Homework_3

Jing Tang

Week6_ISLR Chapter 7 Exercises 9

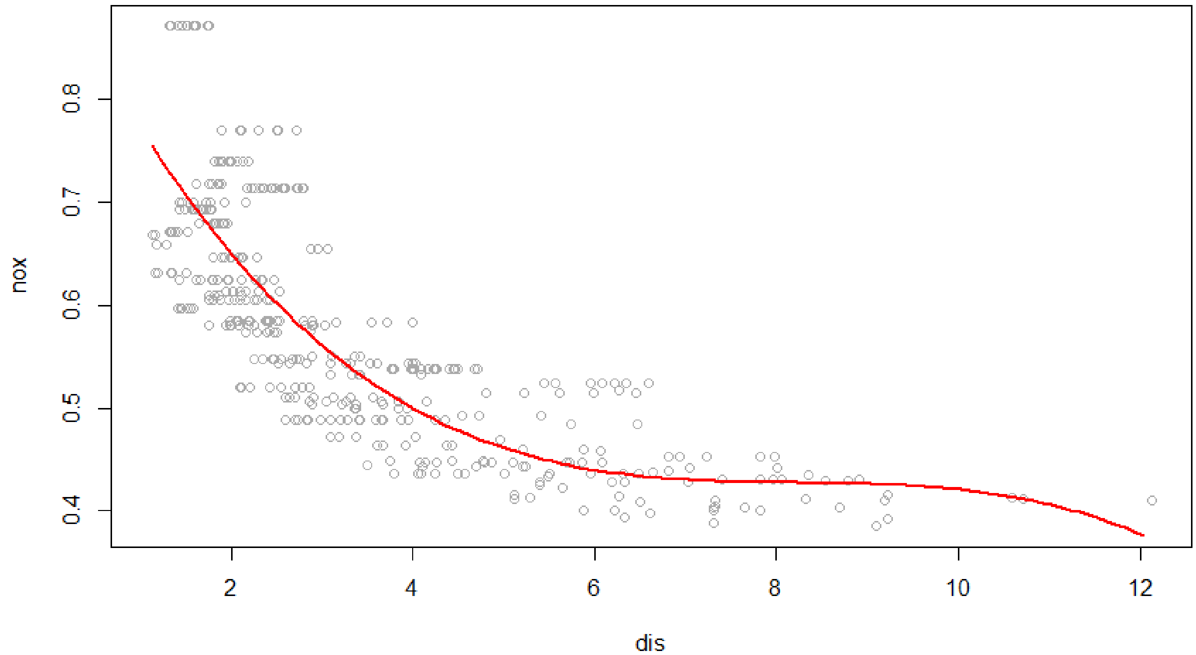
9. This question uses the variables dis (the weighted mean of distances to five Boston employment centers) and nox (nitrogen oxides concentration in parts per 10 million) from the Boston data. We will treat dis as the predictor and nox as the response.

(a) Use the poly() function to fit a cubic polynomial regression to predict nox using dis. Report the regression output, and plot the resulting data and polynomial fits.

```
library(MASS)
library(ISLR)
library(boot)
set.seed(42)
fit <- lm(nox ~ poly(dis, 3), data = Boston)
summary(fit)

dislims <- range(Boston$dis)
dis.grid <- seq(from = dislims[1], to = dislims[2], by = 0.1)
preds <- predict(fit, list(dis = dis.grid))
plot(nox ~ dis, data = Boston, col = "darkgrey")
lines(dis.grid, preds, col = "red", lwd = 2)
```

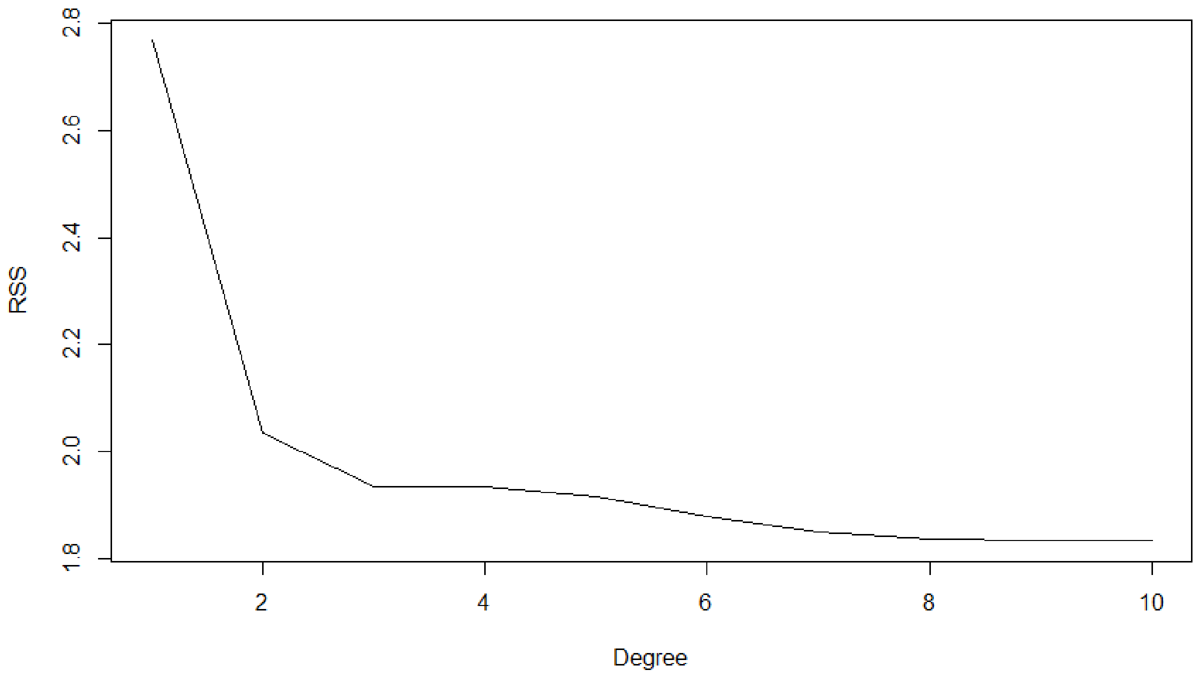
```
Call:
lm(formula = nox ~ poly(dis, 3), data = Boston)

Residuals:
      Min        1Q    Median        3Q       Max 
-0.121130 -0.040619 -0.009738  0.023385  0.194904 

Coefficients:
               Estimate Std. Error t value Pr(>|t|)    
(Intercept)    0.554695   0.002759 201.021  < 2e-16 ***
poly(dis, 3)1 -2.003096   0.062071 -32.271  < 2e-16 ***
poly(dis, 3)2  0.856330   0.062071  13.796  < 2e-16 ***
poly(dis, 3)3 -0.318049   0.062071  -5.124 4.27e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.06207 on 502 degrees of freedom
Multiple R-squared:  0.7148,   Adjusted R-squared:  0.7131 
F-statistic: 419.3 on 3 and 502 DF,  p-value: < 2.2e-16
```

- All terms in spline fit are significant.



(b) Plot the polynomial fits for a range of different polynomial degrees (say, from 1 to 10), and report the associated residual sum of squares.

```
rss <- rep(NA, 10)
for (i in 1:10) {
  fit <- lm(nox ~ poly(dis, i), data = Boston)
  rss[i] <- sum(fit$residuals^2)
}
plot(1:10, rss, xlab = "Degree", ylab = "RSS", type = "l")
rss
```

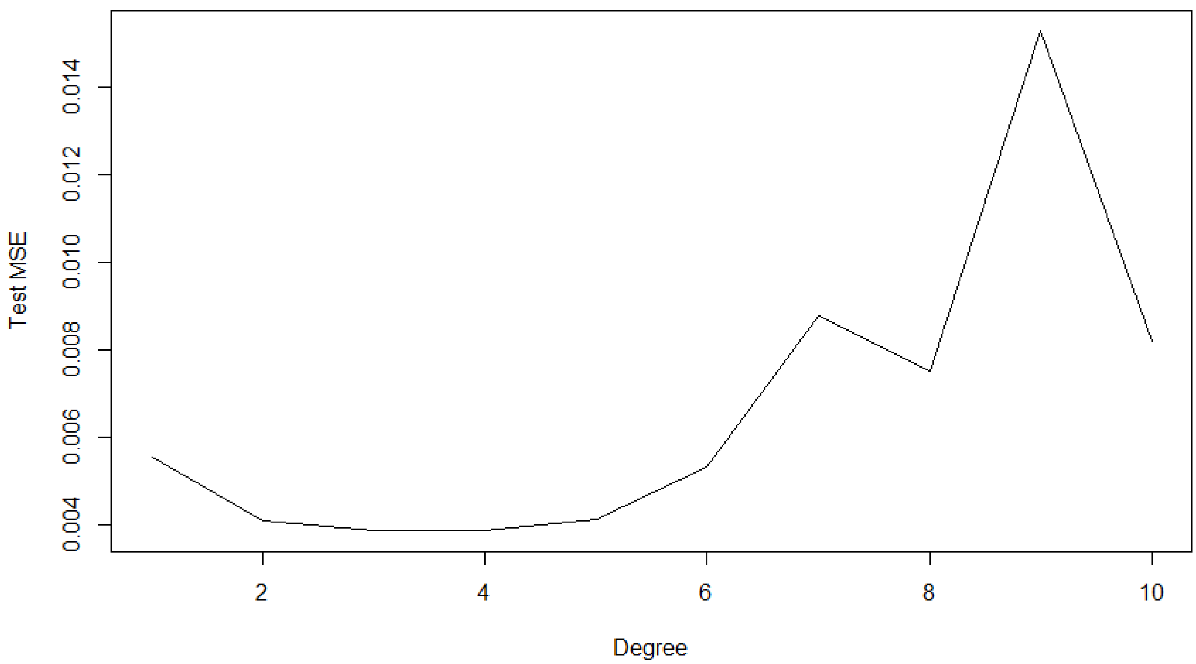



```
rss
[1] 2.768563 2.035262 1.934107 1.932981 1.915290 1.878257 1.849484 1.835630 1.83
3331 1.832171
```

- The RSS decreases with the degree of the polynomial, and so is minimum for a polynomial of degree 10.

(c) Perform cross-validation or another approach to select the optimal degree for the polynomial, and explain your results.

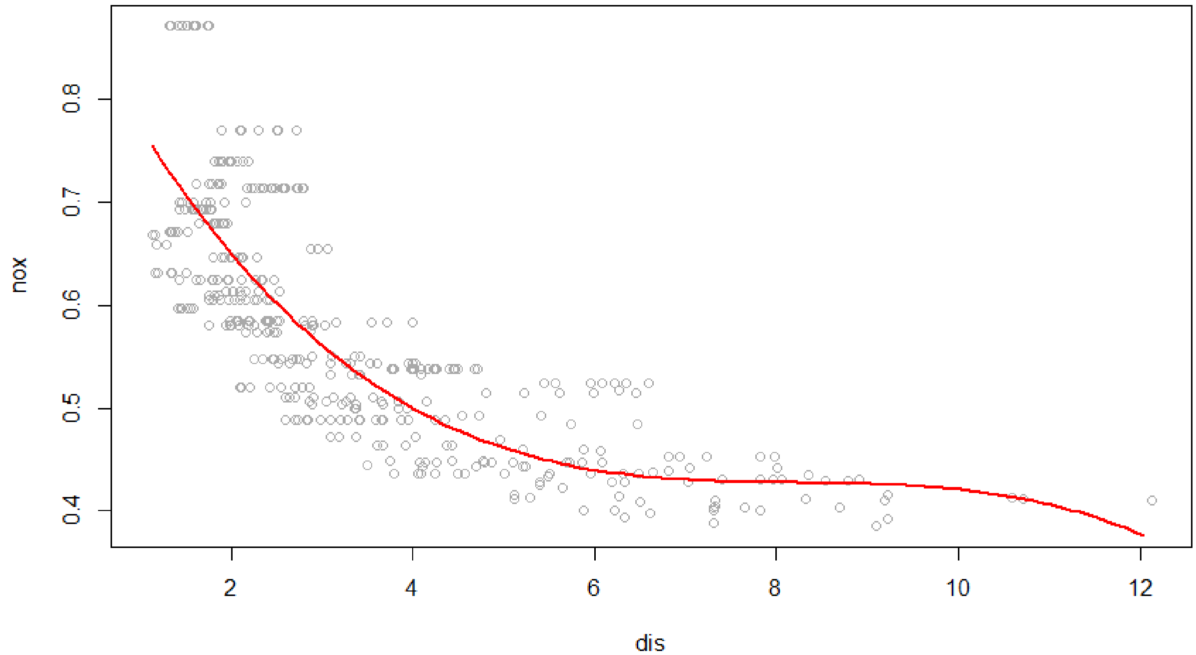
```
deltas <- rep(NA, 10)
for (i in 1:10) {
  fit <- glm(nox ~ poly(dis, i), data = Boston)
  deltas[i] <- cv.glm(Boston, fit, K = 10)$delta[1]
}
plot(1:10, deltas, xlab = "Degree", ylab = "Test MSE", type = "l")
```



- A polynomial of degree 4 minimizes the test MSE.

(d) Use the bs() function to fit a regression spline to predict nox using dis. Report the output for the fit using four degrees of freedom. How did you choose the knots? Plot the resulting fit.

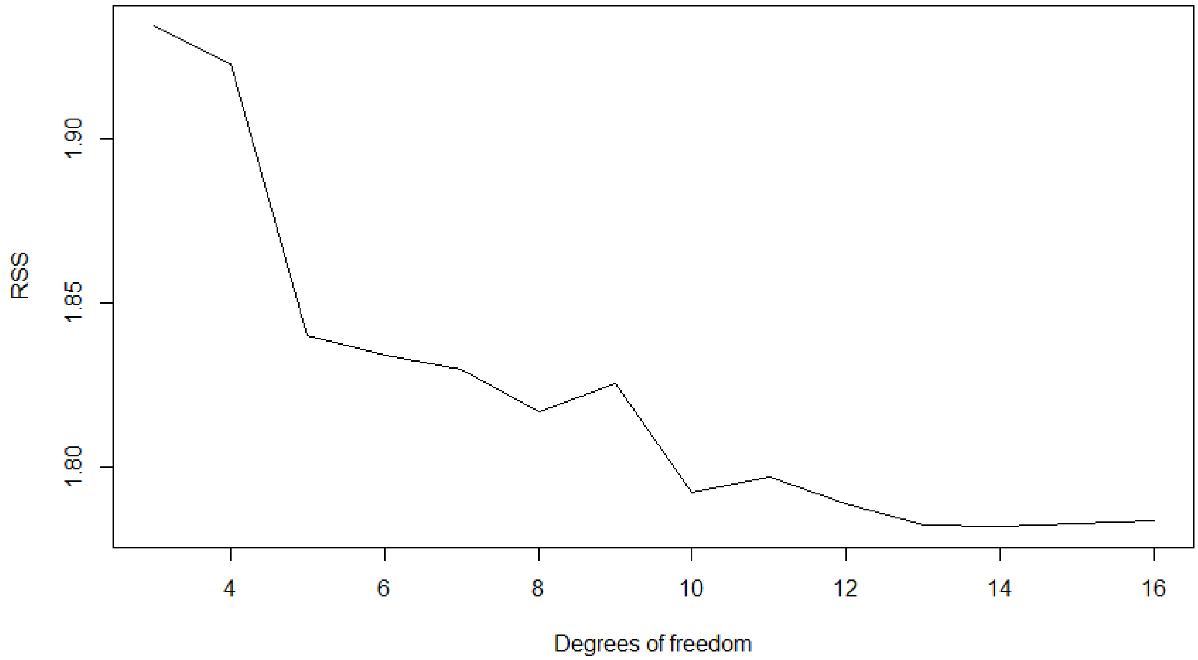
```
Call:
lm(formula = nox ~ bs(dis, knots = c(4, 7, 11)), data = Boston)

Residuals:
      Min        1Q    Median        3Q       Max 
-0.124567 -0.040355 -0.008702  0.024740  0.192920 

Coefficients:
                              Estimate Std. Error t value Pr(>|t|)    
(Intercept)                    0.73926    0.01331  55.537  < 2e-16 ***
bs(dis, knots = c(4, 7, 11))1 -0.08861    0.02504  -3.539  0.00044 ***
bs(dis, knots = c(4, 7, 11))2 -0.31341    0.01680 -18.658  < 2e-16 ***
bs(dis, knots = c(4, 7, 11))3 -0.26618    0.03147  -8.459 3.00e-16 ***
bs(dis, knots = c(4, 7, 11))4 -0.39802    0.04647  -8.565  < 2e-16 ***
bs(dis, knots = c(4, 7, 11))5 -0.25681    0.09001  -2.853  0.00451 ** 
bs(dis, knots = c(4, 7, 11))6 -0.32926    0.06327  -5.204 2.85e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.06185 on 499 degrees of freedom
Multiple R-squared:  0.7185,	Adjusted R-squared:  0.7151 
F-statistic: 212.3 on 6 and 499 DF,  p-value: < 2.2e-16
```



- All terms in spline fit are significant.

