Decision Trees

Objectives

- Linear regression (logistic regression) combining with regularization works for many studies, especially in the case where we try to identify important attributes. However, the strong model assumptions may not be met.
- For the purpose of prediction, a model free approach may gain more in accuracy. We start with decision trees.
- While wild search of optimal boxes is impossible, we use binary, top-down, greedy, recursive algorithm to build a tree.
- To reduce the prediction error from a single equation, one idea is to take the average of many very different prediction equations. Bagging and Random Forest are proposed to aggregate many Bootstrap trees.

Objectives

- In general, we want to bag **MANY** different equations. They can be completely different methods, such as linear model, lasso equations with different lambda, random forests with different mtry, split predictors. . .
- Decision trees, bagging and random forests are all readily/easily extended to classification problems. To build a tree we use sample proportions to estimate the probability of the outcome, instead of sample means.
- The criteria of choosing the best split is the sum of squared errors
- Trees are easily extended to classification problems

Table of Contents

Regression trees

Single Tree

- Model free, flexible
- Easy to interpret, reveal important features among attributes
- Packages
  - `tree()`
  - `rpart()` from CART (Recursive Partitioning and Regression Trees)

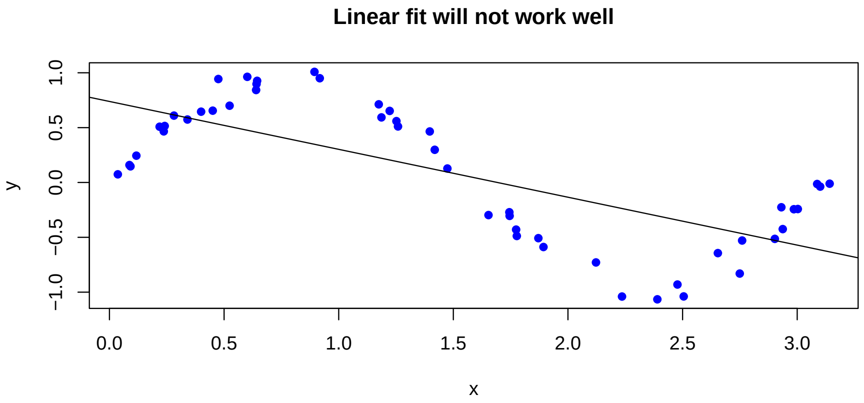
A toy example

The following data is obtained from a simulation:

- 50 pairs of (x, y) are observed.
- The goal is to build a predictive equation to predict y given x.
- The scatter plot below shows that a linear model will not do a good job.

A toy example

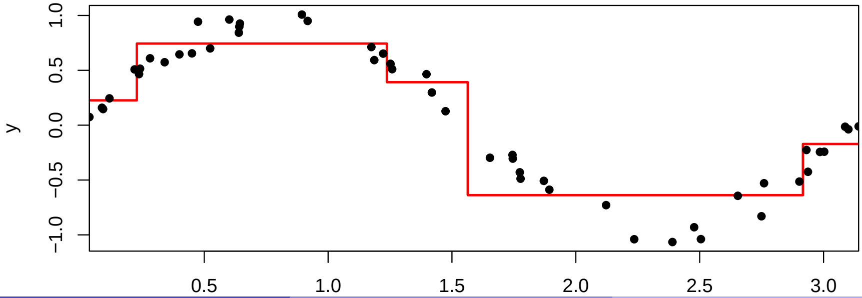
```r
set.seed(571)
x <- runif(50, min = 0, max = pi)
y <- sin(2*x) + rnorm(50, 0, .1)
toy <- data.frame(x, y)
plot(toy$x, toy$y, pch=16, col="blue",
     xlab = "x", ylab = "y",
     main = "Linear fit will not work well")
abline(lm(y~x, toy))
```



A toy example

On the other hand if we could group the x's into a few groups, use the sample mean in each group as predicted value for all x in the group. The step function may capture the curvature shown in the data.

```
# fit the tree
fit <- tree(y~x, toy, control=tree.control(length(y), minsize = 20))
# minsize = 15
partition.tree(fit, col="red", lwd=2)
points(toy$x, toy$y, pch=16, xlab = x, ylab = y)
```

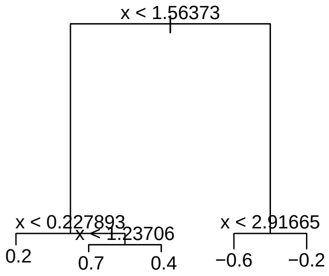


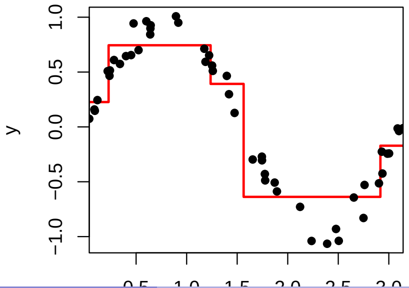
A toy example

The above step function can be presented by a tree with five leaves or terminal nodes.

```
# split the drawing canvas for 2 plots
par(mfrow=c(1, 2))
plot(fit)
text(fit)    # add the split variables

partition.tree(fit, col="red", lwd=2)
points(toy$x, toy$y, pch=16,
       xlab = x,
       ylab = y)
```





How to build trees?

How to split the predictors? How many regions should we have?

How to build trees?

The idea is to partition the space into J boxes $R_1, R_2, \dots, R_J$.

The target is to minimize the RSS

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

where $\hat{y}_{R_j}$ is the sample mean of the training samples in the region R_j.

How to build trees?

Find the optimal split point

Start with one continuous predictor case x. Here is an efficient algorithm to split x into two regions.

- For each possible split point s
  - Use the sample averages to predict the y's on the left side of s and the right side respectively
  - Calculate the RSS_s for this split.
- We then find the minimum point s_1 by minimizing RSS_s across all the possible split points.
- Repeat this process to the left region of s_1 then to the right region until a criterion is met.

How to build trees?

Notes:

Binary, top-down, recursive and greedy trees

Binary: Always a binary split

Greedy: At each step by using the sample means as predicted value on either side

Topdown/Recursive:Continue to split further down to smaller regions

X is categorical: then the levels of x are divided into two non-empty groups.

Binary, top-down, recursive and greedy trees

Pros:

- not a linear model, more flexible
- take interactions among variables
- simple interpretation of the prediction

Cons:

- greedy or forward splitting (not optimal)
- not stable
- over-fitting or not a good prediction

Case study: Predicting baseball players' salaries

EDA:

For the purpose of demonstration we use the Baseball data from the book.
Our Goal: predict log salary given the past performance!

We read in the data and after quick exploration of the data, we remove any "NA" values. We also take the log of salary and replace that as the dependent variable.

```
data.comp <- na.omit(Hitters)  # For simplicity
dim(data.comp)              # We are keeping 263 players
```

```
## [1] 263  20
```

```
data.comp$LogSalary <- log(data.comp$Salary)  # add LogSalary
names(data.comp)
```

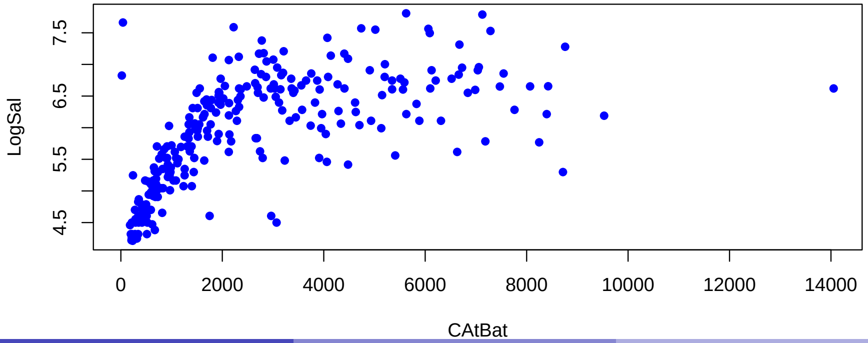
```
##  [1] "AtBat"     "Hits"      "HmRun"     "Runs"      "RBI"       "Walks"
##  [7] "Years"     "CAtBat"    "CHits"     "CHmRun"    "CRuns"     "CRBI"
## [13] "CWalks"    "League"    "Division"  "PutOuts"   "Assists"   "Errors"
## [19] "Salary"    "NewLeague" "LogSalary"
```

```
data1 <- data.comp[,-19]        # Take Salary out
```

Tree using a single predictor (CAtBat)

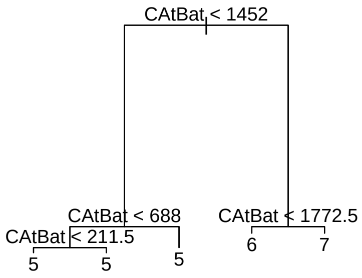
tree():
We need to find the best split point that leads to the greatest reduction in RSS. We use the `tree` package.

