Classification and Regression Trees (CART)

Gradient Boosted Machines

Decision Trees and Regression Trees

Both

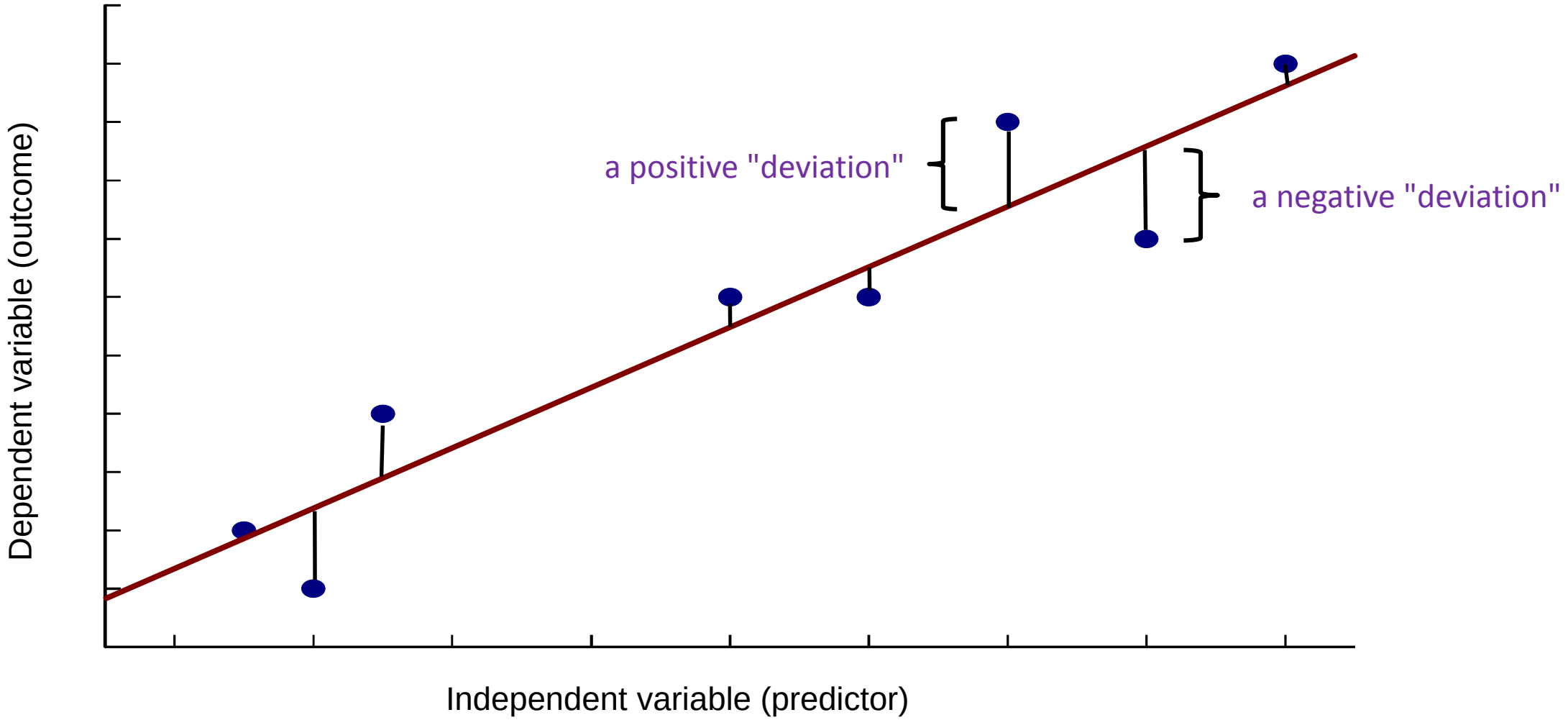
- use continuous, ordinal, binary, or nominal (dummy) predictors
- split the data into subsets - *recursive binary splitting*
- use a *greedy* algorithm - at each split - a branch is forever
- calculate no parameters, rather, sectioning/cut points
- can be tuned and pruned - how "bushy" do you want it?

Regression Trees

- chooses splits that minimize the RSS (Residual sum of squares)

Danger: overfitting (so we need to randomize training data)

Ordinary Least Squares (OLS) Regression minimizes the Residual Sum of Squares



Regression Trees

Chooses splits that minimize the RSS (Residual sum of squares)

For example, the expected value for a normal sample = mean

But often there are other predictors that "cluster" the outcomes

Plus, we can use Ensemble methods such as bagging, boosting, and Random Forests methods

We will use Gradient Boosting too - choosing models based on a "loss function"

Regression Trees: Nodes have means, not accuracies

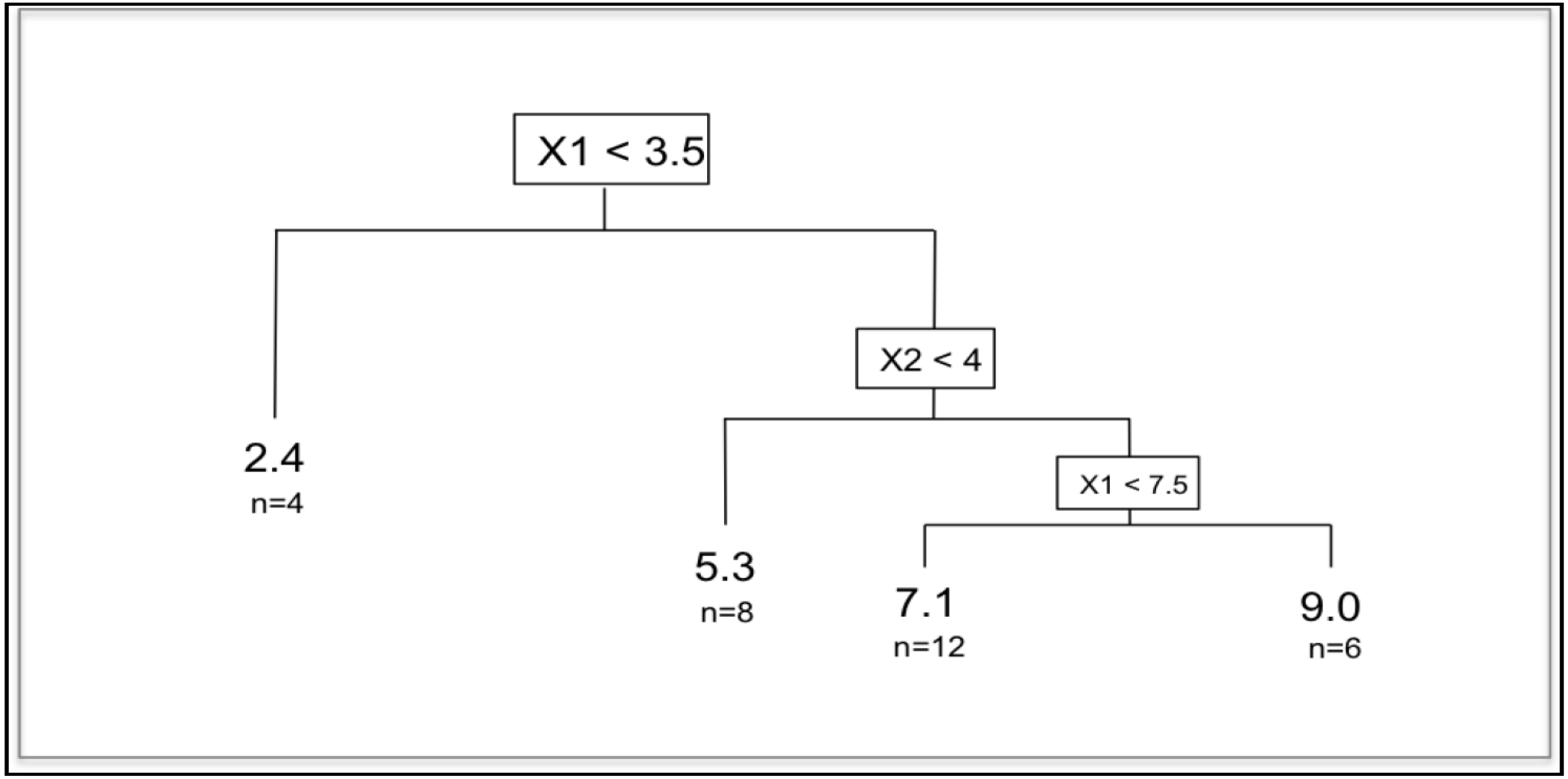
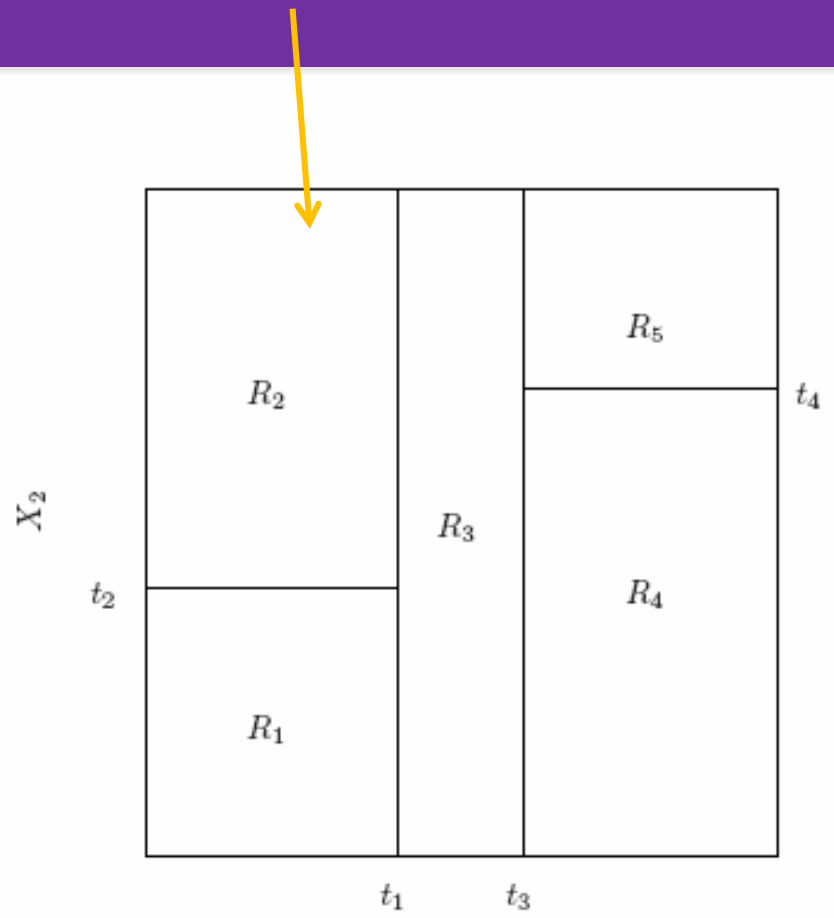
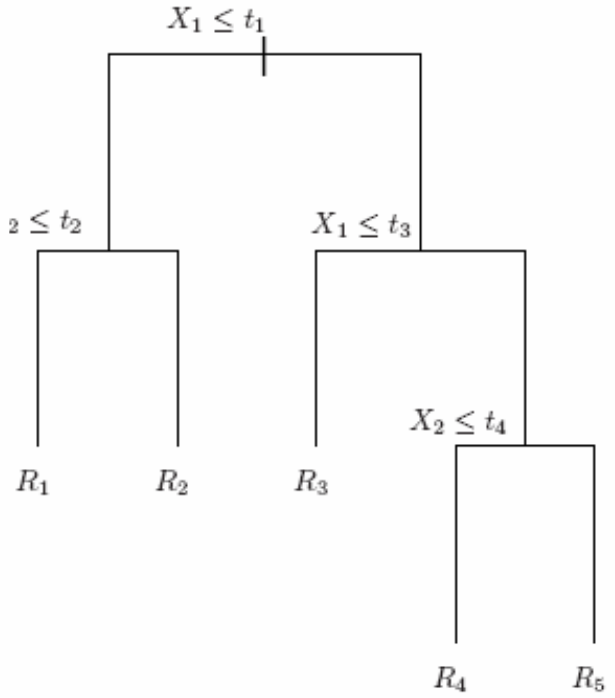


