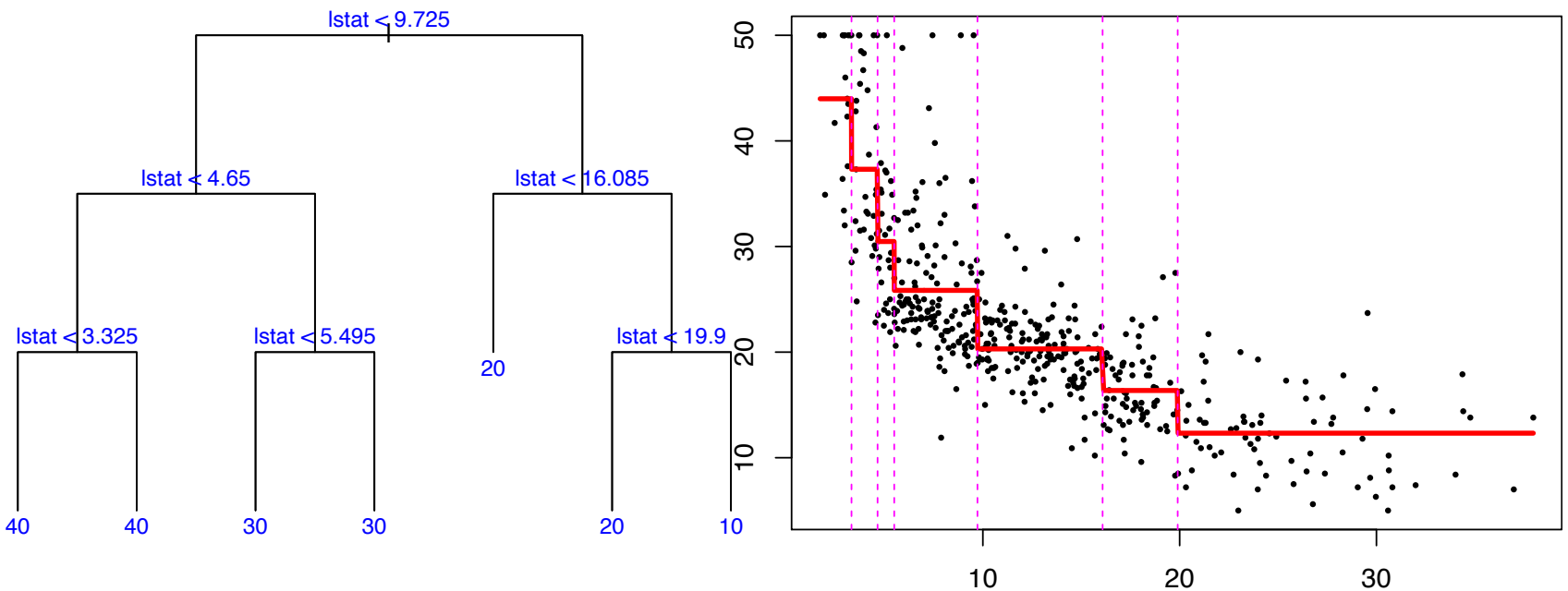
Then prune it down to one with 7 leaves

```
boston.tree=prune.tree(temp,best=7)
length(unique(boston.tree$where)) # pruned tree size
```

```
## [1] 7
```

Bostong Housing

```
plot(boston.tree,type="uniform") # first big tree
text(boston.tree,col="blue",label=c("yval"),cex=.8)
boston.fit = predict(boston.tree) #get training fitted values
plot(lstat,medv,cex=.5,pch=16) #plot data
oo=order(lstat)
lines(lstat[oo],boston.fit[oo],col='red',lwd=3) #step function fit
cvals=c(9.725,4.65,3.325,5.495,16.085,19.9) #cutpoints from tree
for(i in 1:length(cvals)) abline(v=cvals[i],col='magenta',lty=2) #cutpoints
```



Bostong Housing

Pick off `dis,lstat,medv`

```
df2=Boston[,c(8,13,14)]
print(names(df2))
```

```
## [1] "dis"   "lstat" "medv"
```