(e) Now fit a regression spline for a range of degrees of freedom, and plot the resulting fits and report the resulting RSS. Describe the results obtained.

```
library ( splines )
fit <- lm(nox ~ bs(dis, knots = c(4, 7, 11)), data = Boston)
summary(fit)
pred <- predict(fit, list(dis = dis.grid))
plot(nox ~ dis, data = Boston, col = "darkgrey")
lines(dis.grid, preds, col = "red", lwd = 2)
```




```
rss
 [1]1.934107 1.922775 1.840173 1.833966 1.829884 1.816995 1.825653 1.792535 1.79
6992 1.788999
[13] 1.782350 1.781838 1.782798 1.783546
```

- RSS decreases until 14 and then slightly increases after that.

(f) Perform cross-validation or another approach in order to select the best degrees of freedom for a regression spline on this data. Describe your results.

```
cv <- rep(NA, 16)
for (i in 3:16) {
  fit <- glm(nox ~ bs(dis, df = i), data = Boston)
  cv[i] <- cv.glm(Boston, fit, K = 10)$delta[1]
}
plot(3:16, cv[-c(1, 2)], xlab = "Degrees of freedom", ylab = "Test MSE",
type = "l")
```

- Test MSE is minimum for 10 degrees of freedom.

Week7_ ISLR Chapter 8 Exercises 9

9. This problem involves the OJ data set which is part of the ISLR package.

(a) Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations.

```
library(ISLR)
?OJ
fix(OJ)
names(OJ)
dim(OJ) ## variable names and dataset dimensions
summary(OJ)

library(tree)
set.seed(1)
train <- sample(1:nrow(OJ), 800)
OJ.train <- OJ[train, ]
OJ.test <- OJ[-train, ]
```

(b) Fit a tree to the training data, with Purchase as the response and the other variables as predictors. Use the summary() function to produce summary statistics about the tree, and describe the results obtained. What is the training error rate? How many terminal nodes does the tree have?

```
tree.oj <- tree(Purchase ~ ., data = OJ.train)
summary(tree.oj)
```

```
Classification tree:
tree(formula = Purchase ~ ., data = OJ.train)
Variables actually used in tree construction:
[1] "LoyalCH"       "PriceDiff"     "ListPriceDiff" "PctDiscMM"
Number of terminal nodes:  8
Residual mean deviance:  0.7659 = 606.6 / 792
Misclassification error rate: 0.1675 = 134 / 800
```

(c) Type in the name of the tree object in order to get a detailed text output. Pick one of the terminal nodes, and interpret the information displayed.

```
tree.oj
```

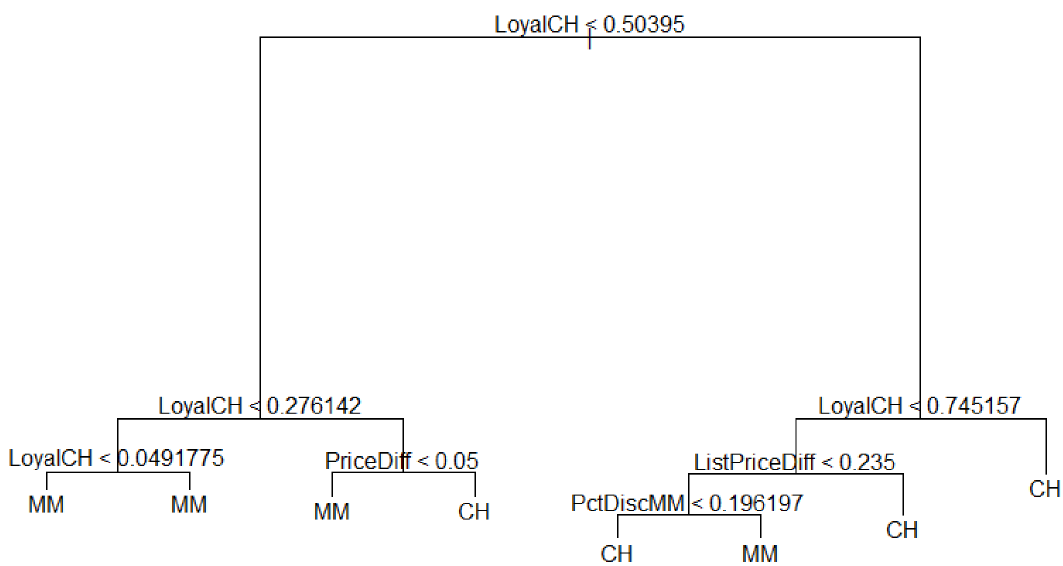
```
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

 1) root 800 1077.00 CH ( 0.60000 0.40000 )
   2) LoyalCH < 0.50395 360  425.40 MM ( 0.27778 0.72222 )
     4) LoyalCH < 0.276142 176  132.60 MM ( 0.12500 0.87500 )
       8) LoyalCH < 0.0491775 63   10.27 MM ( 0.01587 0.98413 ) *
       9) LoyalCH > 0.0491775 113  108.50 MM ( 0.18584 0.81416 ) *
     5) LoyalCH > 0.276142 184  250.80 MM ( 0.42391 0.57609 )
      10) PriceDiff < 0.05 71   75.77 MM ( 0.22535 0.77465 ) *
      11) PriceDiff > 0.05 113  155.60 CH ( 0.54867 0.45133 ) *
   3) LoyalCH > 0.50395 440  350.50 CH ( 0.86364 0.13636 )
     6) LoyalCH < 0.745157 182  210.00 CH ( 0.73626 0.26374 )
```

```
   12) ListPriceDiff < 0.235 70   97.04 CH ( 0.50000 0.50000 )
     24) PctDiscMM < 0.196197 51   66.22 CH ( 0.64706 0.35294 ) *
     25) PctDiscMM > 0.196197 19   12.79 MM ( 0.10526 0.89474 ) *
   13) ListPriceDiff > 0.235 112   80.42 CH ( 0.88393 0.11607 ) *
 7) LoyalCH > 0.745157 258   97.07 CH ( 0.95349 0.04651 ) *
```

- Pick the node labelled 4, which is a terminal node because of the asterisk. The split criterion is LoyalCH < 0.035, the number of observations in that branch is 63 with a deviance of 10.27 and an overall prediction for the branch of MM. Less than 2% of the observations in that branch take the value of CH, and the remaining 98% take the value of MM.

(d) Create a plot of the tree, and interpret the results.

```
plot(tree.oj)
text(tree.oj, pretty = 0)
```



- The most important indicator of “Purchase” appears to be “LoyalCH”, since the first branch differentiates the intensity of customer brand loyalty to CH. In fact, the top three nodes contain “LoyalCH”.

(e) Predict the response on the test data, and produce a confusion matrix comparing the test labels to the predicted test labels. What is the test error rate?

```
tree.pred <- predict(tree.oj, OJ.test, type = "class")
table(tree.pred, OJ.test$Purchase)
```

```
tree.pred  CH  MM
       CH 147  49
       MM  12  62
```

```
> 1 - (147 + 62) / 270
[1] 0.2259259
```

- The test error rate is about 22.6%.

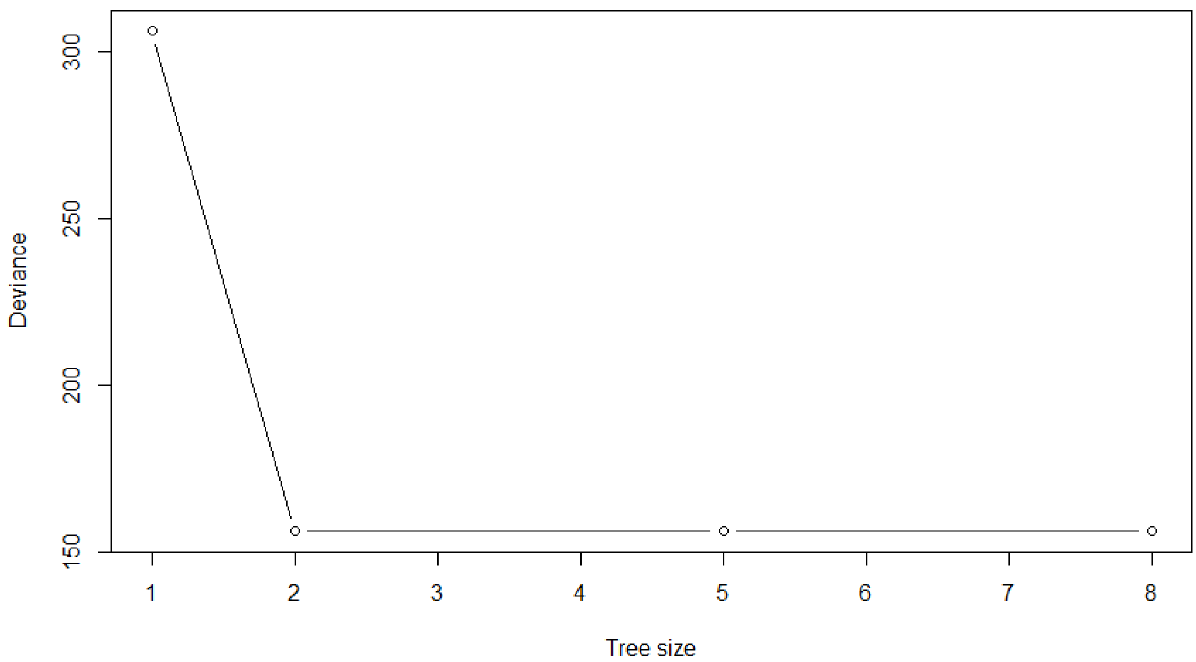
(f) Apply the cv.tree() function to the training set in order to determine the optimal tree size.

```
cv.oj <- cv.tree(tree.oj, FUN = prune.misclass)
cv.oj
```

```
$size
[1] 8 5 2 1

$dev
[1] 156 156 156 306

$k
[1]       -Inf   0.000000   4.666667 160.000000

$method
[1] "misclass"

attr(,"class")
[1] "prune"         "tree.sequence"
```

(g) Produce a plot with tree size on the x-axis and cross-validated classification error rate on the y-axis.

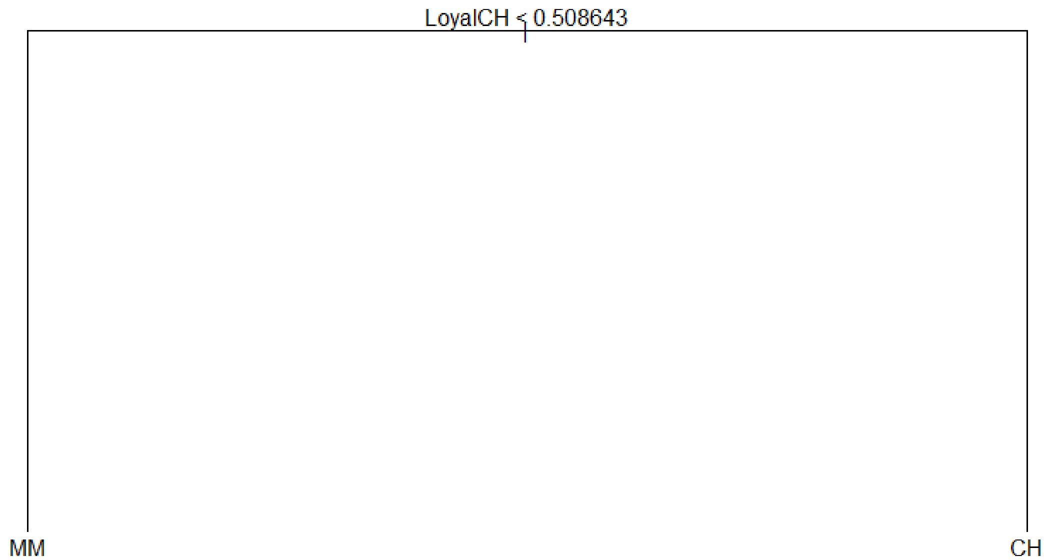
```
plot(cv.oj$size, cv.oj$dev, type = "b", xlab = "Tree size", ylab =
"Deviance")
```



- The 2-node tree is the smallest tree with the lowest classification error rate.

(h) Which tree size corresponds to the lowest cross-validated classification error rate?

```
prune.oj <- prune.misclass(tree.oj, best =7)
plot(prune.oj)
text(prune.oj, pretty = 0)
```



(i) Produce a pruned tree corresponding to the optimal tree size obtained using cross-validation. If cross-validation does not lead to selection of a pruned tree, then create a pruned tree with five terminal nodes.

```
summary(tree.oj)
summary(prune.oj)
```

```
> summary(tree.oj)
Classification tree:
tree(formula = Purchase ~ ., data = OJ.train)
Variables actually used in tree construction:
[1] "LoyalCH"       "PriceDiff"     "SpecialCH"     "ListPriceDiff"
Number of terminal nodes:  8
Residual mean deviance:  0.7305 = 578.6 / 792
Misclassification error rate: 0.165 = 132 / 800

> summary(prune.oj)
Classification tree:
snip.tree(tree = tree.oj, nodes = c(3L, 2L))
Variables actually used in tree construction:
[1] "LoyalCH"
Number of terminal nodes:  2
Residual mean deviance:  0.9115 = 727.4 / 798
Misclassification error rate: 0.1825 = 146 / 800
```

- The misclassification error rate is the same for the pruned tree (0.1825 vs 0.1675).

