ANLY512_HW8

Hongyang Zheng

2019/4/8

```
library(MASS)
library(ISLR)
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
library(tree)
require(gbm)
```

```
## Loading required package: gbm
```

```
## Warning: package 'gbm' was built under R version 3.5.2
```

```
## Loaded gbm 2.1.5
```

Problem #4

a)

See Diagram for part a

b)

See Diagram for part b

Problem #7

```
set.seed(1234)

# Split the data into train(70%) and test(30%)
train = sample(dim(Boston)[1], 0.7*dim(Boston)[1])
X.train = Boston[train, -14]
X.test = Boston[-train, -14]
Y.train = Boston[train, 14]
Y.test = Boston[-train, 14]

# Different mtry
p = dim(Boston)[2] - 1
p.2 = p/2
p.3 = p/3
p.sq = sqrt(p)

# Build different models
rf.p = randomForest(X.train, Y.train, xtest = X.test, ytest = Y.test,
```

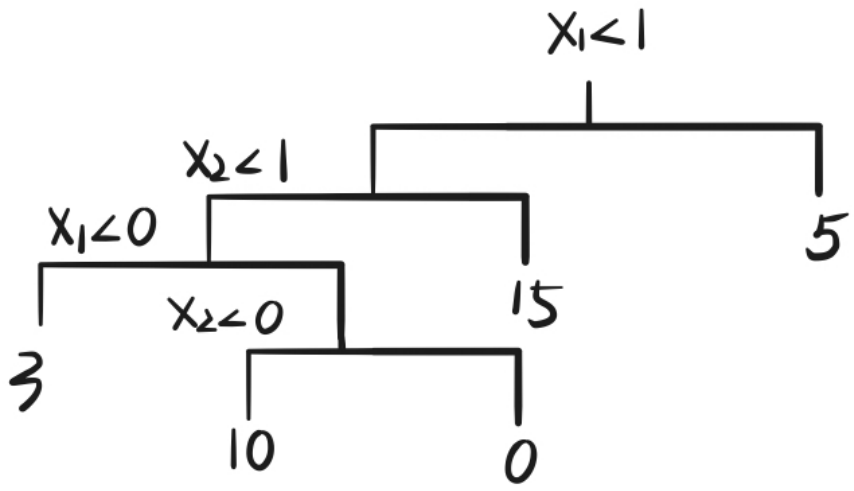


Figure 1: Diagram for part a

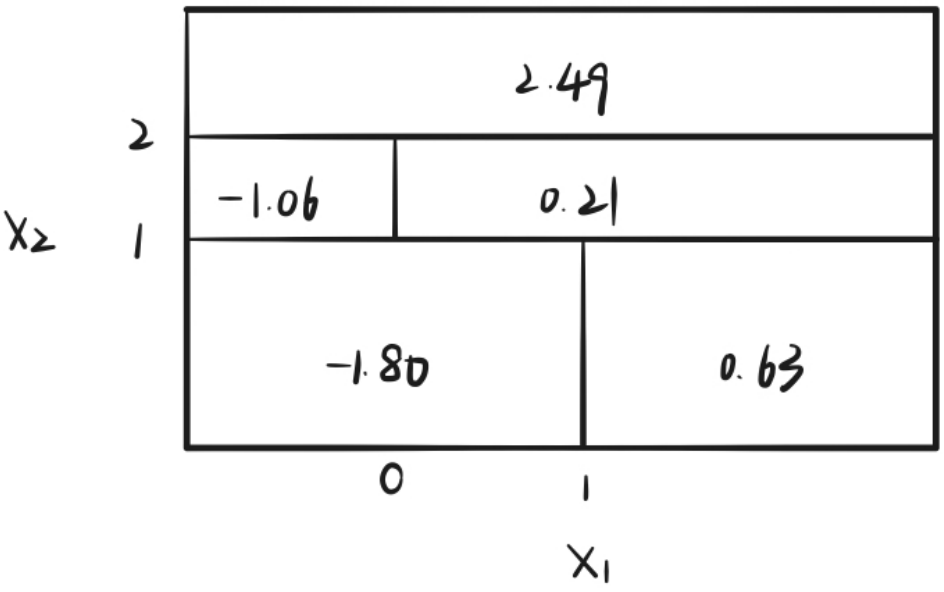
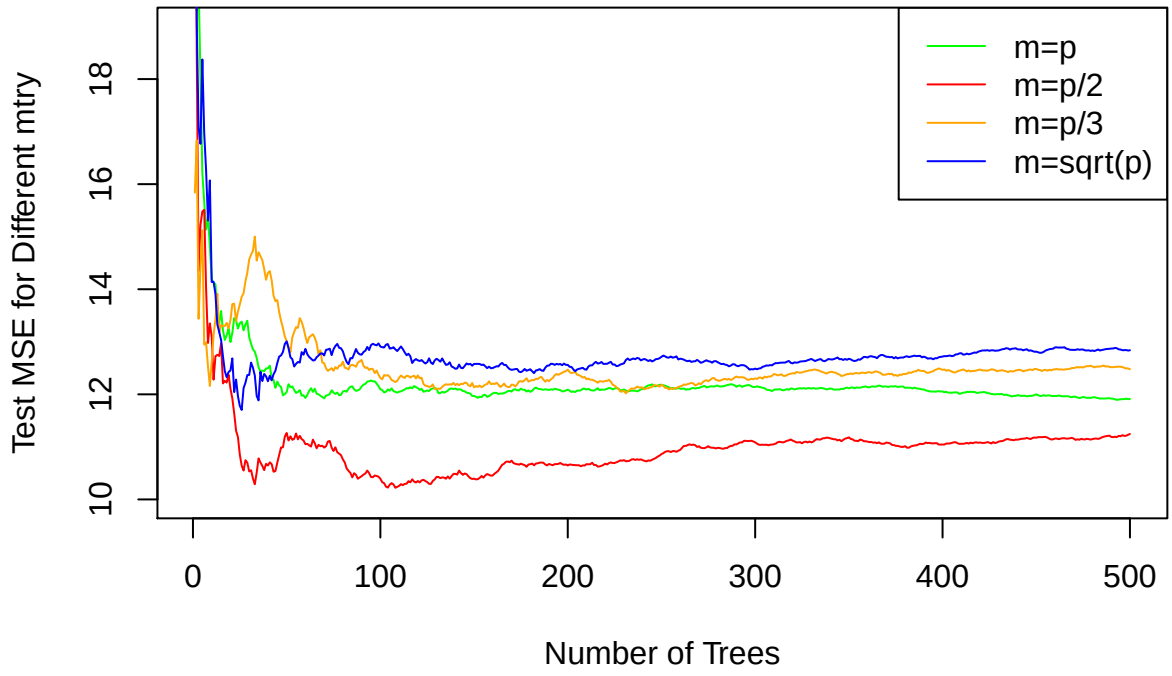


Figure 2: Diagram for part b

```
    mtry = p, ntree = 500)
rf.p.2 = randomForest(X.train, Y.train, xtest = X.test, ytest = Y.test,
    mtry = p.2, ntree = 500)
rf.p.3 = randomForest(X.train, Y.train, xtest = X.test, ytest = Y.test,
    mtry = p.3, ntree = 500)
rf.p.sq = randomForest(X.train, Y.train, xtest = X.test, ytest = Y.test,
    mtry = p.sq, ntree = 500)

# Make a plot
plot(1:500, rf.p$test$mse, col = "green", type = "l", xlab = "Number of Trees",
     ylab = "Test MSE for Different mtry", ylim = c(10, 19))
lines(1:500, rf.p.2$test$mse, col = "red", type = "l")
lines(1:500, rf.p.3$test$mse, col = "orange", type = "l")
lines(1:500, rf.p.sq$test$mse, col = "blue", type = "l")
legend("topright", c("m=p", "m=p/2", "m=p/3", "m=sqrt(p)"), col = c("green", "red", "orange", "blue"),
```



The plot shows that test MSE for single tree is very high and it is reduced by adding more trees to the model and stabilizes after 150-200 trees. The model including half of all variables performs best since its test MSE always below the others after n=20. The test MSE for the model including sqrt of all variables performs not very good (even bad than the model including all variables), which surprises me.