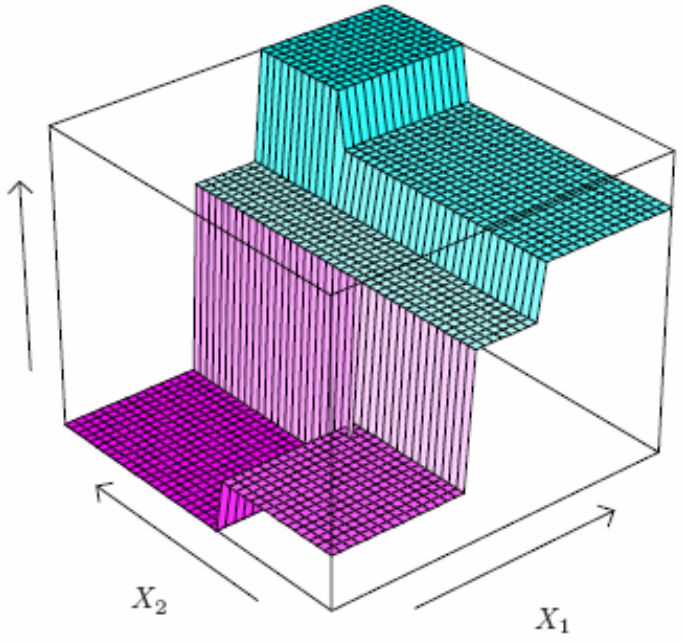
Figure 6.1: Regression Tree with 3 splits and 4 terminal nodes and the corresponding node average and number of observations

2-D, tree, 3-D

Minimize RSS within regions (maximize mean differences among regions)







Deeper Trees

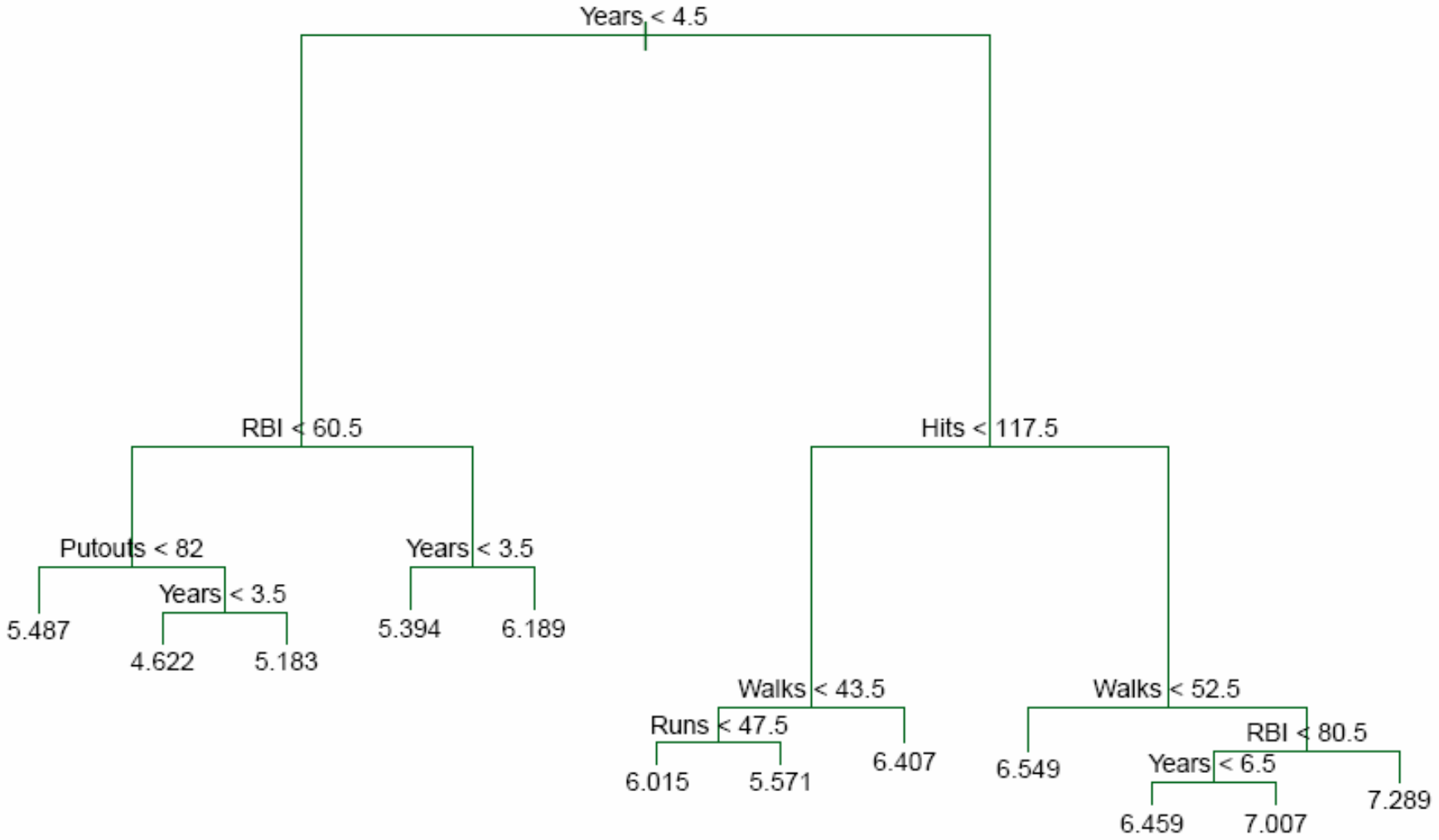


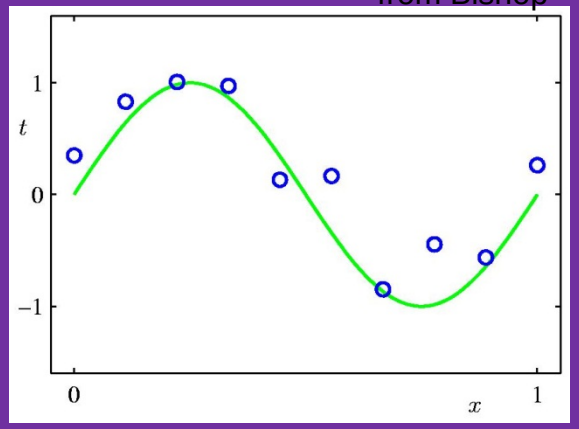
FIGURE 8.4. *Regression tree analysis for the* `Hitters` *data. The unpruned tree that results from top-down greedy splitting on the training data is shown.*

Modeling using a loss function:
Fitting a polynomial model

The green curve is the true function

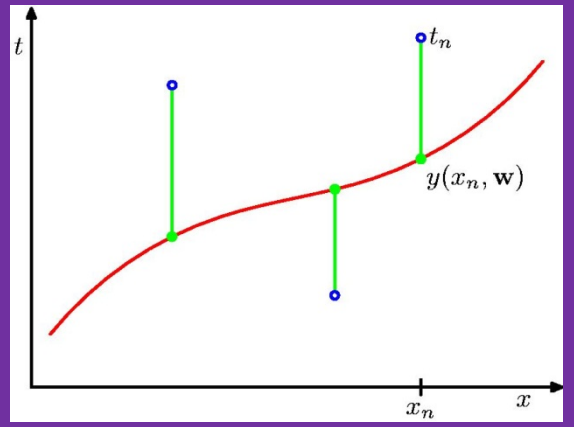
The data points are uniform in x but have noise in y.

from Bishop



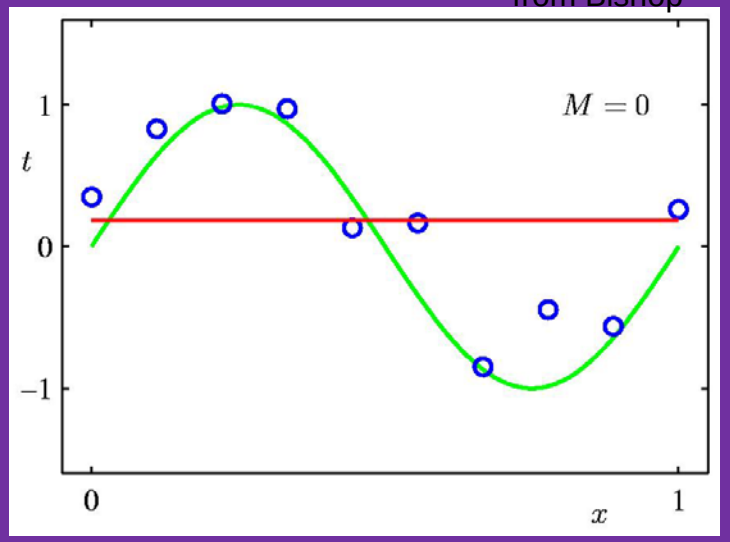
A loss function could measure the squared error in the prediction of y(x) from x. The loss for the red polynomial is the sum of the squared vertical errors.