(j) Compare the training error rates between the pruned and unpruned trees. Which is higher?

```
prune.pred <- predict(prune.oj, OJ.test, type = "class")
table(prune.pred, OJ.test$Purchase)
```

```
prune.pred  CH  MM
        CH 119  30
        MM  40  81
> 1 - (119 + 81) / 270
[1] 0.2592593
```

- The test error rate is about 25.9%.

(k) Compare the test error rates between the pruned and unpruned trees. Which is higher?

- The pruning process increased the test error rate from 22.6% to 25.9%, but it produced a way more interpretable tree.

Week9_ Khan

Homework: ISLR has a dataset Khan. It contains gene expression data for 4 types of small round blue cell tumors. Use help(Khan) to see details. Apply random forest and boosting to the training set, and tuning the hyperparameters to improve the models. Report your main steps and final results.

Random Forest

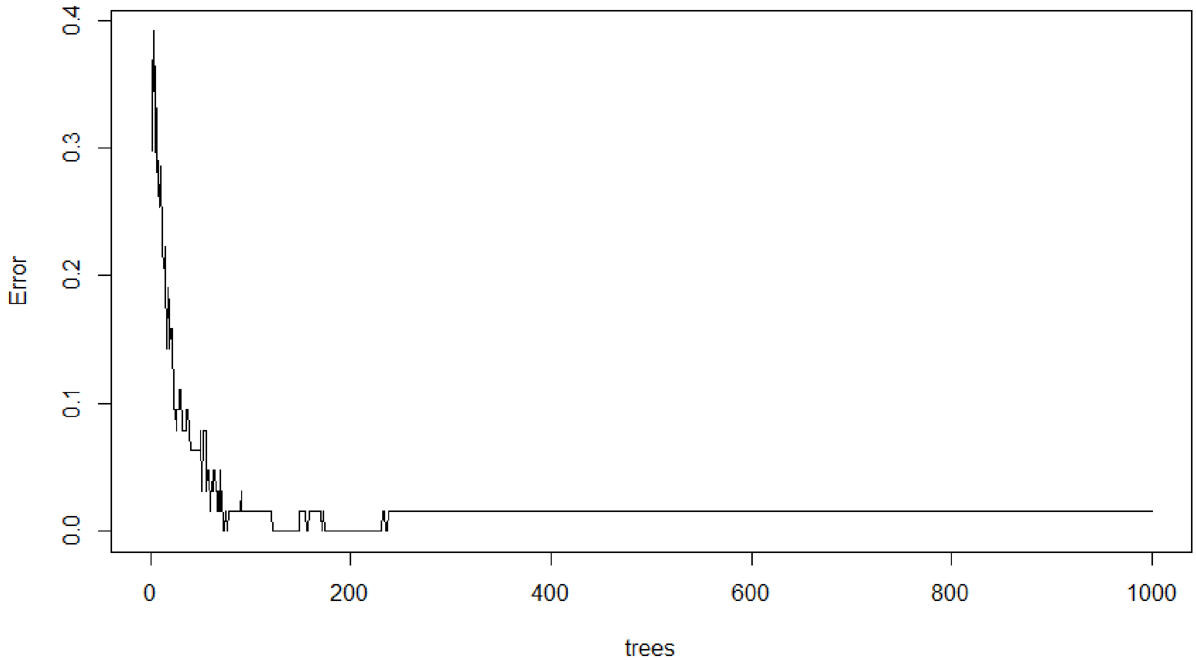
```
library(ISLR)
?Khan
#fix(Khan)
summary(Khan)
str(Khan) ## variable names and dataset dimensions
dd = data.frame(Khan$xtrain)
tt = data.frame(Khan$xtest)
str(dd)
summary(dd)
dd$ytrain = as.factor(Khan$ytrain) ## randomForest() requires categorical
outcome to be a factor
tt$ytest = as.factor(Khan$ytest)

table(Khan$ytrain)
table(Khan$ytest)

apply(is.na(dd), 2, sum) ## check which variable has missing data
apply(is.na(dd), 1, sum) ## check which observation has missing data

# Random Forests and Bagging
library(randomForest)
```

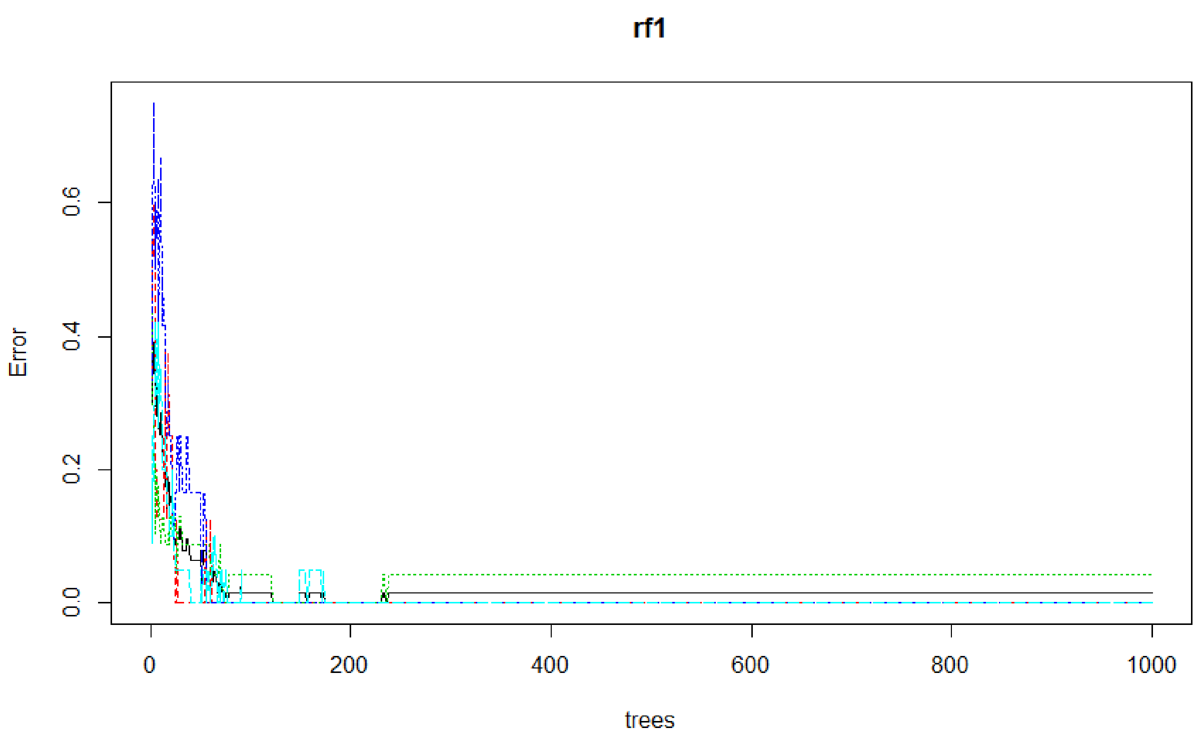
```
set.seed(1)
rf1 = randomForest(ytrain ~ ., data=dd, ntree=1000)
plot(rf1)
plot(rf1$err.rate[,1], type='l', xlab='trees', ylab='Error')
rf1
names(rf1) ## all details are here
rf1$mtry; rf1$ntree ## check what default values were used
rf1$confusion ## same as table(Heart2.train$AHD, rf$predicted)
rf1$err.rate[rf$ntree, ] ## the FPR and FNR can also be obtained here

importance(rf1) ## show rf$importance
varImpPlot(rf1) ## same as dotchart(rf$importance[, 'MeanDecreaseGini'])
except order

varImpPlot(randomForest(ytrain ~ ., data=dd)) ## repeat a few times

set.seed(1)
varImpPlot(randomForest(ytrain ~ ., data=dd, mtry=1))
varImpPlot(randomForest(ytrain ~ ., data=dd, mtry=30))
varImpPlot(randomForest(ytrain ~ ., data=dd, mtry=100))

rf2 = randomForest(ytrain ~ ., data=dd, importance=T)
rf2$importance
importance(rf2) ## different from rf$importance except the last column
varImpPlot(rf2)
rf2
plot(rf2)

#Cross-validation to select m
library(caret)
cvCtrl = trainControl(method="repeatedcv", number=5, repeats=4, ## 5-fold
CV repeated 4 times
                      #summaryFunction=twoClassSummary,
                      classProbs=TRUE)
set.seed(1)
fitRFcaret = train(x=dd[, 1:2308], y=Khan$ytrain, trControl=cvCtrl,
                   tuneGrid=data.frame(mtry=100:150),
                   #tuneLength=4,
                   #metric="ROC", ## when summaryFunction=twoClassSummary
                   method="rf", ntree=1000) ##
fitRFcaret
plot(fitRFcaret)

names(fitRFcaret)
fitRFcaret$results
fitRFcaret$bestTune$mtry
fitRFcaret$finalModel
fitRFcaret$finalModel$confusion ## OOB confusion matrix


set.seed(1)
rf3 = randomForest(ytrain ~ ., data=dd, mtry=142, ntree=1000)
rf3
plot(rf3)
```

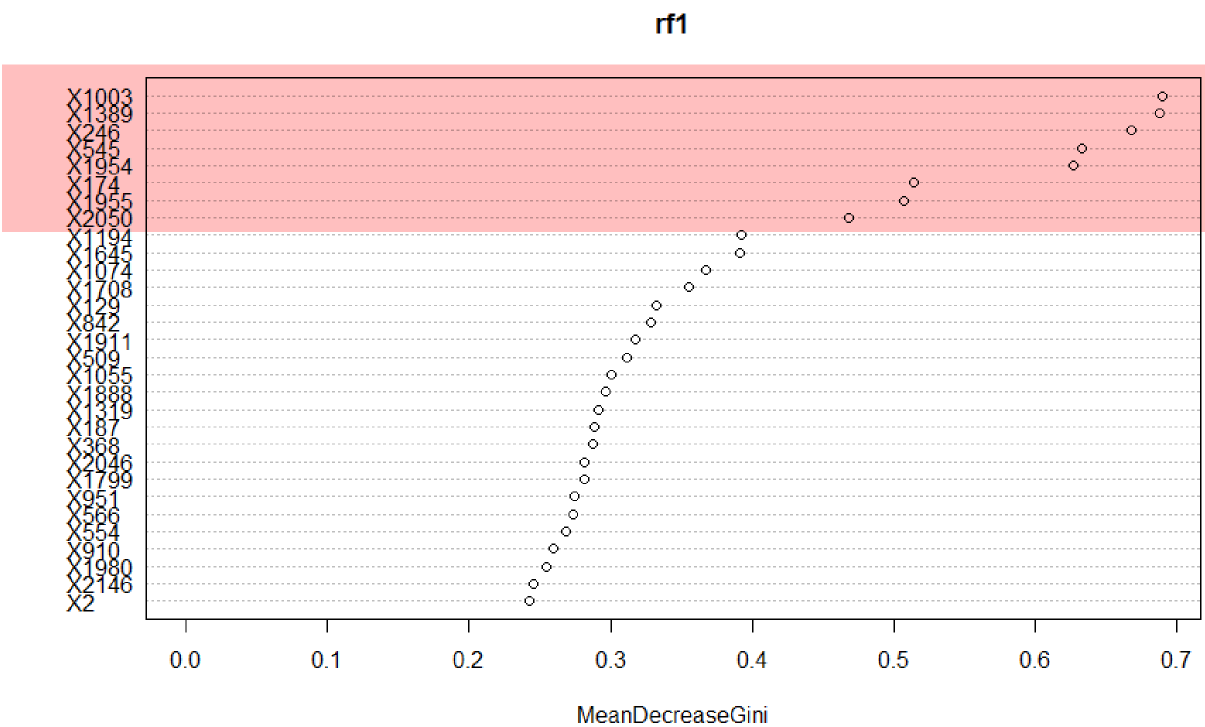
```
Call:
 randomForest(formula = ytrain ~ ., data = dd, ntree = 1000)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 48
```

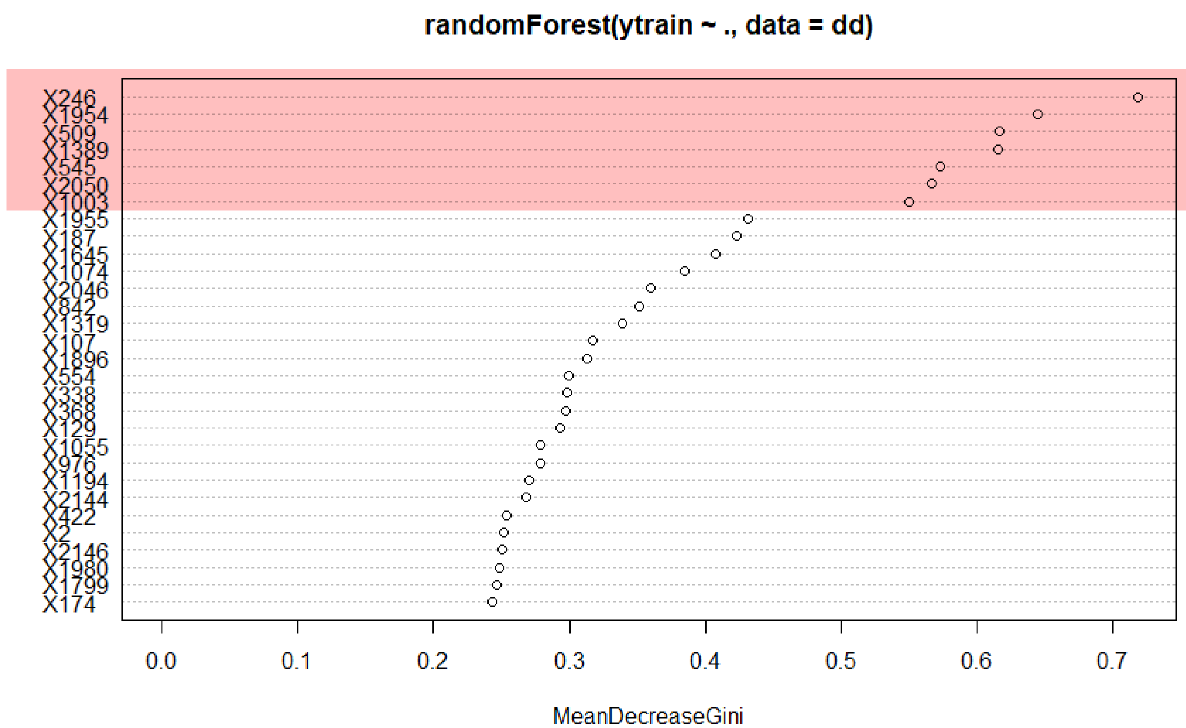
```
        OOB estimate of  error rate: 1.59%
Confusion matrix:
  1  2  3  4 class.error
1 8  0  0  0  0.00000000
2 0 22  0  1  0.04347826
3 0  0 12  0  0.00000000
4 0  0  0 20  0.00000000
```

Step1. Try default RandomForest Model.
Step2. Adjust hyperparameters mtry to see if the model could be improved or not.
Step3. Use cross-validation to choose the best model.