Problem #9

a)

```
set.seed(1234)

train = sample(dim(OJ)[1], 800)
OJ.train = OJ[train, ]
OJ.test = OJ[-train, ]
```

b)

```
# Fit a tree model
tree.fit = tree(Purchase ~ ., data = OJ.train)
summary(tree.fit)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ.train)
## Variables actually used in tree construction:
## [1] "LoyalCH"   "PriceDiff"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7573 = 599.8 / 792
## Misclassification error rate: 0.1688 = 135 / 800
```

From the summary we can see that the training error rate is about 16.88%. The tree contains 8 terminal nodes with only using the variables "LoyalCH" and "PriceDiff".

c)

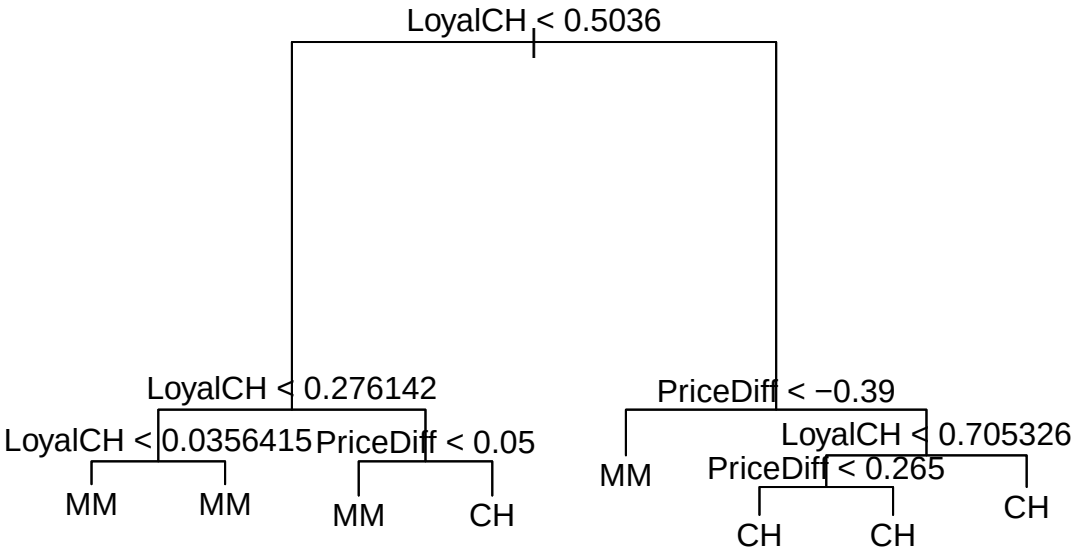
```
tree.fit
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 800 1069.00 CH ( 0.61125 0.38875 )
##    2) LoyalCH < 0.5036 356  428.30 MM ( 0.28933 0.71067 )
##      4) LoyalCH < 0.276142 169  130.70 MM ( 0.13018 0.86982 )
##        8) LoyalCH < 0.0356415 54    0.00 MM ( 0.00000 1.00000 ) *
##        9) LoyalCH > 0.0356415 115  112.30 MM ( 0.19130 0.80870 ) *
##      5) LoyalCH > 0.276142 187  255.90 MM ( 0.43316 0.56684 )
##       10) PriceDiff < 0.05 79   82.28 MM ( 0.21519 0.78481 ) *
##       11) PriceDiff > 0.05 108  146.00 CH ( 0.59259 0.40741 ) *
##    3) LoyalCH > 0.5036 444  344.20 CH ( 0.86937 0.13063 )
##      6) PriceDiff < -0.39 26   34.65 MM ( 0.38462 0.61538 ) *
##      7) PriceDiff > -0.39 418  272.60 CH ( 0.89952 0.10048 )
##       14) LoyalCH < 0.705326 138  141.90 CH ( 0.78986 0.21014 )
##         28) PriceDiff < 0.265 73   95.07 CH ( 0.64384 0.35616 ) *
##         29) PriceDiff > 0.265 65   24.31 CH ( 0.95385 0.04615 ) *
##       15) LoyalCH > 0.705326 280  105.20 CH ( 0.95357 0.04643 ) *
```