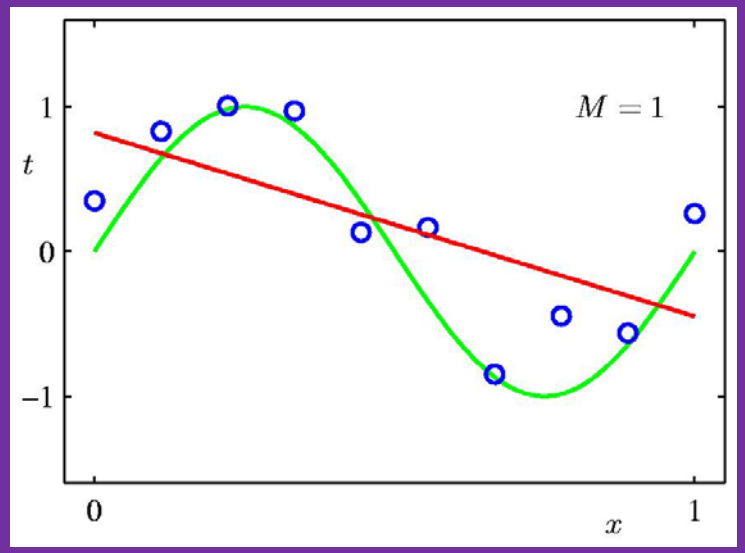
(similar to least-squares regression)

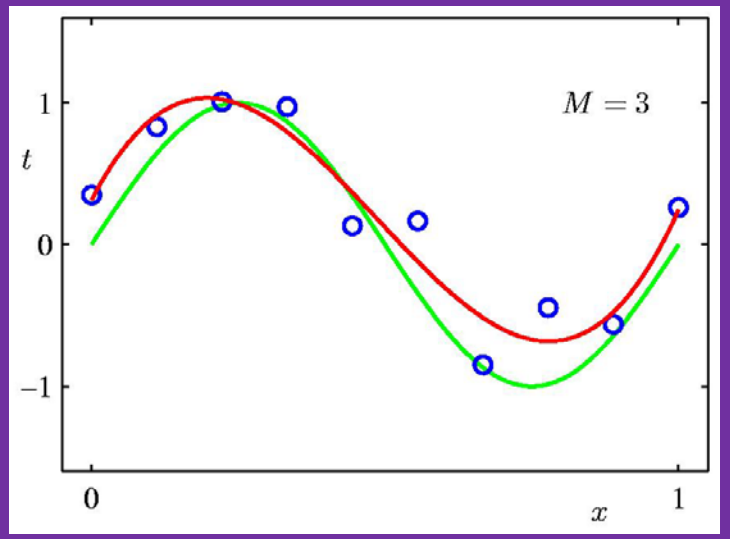


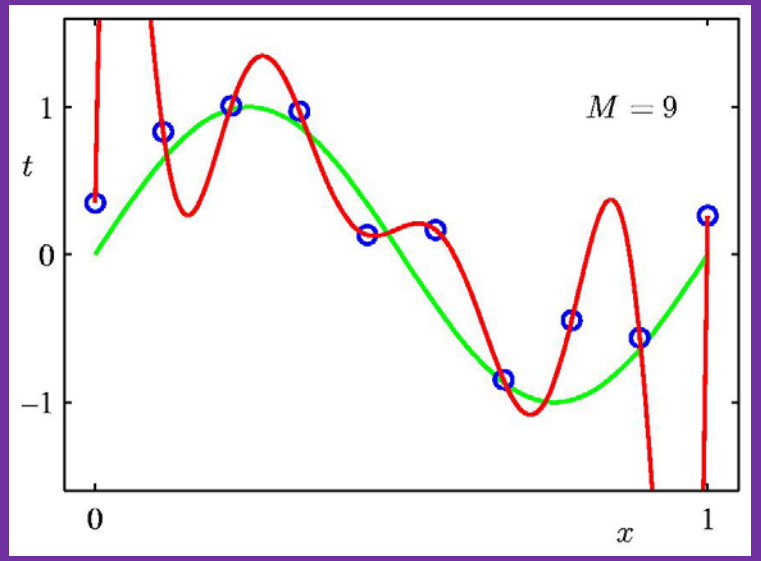
Some fits to the data: which is best?

from Bishop



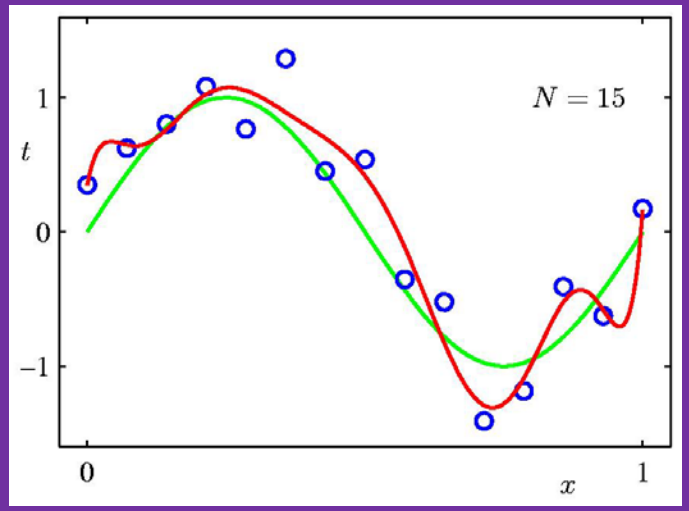






A simple way to reduce model complexity

If we penalize polynomials that have big values for their coefficients, we will get less wiggly solutions:



penalized loss function

regularization parameter

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

target value

CART

```
# check libraries
require(rpart) #classification and regression trees (CART)
require(partykit) #treeplots
require(MASS) #breast and pima indian data
require(ElemStatLearn) #prostate data
require(randomForest) #random forests
require(gbm) #gradient boosting
require(caret) #tune hyper-parameters
```

Prostate surgery data:

lcavol	log cancer volume
lweight	log prostate weight
age	in years
lbph	log of the amount of benign prostatic hyperplasia
svi	seminal vesicle invasion
lcp	log of capsular penetration
gleason	a numeric vector
pgg45	percent of Gleason score 4 or 5
lpsa	**response , PSA levels**
train	a logical vector

CART

```
data(prostate)
```

```
          lcavol  lweight age        lbph svi         lcp gleason pgg45       lpsa train
1  -0.579818495 2.769459  50 -1.38629436   0 -1.38629436       6     0 -0.4307829  TRUE
2  -0.994252273 3.319626  58 -1.38629436   0 -1.38629436       6     0 -0.1625189  TRUE
3  -0.510825624 2.691243  74 -1.38629436   0 -1.38629436       7    20 -0.1625189  TRUE
4  -1.203972804 3.282789  58 -1.38629436   0 -1.38629436       6     0 -0.1625189  TRUE
5   0.751416089 3.432373  62 -1.38629436   0 -1.38629436       6     0  0.3715636  TRUE
6  -1.049822124 3.228826  50 -1.38629436   0 -1.38629436       6     0  0.7654678  TRUE
7   0.737164066 3.473518  64  0.61518564   0 -1.38629436       6     0  0.7654678 FALSE
8   0.693147181 3.539509  58  1.53686722   0 -1.38629436       6     0  0.8544153  TRUE
9  -0.776528789 3.539509  47 -1.38629436   0 -1.38629436       6     0  1.0473190 FALSE
10  0.223143551 3.244544  63 -1.38629436   0 -1.38629436       6     0  1.0473190 FALSE
```

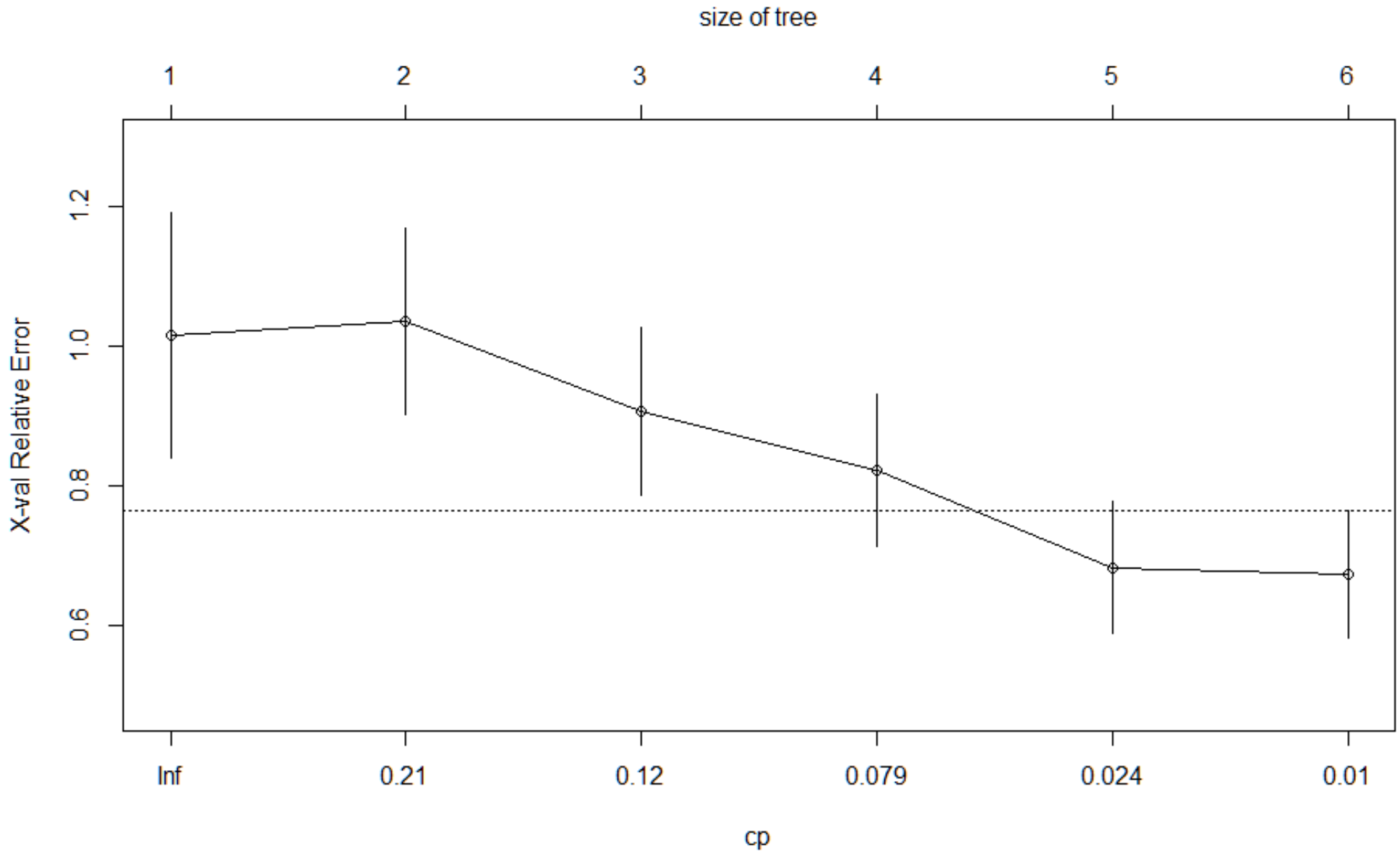
```
prostate$gleason = ifelse(prostate$gleason == 6, 0, 1)
pros.train = subset(prostate, train==TRUE)[,1:9]
pros.test = subset(prostate, train==FALSE)[,1:9]

# very simple call, many defaults
tree.pros = rpart(lpsa~., data=pros.train)

print(tree.pros$cptable)
```