Build the big tree

```
temp = tree(medv~.,df2,mindev=.0001)
length(unique(temp$where)) #
```

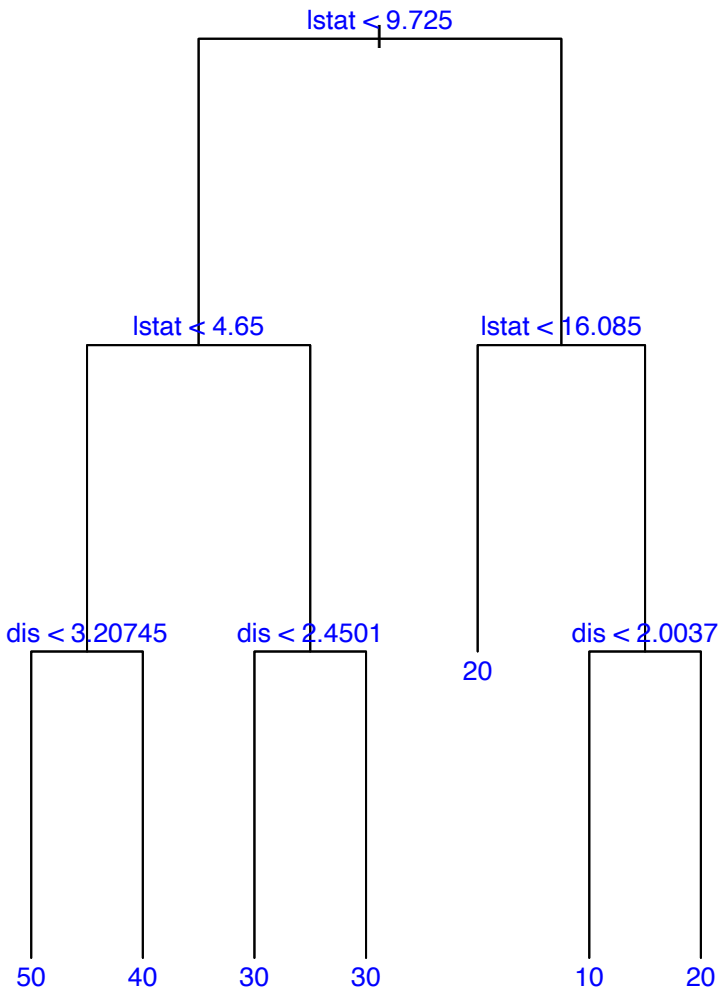
```
## [1] 74
```

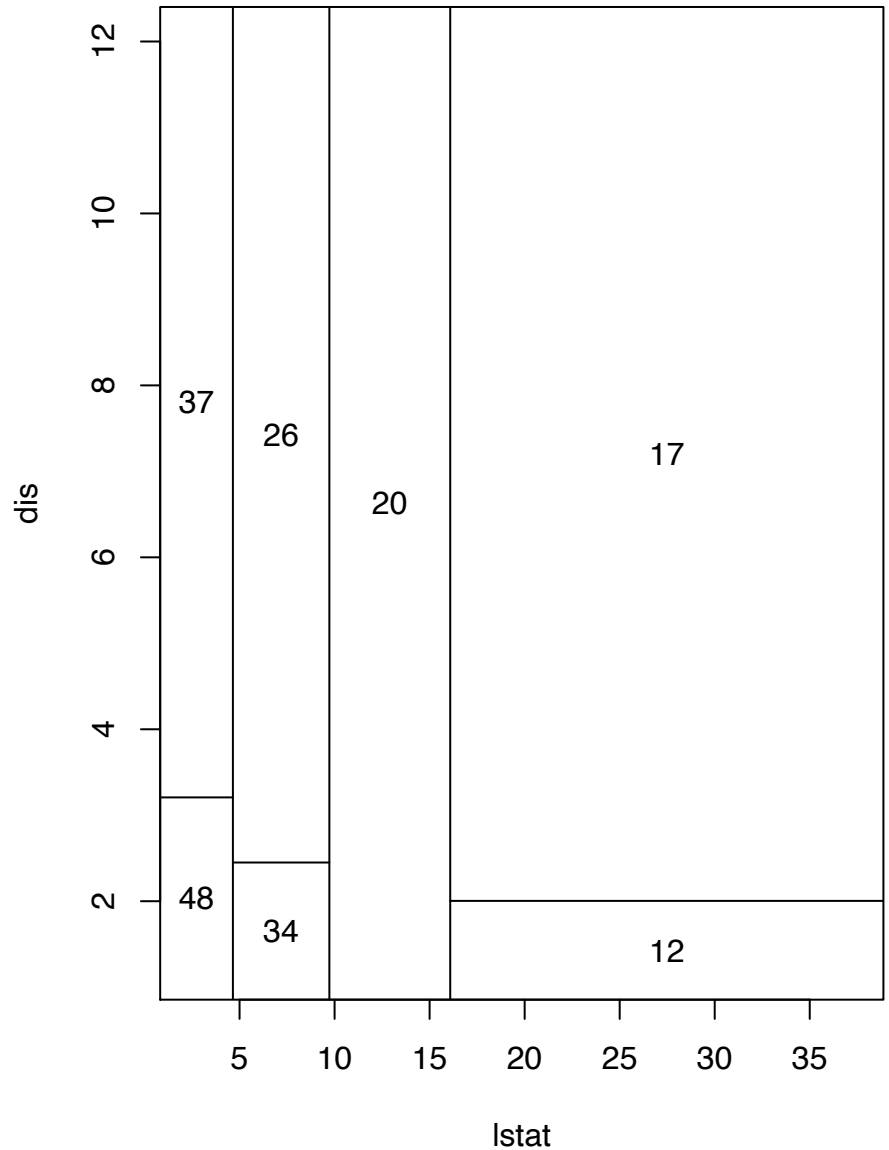
Then prune it down to one with 7 leaves

```
boston.tree=prune.tree(temp,best=7)
```

Bostong Housing

```
plot(boston.tree,type="u")# plot tree and partition in x.
text(boston.tree,col="blue",label=c("yval"),cex=.8)