Pick the terminal node labeled 8). The splitting variable at this node is 'LoyalCH'. The splitting value of this node is 0.0356415. There are 54 points in the subtree. The deviance for all points contained in the sub-region is 0.00, since all customers under this subtree will buy MM.

d)

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



From the graph we can see that 'LoyalCH' is the most important variable for predicting the results of purchasing(the top node is LoyalCH). For example, when 'LoyalCH' < 0.276142, the customers will buy MM. However, when 'LoyalCH' > 0.5036, 'PriceDiff' also has an impact. For example, when 'PriceDiff' < -0.39, the customers will buy MM(but when 'PriceDiff' > -0.39, the PriceDiff will lose the influence, since the customers will buy CH).

e)

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

```
##     tree.pred
##       CH  MM
##   CH 152  12
##   MM  35  71
```

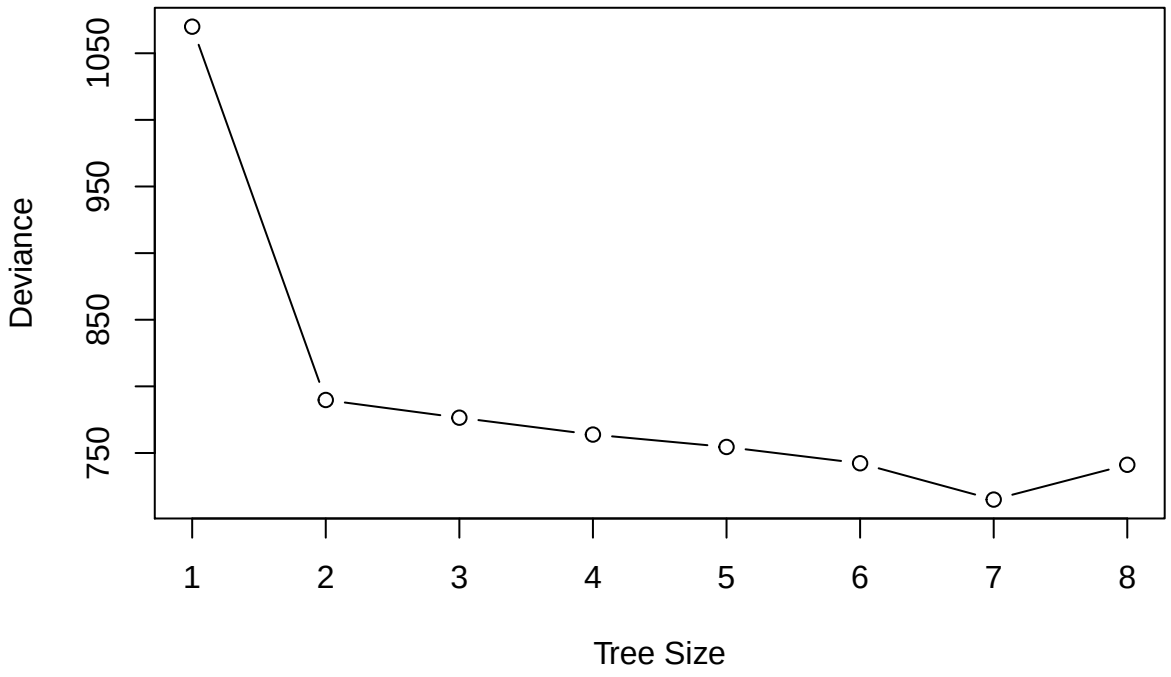
The test error rate is (12+35)/(12+35+152+71)=17.41%

f)

```
cv.fit = cv.tree(tree.fit, FUN = prune.tree)
```

g)