CART

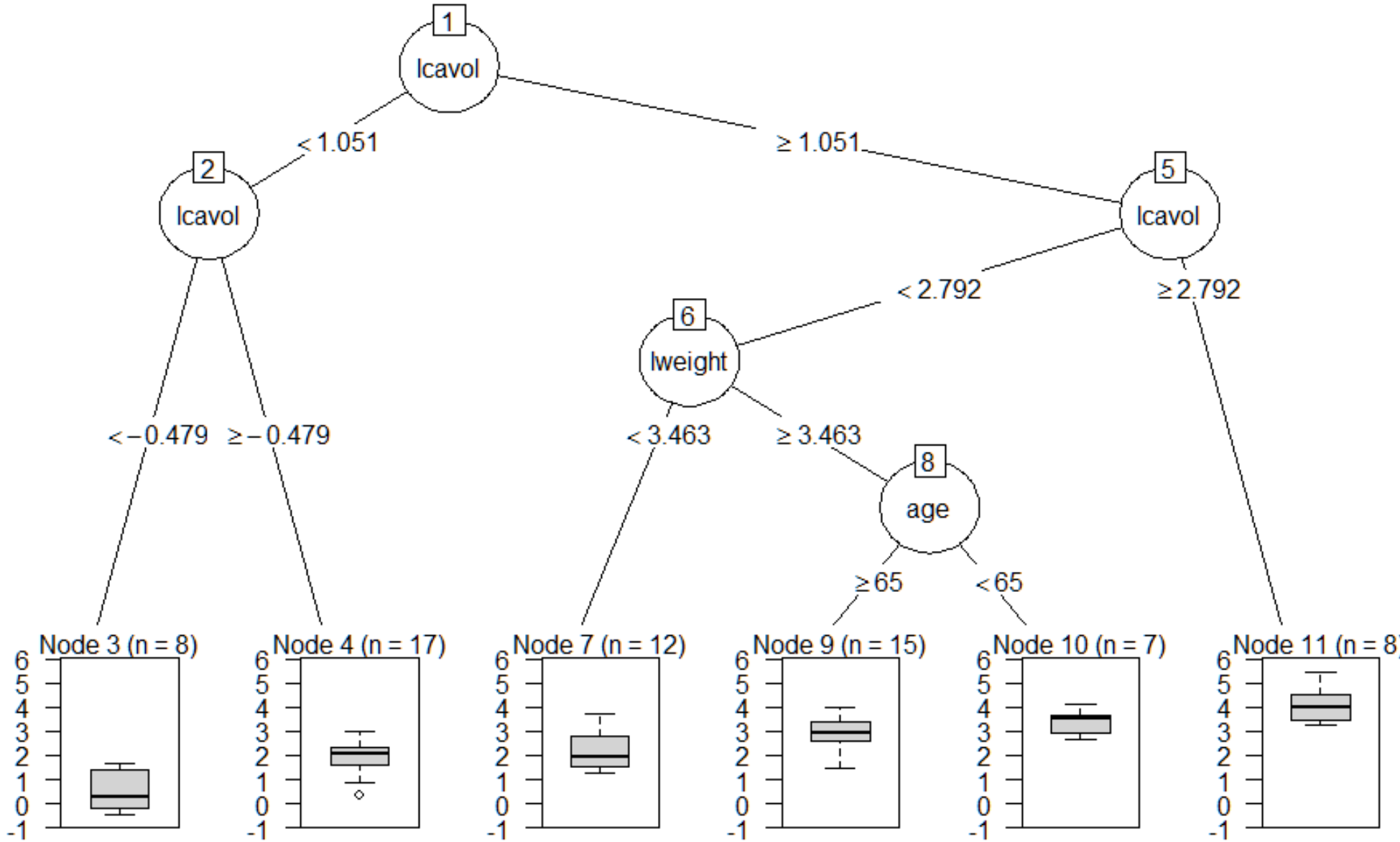
```
# get relative RSS error, cv error for number of tree splits
print(tree.pros$cptable)
          CP nsplit rel error    xerror       xstd
1 0.35852251      0 1.0000000 1.0160388 0.17588408
2 0.12295687      1 0.6414775 1.0356473 0.13339844
3 0.11639953      2 0.5185206 0.9059909 0.12031309
4 0.05350873      3 0.4021211 0.8212585 0.10894600
5 0.01032838      4 0.3486124 0.6819445 0.09507033
6 0.01000000      5 0.3382840 0.6720579 0.09113313

^^^ these are different from the book chapter!
```

CART uses a complexity parameter, which is a "penalty" or "cost" for using another predictor in the model.

cp influences RMSE



plotcp(tree.pros)



plot(as.party(tree.pros))

CART

```
# Predict test set using training model
party.pros.test = predict(tree.pros, newdata=pros.test)

rpart.resid = party.pros.test - pros.test$lpsa #calculate residuals

rpart.resid
           7            9           10           15           22           25
 1.215827506 -0.501339500  0.933976306  0.822812133  1.689080943  0.249639806
          26           28           32           34           36           42
 0.455087333  0.164843206 -0.026918694 -0.040252294  1.261636943  0.583275053
          44           48           49           50           53           54
 0.972403243  0.322059553  0.755792643  0.299331253 -0.703144994  0.199604553
          55           57           62           64           65           66
 1.395918450 -0.806797594  0.493716543  0.008844153  0.465305543  0.459718943
          73           74           80           84           95           97
-0.165509247 -0.853476467 -0.165727857 -0.680092547 -1.042494750 -1.482302450

# calculate RMSE - what we want to minimize
mean(rpart.resid^2)
[1] 0.6136057
```

We can use caret to find the best value for cp using cross-validation

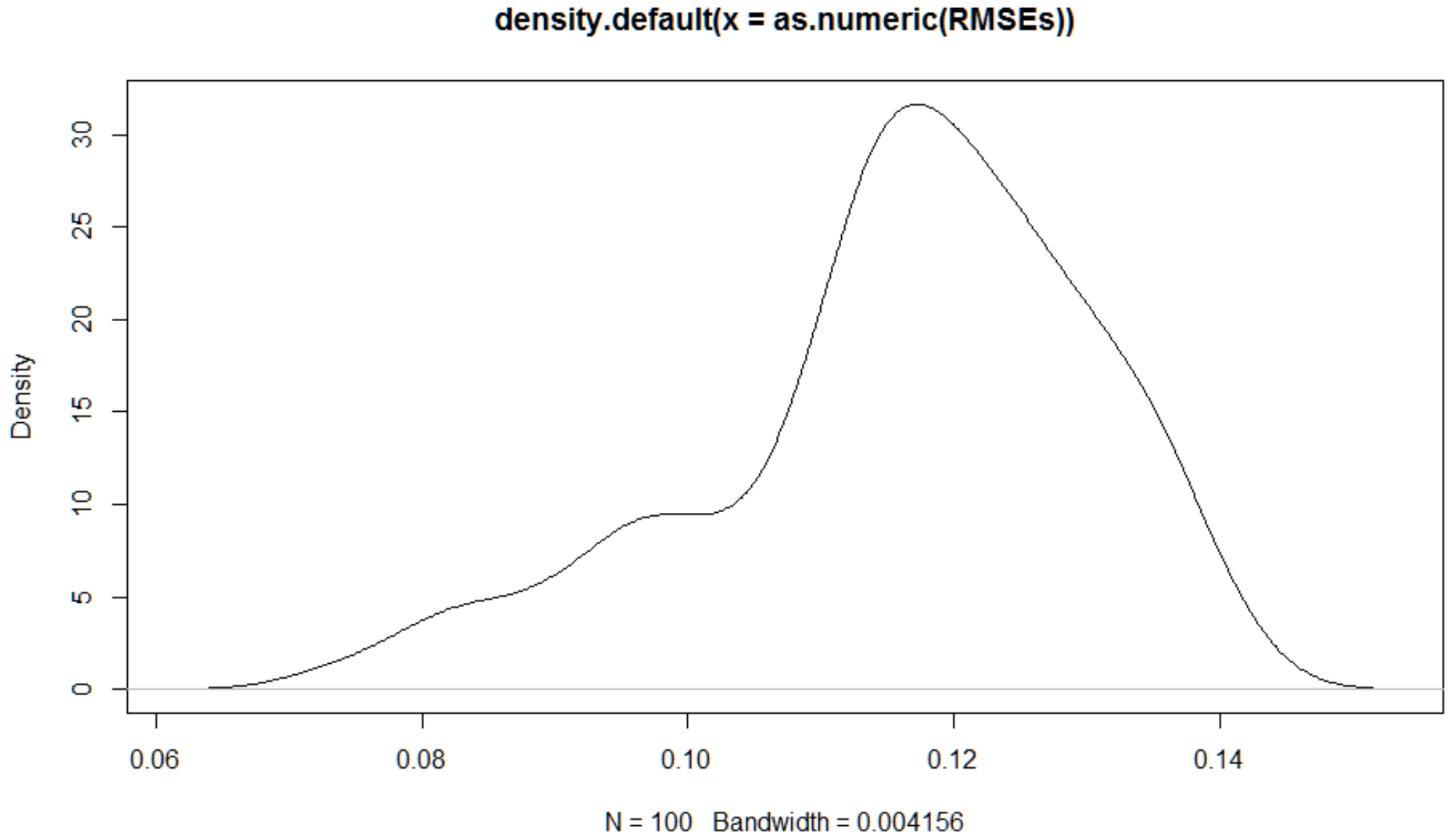
Training CART with caret

```
# sample
inTrain <- createDataPartition(prostate$lpsa, p = .80, list = FALSE)
training <- prostate[inTrain,]
testing <- prostate[-inTrain,]

# fit training sample
PT <- train(training[,c(1,2,3,5:9)], training[,"lpsa"], method = "rpart", metric = "RMSE", trControl =
trainControl(method = "cv"))

PT
CART

79 samples
 8 predictor

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 71, 71, 71, 71, 71, 71, ...
Resampling results across tuning parameters:

  cp         RMSE       Rsquared   MAE
  0.1129282  0.5948233  0.8336681  0.4822880
  0.1580465  0.6518505  0.7579835  0.5358285
  0.6027546  0.9372021  0.5727659  0.7819939

RMSE was used to select the optimal model using  the smallest value.
The final value used for the model was cp = 0.1129282.
```