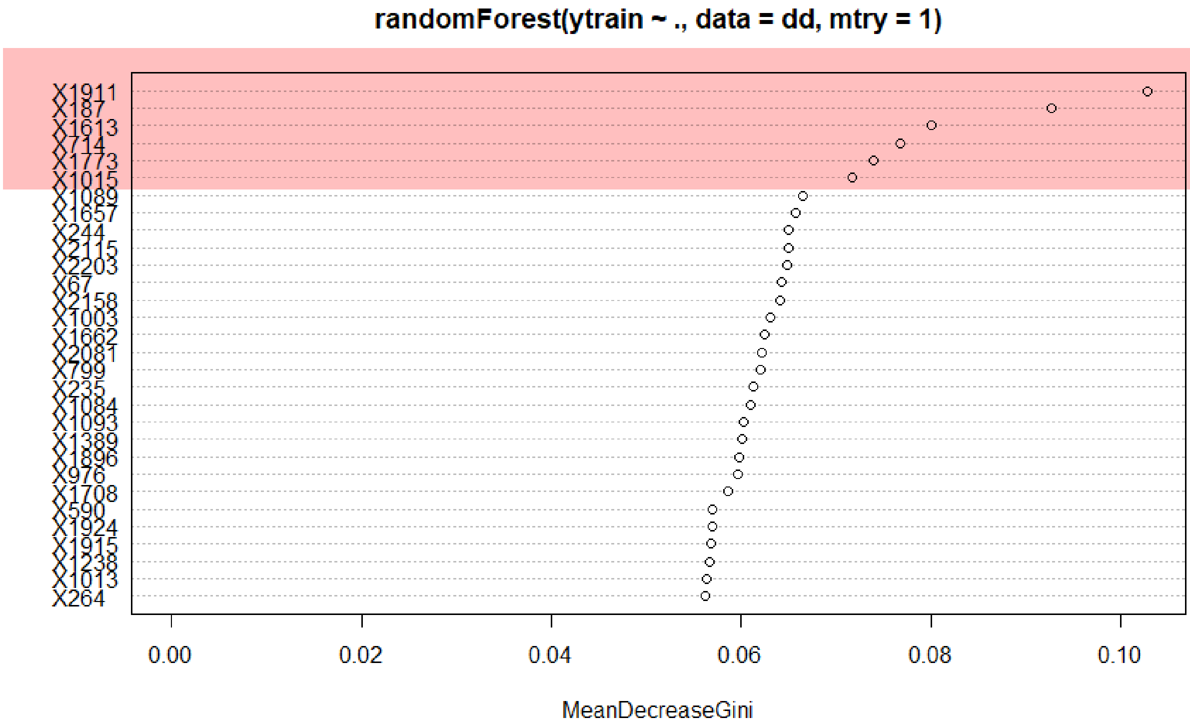

Error
0.0
0.1
0.2
0.3
0.4
trees
0
200
400
600
800
1000

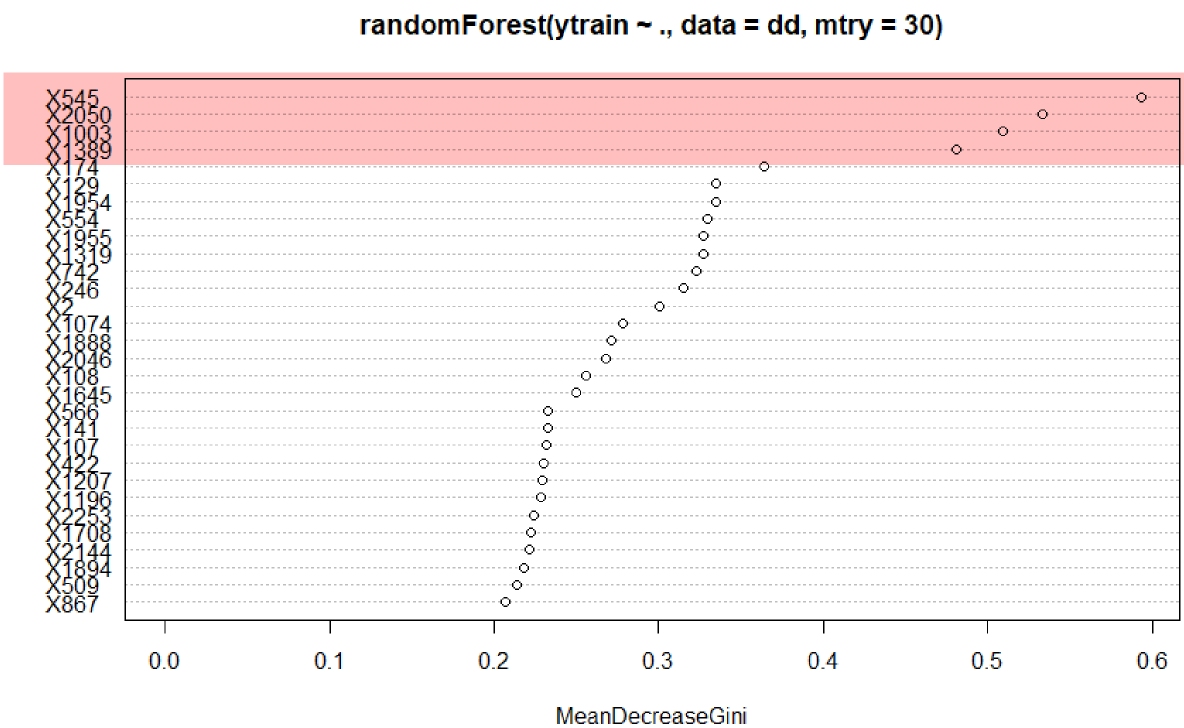

rf1
Error
0.0
0.2
0.4
0.6
trees
0
200
400
600
800
1000

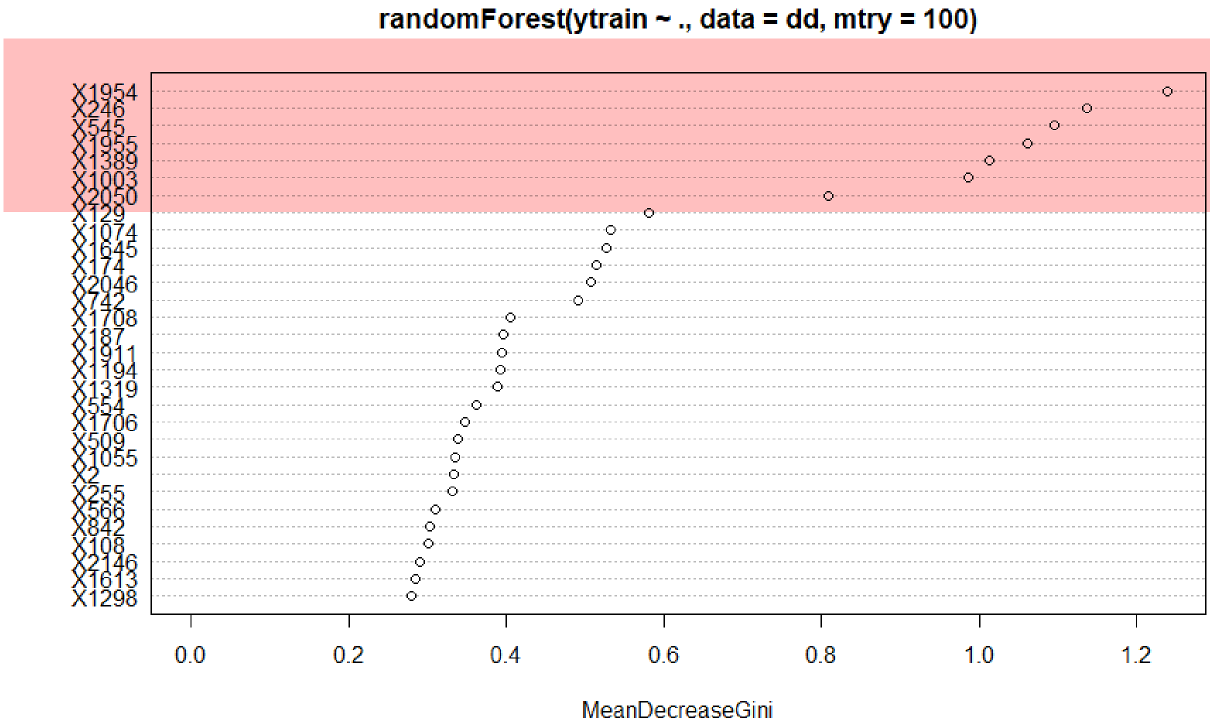




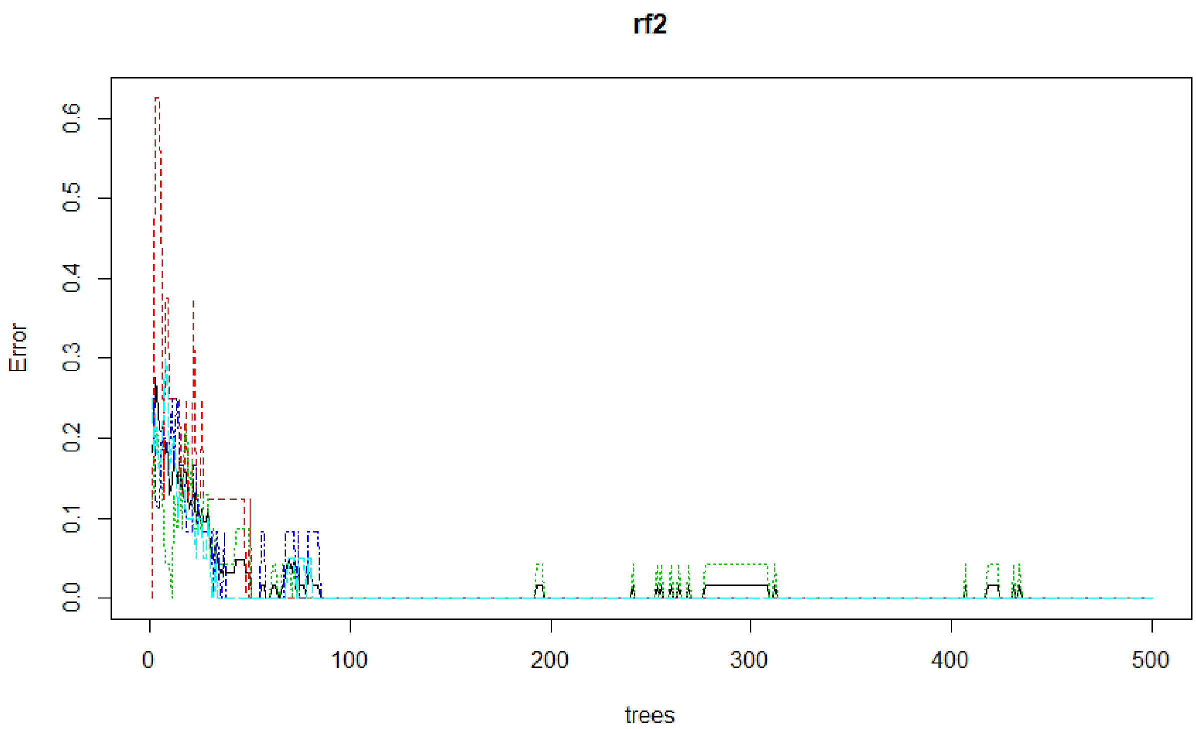

```
r
f1$mtry; rf1$ntree ## check what default values were used
[1] 48
[1] 1000
```

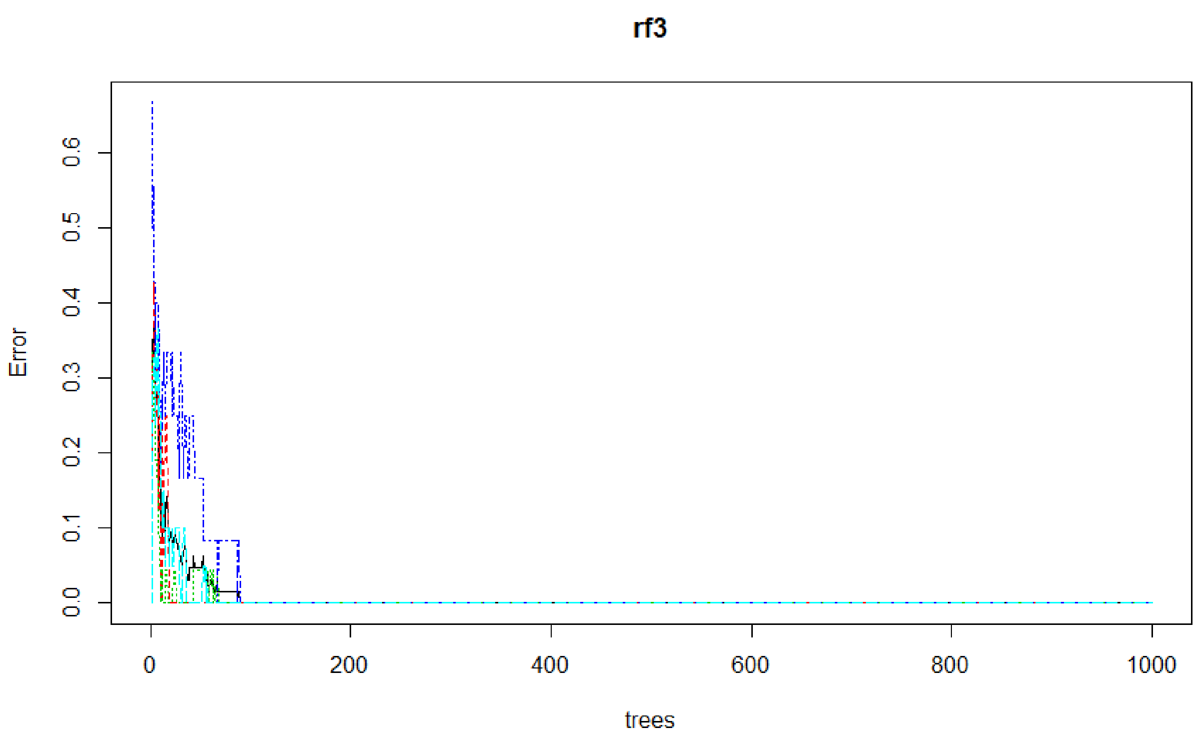

randomForest(ytrain ~ ., data = dd, mtry = 1)
X1911
X187
X1613
X714
X1773
X1015
X1089
X1657
X244
X2115
X2203
X67
X2158
X1003
X1662
X2081
X799
X235
X1084
X1093
X1389
X1896
X976
X1708
X590
X1924
X1915
X1238
X1013
X264
0.00
0.02
0.04
0.06
0.08
0.10
MeanDecreaseGini


randomForest(ytrain ~ ., data = dd, mtry = 30)
X545
X2050
X1003
X1389
X174
X129
X1954
X554
X1955
X1319
X742
X246
X2
X1074
X1888
X2046
X108
X1645
X566
X141
X107
X422
X1207
X1196
X2253
X1708
X2144
X1894
X509
X867
0.0
0.1
0.2
0.3
0.4
0.5
0.6
MeanDecreaseGini




```
> rf3

Call:
 randomForest(formula = ytrain ~ ., data = dd, mtry = 100, ntree = 1000)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 100

        OOB estimate of  error rate: 0%
Confusion matrix:
  1  2  3  4 class.error
1 8  0  0  0           0
2 0 23  0  0           0
3 0  0 12  0           0
4 0  0  0 20           0
```