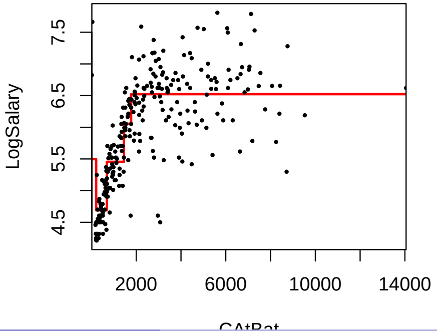
```
# take a look at the scatter plot
plot(data1$CAtBat, data1$LogSalary, pch=16, cex=1, col = "blue",
     xlab = "CAtBat",
     ylab = "LogSal")
```



'tree()'

```
# build a single tree
fit0.single <- tree(LogSalary~CAtBat,data1)
# split the canvas
par(mfrow=c(1,2))
# plot the tree
plot(fit0.single)
text(fit0.single)    # add the split variables
# plot it on the scatter plot
partition.tree(fit0.single, col="red", lwd=2)
points(data1$CAtBat, data1$LogSalary, pch=16, cex=.5)
```





‘tree()‘

Let’s take a look at the return from `fit0.single`. `fit0.single$frame` stores the details splitting process and the results in each best split.

```r
fit0.single <- tree(LogSalary~CAtBat,data1)
names(fit0.single)
```

```
## [1] "frame"   "where"   "terms"   "call"    "y"       "weights"
```

‘tree()‘

```
data.table(fit0.single$frame)
```

```
##        var   n    dev yval splits.cutleft splits.cutright
## 1: CAtBat 263 207.15 5.93          <1452           >1452
## 2: CAtBat 103  36.22 5.09           <688            >688
## 3: CAtBat  54  18.32 4.76         <211.5          >211.5
## 4: <leaf>   5  10.55 5.50
## 5: <leaf>  49   4.82 4.69
## 6: <leaf>  49   5.63 5.46
## 7: CAtBat 160  53.08 6.46        <1772.5         >1772.5
## 8: <leaf>  20   3.59 6.06
## 9: <leaf> 140  45.67 6.52
```

```
# fit0.single$frame should print the result
```

‘tree()‘

- **var**: the variable (internal node) we are looking into: `CAtBat` as our predictor, `<leaf>` means the end nodes
- **n**: number of observations in the node of interest
- **dev**: RSS of the node of interest
  - Check: `sum((data1$LogSalary - mean(data1$LogSalary))^2)`
- **yval**: prediction/mean of the responses of the node of interest
  - Check: `mean(data1$LogSalary)`
- **splits.cutleft**: the split to the left branch
- **splits.cutright**: the split to the right branch

i. For example at the very top, without any split we have `n`=263, yval= mean(data1$LogSalary) =5.927, dev= 207.154
ii. The best firsts split point is at 1452:

- If CAtBat $<$ 1452, then the estimated LogSalary is 5.093
- If CAtBat $>$ 1452, then the estimated LogSalary is 6.464

‘tree()‘

The summary statistics

```
fit0.single.s <- summary(fit0.single)     #fit0.single.s$dev
names(fit0.single.s)
```

```
## [1] "call"        "type"        "size"        "df"          "dev"         "residuals"
```

```
fit0.single.s$size
```

```
## [1] 5
```

```
fit0.single.s$dev
```

```
## [1] 70.3
```

- `size`: the number of terminal nodes is 5
- `df`: degree of freedom 258 = number of observations 263 - the number of terminal nodes
- `dev`: deviance 70.251 = sum of deviance of each terminal nodes (RSS)

Depth of the tree

How deep a tree should we build? Too many splits/too deep a tree yields smaller bias but the variance will be large. We can control the size of a tree with various criteria. Or stop until the reduction in deviance is not small enough:

minsize: minimal number of observations in the end node

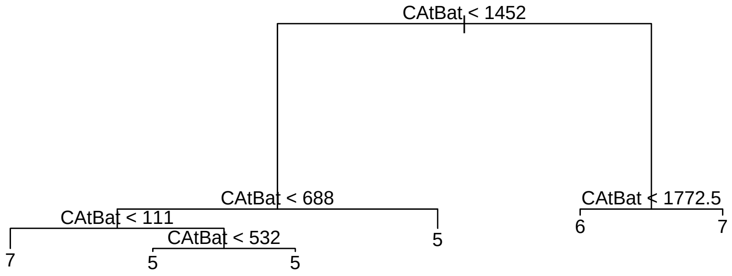
mindev: deviance in each new split node must be at least `mindev` times deviance of the **root node**

mincut: minimal number of observations in each partition in each split. (Not really a tuning parameter. Default is set at 5.)

We can use `tree.control` to control how deep the tree grows.

Depth of the tree

```
fit0.single <- tree(LogSalary~CAtBat, data1,
                    control=tree.control(nobs=nrow(data1), minsize=5, mindev=.008))
# try different set up to see the fit set as .008(6), .01(default, 5), .1(one split)
plot(fit0.single)
text(fit0.single)
```



- nobs: number of observations

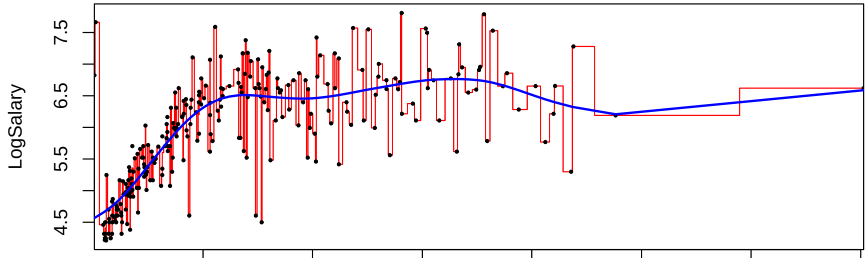
Depth of the tree

An over-fitting tree

By setting minsize=2 and mindev=0, we get a perfectly fit tree, namely an over-fitting tree (a very deep tree).

```
fit0.single.deep <- tree(LogSalary~CAtBat, data1,
                         control=tree.control(nobs=nrow(data1), minsize=2, mindev=0))

partition.tree(fit0.single.deep, col="red", lwd=1) # plot it on the scatter plot
points(data1$CAtBat, data1$LogSalary, pch=16, cex=.5)
smoothingSpline <- smooth.spline(data1$CAtBat, data1$LogSalary, spar=1) # to get a smooth line
lines(smoothingSpline, pch=16, cex=.5, col="blue", lwd=2)
```



Tree using two predictors (CHits and CAtBat)

Algorithm:

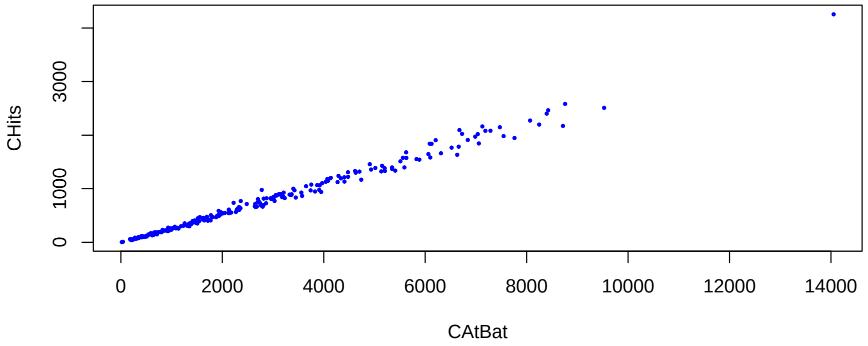
Similar to the tree we built using only one variable, the recursive binary splitting algorithm for building tree using two variables is as follows:

1. Find the best split by `CHits` and `CAtBat` respectively. Denote the RSS's of the two splits as RSS_{CHits}, RSS_{CAtBat}
2. Pick the variable with $\min(RSS_{CHits}, RSS_{CAtBat})$ to be split first
3. Repeat 1. - 2. on either side, until reaching the stopping rule.