Training CART with caret: cp

```
PT$bestTune
         cp
1 0.1129282

# the cp changes a little every time, so FIRST let's get the mean of the best cp to use.
bestcps <- c(0.00)
for (i in seq(100))
{
# sample
inTrain <- createDataPartition(prostate$lpsa, p = .80, list = FALSE)
training <- prostate[inTrain,]
testing <- prostate[-inTrain,]

# fit training sample
PT <- train(training[,c(1,2,3,5:9)], training[,"lpsa"], method = "rpart", metric = "RMSE", trControl
= trainControl(method = "cv"))

bestcps[i] <- PT$bestTune
print(i) # give feedback
}
median(as.numeric(bestcps))
[1] 0.117201
plot(density(as.numeric(bestcps)))
```

Training CART with caret: cp

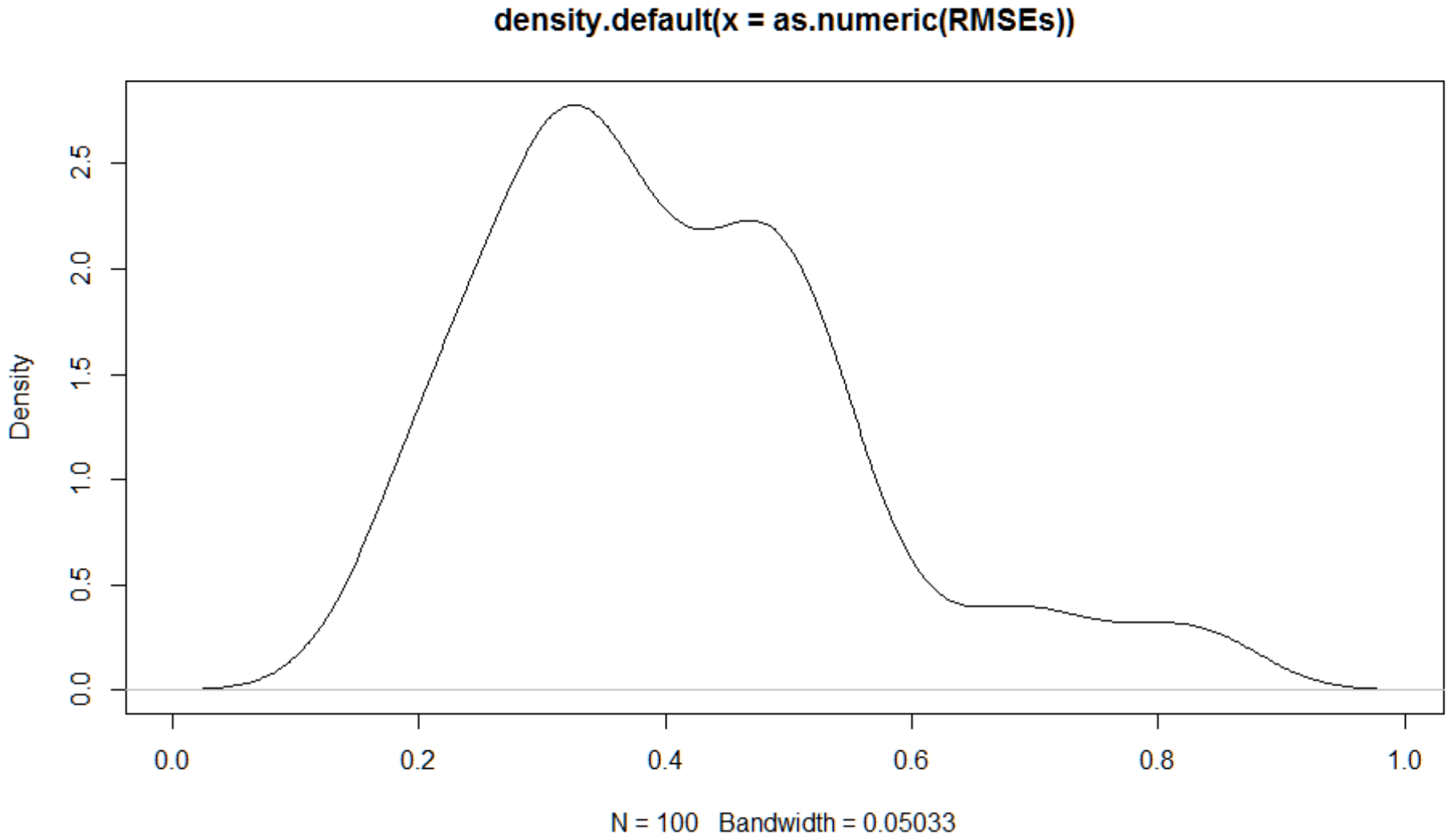



```
plot(density(as.numeric(RMSEs)))
```

Training CART with caret: RMSE

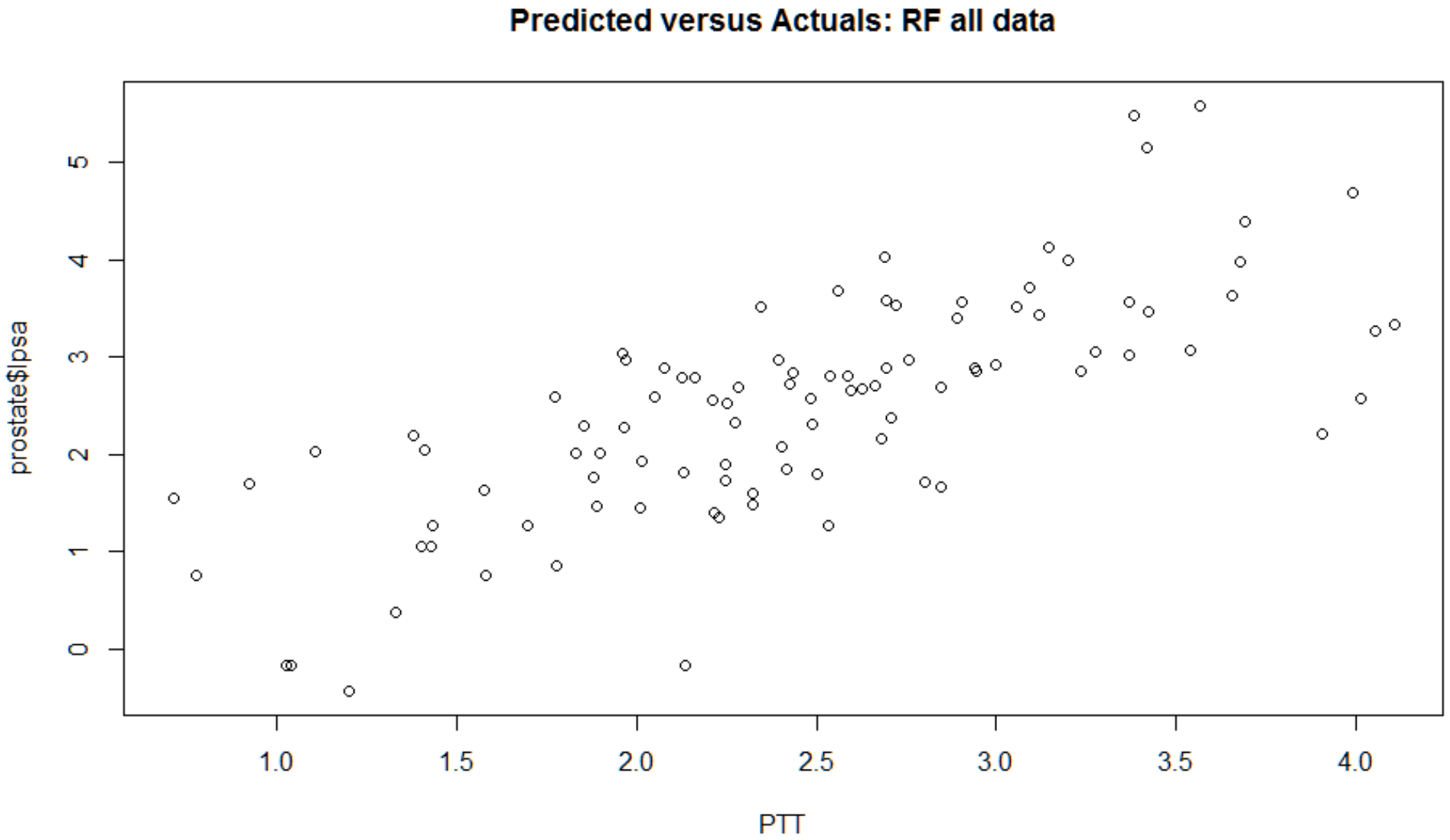
```
# the median cp was 0.117201. Now we want RMSE
RMSEs <- c(0.00)
for (i in seq(100)) # 100 took 40 seconds on my office computer
{
# sample
inTrain <- createDataPartition(prostate$lpsa, p = .80, list = FALSE)
training <- prostate[inTrain,]
testing <- prostate[-inTrain,]

# fit training sample
PT <- train(training[,c(1,2,3,5:9)], training[,"lpsa"], method = "rpart", metric = "RMSE", trControl
= trainControl(method = "cv"))

PTT = predict(PT, newdata = testing)
PTT.resid = PTT - testing$lpsa #calculate residuals

RMSEs [i] <- mean(PTT.resid^2)

print(i) # give feedback
}

median(as.numeric(RMSEs))
[1] 0.3775288
plot(density(as.numeric(RMSEs)))
```