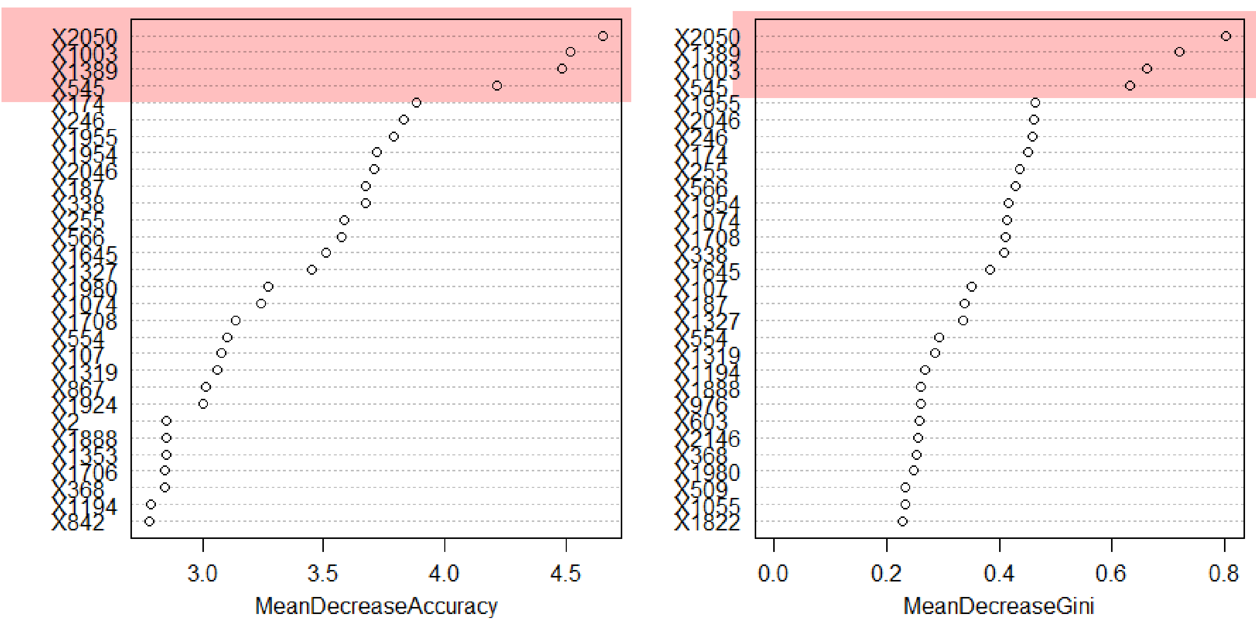




Model is stable after n.trees >200.

```
> rf2

Call:
 randomForest(formula = ytrain ~ ., data = dd, importance = T)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 48

        OOB estimate of  error rate: 0%
Confusion matrix:
  1  2  3  4 class.error
1 8  0  0  0           0
2 0 23  0  0           0
3 0  0 12  0           0
4 0  0  0 20           0
```

rf2



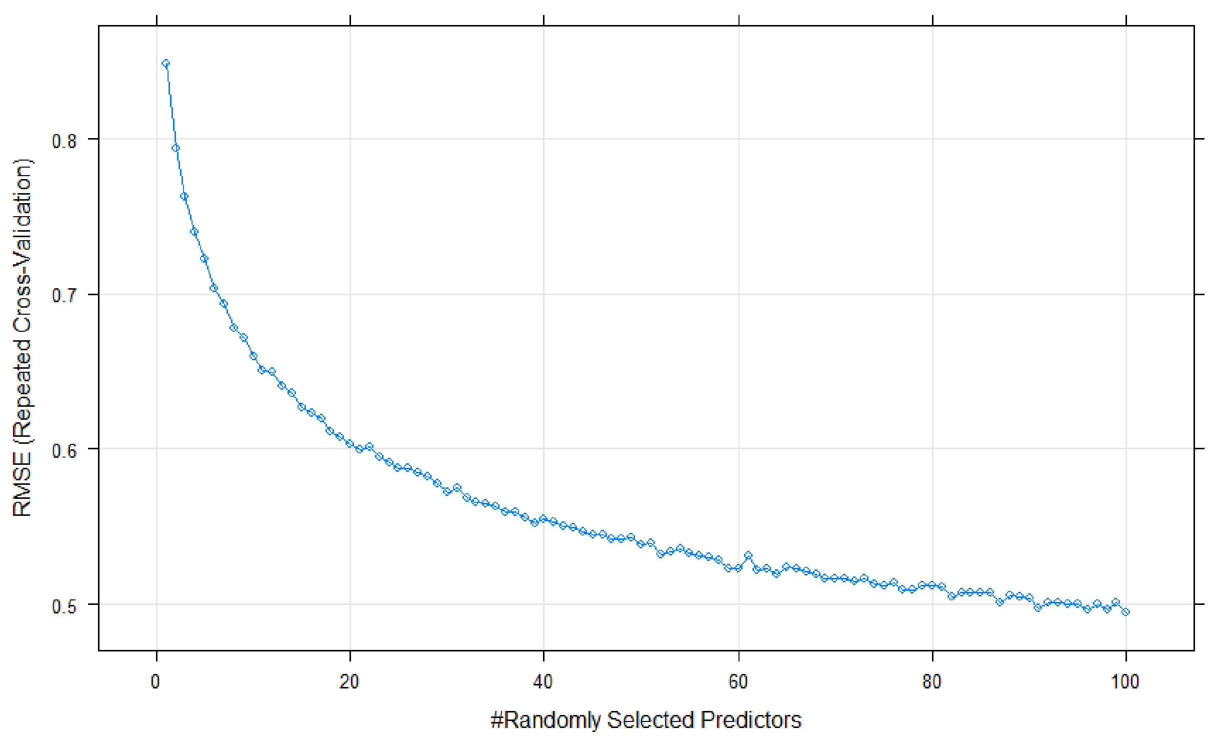
#Cross-validation to select m

```
Random Forest

  63 samples
2308 predictors

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 4 times)
Summary of sample sizes: 51, 49, 51, 50, 51, 51, ...
Resampling results across tuning parameters:

  mtry  RMSE       Rsquared   MAE
```

```
    1   0.8482015  0.7403767  0.7659427
    2   0.7940902  0.7938442  0.7153580
    3   0.7630002  0.7980486  0.6878453
…………
   95   0.5001371  0.9139016  0.4326676
   96   0.4963442  0.9167989  0.4292185
   97   0.4996833  0.9130528  0.4320739
   98   0.4963625  0.9128346  0.4287431
   99   0.5007732  0.9119197  0.4315655
  100   0.4948107  0.9169527  0.4274850

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was mtry = 100.
x731       0.046358214
x732       0.001777778
x733       0.059856087
x734       0.000000000
x735       0.003589744
x736       0.000000000
………..
x988       0.039416624
x989       0.005564103
x990       0.010791234
x991       0.009936364
x992       0.003750000
x993       0.002909091
x994       0.007100000
x995       0.019689542
x996       0.001857143
x997       0.002400000
x998       0.066717771
x999       0.005866667
x1000      0.000000000
 [ reached getOption("max.print") -- omitted 1308 rows ]
> varImpPlot(rf1) ## same as dotchart(rf$importance[, 'MeanDecreaseGini']) excep
t order
> varImpPlot(randomForest(ytrain ~ ., data=dd)) ## repeat a few times
> importance.multirun = matrix(,20,13)
> for(i in 1:20)
+   importance.multirun[i,] = randomForest(ytrain ~ ., data=dd, ntree=500)$impor
tance
> set.seed(1)
> varImpPlot(randomForest(ytrain ~ ., data=dd, mtry=1))
> varImpPlot(randomForest(ytrain ~ ., data=dd, mtry=30))
> varImpPlot(randomForest(ytrain ~ ., data=dd, mtry=100))
> plot(rf1)
> rf1$mtry; rf1$ntree ## check what default values were used
[1] 48
[1] 1000
> rf2 = randomForest(ytrain ~ ., data=dd, importance=T)
> rf2$importance
                 1             2             3             4 MeanDecreaseAccuracy MeanDecreaseGini
x1    2.500000e-03  2.222222e-05 -5.000000e-04  0.000000e+00         1.844637e-04      0.047726677
x2    1.000000e-03  1.712454e-03  4.152381e-03  7.388095e-03         4.024664e-03      0.212860868
x3    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00      0.003666667
x4    0.000000e+00 -2.857143e-04  0.000000e+00 -2.222222e-04        -1.569231e-04      0.011317345
x5    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00      0.000000000
```

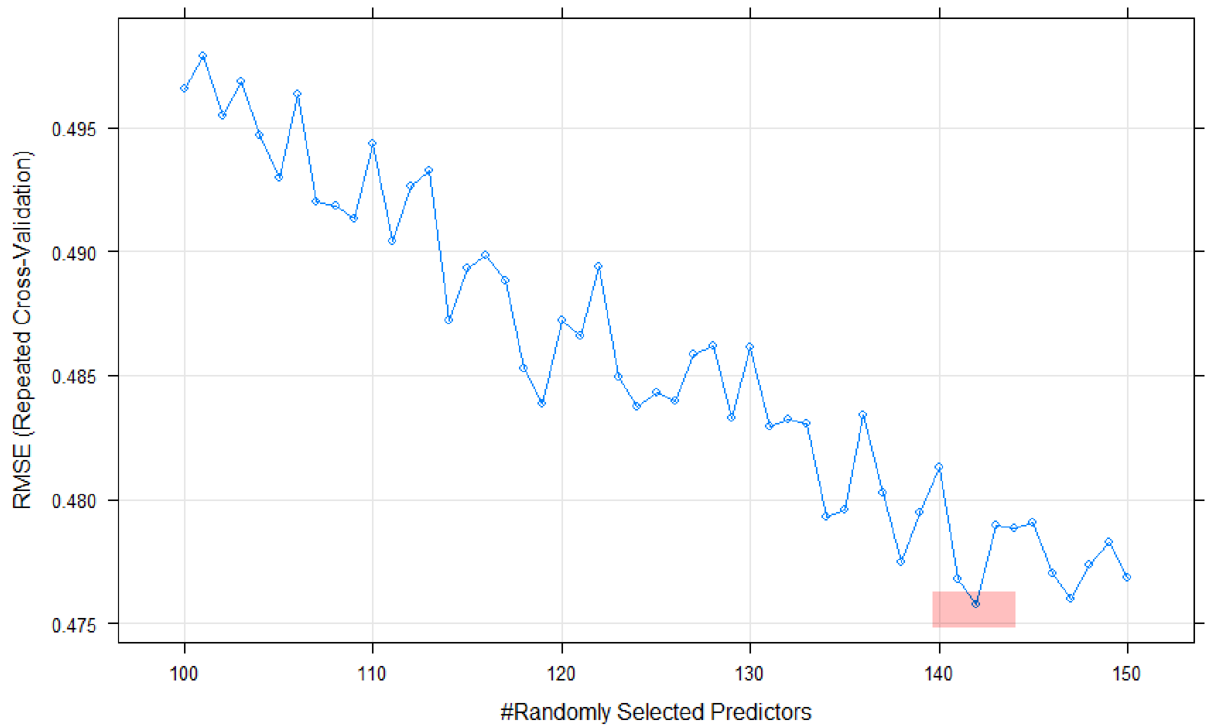
```
x6      0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00       0.000000000
x7      0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00       0.000000000
x8      0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00       0.000000000
x9      0.000000e+00  2.222222e-04  0.000000e+00  0.000000e+00         9.090909e-05       0.014233670
x10     0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00       0.000000000
x11     0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00       0.000000000
x12     0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00       0.000000000
x13     0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00       0.002000000
x14     0.000000e+00  2.000000e-04  5.000000e-04  0.000000e+00         1.904762e-04       0.019171717
x15    -5.000000e-04  1.043651e-03  1.333333e-04  0.000000e+00         3.403614e-04       0.054235675
x16     0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00         0.000000e+00       0.011076923
x17     1.000000e-03  4.000000e-04  1.000000e-03 -2.857143e-04         2.857143e-04       0.017507692
 [ reached getOption("max.print") -- omitted 2142 rows ]
> importance(rf2) ## different from rf$importance except the last column
 [ reached getOption("max.print") -- omitted 2142 rows ]
> varImpPlot(rf2)
> rf2

Call:
 randomForest(formula = ytrain ~ ., data = dd, importance = T)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 48

        OOB estimate of  error rate: 0%
Confusion matrix:
  1  2  3  4 class.error
1 8  0  0  0           0
2 0 23  0  0           0
3 0  0 12  0           0
4 0  0  0 20           0
> set.seed(1)
> rf3 = randomForest(ytrain ~ ., data=dd, mtry=100, ntree=1000)
> rf3

Call:
 randomForest(formula = ytrain ~ ., data = dd, mtry = 100, ntree = 1000)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 100

        OOB estimate of  error rate: 0%
Confusion matrix:
  1  2  3  4 class.error
1 8  0  0  0           0
2 0 23  0  0           0
3 0  0 12  0           0
4 0  0  0 20           0

> set.seed(1)
> fitRFcaret = train(x=dd[, 1:2308], y=Khan$ytrain, trControl=cvCtrl,
+                    tuneGrid=data.frame(mtry=1:100),
+                    #tuneLength=4,
+                    #metric="ROC", ## when summaryFunction=twoClassSummary
+                    method="rf", ntree=1000) ##
> fitRFcaret
```

```
> fitRFcaret
Random Forest

  63 samples
2308 predictors

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 4 times)
Summary of sample sizes: 51, 49, 51, 50, 51, 51, ...
Resampling results across tuning parameters:

  mtry  RMSE       Rsquared   MAE
    1   0.8482015  0.7403767  0.7659427
    2   0.7940902  0.7938442  0.7153580
    3   0.7630002  0.7980486  0.6878453
    4   0.7395510  0.8231199  0.6638888
    5   0.7223881  0.8253714  0.6488759
    6   0.7034210  0.8453957  0.6307501
    7   0.6930562  0.8552637  0.6209658
……………
   96   0.4963442  0.9167989  0.4292185
   97   0.4996833  0.9130528  0.4320739
   98   0.4963625  0.9128346  0.4287431
   99   0.5007732  0.9119197  0.4315655
  100   0.4948107  0.9169527  0.4274850

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was mtry = 100.
> plot(fitRFcaret)
```