partition.tree(boston.tree)
```





Bostong Housing

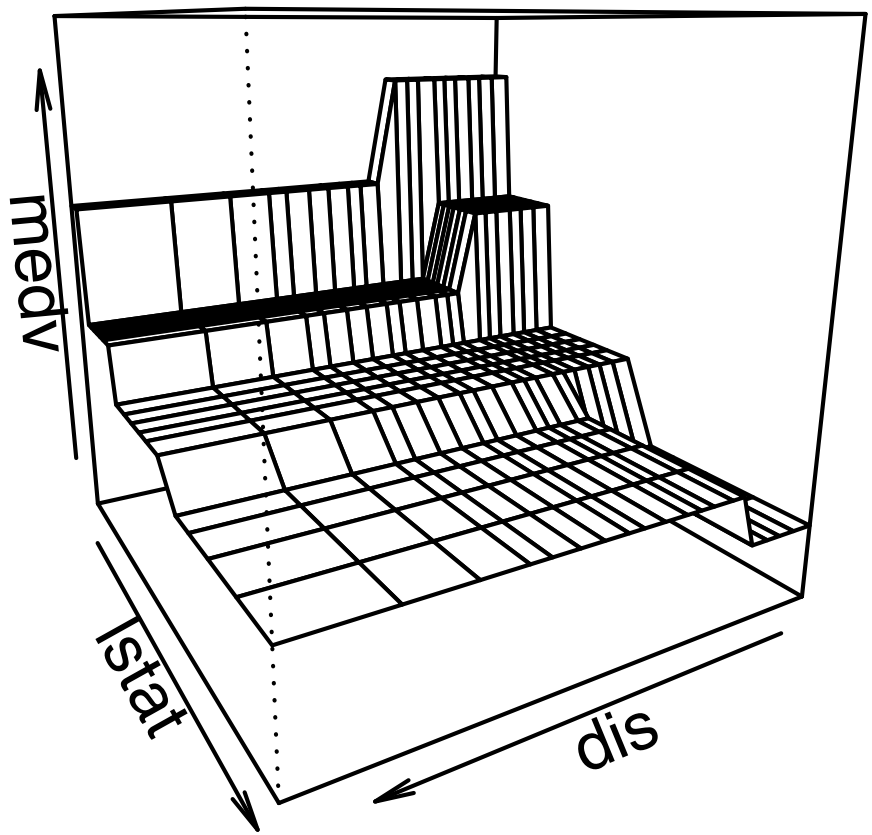
Get predictions on 2d grid

```
pv=seq(from=.01,to=.99,by=.05)
x1q = quantile(df2$lstat,probs=pv)
x2q = quantile(df2$dis,probs=pv)
xx = expand.grid(x1q,x2q) #matrix with two columns using all combinations of x1q
dfpred = data.frame(dis=xx[,2],lstat=xx[,1])
lmedpred = predict(boston.tree,dfpred)
```