In other words, at each split, we consider all the variables and all possible split points and then pick the predictor and the split point that minimize the RSS (that is the reason why it is greedy).

Tree using two predictors (CHits and CAtBat)

```
# get scatter plot of CHits and CAtBat
plot(data1$CAtBat, data1$CHits, pch=16, cex=.5,
     col = "blue",
     xlab="CAtBat",
     ylab="CHits")
```



Best split for CHits

```
### Get the single tree for fit0.CHits
fit0.CHits <- tree(LogSalary~CHits, data1)

## use prune.tree() to get a 1-split tree
## best: number of leafs
## see appendix for prune.tree()
fit0.CHits.1split <- prune.tree(fit0.CHits, best=2)

## plot the tree
# plot(fit0.CHits.1split)
# text(fit0.CHits)
# partition.tree(fit0.CHits, col="red", lwd=2)
# points(data1$CHits, data1$LogSalary, pch=16, cex=.5)

# best split 358: RSS_{358}=90.31
data.table(fit0.CHits.1split$frame)
summary(fit0.CHits.1split)$dev
```

```
##       var   n   dev yval splits.cutleft splits.cutright
## 1:  CHits 263 207.2 5.93           <358            >358
## 2: <leaf> 101  35.2 5.08
## 3: <leaf> 162  55.1 6.45
## [1] 90.3
```

The deviance for the best CHits split is 90.312.

Best split CatBat

```
fit0.CAtBat <- tree(LogSalary~CAtBat, data1)
fit0.CAtBat.1split <- prune.tree(fit0.CAtBat, best=2)

data.table(fit0.CAtBat.1split$frame)  # best split 452: RSS_{1452}=89
# So we split CAtBat first at 1452 then split to either side, continue
```

```
##       var   n   dev yval splits.cutleft splits.cutright
## 1: CAtBat 263 207.2 5.93          <1452           >1452
## 2: <leaf> 103  36.2 5.09
## 3: <leaf> 160  53.1 6.46
```

- The deviance for the best CHits split is 90.312.
- The deviance for the best CAtBat split is 89.296.
- CAtBat wins! So we first split CAtBat at $<$1452.

Best 1st split: CatBat wins

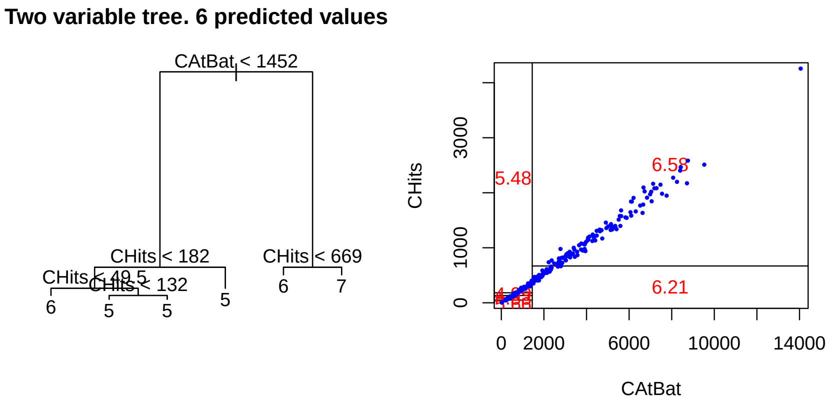
```
fit1.single <- tree(LogSalary~CAtBat+CHits, data1) # The order plays no role
fit1.single.1split <- prune.tree(fit1.single, best = 2)
# fit1.single.1split
data.table(fit1.single.1split$frame)
```

```
##        var   n   dev yval splits.cutleft splits.cutright
## 1: CAtBat 263 207.2 5.93          <1452           >1452
## 2: <leaf> 103  36.2 5.09
## 3: <leaf> 160  53.1 6.46
```

Similarly, we repeat this splitting process until stop.

Tree using two predictors (CHits and CAtBat)

```
par(mfrow=c(1,2))
# plot tree
plot(fit1.single)
title(main="Two variable tree. 6 predicted values")
text(fit1.single)
# scatter plot
partition.tree(fit1.single, col="red", lwd=1)
points(data1$CAtBat, data1$CHits, pch=16, cex=.5, col = "blue")
```



Tree using two predictors (CHits and CAtBat)

```
data.table(fit1.single$frame)
```

```
##        var   n     dev yval splits.cutleft splits.cutright
##  1: CAtBat 263 207.154 5.93          <1452           >1452
##  2:  CHits 103  36.220 5.09           <182            >182
##  3:  CHits  56  18.359 4.77          <49.5           >49.5
##  4: <leaf>   5   9.332 5.66
##  5:  CHits  51   4.691 4.68           <132            >132
##  6: <leaf>  34   1.621 4.53
##  7: <leaf>  17   0.751 4.99
##  8: <leaf>  47   5.165 5.48
##  9:  CHits 160  53.077 6.46           <669            >669
## 10: <leaf>  50   9.500 6.21
## 11: <leaf> 110  39.005 6.58
```

Tree using two predictors (CHits and CAtBat)

Summary:

Using CAtBat and CHits

- The single tree has 6 terminal nodes, i.e., we partition CAtBat and Chits into six boxes.
- The predicted values are the sample means in each box.

```
fit1.single.s <- summary(fit1.single)
fit1.single.s$size
```

```
## [1] 6
```

```
fit1.single.s$dev
```

```
## [1] 65.4
```

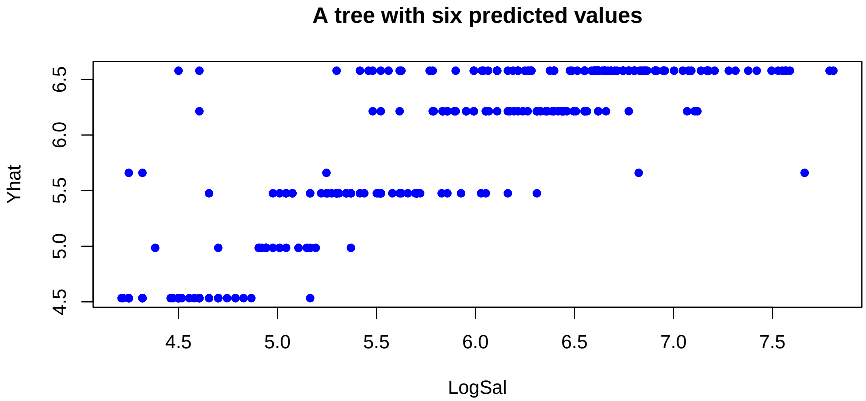
The deviance of the tree using CAtBat and CHits is 65.374, comparing to the one using only CAtBat 70.251.