Training CART with caret: RMSE




```
plot(density(as.numeric(RMSEs)))
```

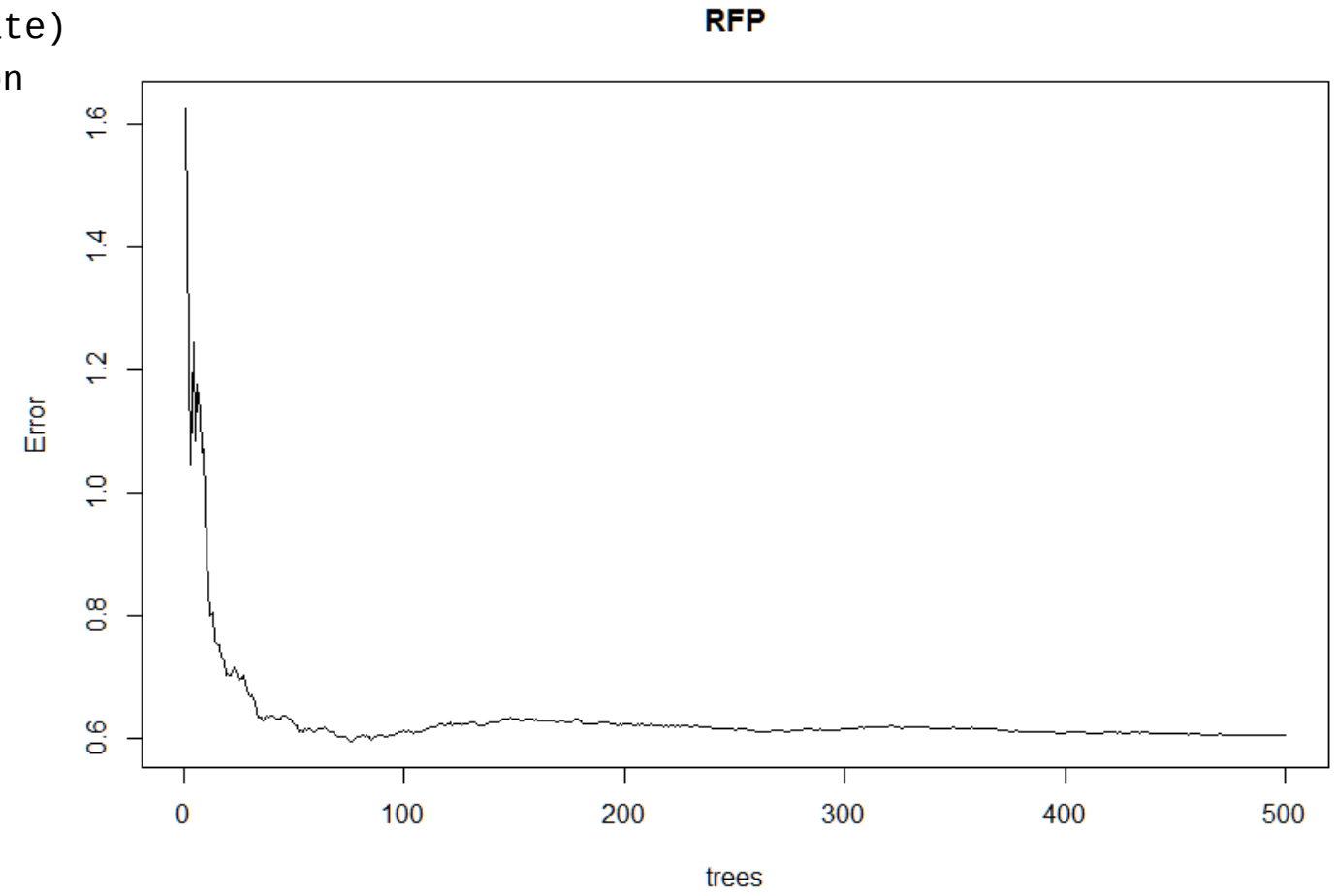
Training CART with caret: RMSE




```
plot(PTT, prostate$lpsa, main="Predicted versus Actuals: RF all data")
```

Random Forest

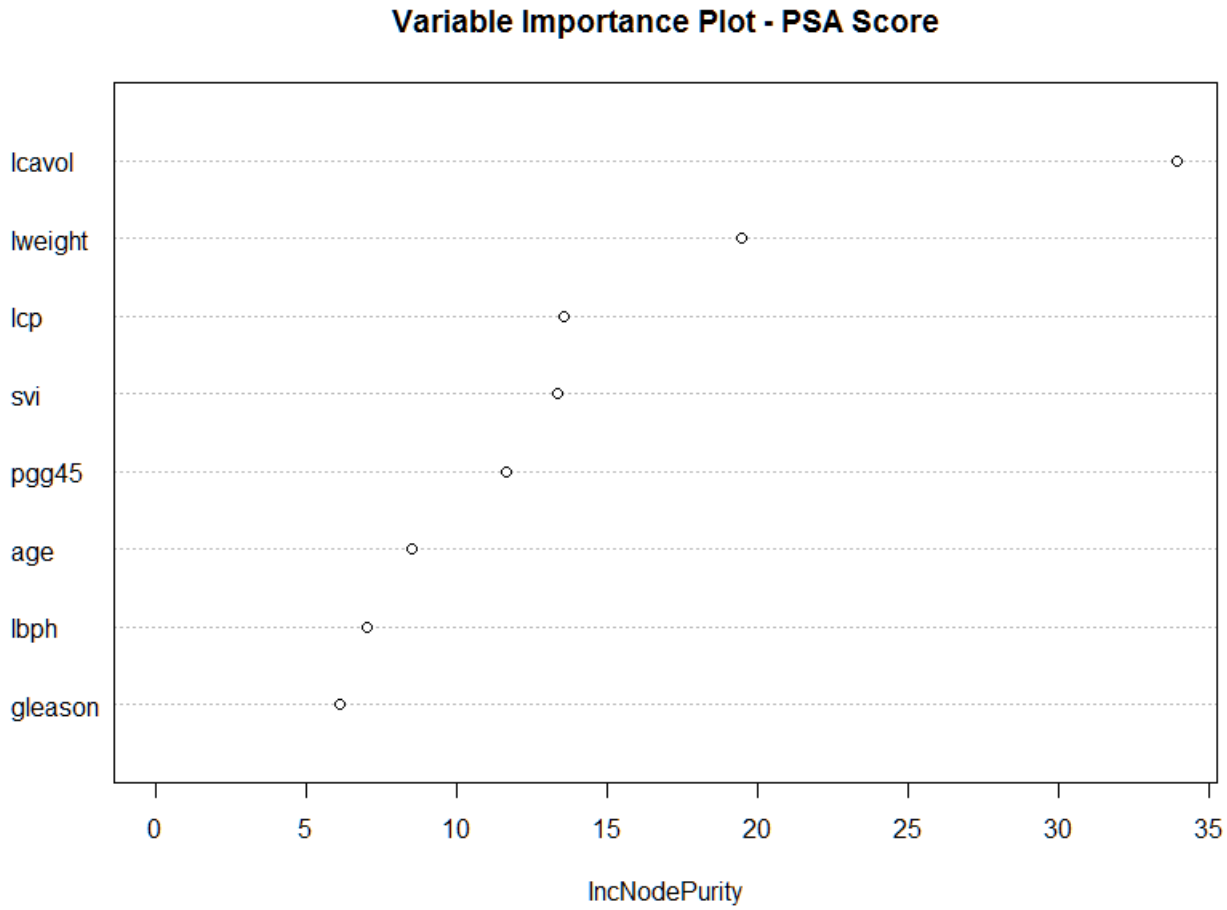
```
# get rid of train column
prostate$train <- NULL

RFP = randomForest(lpsa~., data=prostate) # no need to worry about overfitting
print(RFP)
Call:
 randomForest(formula = lpsa ~ ., data = prostate)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 2

          Mean of squared residuals: 0.6062223
                    % Var explained: 54.03
plot(RFP)

which.min(RFP$mse)
[1] 76
```



Random Forest

```
# plot importance
varImpPlot(RFP, main="Variable Importance Plot - PSA Score")
# get numbers
importance(RFP)
```

```
        IncNodePurity
lcavol      33.900049
lweight     19.452884
age          8.526544
lbph         7.018917
svi         13.324980
lcp         13.561964
gleason      6.116718
pgg45       11.638975
```



Random Forest

```
# run again using 76 trees
RFP = randomForest(lpsa~., data=prostate, ntree = 76)
print(RFP)

Call:
 randomForest(formula = lpsa ~ ., data = prostate, ntree = 76)
               Type of random forest: regression
                     Number of trees: 76
No. of variables tried at each split: 2

          Mean of squared residuals: 0.5940054
                    % Var explained: 54.96
```

A slight improvement

How can we improve our models?

Gradient Booting Regression (GBR)

Gradient Boosting: an Ensemble method that produces a prediction model based on a "loss function" ("cost")

- uses a series of (usually weak) prediction models (decision/regression trees)
- the loss function prevents overfitting
- iteratively tunes previous models
- can be especially useful when you have MANY predictors

Some parameters that influence performance:

Number of trees: number of (boosted) trees to be run iteratively

Interaction depth: How many simultaneous predictor interactions? (Tree Depth)

Shrinkage: rate of learning; contribution of each tree to model (regularizes)

GBR with Caret

We will set up a "grid" of values to be tested to find the best combination using `expand.grid()`

number of trees: 100,300,500

interaction depth: 1,2,3,4

shrinkage: 0.001, 0.01, 0.1

How many combinations? 3 x 4 x 3 = 36

```
grid = expand.grid(.n.trees=seq(100,500, by=200), .interaction.depth=seq(1,4, by=1),
.shrinkage=c(.001,.01,.1), .n.minobsinnode=10)
grid
```

```
   .n.trees .interaction.depth .shrinkage .n.minobsinnode
1       100                 1      0.001             10
2       300                 1      0.001             10
3       500                 1      0.001             10
4       100                 2      0.001             10
5       300                 2      0.001             10
6       500                 2      0.001             10
...
21      500                 3      0.010             10
22      100                 4      0.010             10
...
31      100                 3      0.100             10
32      300                 3      0.100             10
33      500                 3      0.100             10
34      100                 4      0.100             10
35      300                 4      0.100             10
36      500                 4      0.100             10
```