Bostong Housing

Make perspective plot

```
par(mfrow=c(1,1))
persp(x1q,x2q,matrix(lmedpred,ncol=length(x2q),byrow=T),
      theta=150,xlab='dis',ylab='lstat',zlab='medv',
      zlim=c(min(df2$medv),1.1*max(df2$medv)))
```



Trees Pluses

- Trees are very easy to explain to people. In fact, they are even easier to explain than linear regression!
- Some people believe that decision trees more closely mirror human decision-making than do the regression and classification approaches seen in previous chapters.
- Trees can be displayed graphically, and are easily interpreted even by a non-expert (especially if they are small).
- Trees can easily handle qualitative predictors without the need to create dummy variables.

Trees Minuses

- Large tress are of high variance (a small change in the data can cause a large change in the final estimated tree)
- Small trees are not good predictors
- Often hard to find a good model on the bias-variance scale

Bagging

- Treat the sample as if it were the population and then take iid draws.
- That is, you sample with replacement so that you can get the same original sample value more than once in a bootstrap sample.

To Bootsrap Aggregate (Bag) we:

- Take B bootstrap samples from the training data, each of the same size as the training data.
- Fit a large tree to each bootstrap sample (we know how to do this fast!). This will give us B trees.
- Combine the results from each of the B trees to get an overall prediction.

Bagging

- For numeric y we can combine the results easily by making our overall prediction the average of the predictions from each of the B trees.
- For categorical y, it is not quite so obvious how you want to combine the results from the different trees.
- Often people let the trees vote: given x get a prediction from each tree and the category that gets the most votes (out of B ballots) is the prediction.
- Alternatively, you could average the $\hat{p}$ from each tree. Most software seems to follow the vote plan.

Baggining

- The simple idea behind every ensemble modes is that variance of the average is lowe than variance of individual.
- Say we have B models $f_1(x), \ldots, f_B(x)$ then we combine those

$$f_{avg}(x) = \frac{1}{B} \sum_{b=1}^{B} f_b(x)$$

- Combining models helps fighting overfilling
- On the negative side, it is harder to interpret those ensembles