Tree using two predictors (CHits and CAtBat)

Summary:

Let's look at the plot of predicted y values:

```
yhat <- predict(fit1.single, data1)
plot(data1$LogSalary, yhat, pch=16, col="blue",
     xlab="LogSal",
     ylab="Yhat",
     main = "A tree with six predicted values")
```



Tree using two predictors

Compare a single tree with lm:

As we already knew there are only 6 predicted values being used.

How does the above tree perform comparing with our old friend lm in terms of the in sample errors?

```
fit.lm <- summary(lm(LogSalary~CAtBat+CHits, data1))
RSS.lm <- (263-2)*(fit.lm$sigma)^2
RSS.lm   # fit1.single.s$dev
```

```
## [1] 127
```

Oops much worse than even a single tree. RSS.lm = 127.251 vs. RSS.tree=65.374.

- Remember these are training errors!

rpart: An alternative tree package CART (like its output)

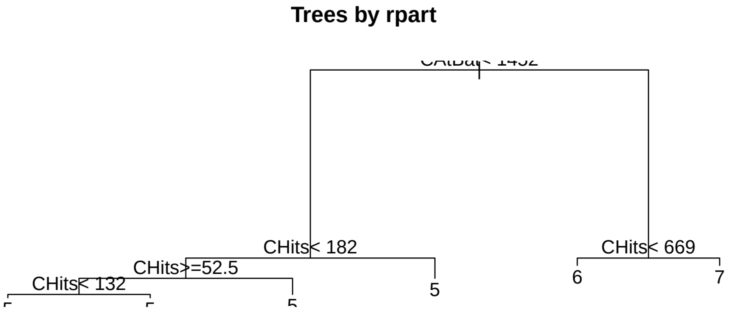
```
fit.single.rp <- rpart(LogSalary~CAtBat+CHits, data1, minsplit=20, cp=.009)
fit.single.rp
```

```
## n= 263
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 263 207.000 5.93
##    2) CAtBat< 1.45e+03 103  36.200 5.09
##      4) CHits< 182 56  18.400 4.77
##        8) CHits>=52.5 49   4.260 4.70
##         16) CHits< 132 32   1.420 4.55 *
##         17) CHits>=132 17   0.751 4.99 *
##        9) CHits< 52.5 7  12.300 5.25 *
##      5) CHits>=182 47   5.160 5.48 *
##    3) CAtBat>=1.45e+03 160  53.100 6.46
##      6) CHits< 669 50   9.500 6.21 *
##      7) CHits>=669 110  39.000 6.58 *
```

rpart: An alternative tree package CART (like its output)

Plot method 1.

```
plot(fit.single.rp, main = "Trees by rpart")
text(fit.single.rp, pretty = TRUE)    # plot method 1
```



rpart: An alternative tree package CART (like its output)

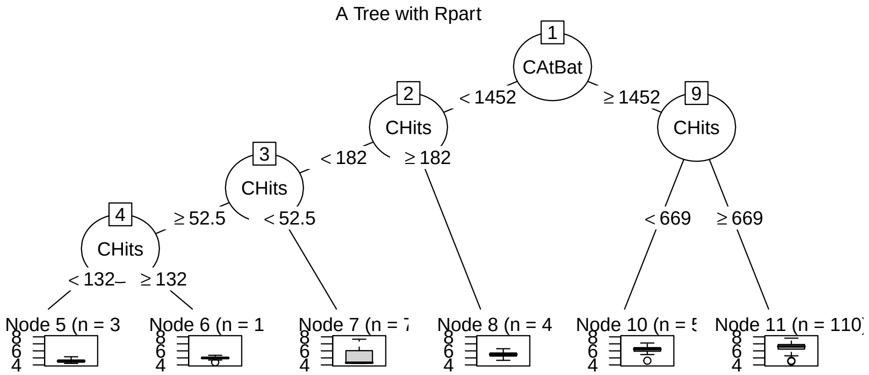
```
summary(fit.single.rp)
```

```
## Call:
## rpart(formula = LogSalary ~ CAtBat + CHits, data = data1, minsplit = 20,
##     cp = 0.009)
##   n= 263
##
##          CP nsplit rel error xerror   xstd
## 1 0.56894      0     1.000  1.005 0.0652
## 2 0.06129      1     0.431  0.460 0.0521
## 3 0.02206      2     0.370  0.403 0.0577
## 4 0.00949      3     0.348  0.382 0.0589
## 5 0.00900      5     0.329  0.397 0.0592
##
## Variable importance
## CAtBat  CHits
##     50     50
##
## Node number 1: 263 observations,    complexity param=0.569
##   mean=5.93, MSE=0.788
##   left son=2 (103 obs) right son=3 (160 obs)
##   Primary splits:
##       CAtBat < 1450 to the left,  improve=0.569, (0 missing)
##       CHits  < 358  to the left,  improve=0.564, (0 missing)
##   Surrogate splits:
##       CHits < 358  to the left,  agree=0.985, adj=0.961, (0 split)
##
## Node number 2: 103 observations,    complexity param=0.0613
##   mean=5.09, MSE=0.352
##   left son=4 (56 obs) right son=5 (47 obs)
##   Primary splits:
##       CHits  < 182  to the left,  improve=0.351, (0 missing)
##       CAtBat < 688  to the left,  improve=0.339, (0 missing)
```

rpart: An alternative tree package CART (like its output)

Another cool plot of a single tree:

```
plot(as.party(fit.single.rp), main="A Tree with Rpart") # method 3
```



The plots are only useful for a small tree!

Tree using all available predictors

We conclude the section by building a final tree with all the variables. To get a testing error, let's split the data first.

```
n <- nrow(data1)
set.seed(1)
train.index <- sample(n, n*3/4) # we use about 3/4 of the subjects as the training data.
train.index
data.train <- data1[train.index,]
data.test <- data1[-train.index, ]
```

Tree using all available predictors

Build final tree:

We will build a final tree with `data.train`.

At each splitting,

1. for each variables X_i, we get the best split point s that minimizes RSS_{X_i}
2. compare all the RSS_{X_i}, pick the variable X_j and the corresponding split point s that minimize RSS

Repeat until reaching the stopping rule.

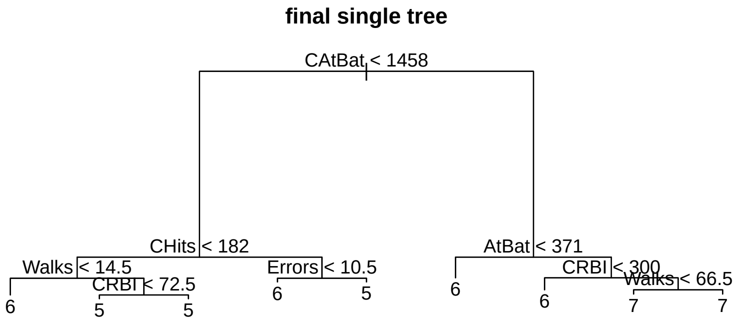
```
fit1.single.full <- tree(LogSalary~., data.train)
```

Tree using all available predictors

Build final tree:

The above tree looks like this:

```
plot(fit1.single.full)
title(main =  "final single tree")
text(fit1.single.full, pretty=0)
```



Tree using all available predictors

Build final tree:

```
fit1.single.full
```

Tree using all available predictors

Build final tree:

Summary Statistics of the above tree:

1) Names

```
fit1.single.full.s <- summary(fit1.single.full)
names(fit1.single.full.s)
```

```
## [1] "call"      "type"      "used"      "size"      "df"        "dev"
## [7] "residuals"
```

```
names(fit1.single.full)    #fit1.single.full.s$size
```

```
## [1] "frame"   "where"   "terms"   "call"    "y"       "weights"
```

2) RSS

```
fit1.single.full.s$dev  # training RSS=29.28
```

```
## [1] 29.3
```

```
sum((fit1.single.full.s$residuals)^2) # to check the dev is the RSS
```

```
## [1] 29.3
```

3) Variables used

```
fit1.single.full.s$used # Var's included
```

```
## [1] CAtBat CHits  Walks  CRBI   Errors AtBat
```

Tree using all available predictors

Compare a full tree with lm:

4) Finally comparing the RSS from the lm function.

```
n.train <- nrow(data.train)
fit.lm <- lm(LogSalary~CAtBat+CHits + Walks + CRBI +  Errors+
RSS.lm <- (n.train - 6+1)*(summary(fit.lm)$sigma)^2  # (n-p+1,
RSS.lm
```

```
## [1] 80.1
```

Do you see that RSS=80.103 from lm() $>$ RSS=29.282 from the tree. Is this always true and why or why not?