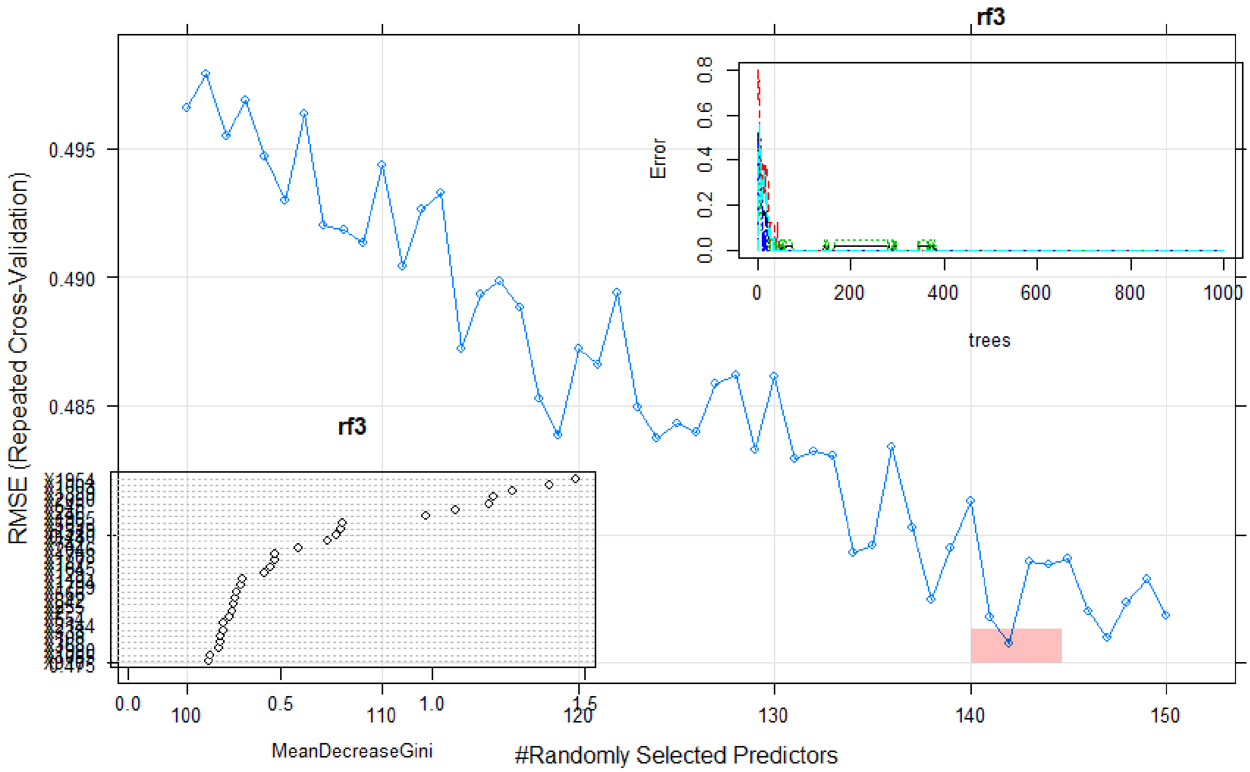



```
> #Cross-validation to select m
> library(caret)
```

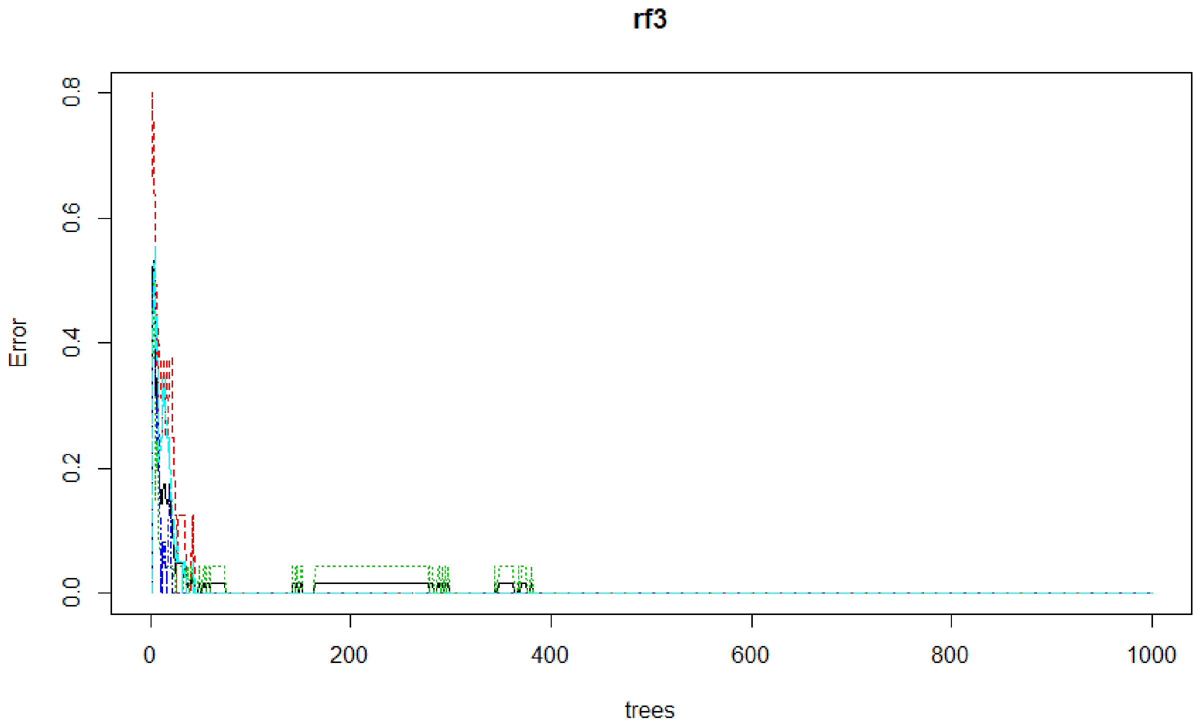
```
> cvCtrl = trainControl(method="repeatedcv", number=5, repeats=4, ## 5-fold CV r
epeated 4 times
+                        #summaryFunction=twoClassSummary,
+                        classProbs=TRUE)
> set.seed(1)
> fitRFcaret = train(x=dd[, 1:2308], y=Khan$ytrain, trControl=cvCtrl,
+                     tuneGrid=data.frame(mtry=100:150),
+                     #tuneLength=4,
+                     #metric="ROC", ## when summaryFunction=twoClassSummary
+                     method="rf", ntree=1000) ##
> fitRFcaret
Random Forest

  63 samples
2308 predictors

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 4 times)
Summary of sample sizes: 51, 49, 51, 50, 51, 51, ...
Resampling results across tuning parameters:

  mtry  RMSE       Rsquared   MAE
  100   0.4965517  0.9151116  0.4274355
  101   0.4978712  0.9149261  0.4293641
  102   0.4954864  0.9154573  0.4281065
  103   0.4968707  0.9154882  0.4284154
  104   0.4946781  0.9134506  0.4264180
  ……………..
  138   0.4774528  0.9147392  0.4083156
  139   0.4794838  0.9146149  0.4107703
  140   0.4813043  0.9124724  0.4110739
  141   0.4767976  0.9192609  0.4087820
  142   0.4757777  0.9140706  0.4055893
  143   0.4789702  0.9145817  0.4097683
  144   0.4788150  0.9111041  0.4094942
  145   0.4790876  0.9136148  0.4094785
  146   0.4770247  0.9168655  0.4086519
  147   0.4759817  0.9152729  0.4069054
  148   0.4773655  0.9139032  0.4076564
  149   0.4782550  0.9130153  0.4078687
  150   0.4768591  0.9157534  0.4079938

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was mtry = 142.
```




```
> fitRFcaret$bestTune$mtry
[1] 142
> fitRFcaret$finalModel

Call:
 randomForest(x = x, y = y, ntree = 1000, mtry = param$mtry)
               Type of random forest: regression
                     Number of trees: 1000
No. of variables tried at each split: 142

          Mean of squared residuals: 0.1986616
                    % Var explained: 81.93
```




```
> rf3

Call:
 randomForest(formula = ytrain ~ ., data = dd, mtry = 142, ntree = 1000)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 142

        OOB estimate of  error rate: 0%
Confusion matrix:
  1  2  3  4 class.error
1 8  0  0  0           0
2 0 23  0  0           0
3 0  0 12  0           0
4 0  0  0 20           0
```



Boosting Tree

```
#Boosting
library(survival)
library(lattice)
library(splines)
library(parallel)
library(gbm)
bt1 = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=500)
bt1
names(bt1)
bt1$interaction.depth ## stumps
bt1$cv.folds ## no CV was done

bt4 = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=5000
interaction.depth=4)
mse = function(a,b) mean((a-b)^2)
mse(Khan$ytest, predict(bt1, tt, n.trees=500)) ## MSE=
mse(Khan$ytest, predict(bt4, tt, n.trees=5000)) ## MSE=

summary(bt1) ## results and a plot
summary(bt4) ## with d=4, the influence of lstat is smaller
summary(bt1, plotit=F) ## without the plot
sum(summary(bt1, plotit=F)$rel.inf) ## 100
sum(summary(bt4, plotit=F)$rel.inf) ## 100

bt.try = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=100,
bag.fraction=1)
summary(bt.try, plotit=F)$rel.inf
```

```
par(mfrow=c(2,2))
plot(bt4, i="X1194", main='bt4')
plot(bt4, i="X1003", main='bt4')
plot(bt1, i="X1194", main='bt1')
plot(bt1, i="X1003", main='bt1')

#a 2-dimensional partial dependence plot.
plot(bt4, i=c("X1194", "X1003"))

### look at model performance at the end of 1000, 2000, etc. trees.
set.seed(1)
bt4b = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=2000,
interaction.depth=4,shrinkage =0.01)
mse(Khan$ytest, predict(bt4b, tt, n.trees=2000))
bt4c = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=5000,
interaction.depth=4,shrinkage =0.01)
mse(Khan$ytest, predict(bt4c, tt, n.trees=5000))

#Cross-validation using caret
library(caret)
set.seed(1)
ctr = trainControl(method="cv", number=3) ## 3-fold CV
mygrid = expand.grid(n.trees=seq(50, 1000, 50), interaction.depth=1:8,
                     shrinkage=0.01, n.minobsinnode=5)
boost.caretk <- train(ytrain ~ ., dd, trControl=ctr, method='gbm',
                     tuneGrid=mygrid,
                     preProc=c('center','scale'), verbose=F)
boost.caretk
plot(boost.caretk)

#Using the optimal hyperparameters selected by train() improves the result!
boost.caretk$bestTune
mse(Khan$ytest, predict(boost.caretk, tt)) ## MSE=10.8
names(boost.caretk)
boost.caretk$results
boost.caretk$finalModel
```

```
> bt1
gbm(formula = ytrain ~ ., distribution = "gaussian", data = dd,
    n.trees = 500)
A gradient boosted model with gaussian loss function.
500 iterations were performed.
There were 2308 predictors of which 62 had non-zero influence.
> bt1$interaction.depth ## stumps
[1] 1
> bt1$cv.folds ## no CV was done
[1] 0
```

adjust n.trees to see if the model could be improve or not.

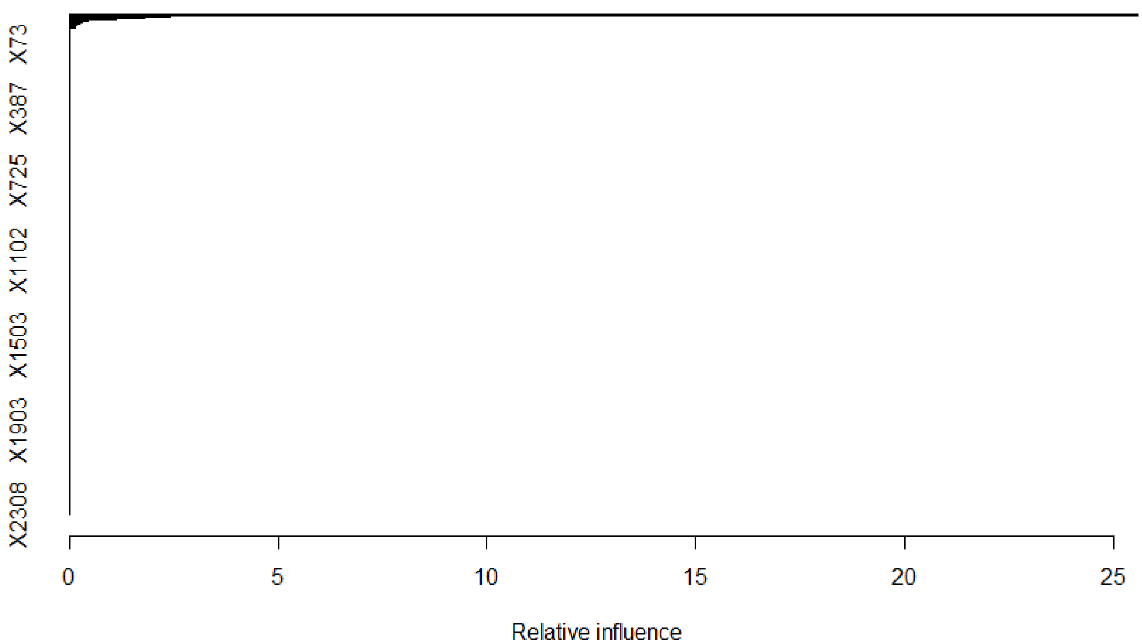
```
> bt4
gbm(formula = ytrain ~ ., distribution = "gaussian", data = dd,
    n.trees = 5000, interaction.depth = 4)
A gradient boosted model with gaussian loss function.
5000 iterations were performed.
There were 2308 predictors of which 1271 had non-zero influence.
> mse = function(a,b) mean((a-b)^2)
```

```
> mse(Khan$ytest, predict(bt1, tt, n.trees=5000)) ## MSE=
[1] 0.6805692

> mse(Khan$ytest, predict(bt4, tt, n.trees=5000)) ## MSE=
[1] 0.123483
```

n.trees = 5000, show better model

```
> summary(bt1)
          var    rel.inf
x1194 x1194 25.5849726
x1003 x1003  9.3920327
x1706 x1706  8.2145581
x1888 x1888  6.3004250
x187   x187  5.2335208
x2046 x2046  4.9996605
x509   x509  3.3126329
x1207 x1207  2.4008414
x188   x188  2.3390119
x129   x129  2.3348677
x2247 x2247  1.9585361
x1955 x1955  1.9389083
x867   x867  1.8938311
x2       x2  1.8151077
x2146 x2146  1.6915503
x153   x153  1.5511718
x970   x970  1.3651166
x1093 x1093  1.3296152
x1110 x1110  1.2203850
x1954 x1954  1.1922254
x1723 x1723  1.1377847
x1911 x1911  0.9731629
x1634 x1634  0.9413626
x141   x141  0.9412257
x979   x979  0.8898331
x251   x251  0.7014818
x1055 x1055  0.5844622
x1994 x1994  0.4426225
x849   x849  0.3798042
x554   x554  0.3513605
x1112 x1112  0.3494695
x1203 x1203  0.3369867
x419   x419  0.3126070
x229   x229  0.2962180
x1914 x1914  0.2772291
x1655 x1655  0.2647783
x779   x779  0.2436837
x2167 x2167  0.2426966
x603   x603  0.2421203
x2115 x2115  0.2374324
x910   x910  0.2350079
x1627 x1627  0.2232165
x1283 x1283  0.2162547
x348   x348  0.2123959
x335   x335  0.1998452
x980   x980  0.1970491
x1298 x1298  0.1931072
x2227 x2227  0.1863062
```