GBR with Caret

```
# Use caret to learn the best combinations using cv
inTrain <- createDataPartition(prostate$lpsa, p = .80, list = FALSE)
training <- prostate[inTrain,]
testing <- prostate[-inTrain,]
PGBT = train(lpsa~., data=training, method="gbm", trControl = trainControl(method="cv"), tuneGrid=grid)
PGBT
```

```
Stochastic Gradient Boosting

79 samples
 8 predictor

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 71, 71, 71, 71, 71, 71, ...
Resampling results across tuning parameters:

  shrinkage  interaction.depth  n.trees  RMSE       Rsquared   MAE
  0.001      1                  100      1.0085731  0.5074927  0.8164992
  0.001      1                  300      0.9536536  0.5299330  0.7673633
...
  0.100      4                  500      0.7861750  0.5487322  0.6234222

Tuning parameter 'n.minobsinnode' was held constant at a value of 10
RMSE was used to select the optimal model using  the smallest value.
The final values used for the model were n.trees = 500, interaction.depth = 3,
 shrinkage = 0.01 and n.minobsinnode = 10.
```

GBR with Caret

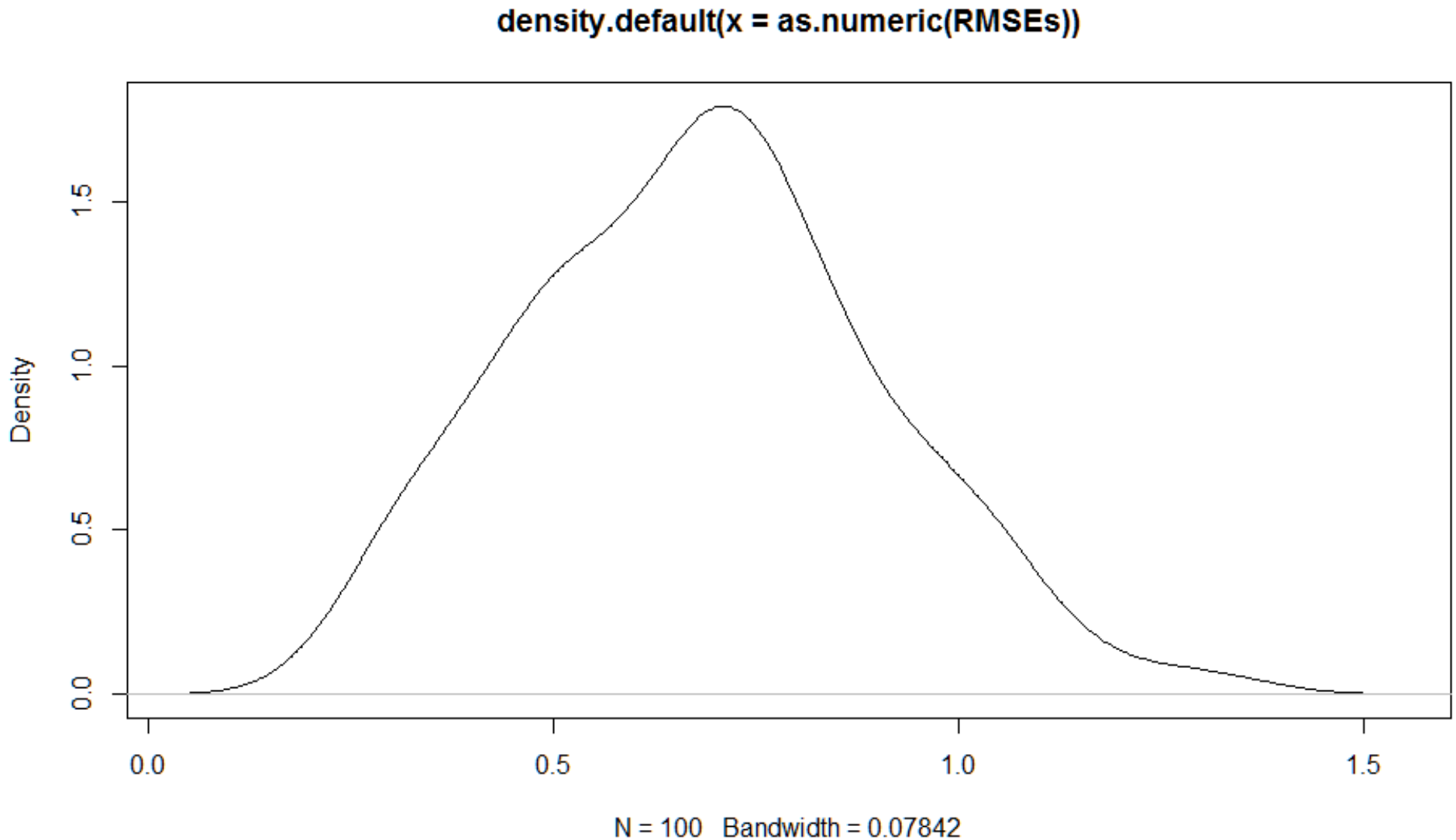
```
# Use caret to learn the best combinations
# I reset the maximum trees to 900 because 500 was the best but is also the max
PGBT = train(lpsa~., data=training, method="gbm", trControl = trainControl(method="cv"), tuneGrid=grid)
PGBT
Stochastic Gradient Boosting

79 samples
 8 predictor

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 71, 71, 72, 71, 71, 71, ...
Resampling results across tuning parameters:

  shrinkage  interaction.depth  n.trees  RMSE       Rsquared   MAE
  0.001      1                  100      1.0262697  0.4900559  0.8182234
  0.001      1                  300      0.9745688  0.5025500  0.7680291
...
  0.100      4                  300      0.7520271  0.5422072  0.5955507
  0.100      4                  500      0.7854956  0.5087766  0.6163335
  0.100      4                  700      0.8206882  0.4797294  0.6470450
  0.100      4                  900      0.8471198  0.4545826  0.6714290

Tuning parameter 'n.minobsinnode' was held constant at a value of 10
RMSE was used to select the optimal model using  the smallest value.
The final values used for the model were n.trees = 700, interaction.depth = 2,
 shrinkage = 0.01 and n.minobsinnode = 10.
```

GBR with Caret

```
# The best combinations are 700 trees, interaction depth = 2, shrinkage = 0.01
# We use "gaussian" for a squared error from a continuous outcome

GBPR = gbm(lpsa~., data = training, n.trees=700, interaction.depth=2, shrinkage=0.01, distribution="gaussian")

GBPP = predict(GBPR, newdata = testing, n.trees = 700)

# residuals
GBPres = GBPP - testing$lpsa
mean(GBPres^2)
[1] 1.037583

# equivalent
mean((GBPP - testing$lpsa)^2)
[1] 1.03166
```

GBR with Caret

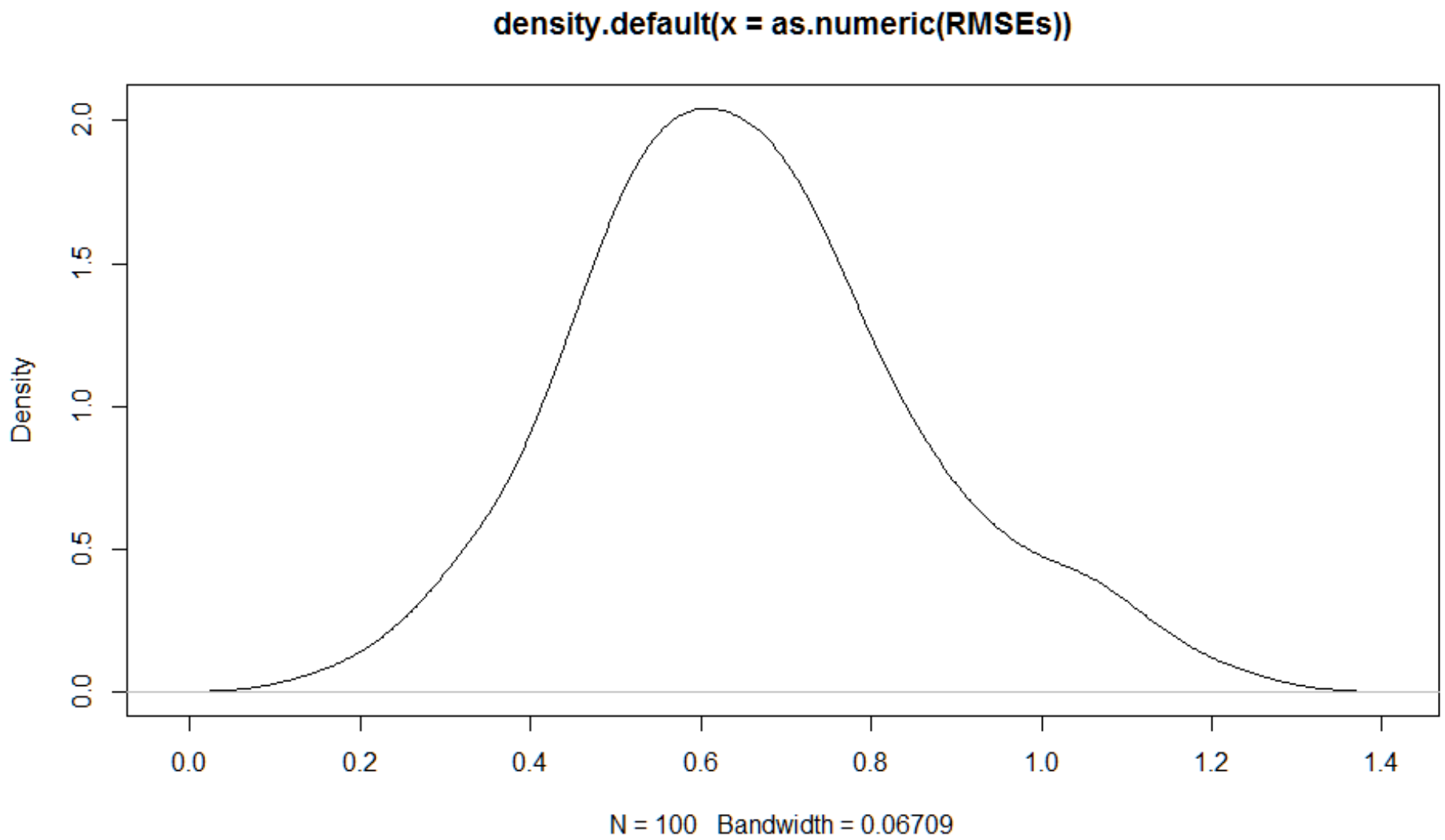
```
RMSEs <- c(0.00)
for (i in seq(100)) # 100 is fast
{
# sample
inTrain <- createDataPartition(prostate$lpsa, p = .80, list = FALSE)
training <- prostate[inTrain,]
testing <- prostate[-inTrain,]

# fit training sample
GBPR = gbm(lpsa~., data = training, n.trees=700, interaction.depth=2, shrinkage=0.01, distribution="gaussian")

GBPP = predict(GBPR, newdata = testing, n.trees = 700)

# residuals

RMSEs [i] <- mean((GBPP - testing$lpsa)^2)

print(i) # give feedback
}

median(as.numeric(RMSEs))
[1] 0.6920637
plot(density(as.numeric(RMSEs)))
```




```
plot(density(as.numeric(RMSEs)))
```