Bagging

Let's experiment with the number of trees in the model

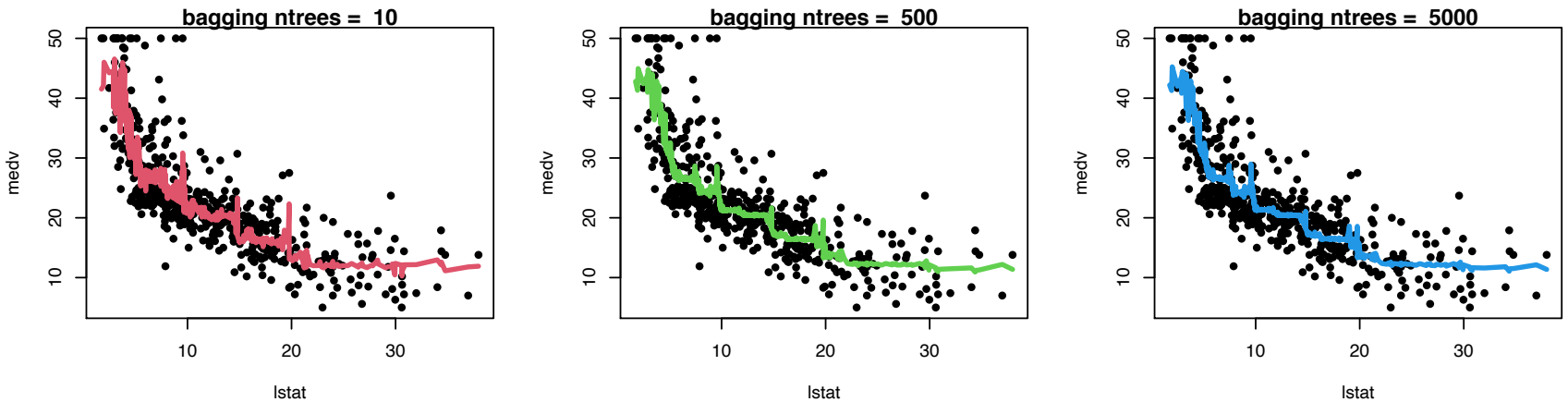
```
library(randomForest)
n = nrow(Boston)
ntreev = c(10,500,5000)
fmat = matrix(0,n,3)
for(i in 1:3) {
  rffit = randomForest(medv~lstat,data=Boston,ntree=ntreev[i],maxnodes=15)
  fmat[,i] = predict(rffit)
  print(mean((fmat[,i] - medv)^2, na.rm = TRUE))
}
```

```
## [1] 32
## [1] 29
## [1] 29
```

Bagging

Let's plot the results

```
for(i in 1:3) {
  plot(Boston$lstat,Boston$medv,xlab='lstat',ylab='medv',pch=16)
  lines(Boston$lstat[oo],fmat[oo,i],col=i+1,lwd=3)
  title(main=paste('bagging ntrees = ',ntreev[i]))
}
```



- With 10 trees our fit is too jumbly.
- With 1,000 and 5,000 trees the fit is not bad and very similar.
- Note that although our method is based multiple trees (average over) so we no longer have a simple step function!!