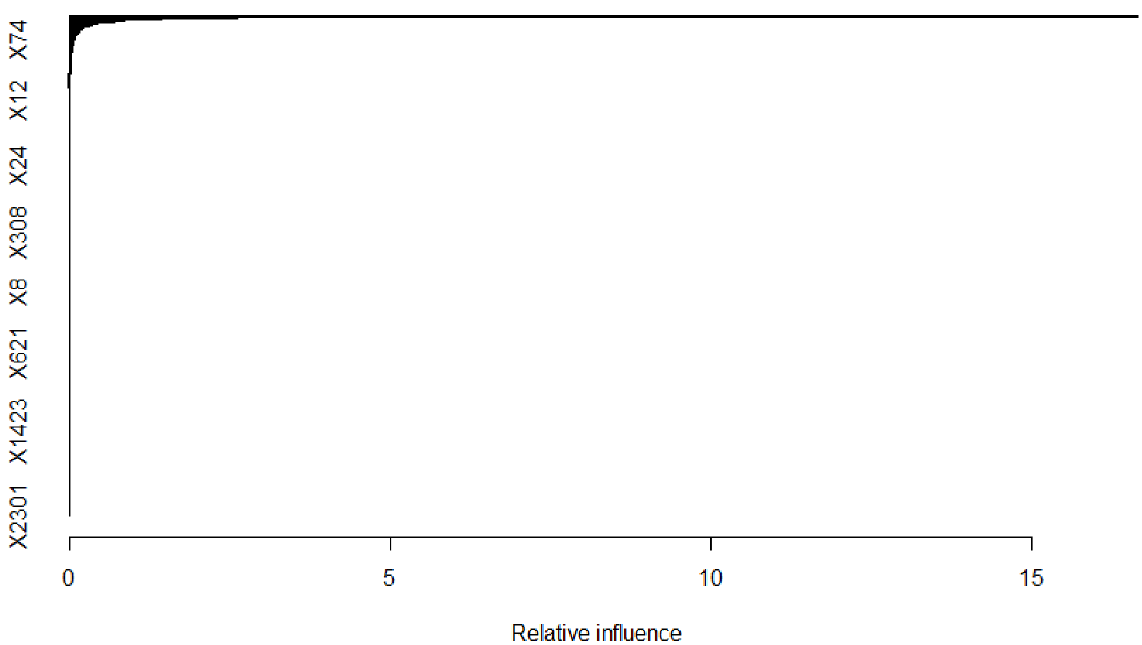
```
x2050 x2050  0.1782201
x1822 x1822  0.1773343
x1074 x1074  0.1722375
x2114 x2114  0.1705651
x965   x965  0.1653687
x67     x67  0.1577687
x941   x941  0.1548919
x1105 x1105  0.1537763
x174   x174  0.1537202
x761   x761  0.1486440
x469   x469  0.1434305
x1345 x1345  0.1363478
x1799 x1799  0.1065878
x1090 x1090  0.1026317
```




```
> summary(bt4)
        var      rel.inf
x1194 x1194 1.664393e+01
x1003 x1003 9.361631e+00
x2046 x2046 6.072878e+00
x1706 x1706 5.088604e+00
x187   x187 4.315902e+00
x1888 x1888 3.413963e+00
x509   x509 3.381628e+00
x1536 x1536 2.629198e+00
x1110 x1110 1.925364e+00
x1955 x1955 1.922930e+00
x1207 x1207 1.704898e+00
x1634 x1634 1.659425e+00
x129   x129 1.609944e+00
```

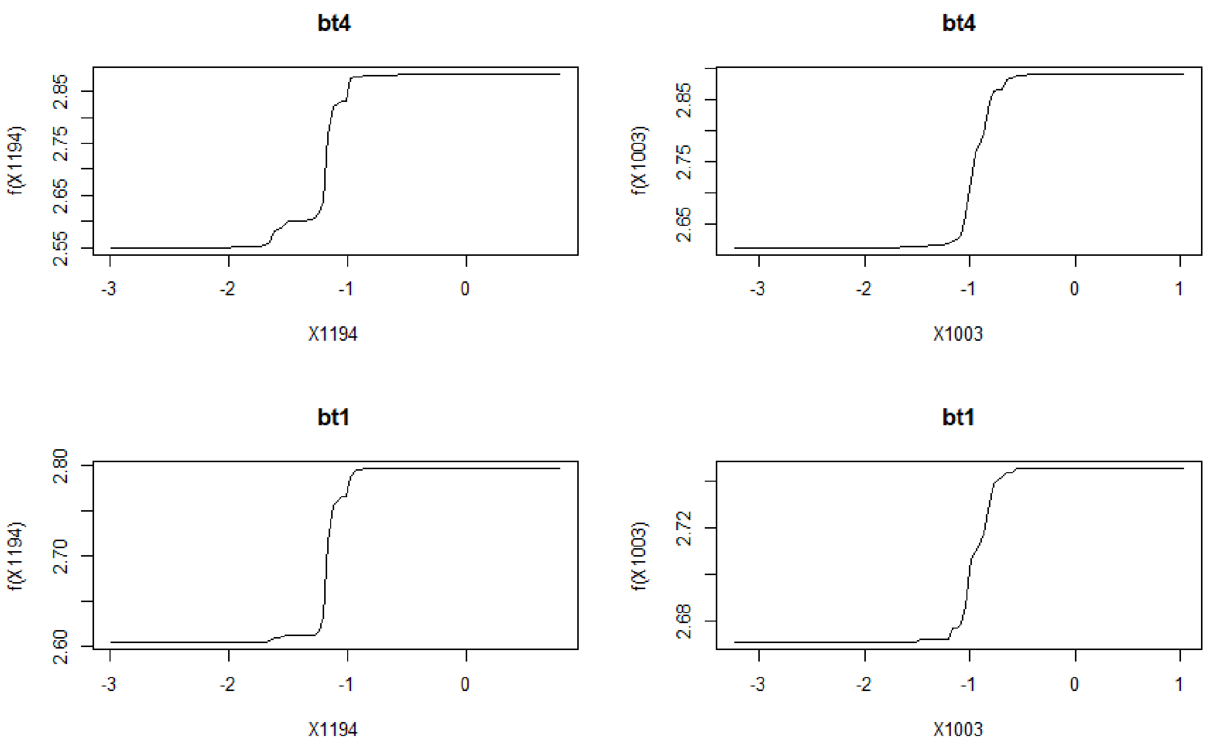
x1093	x1093	1.431456e+00
x153	x153	1.310559e+00
x188	x188	1.217891e+00
x2146	x2146	1.200711e+00
x2247	x2247	1.033930e+00
x979	x979	9.839037e-01
x1760	x1760	8.706712e-01
x469	x469	8.656647e-01
x2	x2	8.444401e-01
x761	x761	8.317947e-01
x1723	x1723	8.200500e-01
x174	x174	7.799429e-01
x335	x335	7.594749e-01
x867	x867	7.129045e-01
x1606	x1606	7.042687e-01
x941	x941	6.442353e-01
x1911	x1911	6.102285e-01
x849	x849	5.914341e-01
x151	x151	4.728010e-01
x970	x970	4.702271e-01
x141	x141	4.424153e-01
x1954	x1954	4.250515e-01
x1090	x1090	3.987292e-01
x67	x67	3.963805e-01
x2081	x2081	3.962415e-01
x2235	x2235	3.847508e-01
x1393	x1393	3.822786e-01
x1914	x1914	3.485191e-01
x1994	x1994	3.321619e-01
x85	x85	3.243962e-01
x1345	x1345	3.201392e-01
x910	x910	3.185412e-01
x554	x554	3.088031e-01
x1610	x1610	3.057559e-01
x1647	x1647	3.008315e-01
x338	x338	2.963001e-01
x1196	x1196	2.942557e-01
x1577	x1577	2.850909e-01
x1245	x1245	2.816273e-01
x780	x780	2.565527e-01
x380	x380	2.285942e-01
x1916	x1916	2.264626e-01
x2227	x2227	2.246893e-01
x251	x251	2.215651e-01
x1822	x1822	2.159333e-01
x1105	x1105	2.113934e-01
x276	x276	1.933386e-01
x714	x714	1.928035e-01
x603	x603	1.912856e-01
x715	x715	1.856958e-01
x783	x783	1.794961e-01
x1884	x1884	1.680045e-01
x1066	x1066	1.622347e-01
x1727	x1727	1.619629e-01
x1427	x1427	1.603017e-01
x123	x123	1.580604e-01
x483	x483	1.563786e-01

```
X1896 X1896 1.561820e-01
X1735 X1735 1.543707e-01
X1937 X1937 1.540193e-01
X2083 X2083 1.529346e-01
X1076 X1076 1.505201e-01
X1387 X1387 1.423638e-01
X808   X808 1.398355e-01
X1645 X1645 1.378101e-01
X442   X442 1.376118e-01
X1389 X1389 1.375247e-01
X1799 X1799 1.354140e-01
X1116 X1116 1.321542e-01
X1655 X1655 1.296631e-01
X2000 X2000 1.267228e-01
X1738 X1738 1.196998e-01
X1964 X1964 1.165778e-01
X1917 X1917 1.141107e-01
X1006 X1006 1.070502e-01
X230   X230 1.066377e-01
X796   X796 1.061548e-01
X632   X632 1.059205e-01
X965   X965 1.053068e-01
X419   X419 1.022083e-01
```




```
> bt.try = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=5000, bag.f
raction=1)
> summary(bt.try, plotit=F)$rel.inf
```

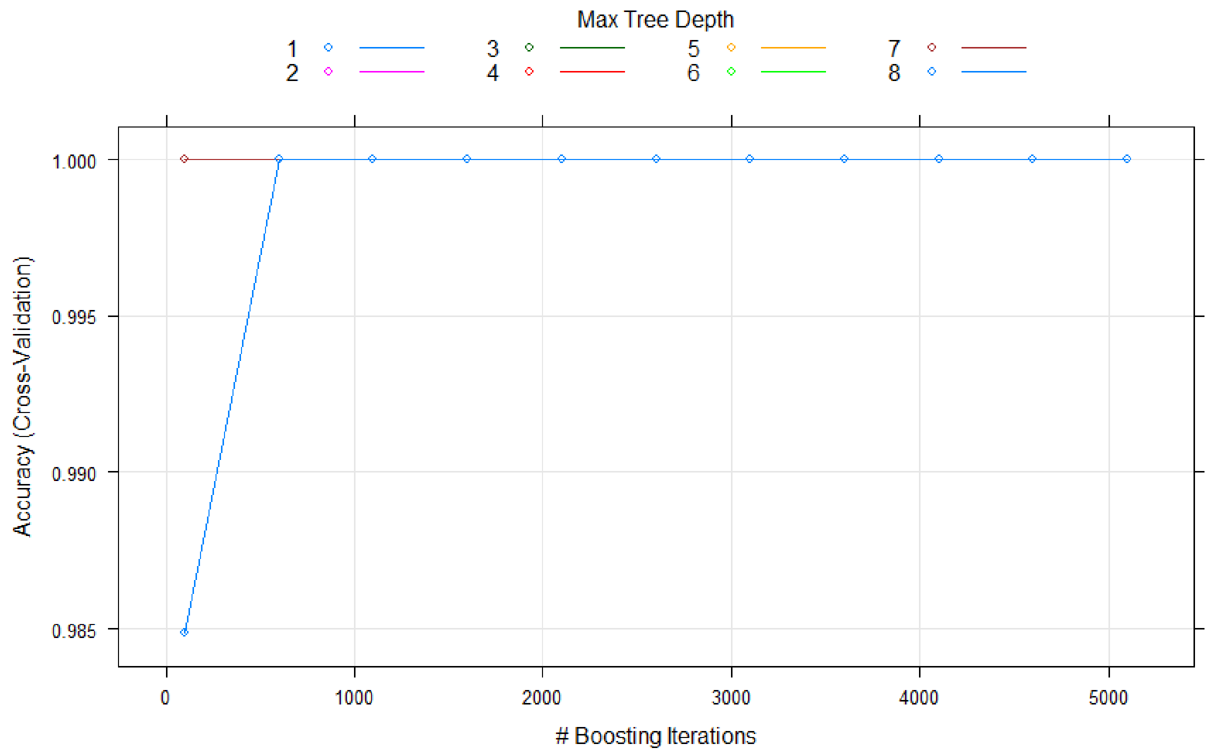
```
   [1] 33.969423507  9.809201193  9.700345785  6.914255696  6.650711941  6.22561
6214  4.721334058  4.680644747
   [9]  2.724992992  2.419005373  1.980908043  1.224534725  1.164646137  1.12437
5093  1.107265831  0.841707751
  [17]  0.755638523  0.685814271  0.649475642  0.619591660  0.498416176  0.32208
7042  0.315982267  0.143280925
  [25]  0.138949413  0.138544182  0.106216733  0.101027284  0.072993308  0.05859
4321  0.054544407  0.041248216
  [33]  0.032171098  0.006455447  0.000000000  0.000000000  0.000000000  0.00000
0000  0.000000000  0.000000000
```




```
> set.seed(1)
> bt4b = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=2000, interac
tion.depth=4,shrinkage =0.01)
> mse(Khan$ytest, predict(bt4b, tt, n.trees=2000))
[1] 0.09916075
> bt4c = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=5000, interac
tion.depth=4,shrinkage =0.01)
> mse(Khan$ytest, predict(bt4c, tt, n.trees=5000))
[1] 0.09661525


#Cross-validation using caret
```

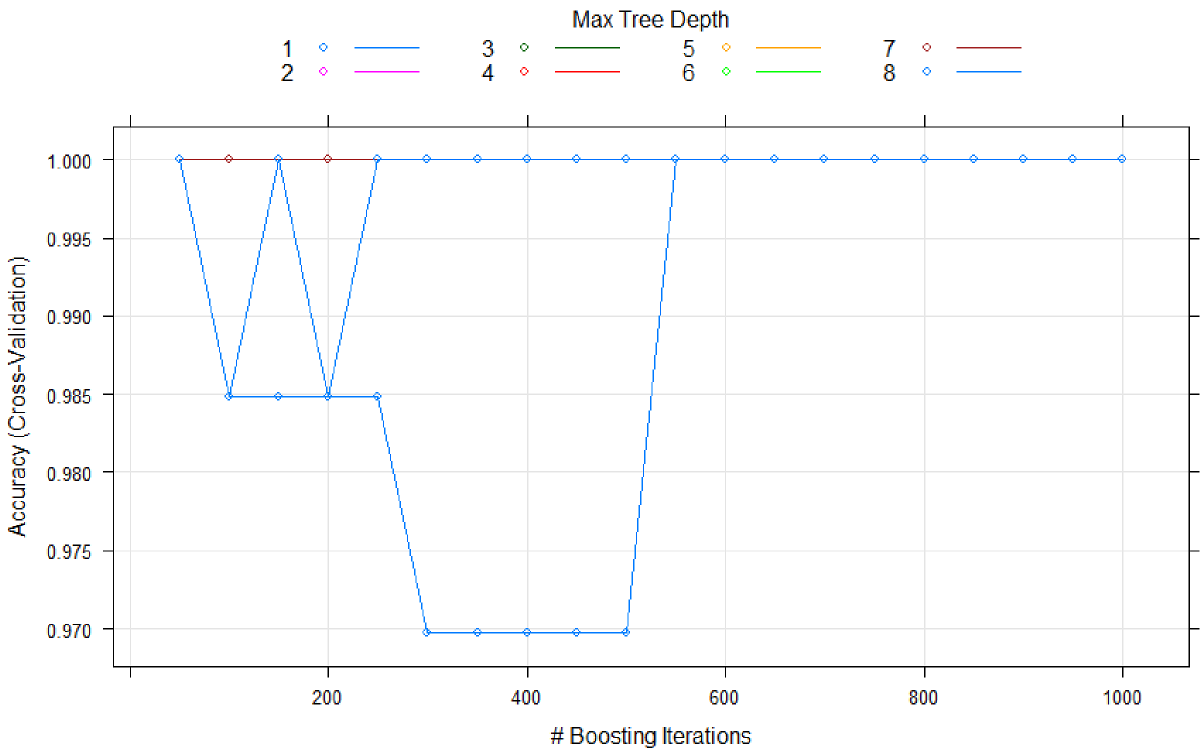
> bt4d = gbm(ytrain ~ ., data=dd, distribution="gaussian", n.trees=5000,
interaction.depth=4,shrinkage =0.1)
> mse(Khan$ytest, predict(bt4d, tt, n.trees=5000))
[1] 0.0715025