Question: do you expect that the testing RSS(lm) $>$ RSS(single tree)?

Tree using all available predictors

Report the testing error:

We apply the testing data, `data.test` to output the testing error of the final tree

```
test.error.tree <- sum((predict(fit1.single.full, data.test) - data.test$LogSalary)^2)
test.error.twoVar <- sum((predict(fitTwoVar, data.test) - data.test$LogSalary)^2)
test.error.lm <- sum((predict(fit.lm, data.test)-data.test$LogSalary)^2)

data.frame(test.error.lm = test.error.lm,
           test.error.tree = test.error.tree,
           test.error.twoVar = test.error.twoVar)
```

```
##   test.error.lm test.error.tree test.error.twoVar
## 1          30.5              28              18.7
```

Notice:

- The testing error from the tree being 28.009 is quite smaller than that from the linear model 30.483
- Also notice the testing errors are quite different from the training errors!!!!! Which once again shows the necessity of using the notion of testing or validation data.

Recap

- Model free, flexible
- Easy to interpret, reveal important features among attributes
- Packages
  - tree()
  - rpart() from CART (Recursive Partitioning and Regression Trees)

Ensemble Methods

A single predictive model (e.g. a single tree):

- Pros: flexible, small training error
- Cons: large variability

Solution to large variability:

- Ensemble method: Get many different (uncorrelated) trees/models and take average.
- Similar bias but reduce the variance.
- The testing error maybe reduced significantly.

Bootstrap method offers one solution to produce many trees and **random trees** will produce uncorrelated trees. This leads to Random Forest.

Ensemble Methods: Bagging

Bootstrap:

Developed by Bradley Efron in 1977, the **bootstrap** is a simple re-sampling scheme.

- Given a data of iid sample of size n, we will sample n observations with replacement to create another sample of size n. This new sample is called a bootstrap sample.
- We can produce many bootstrap samples and get a new sample from the population for free.
- It is a powerful tool to estimate the variance of estimators, so we can often construct confidence intervals.
- The International Prize in Statistics has been awarded to BRADLEY EFRON, professor of statistics and biomedical data science, in recognition of the "bootstrap", he developed that has had extraordinary impact across many scientific fields.

Ensemble Methods: Bootstrap Magic

Bootstrap sample resembles the original sample:

We explain how bootstrap samples can be used as a new random sample from the population through an example.

1. We have a random sample $x_1, x_2, \ldots, x_{1000}$ of size n=1000. Here we show the histogram of the sample.

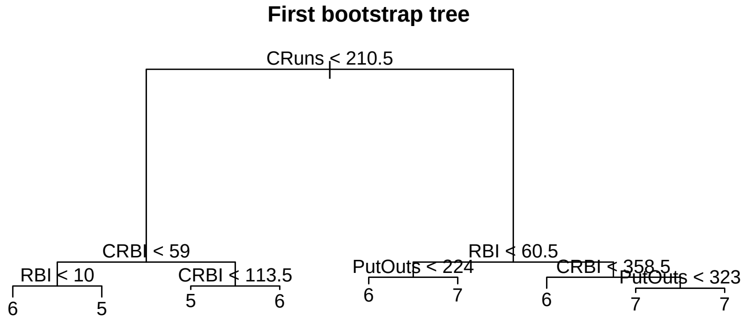
Ensemble Methods

A single Bootstrap tree:

We first take a bootstrap sample, then build a single tree. Notice the structure of the trees are very similar to that of using the whole dataset. We then built another bootstrap tree. A better way to aggregate the two free trees is to take the average of fitted values, namely bagging of two trees.

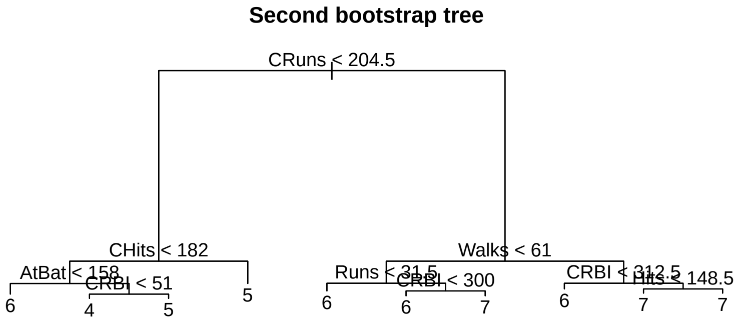
A single Bootstrap tree

```r
# bootstrap tree 1
n=263
set.seed(1)
index1 <- sample(n, n, replace = TRUE)
data2 <- data1[index1, ]  # data2 here is a bootstrap sample
boot.1.single.full <- tree(LogSalary~., data2)
plot(boot.1.single.full)
title(main = "First bootstrap tree")
text(boot.1.single.full, pretty=0)
```



Another Bootstrap tree

```
# bootstrap tree 2
set.seed(2)
index1 <- sample(n, n, replace = TRUE)
data2 <- data1[index1, ]  # data2 here is a bootstrap sample
boot.2.single.full <- tree(LogSalary~., data2)
plot(boot.2.single.full)
title(main = "Second bootstrap tree")
text(boot.2.single.full, pretty=0)
```



Bagging

A new predictive equation: Average two trees

```
# bag of two trees by averaging the two fitted equations. predict the response for the 10th player:
data1[10, ]
fit.bag.2.predict <- (predict(boot.1.single.full, data1[10, ]) + predict(boot.2.single.full, data1[10, ]))/2 #b
data.frame(fitted=fit.bag.2.predict,  obsy=data1[10, "LogSalary"])  # not bad
```

```
##                 AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns
## -Alan Trammell   574  159    21  107  75    59    10   4631  1300     90   702
##                 CRBI CWalks League Division PutOuts Assists Errors NewLeague
## -Alan Trammell   504    488      A        E     238     445     22         A
##                 LogSalary
## -Alan Trammell       6.25
##                 fitted obsy
## -Alan Trammell     6.7 6.25
```

Remark:

- A bootstrap tree will resemble the tree built using the original data
- The tree built depends on the bootstrap sample.
- We have aggregated two trees by taking average of the two predictive equations.
- We can of course aggregate more bootstrap trees. We can get this using randomForest function with specific settings.

Ensemble Methods: Bagging

To bag many trees:

Step1: Build m many bootstrap trees, say $T_1, T2, \ldots, T_m$.

Step2: The final Bagging equation T is simply the average of $T_1, T2, \ldots, T_m$.

We can use Random Forest package with one parameter `mtry` setting as the total number of predictors. You will see why once Random Forest is explained.

For example to bag 10 bootstrap trees with minsize setting at 5, we can get it by specifying mtry = 19, ntree=10 in randomRorest() as follows

Ensemble Methods: Bagging

R functions:

- We borrow package randomForest()
- Begging can be done using this package
- With mtry = total number of variables

```
set.seed(1)
fit.bagging.10 <- randomForest(LogSalary~., data1, mtry=19, nt
```

Ensemble Methods: Bagging

Compare one tree with Bagging 10 trees:

How does this work comparing with a single tree? Here are the training errors

```
set.seed(2)
fit.bagging.1 <- randomForest(LogSalary~., data1, mtry=19, ntree=1)  # a single tree
m1 <- mean((predict(fit.bagging.1, data1) - data1$LogSalary)^2)
m10 <- mean((predict(fit.bagging.10, data1) - data1$LogSalary)^2)
data.frame(MSE_Single=m1, MSE_10=m10)
```

```
##   MSE_Single MSE_10
## 1       0.16 0.0523
```

Ensemble Methods: Bagging

Wow, a huge reduction in bagging already! But we are only showing the deduction in training errors through.