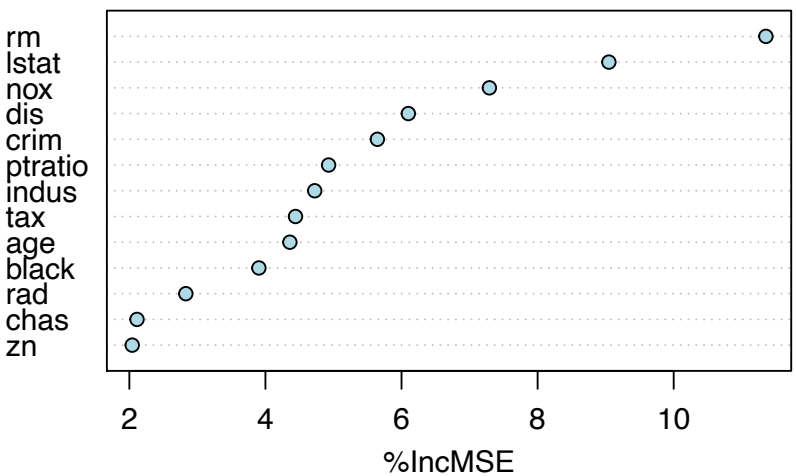
Random Forest

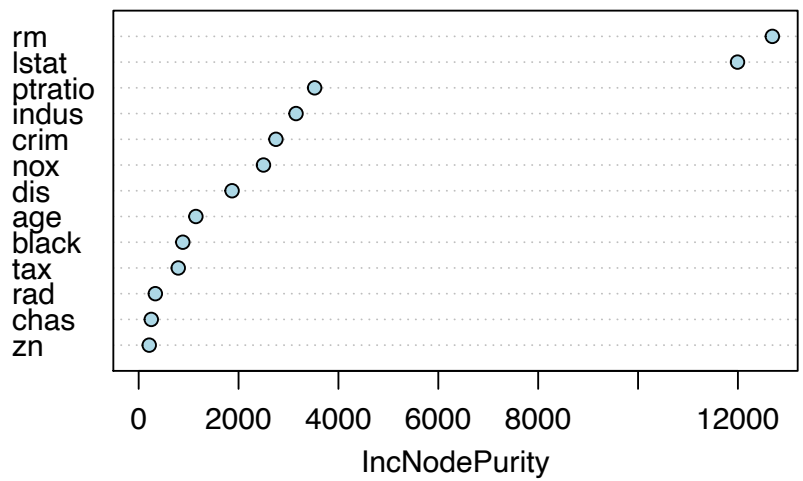
- In bagging, the models become correlated and you do not achieve $1/n$ reduction in variance: most or all of the trees will use the strongest predictor in the top split
- Bagged trees will look similar!
- Random forests decorrelates the trees: each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors
- Typically $m = \sqrt{p}$

Random Forest

Ove of the "interpretation" tools that comes with ensemble models is importance rank: total amount that the deviance (loss) is decreased due to splits over a given predictor, averaged over all tree

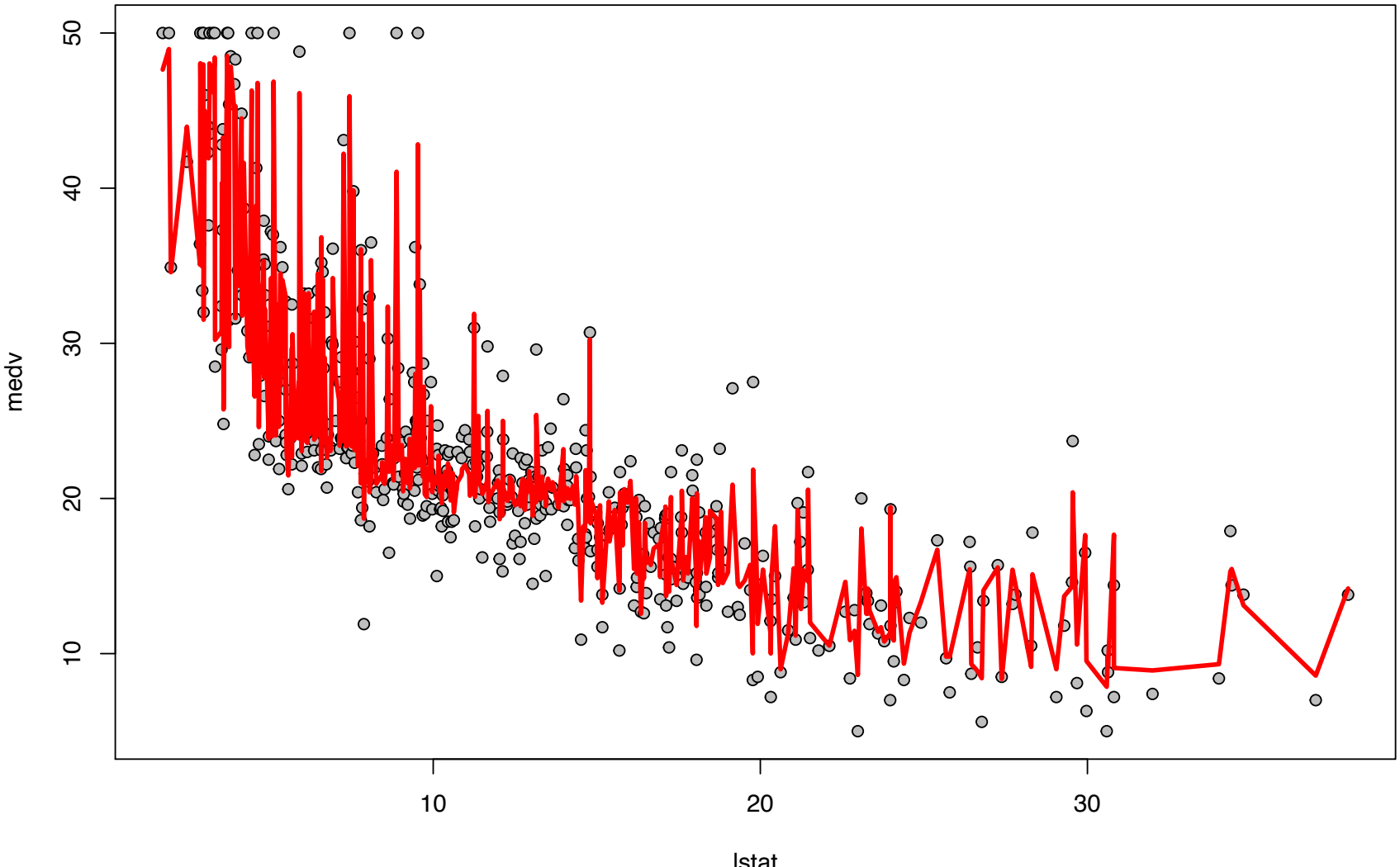
```
rf.boston = randomForest(medv~.,data=Boston,mtry=4,importance=TRUE,ntree=50)
varImpPlot(rf.boston,pch=21,bg="lightblue",main="")
```