shrinkage =0.1 show better model results.




```
   n.trees interaction.depth shrinkage n.minobsinnode
12     100                 2      0.01              5
```

```
> boost.caretk$results
   shrinkage interaction.depth n.minobsinnode n.trees  Accuracy     Kappa Accura
cySD     KappaSD
1       0.01                 1              5     100 0.9848485 0.9789876 0.0262
4319 0.03639457
12      0.01                 2              5     100 1.0000000 1.0000000 0.0000
0000 0.00000000
23      0.01                 3              5     100 1.0000000 1.0000000 0.0000
0000 0.00000000
34      0.01                 4              5     100 1.0000000 1.0000000 0.0000
0000 0.00000000
45      0.01                 5              5     100 1.0000000 1.0000000 0.0000
0000 0.00000000
56      0.01                 6              5     100 1.0000000 1.0000000 0.0000
0000 0.00000000
67      0.01                 7              5     100 1.0000000 1.0000000 0.0000
0000 0.00000000
78      0.01                 8              5     100 0.9848485 0.9789876 0.0262
4319 0.03639457
2       0.01                 1              5     600 1.0000000 1.0000000 0.0000
0000 0.00000000
13      0.01                 2              5     600 1.0000000 1.0000000 0.0000
0000 0.00000000
24      0.01                 3              5     600 1.0000000 1.0000000 0.0000
0000 0.00000000
```




```
> boost.caretk
Stochastic Gradient Boosting

  63 samples
2308 predictors
   4 classes: '1', '2', '3', '4'

Pre-processing: centered (2308), scaled (2308)
Resampling: Cross-Validated (3 fold)
Summary of sample sizes: 42, 41, 43
Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy   Kappa
  1                    50     1.0000000  1.0000000
  1                   100     0.9848485  0.9789876
  1                   150     0.9848485  0.9789876
  1                   200     0.9848485  0.9789876
  1                   250     0.9848485  0.9789876
  1                   300     0.9696970  0.9583333
  1                   350     0.9696970  0.9583333
  1                   400     0.9696970  0.9583333
  1                   450     0.9696970  0.9583333
  1                   500     0.9696970  0.9583333
  1                   550     1.0000000  1.0000000
  1                   600     1.0000000  1.0000000
  1                   650     1.0000000  1.0000000
  1                   700     1.0000000  1.0000000
  1                   750     1.0000000  1.0000000
  1                   800     1.0000000  1.0000000
  1                   850     1.0000000  1.0000000
  1                   900     1.0000000  1.0000000
```

```
  1                    950     1.0000000  1.0000000
  1                   1000     1.0000000  1.0000000
  2                     50     1.0000000  1.0000000
  2                    100     1.0000000  1.0000000
  2                    150     1.0000000  1.0000000
  2                    200     1.0000000  1.0000000
  2                    250     1.0000000  1.0000000
  2                    300     1.0000000  1.0000000
  2                    350     1.0000000  1.0000000
  2                    400     1.0000000  1.0000000
  2                    450     1.0000000  1.0000000
  2                    500     1.0000000  1.0000000
  2                    550     1.0000000  1.0000000
  2                    600     1.0000000  1.0000000
  2                    650     1.0000000  1.0000000
  2                    700     1.0000000  1.0000000
  2                    750     1.0000000  1.0000000
  2                    800     1.0000000  1.0000000
  2                    850     1.0000000  1.0000000
  2                    900     1.0000000  1.0000000
  2                    950     1.0000000  1.0000000
  2                   1000     1.0000000  1.0000000
  3                     50     1.0000000  1.0000000
  3                    100     1.0000000  1.0000000
  3                    150     1.0000000  1.0000000
  3                    200     1.0000000  1.0000000
  3                    250     1.0000000  1.0000000
  3                    300     1.0000000  1.0000000
  3                    350     1.0000000  1.0000000
  3                    400     1.0000000  1.0000000
  3                    450     1.0000000  1.0000000
  3                    500     1.0000000  1.0000000

Tuning parameter 'shrinkage' was held constant at a value of 0.01
Tuning parameter 'n.minobsinnode' was held
 constant at a value of 5
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 50, interaction.depth = 1, sh
rinkage = 0.01 and n.minobsinnode = 5.
> plot(boost.caretk)
>
> #Using the optimal hyperparameters selected by train() improves the result!
> boost.caretk$bestTune
  n.trees interaction.depth shrinkage n.minobsinnode
1      50                 1      0.01              5



> boost.caretk$finalModel
A gradient boosted model with multinomial loss function.
50 iterations were performed.
There were 2308 predictors of which 27 had non-zero influence.

> predict(boost.caretk, tt)
 [1] 3 2 4 2 1 3 4 2 3 1 3 4 1 2 2 2 4 3 4 3
Levels: 1 2 3 4
> Khan$ytest
```

```
 [1] 3 2 4 2 1 3 4 2 3 1 3 4 1 2 2 2 4 3 4 3
```
same as the predict value

```
> mse(Khan$ytest, predict(boost.caretk, tt))
[1] NA
```

Week10_ISLR Chapter 9 Exercises 8

8. This problem involves the OJ data set which is part of the ISLR package.

(a) Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations.

```
library(ISLR)
?OJ
fix(OJ)
names(OJ)
dim(OJ) ## variable names and dataset dimensions
summary(OJ)

set.seed(42)
train <- sample(nrow(OJ), 800)
OJ.train <- OJ[train, ]
OJ.test <- OJ[-train, ]
```

(b) Fit a support vector classifier to the training data using cost=0.01, with Purchase as the response and the other variables as predictors. Use the summary() function to produce summary statistics, and describe the results obtained.

```
library(e1071)
svm.linear <- svm(Purchase ~ ., data = OJ.train, kernel = "linear", cost =
0.01)
summary(svm.linear)
```

```
Call:
svm(formula = Purchase ~ ., data = OJ.train, kernel = "linear", cost = 0.01)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  0.01
      gamma:  0.05555556

Number of Support Vectors:  439

 ( 219 220 )


Number of Classes:  2

Levels:
 CH MM
```

- Support vector classifier creates 439 support vectors out of 800 training points. Out of these, 220 belong to level MM and remaining 219 belong to level CH.

(c) What are the training and test error rates?

```
train.pred <- predict(svm.linear, OJ.train)
table(OJ.train$Purchase, train.pred)
test.pred <- predict(svm.linear, OJ.test)
table(OJ.test$Purchase, test.pred)
```

```
   train.pred
      CH  MM
  CH 428  54
  MM  78 240
>
    test.pred
      CH  MM
  CH 150  21
  MM  29  70

> (78 + 54) / (428 + 240 + 78 + 54)
[1] 0.165
> (29 + 21) / (150 + 70 + 29 + 21)
[1] 0.1851852
```

- The test error rate is about 16.5%.
- The test error rate is about 18.5%.

(d) Use the tune() function to select an optimal cost. Consider values in the range 0.01 to 10.

```
set.seed(2)
tune.out <- tune(svm, Purchase ~ ., data = OJ.train, kernel = "linear",
ranges = list(cost = 10^seq(-2, 1, by = 0.25)))
summary(tune.out)
```

```
Parameter tuning of ‘svm’:

- sampling method: 10-fold cross validation

- best parameters:
      cost
 0.3162278

- best performance: 0.16875

- Detailed performance results:
          cost   error dispersion
1   0.01000000 0.17375 0.05285265
2   0.01778279 0.17000 0.05210833
3   0.03162278 0.17750 0.04958158
4   0.05623413 0.17625 0.04466309
5   0.10000000 0.17125 0.04372023
```

```
6    0.17782794 0.17125 0.05104804
7    0.31622777 0.16875 0.04649149
8    0.56234133 0.16875 0.04759858
9    1.00000000 0.17000 0.05006940
10   1.77827941 0.17000 0.05210833
11   3.16227766 0.17000 0.05374838
12   5.62341325 0.17000 0.05439056
13  10.00000000 0.17125 0.05560588
```

- The optimal cost is 0.3162278.

(e) Compute the training and test error rates using this new value for cost.

```
svm.linear <- svm(Purchase ~ ., kernel = "linear", data = OJ.train, cost =
tune.out$best.parameter$cost)
train.pred <- predict(svm.linear, OJ.train)
table(OJ.train$Purchase, train.pred)

test.pred <- predict(svm.linear, OJ.test)
table(OJ.test$Purchase, test.pred)
```

```
    train.pred
      CH  MM
  CH 425  57
  MM  74 244
>

    test.pred
      CH  MM
  CH 151  20
  MM  25  74
> (74 + 57) / (425 + 244 + 74 + 57)
[1] 0.16375
> (25 + 20) / (151 + 74 + 25 + 20)
[1] 0.1666667
```

- The test error rate is about 16.4%.
- The test error rate is about 16.7%.
- With the best cost, the training error rate is now 16.4% and the test error rate is 16.7%. The results get better.

(f) Repeat parts (b) through (e) using a support vector machine with a radial kernel. Use the default value for gamma.

```
svm.radial <- svm(Purchase ~ ., kernel = "radial", data = OJ.train)
summary(svm.radial)

train.pred <- predict(svm.radial, OJ.train)
table(OJ.train$Purchase, train.pred)
test.pred <- predict(svm.radial, OJ.test)
table(OJ.test$Purchase, test.pred)

set.seed(2)
```

```
tune.out <- tune(svm, Purchase ~ ., data = OJ.train, kernel = "radial",
ranges = list(cost = 10^seq(-2, 1, by = 0.25)))
summary(tune.out)

svm.radial <- svm(Purchase ~ ., kernel = "radial", data = OJ.train, cost =
tune.out$best.parameter$cost)
summary(svm.radial)
train.pred <- predict(svm.radial, OJ.train)
table(OJ.train$Purchase, train.pred)
test.pred <- predict(svm.radial, OJ.test)
table(OJ.test$Purchase, test.pred)
```

```
Call:
svm(formula = Purchase ~ ., data = OJ.train, kernel = "radial")


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.05555556

Number of Support Vectors:  382

 ( 191 191 )


Number of Classes:  2

Levels:
 CH MM


    train.pred
      CH  MM
  CH 443  39
  MM  78 240

    test.pred
      CH  MM
  CH 152  19
  MM  24  75
```

- The test error rate is about 14.6%.
- The test error rate is about 15.9%.

```
Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
      cost
 0.5623413

- best performance: 0.1725
```

```
- Detailed performance results:
          cost   error dispersion
1   0.01000000 0.39750 0.04632314
2   0.01778279 0.39750 0.04632314
3   0.03162278 0.35750 0.04721405
4   0.05623413 0.20875 0.04641674
5   0.10000000 0.18375 0.04860913
6   0.17782794 0.18000 0.04901814
7   0.31622777 0.17625 0.05726704
8   0.56234133 0.17250 0.05096295
9   1.00000000 0.17625 0.04016027
10  1.77827941 0.17750 0.03574602
11  3.16227766 0.17500 0.03584302
12  5.62341325 0.18000 0.03395258
13 10.00000000 0.18875 0.02972676


Call:
svm(formula = Purchase ~ ., data = OJ.train, kernel = "radial", cost = tune.out
$best.parameter$cost)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  0.5623413
      gamma:  0.05555556

Number of Support Vectors:  408

 ( 202 206 )


Number of Classes:  2

Levels:
 CH MM



    train.pred
      CH  MM
  CH 442  40
  MM  79 239

    test.pred
      CH  MM
  CH 154  17
  MM  23  76
```

- The test error rate is about 14.8%.
- The test error rate is about 14.8%.

(g) Repeat parts (b) through (e) using a support vector machine with a polynomial kernel. Set degree=2.

```
svm.poly <- svm(Purchase ~ ., kernel = "polynomial", data = OJ.train,
degree = 2)
summary(svm.poly)
train.pred <- predict(svm.poly, OJ.train)
table(OJ.train$Purchase, train.pred)
test.pred <- predict(svm.poly, OJ.test)
table(OJ.test$Purchase, test.pred)

set.seed(2)
tune.out <- tune(svm, Purchase ~ ., data = OJ.train, kernel = "polynomial",
degree = 2, ranges = list(cost = 10^seq(-2, 1, by = 0.25)))
summary(tune.out)

svm.poly <- svm(Purchase ~ ., kernel = "polynomial", degree = 2, data =
OJ.train, cost = tune.out$best.parameter$cost)
summary(svm.poly)

train.pred <- predict(svm.poly, OJ.train)
table(OJ.train$Purchase, train.pred)
test.pred <- predict(svm.poly, OJ.test)
table(OJ.test$Purchase, test.pred)
```

```
    train.pred
      CH  MM
  CH 444  38
  MM 113 205

    test.pred
      CH  MM
  CH 157  14
  MM  33  66
```

- The test error rate is about 17.1%.
- The test error rate is about 17.4%.

```
Parameter tuning of ‘svm’:

- sampling method: 10-fold cross validation

- best parameters:
 cost
   10

- best performance: 0.18625

- Detailed performance results:
         cost   error dispersion
1   0.01000000 0.39750 0.04632314
2   0.01778279 0.38125 0.03963812
3   0.03162278 0.37375 0.03839216
4   0.05623413 0.34750 0.02934469
5   0.10000000 0.33875 0.04226652
6   0.17782794 0.25000 0.03726780
7   0.31622777 0.22375 0.02729087
```

```
8    0.56234133 0.22000 0.02898755
9    1.00000000 0.20500 0.03496029
10   1.77827941 0.20250 0.03525699
11   3.16227766 0.19750 0.02687419
12   5.62341325 0.19000 0.03050501
13 10.00000000 0.18625 0.03884174

Call:
svm(formula = Purchase ~ ., data = OJ.train, kernel = "polynomial", degree = 2,
cost = tune.out$best.parameter$cost)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  10
     degree:  2
      gamma:  0.05555556
     coef.0:  0

Number of Support Vectors:  350

 ( 170 180 )


Number of Classes:  2

Levels:
 CH MM


    train.pred
      CH  MM
  CH 441  41
  MM  80 238

    test.pred
      CH  MM
  CH 152  19
  MM  29  70
```

- The test error rate is about 15.1%.
- The test error rate is about 14.1%.

(h) Overall, which approach seems to give the best results on this data?

	training error (basis)	*test error(basis)*
Linear kernel	*16.5%*	*18.5%*
Radial kernel	*14.6%*	*15.9%*
Polynomial kernel	*17.1%*	*17.4%*

	training error (best)	*test error(best)*

Linear kernel	*16.4%*	*16.7%*
Radial kernel	*14.8%*	*14.8%*
Polynomial kernel	*15.1%*	*14.1%*

- Overall, radial basis kernel seems to be producing minimum misclassification error on both train and test data.
- Radical Kernel and Polynomial Kernel all show better performance than Linear kernel when selecting an optimal cost.