Since bagging trees is a special case of Random Forest. We will not go further in this topic.

Summary: Bagging

- May reduce testing errors
- Lose interpretation of effects!

Random forest

A single tree will be very similar in the way how variables are split at each node. Take a look at the bootstrap trees built here. What have you noticed???

Illustration of similarity of a single bootstrap tree

```
n=263
set.seed(1)
index1 <- sample(n, n, replace = TRUE)
data2 <- data1[index1, ]  # data2 here is a bootstrap sample
boot.1.single.full <- tree(LogSalary~., data2)
plot(boot.1.single.full)
title(main = " Bootstrap trees")
text(boot.1.single.full, pretty=0)
```

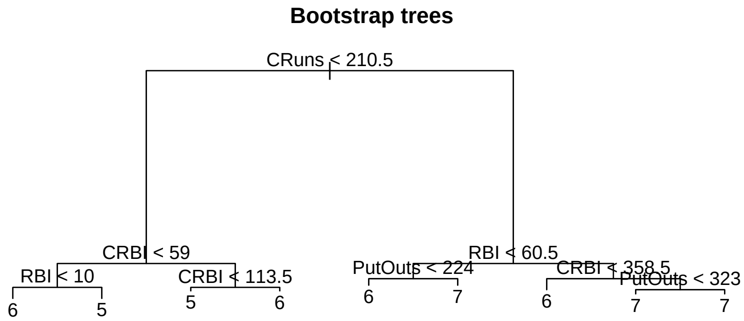
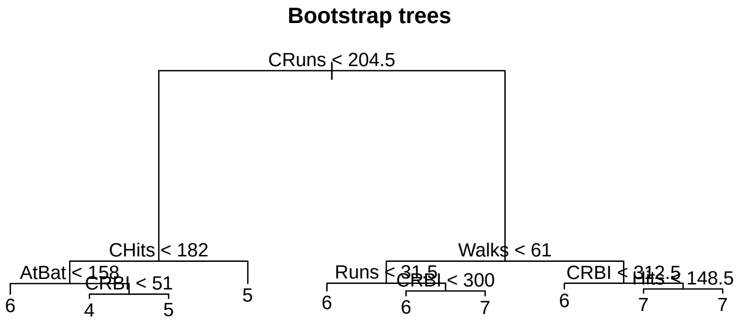


Illustration of similarity of a single bootstrap tree

```
n=263
set.seed(2)
index1 <- sample(n, n, replace = TRUE)
data2 <- data1[index1, ]  # data2 here is a bootstrap sample
boot.1.single.full <- tree(LogSalary~., data2)
plot(boot.1.single.full)
title(main = " Bootstrap trees")
text(boot.1.single.full, pretty=0)
```



Similarity of a single bootstrap tree

All the nodes are more or less similar which means all the bootstrap trees are similar.

- Topdown/greedy search are not flexible
- Bagging similar equations will not reduce the variance

Suggestions needed: how to break the still pattern of similar splitting points????

Random forest: Random tree

Goal: build many uncorrelated bootstrap trees

- How? hoping to reduce the prediction errors!
- Randomness

Given $x_1, x_2, ...x_p$. We will modify our top-down, binary tree to give a chance for any variables to be split earlier.

To find best variable/split point

- Randomly chosen m many variables
- Find the best split variable and the best split point
- Keep the node
- Repeat the process to build a random tree

Random forest: Bagging

Take B many bootstrap samples of size n. Build a m split random trees. Record the prediction equation as $f_i(x_1, x_2, ...x_p, m)$ we then bag then by taking average of these B many random trees. The final Random Forest equation is simply

$$\hat{y}|(x_1, x_2, ...x_p) = \sum_{i=1}^{i=B} f_i(x_1, x_2, ...x_p, m)/B$$

Pros:

- testing error might be reduced

Cons:

- can not interpret the prediction equation any more
- introduce another tuning parameter m and B

Random forest: randomForest()

Created by Leo Breiman, randomForest() has the following algorithm:

1. Take B many bootstrap samples
2. Build a deep random tree for each bootstrap sample by splitting only m (`mtry`) randomly chosen predictors at each split
3. Bag all the random trees by taking average => prediction of y given $x_1, ..., x_p$
4. Use **Out of Bag** observations to provide testing errors to tune `mtry`.

Remarks:

1. The only tuning parameter is mtry, the number of variables to be split at each node.
2. For each tree specify `nodesize`: 5 for regression trees and 1 for classification trees
3. When mtry = p (= 19), randomForest gives us bagging estimates of non-random trees.

Random forest: randomForest()

We now proceed to investigate Random Forest in details. To tune the parameter `m`, the number of variables to be split at any node, we will use testing errors. Cross Validation can be used to evaluate each tree. Breiman introduces notion of **Out Of Bag, OOB errors** to estimate the testing error of the random forest build.

Build RF with fixed parameters (mtry=5)

Now let's build 100 random trees with m (mtry=5).

- For each tree, first take a Bootstrap sample of size n.
- Build a single deep tree but at each split only pick up 5 randomly chosen variables to be split.
- Then average the 100 trees to get the RF prediction equation.

Build RF with fixed parameters (mtry=5)

```
set.seed(1)
fit.rf.5 <- randomForest(LogSalary~., data1, mtry=5, ntree=100)
# by default, the minsize = 5 in regression tree.
names(fit.rf.5)  #summary(fit.rf.5)
```

```
##  [1] "call"            "type"            "predicted"       "mse"
##  [5] "rsq"             "oob.times"       "importance"      "importanceSD"
##  [9] "localImportance" "proximity"       "ntree"           "mtry"
## [13] "forest"          "coefs"           "y"               "test"
## [17] "inbag"           "terms"
```

```
# fit.rf.5$predicted is for the purpose of creating testing errors
```

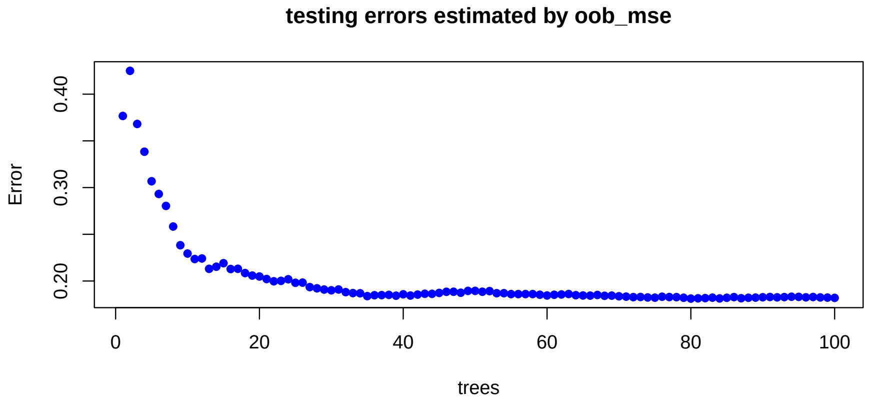
Predict y using fit.rf.5

```
fit.rf.5.pred <- predict(fit.rf.5, data1)
MSE_Train <- mean((data1$LogSalary - fit.rf.5.pred)^2)      # training error
```

The training error is 0.032

Testing errors of fit.rf.5

```
plot(fit.rf.5, type="p", pch=16,col="blue", main = "testing errors estimated by oob_mse" )
```



```
fit.rf.5$mse[100]
```