Random Forest

```
rf.boston = randomForest(medv~.,data=Boston,mtry=6,ntree=50, maxnodes=50)
yhat.rf = predict(rf.boston,newdata=Boston)
oo=order(lstat)
plot(lstat[oo],medv[oo],pch=21,bg="grey", xlab="lstat", ylab="medv") #plot data
lines(lstat[oo],yhat.rf[oo],col='red',lwd=3) #step function fit
```



Boosting

Like Random Forests, boosting is an ensemble method is that the overall fit it produced from many trees. The idea however, is totally different!!

In Boosting we:

- Fit the data with a single tree.
- Crush the fit so that it does not work very well.
- Look at the part of y not captured by the crushed tree and fit a new tree to what is "left over"
- Crush the new tree. Your new fit is the sum of the two trees.
- Repeat the above steps iteratively. At each iteration you fit "what is left over" with a tree, crush the tree, and then add the new crushed tree into the fit.
- Your final fit is the sum of many trees.

Boosting

Pick a loss function L that reflects setting; e.g., for continuous y, could take $L(y_i, \theta_i) = (y_i - \theta_i)^2$ Want to solve

$$\text{minimize}_{\beta \in R^M} \sum_{i=1}^{n} L\left(y_i, \sum_{j=1}^{M} \beta_j \cdot T_j(x_i)\right)$$

- Indexes all trees of a fixed size (e.g., depth $= 5$), so M is huge

- Space is simply too big to optimize
- Gradient boosting: basically a version of gradient descent that is forced to work with trees
- First think of optimization as $\min_\theta f(\theta)$, over predicted values θ (subject to θ coming from trees)

Boosting

Set $f_1(x) = 0$ (constant predictor) and $r_i = y_i$

For $b = 1, 2, \ldots, B$

(a) Fit a tree f_b with d splits to the training set (X, r)
(b) Update the model

$$f(x) = f(x) + \lambda f_b(x)$$

(c) Update the residuals

$$r_i = r_i - \lambda f_b(x)$$

Boosting

Here are some boosting fits where we vary the number of trees, but fix the depth at 2 (suitable with 1 x) and shrinkage $= \lambda$ at .2.

```
library(gbm)
boost.boston=gbm(medv~.,data=Boston,distribution="gaussian",n.trees=5000,intera
yhat.boost=predict(boost.boston,newdata=Boston,n.trees=5000)
mean((yhat.boost-Boston$medv)^2)
```

```
## [1] 4e-04
```

Boosting

```
summary(boost.boston, plotit=FALSE)
```

```
##                var rel.inf
## lstat      lstat   36.32
## rm            rm   30.98
## dis          dis    7.63
## crim        crim    5.09
## nox          nox    4.63
## age          age    4.50
## black      black    3.45
## ptratio ptratio    3.11
## tax          tax    1.74
## rad          rad    1.17
## indus      indus    0.87
## chas        chas    0.39
## zn             zn    0.13
```

Boosting

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