GBR with Caret

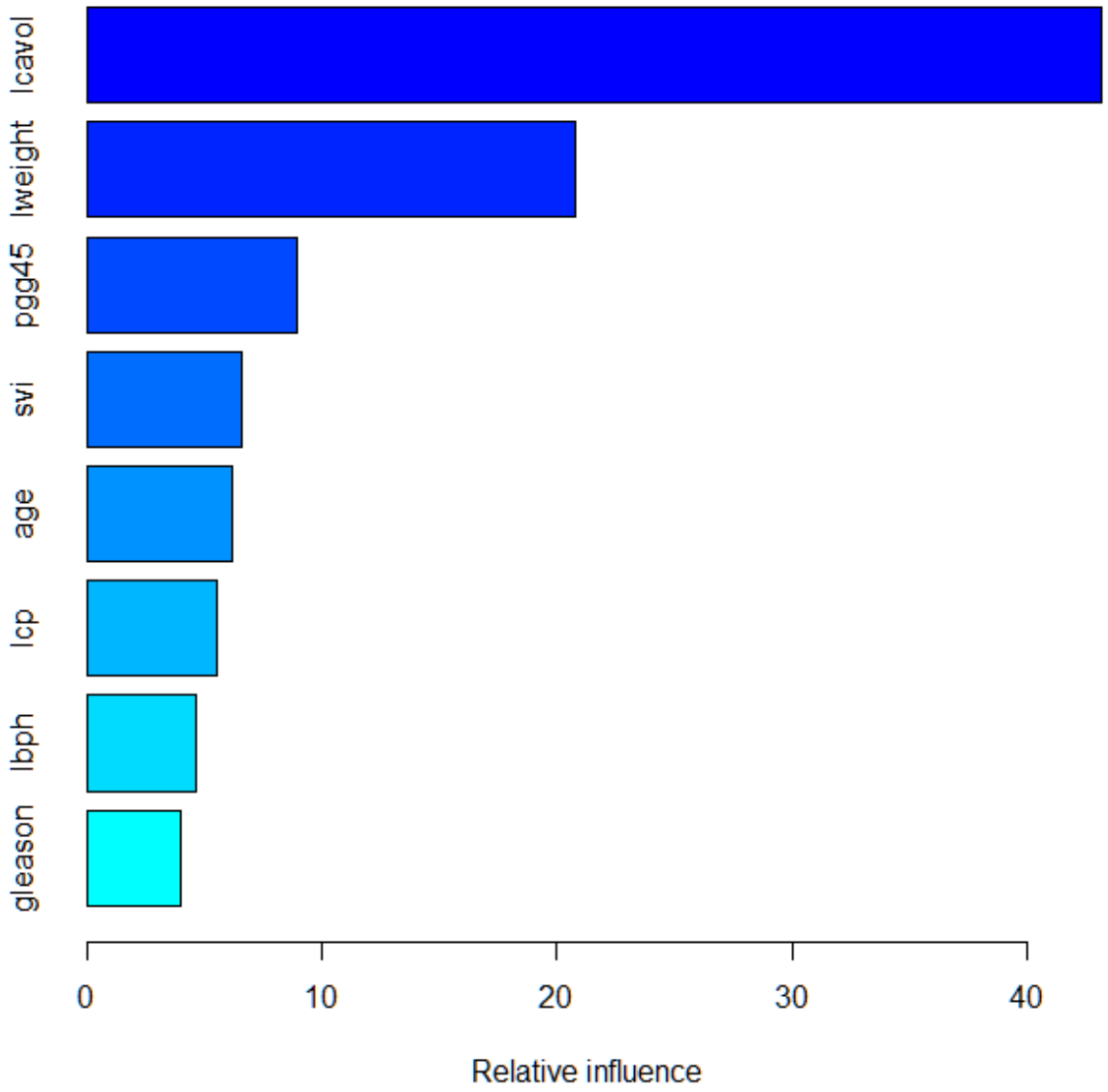
```
require(parallel) # for extra cores
RMSEs <- c(0.00)
for (i in seq(100)) # 100 runs fast
{
# sample
inTrain <- createDataPartition(prostate$lpsa, p = .80, list = FALSE)
training <- prostate[inTrain,]
testing <- prostate[-inTrain,]

# fit training sample
GBPR = gbm(lpsa~., data = training, n.trees=300, interaction.depth=4, shrinkage=0.01, distribution="gaussian",
n.cores = 4)

GBPP = predict(GBPR, newdata = testing, n.trees = 700)

# residuals

RMSEs [i] <- mean((GBPP - testing$lpsa)^2)

print(i) # give feedback
}
median(as.numeric(RMSEs))
[1] 0.6543133
plot(density(as.numeric(RMSEs)))
```



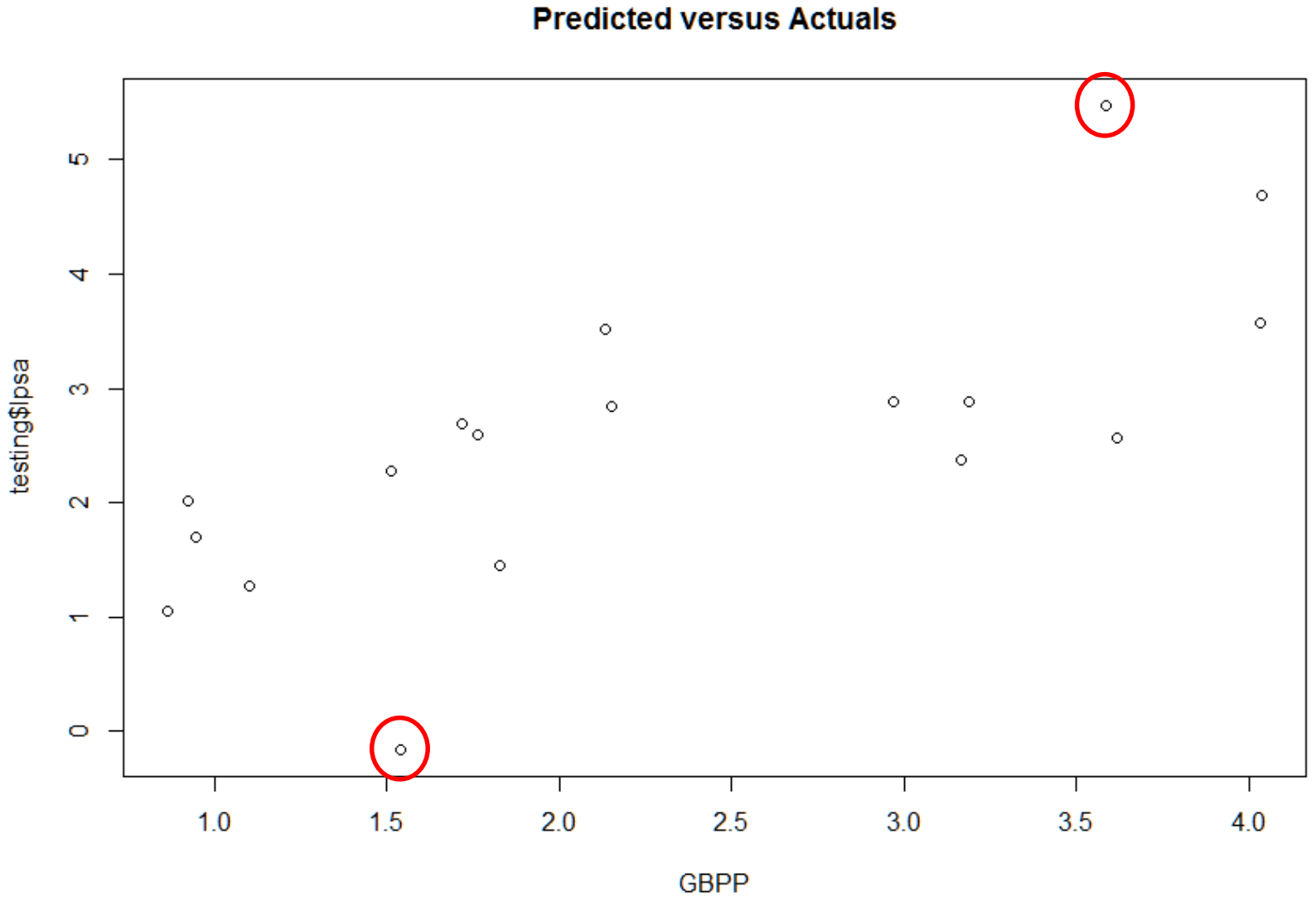

```
plot(density(as.numeric(RMSEs))) # 300 trees, depth = 4
```

GBR

```
# We can get variable importance too, as in RF:
summary(GBPR)
             var   rel.inf
lcavol   lcavol 43.172262
lweight lweight 20.817088
pgg45     pgg45  8.913133
svi         svi  6.623622
age         age  6.221532
lcp         lcp  5.577872
lbph       lbph  4.680782
gleason gleason  3.993709
```



GBR




```
# because it is continuous, we can do a plot of predicted vs. actual
plot(GBPP, testing$lpsa, main="Predicted versus Actuals")
```

Revised Syllabus

14	PLS and PCA variants	CH_6-ISLR;
	ridge regression, lasso	
16	Nonlinear models: GAM, MARS	CH_9-ESL; CH_10-ESL;
21	**Thanksgiving break**	
	I may be available to discuss projects	
23	**Thanksgiving break**	
28	Too few data: Missing Data	CH_18-ESL; CH_25-Gelman-2007;
	Too many data: p > N	CH_18-Kabacoff-2015;
30	Model Tuning III	CH_15-UML; CH_16-UML;
Dec	Which method to use?	
5	Presentations I	
7	Presentations II	
11-15	**Final Exam (take home)**	