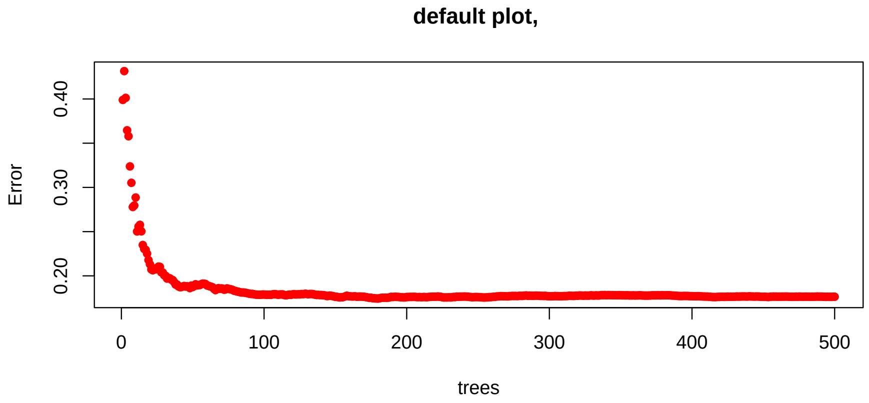
```
## [1] 0.182
```

Tuning ‘ntree‘ and ‘mtry

Ready to tune mtry and B=number of the trees in the bag

a) ntree: given mtry, we see the effect of ntree first

```
fit.rf <- randomForest(LogSalary~., data1, mtry=5, ntree=500)    # change ntree
plot(fit.rf, col="red", pch=16, type="p",
     main="default plot, ")
```



Tuning ‘ntree‘ and ‘mtry

b) mtry: the number of random split at each leaf

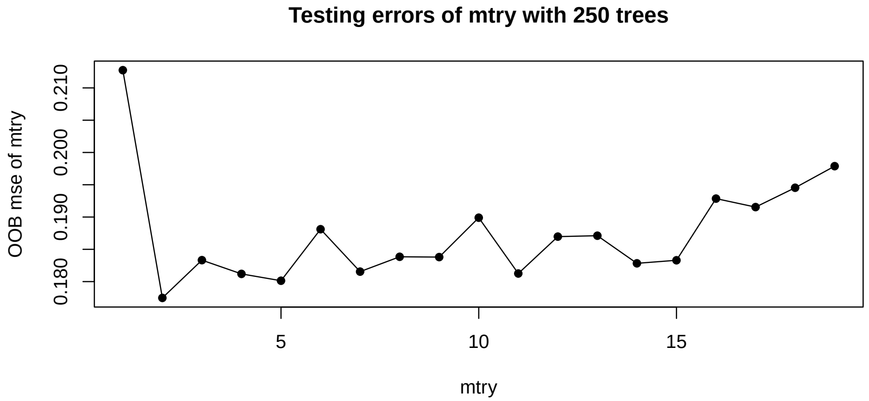
Now we fix ntree=250, We only want to compare the mse[250] to see the mtry effects. Here we loop mtry from 1 to 19 and return the testing errors of 250 trees

```
rf.error.p <- 1:19  # set up a vector of length 19
for (p in 1:19)  # repeat the following code inside { } 19 times
{
  fit.rf <- randomForest(LogSalary~., data1, mtry=p, ntree=250)
  #plot(fit.rf, col= p, lwd = 3)
  rf.error.p[p] <- fit.rf$mse[250]  # collecting oob mse based on 250 trees
}
rf.error.p   # oob mse returned: should be a vector of 19
```

```
##  [1] 0.213 0.177 0.183 0.181 0.180 0.188 0.182 0.184 0.184 0.190 0.181 0.187
## [13] 0.187 0.183 0.183 0.193 0.192 0.195 0.198
```

Tuning ‘ntree‘ and ‘mtry

```
plot(1:19, rf.error.p, pch=16,
     main = "Testing errors of mtry with 250 trees",
     xlab="mtry",
     ylab="OOB mse of mtry")
lines(1:19, rf.error.p)
```



Tuning ‘ntree‘ and ‘mtry

Run above loop a few time, it is not very unstable. Notice

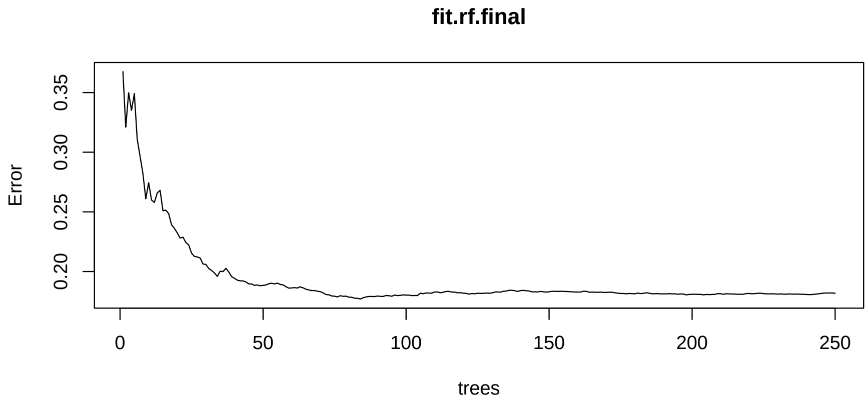
1. mtry = 1 is clearly not a good choice.
2. mtry = 19 is bagging, better but not the best.
3. The recommended mtry for reg trees are mtry=p/3=19/3 about 6 or 7. Seems to agree with this example. Are you convinced with p/3?

We should treat mtry to be a tuning parameter!

The final fit: we take mtry = 6

Tuning ‘ntree‘ and ‘mtry

```
fit.rf.final <- randomForest(LogSalary~., data1, mtry=6, ntree=250)
plot(fit.rf.final)
```



Tuning ‘ntree‘ and ‘mtry

Remark:

- We don’t need to split data to get testing error
- The testing error is provided using OOB errors. In this case since our final model uses `mtry`=6, `ntree`=250, so the testing error is estimated as 0.182

Predictions using RF

```r
# Let's predict rownames(data1)[1]: "-Alan Ashby"
person <- data1[1, ]
fit.person <- predict(fit.rf.final, person)
fit.person
```

```
## -Alan Ashby
##        6.16
```

```r
# the fitted salary in log scale is 6.196343
# (may not be the same each time we run the rf, why?)
```

Alan Ashby's true log salary is:

```r
alan_logsalary <- data1$LogSalary[1]
alan_logsalary
```

```
## [1] 6.16
```

Recap

randomForest() has the following algorithm:

1. Take B many bootstrap samples
2. Build a deep random tree for each bootstrap sample by splitting only m (`mtry`) randomly chosen predictors at each split
3. Bag all the random trees by taking average => prediction of y given $x_1, ..., x_p$
4. Use **Out of Bag** observations to provide testing errors to tune `mtry`.

Remarks:

1. The only tuning parameter is mtry, the number of variables to be split at each node.
2. For each tree specify `nodesize`: 5 for regression trees and 1 for classification trees
3. When mtry = p (= 19), randomForest gives us bagging estimates of non-random trees.

Pruning

The binary/top-down greedy algorithm is a forward splitting. One way to improve the perdition error is to find best sub-tree for given size, then use cross-validation to find a tree with small testing error while penalizing the size of the tree. `prune.tree` can be used to find a tree pruned.

We first train a tree using `tree` then send it back to `prune.tree` to get best sub-tree. For detailed information, `help(prune.tree)`.

The following example shows how to get a final tree with 4 nodes with the smallest testing error among all size 4 trees.

Pruning

```
# fit a tree with large size
fit0.single <- tree(LogSalary~CAtBat, data1,
          control=tree.control(nobs=nrow(data1),
                minsize = 4,
                mindev=0.007))
summary(fit0.single)
```

```
Regression tree:
tree(formula = LogSalary ~ CAtBat, data = data1, control = tree.control(nobs = nrow(data1),
    minsize = 4, mindev = 0.007))
Number of terminal nodes:  8
Residual mean deviance:  0.217 = 55.4 / 255
Distribution of residuals:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.970  -0.226   0.019   0.000   0.249   1.160
```

```
(summary(fit0.single))$size # size of the tree
```

```
[1] 8
```

Pruning

```
# prune fit0.single using prune.tree()
fit0.single.p <- prune.tree(fit0.single, best=4)  # change best to see the best sub-tree
# option "best" controls the number of terminal nodes (i.e. leaves)
# best=2 means taking 2 leaves
summary(fit0.single.p)
```

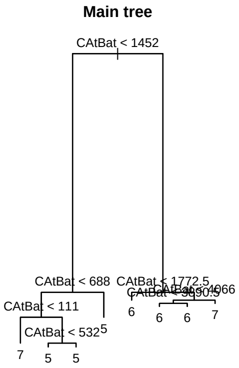
```
Regression tree:
snip.tree(tree = fit0.single, nodes = c(9L, 3L))
Number of terminal nodes:  4
Residual mean deviance:  0.248 = 64.3 / 259
Distribution of residuals:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.960  -0.296   0.032   0.000   0.281   1.340
```

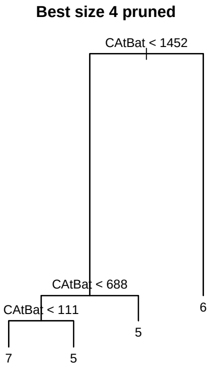
```
data.table(fit0.single.p$frame)
```

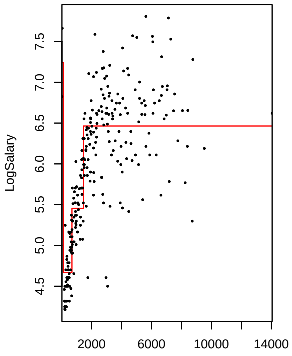
```
      var   n     dev yval splits.cutleft splits.cutright
1: CAtBat 263 207.154 5.93          <1452           >1452
2: CAtBat 103  36.220 5.09           <688            >688
3: CAtBat  54  18.324 4.76           <111            >111
4: <leaf>   2   0.351 7.24
5: <leaf>  52   5.206 4.67
6: <leaf>  49   5.626 5.46
7: <leaf> 160  53.077 6.46
```

Pruning

```
#plot the best subtrees
par(mfrow=c(1, 3))
plot(fit0.single)
title(main="Main tree")
text(fit0.single) # main tree
plot(fit0.single.p)
title(main="Best size 4 pruned")
text(fit0.single.p) # pruned tree
partition.tree(fit0.single.p, col="red", lwd=1)
points(data1$CAtBat, data1$LogSalary, pch=16, cex=.5)
```

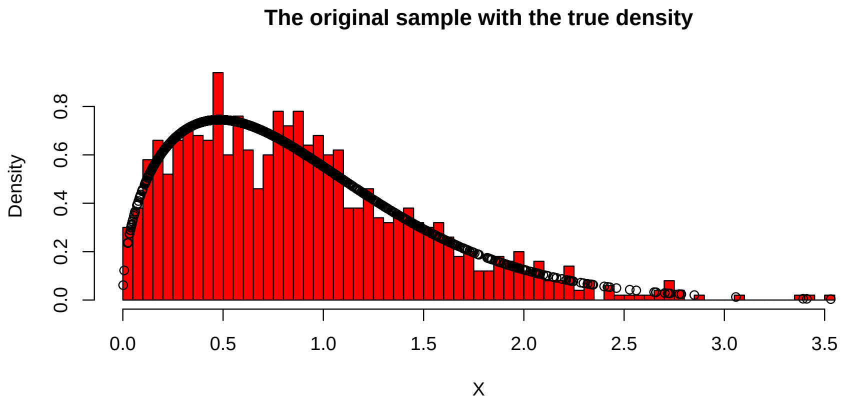






Ensemble Methods: Bootstrap Magic

```r
set.seed(1)
X <- rweibull(1000, shape=1.5)
hist(X, breaks=100, col="red", main="The original sample with the true density",
     freq = FALSE)
points(X, dweibull(X, shape=1.5))
```



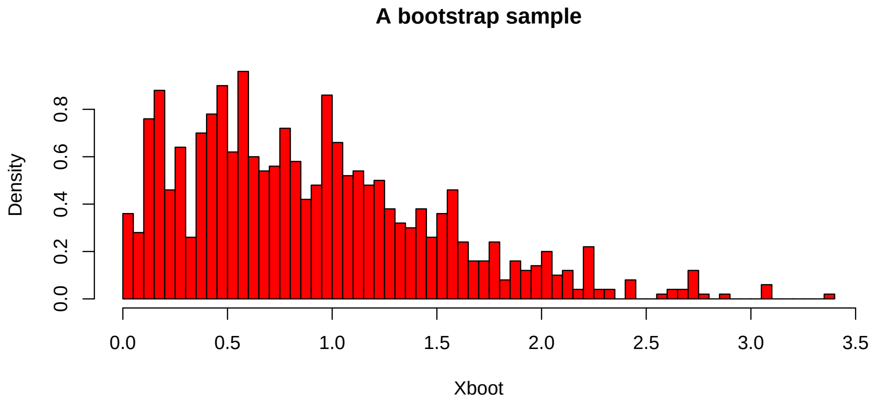
Ensemble Methods: Bootstrap Magic

This shows that the histogram is approximately same as the true density.

ii. Now we take a bootstrap sample: randomly take 1000 obs'n from $x_1, x_2, \ldots, x_{1000}$ with replacement. We also show the histogram of the bootstrap sample.

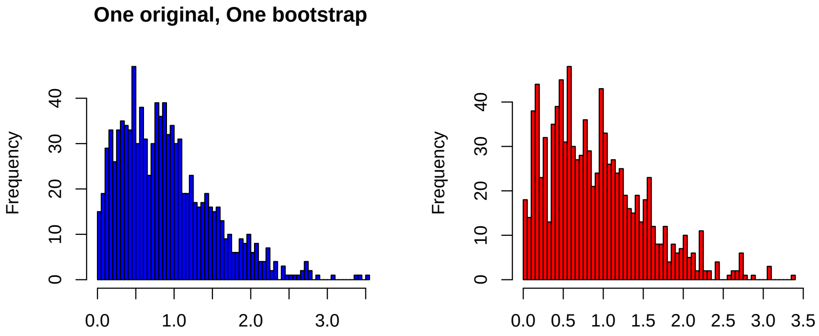
Ensemble Methods: Bootstrap Magic

```r
index <- sample(1000, 1000, replace=TRUE)
Xboot <- X[index]
hist(Xboot, breaks=100, col="red",
     freq = FALSE, main="A bootstrap sample")
```



Ensemble Methods: Bootstrap Magic

iii. Putting two histograms together



Ensemble Methods: Bootstrap Magic

Can you tell the difference between the two histograms? **Not really! They are very similar**.

The reason:

- When n is large, the cdf of X can be well approximated by the empirical cdf which corresponds to the histogram of X.
- A bootstrap sample is same as taking a random sample from the histogram of the first sample which is similar to the true density.

Implication:

- We can get another sample from the population for free!
- We can then produce any statistics we want for free.

In this lecture, we produce many threes from bootstrap samples. Then take average of the trees as a final predictive model.

Random forest: randomForest()

Out-of-Bag (OOB) prediction and testing error estimation:

When a bootstrap sample is drown, on average there will be about $1/e = 37\%$ of the observations not chosen. The proof is simple: P(one person not chosen)=P(not chosen at any round)= $(1 - P(chosen))^n = (1 - 1/n)^n \approx 1/e$Those observations are collected for this particular bootstrap sample and they are called Out of Bag observations (OOB). Since for each tree we only use the bootstrapped observations, hence the OOB observations can be used as a testing data for this tree.

Here is how RF estimate the testing errors using OOB observations.

OOB prediction To get the predicted value for each observation, we can find the trees in which this observation is OOB and then get predicted value from each of these trees and take average. This will be only used as testing predicted values for each observation.

OOB testing errors We then take the residual squared of errors using OOB predicted values.