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



h)

Tree size = 7 has the lowest cross-validation classification error rate.

i)

```
tree.pruned = prune.tree(tree.fit, best = 7)
```

j)

```
summary(tree.pruned)
```

```
##
## Classification tree:
## snip.tree(tree = tree.fit, nodes = 4L)
## Variables actually used in tree construction:
## [1] "LoyalCH"   "PriceDiff"
## Number of terminal nodes:  7
## Residual mean deviance:  0.7796 = 618.2 / 793
## Misclassification error rate: 0.1688 = 135 / 800
```

From the summary we can see that the training error rate is about 16.88%, which is the same as the unpruned tree.

k)

```
pred.pruned = predict(tree.pruned, OJ.test, type = "class")
misclass.pruned = sum(OJ.test$Purchase != pred.pruned)
misclass.pruned/length(pred.pruned)
```

```
## [1] 0.1740741
```

The test error rate for this pruned tree is about 0.1741, which is the same as the unpruned tree.

Problem Extra 59

```
set.seed(1234)
MNIST=load('~/Desktop/other/Data/mnist_all.RData')

# Generate dataframe
# Extract data for digit4 and digit5
y.nist = train$y
index <- (y.nist == 4 | y.nist == 5)
x.nist <- train$x[index,]
x.train <- as.data.frame(x.nist)

y.nist1 = test$y
index1 <- (y.nist1 == 4 | y.nist1 == 5)
x.nist1 <- test$x[index1,]
x.test <- as.data.frame(x.nist1)

# y=1 if it is digit4, y=0 if it is digit5
x.train$y <- 0
x.train$y[y.nist[index] == 4] <- 1
x.test$y <- 0
x.test$y[y.nist1[index1] == 4] <- 1

# Convert y to factor
x.train$y <- as.factor(x.train$y)
x.test$y <- as.factor(x.test$y)
summary(x.train$y)
```

```
##    0    1
## 5421 5842
```

a)

```
# Fit a classification tree to the training data
tree.1=tree(y~., data=x.train)
summary(tree.1)
```

```
##
## Classification tree:
## tree(formula = y ~ ., data = x.train)
## Variables actually used in tree construction:
##  [1] "V597" "V327" "V325" "V357" "V324" "V436" "V438" "V409" "V323" "V517"
## [11] "V355"
## Number of terminal nodes:  12
## Residual mean deviance:  0.2215 = 2492 / 11250
## Misclassification error rate: 0.03205 = 361 / 11263
```

```
# Predict using test data
pred.1.test=predict(tree.1, newdata=x.test, type='class')
misclass.1 = sum(x.test$y != pred.1.test)
error_rate.1=misclass.1/length(pred.1.test)
```

The prediction error is about 0.0336179.

b)

```
# Fit a random forest model using training data
forest.1 = randomForest(y ~ ., data=x.train, ntree = 100)
```

```
# Predict using test data
pred.2.test=predict(forest.1, newdata=x.test)
misclass.2 = sum(x.test$y != pred.2.test)
error_rate.2=misclass.2/length(pred.2.test)
```

The prediction error is about 0, which is lower than using a single tree.

c)

```
ntrain = dim(x.train)[1]

# Number of bootstrap samples
N = 100
# List of 100 trees
mybag = as.list(rep(NA,N))

# Fit a bagging model to the training data
for (j in 1:N)
{
  bootstrap = sample(ntrain, replace = T)
  mybag[[j]] = tree(y ~ ., data = x.train[bootstrap,])
}
```

```
# Predict using test data
error_rate.3=rep(NA,100)

for (j in 1:N){
  pred.3.test=predict(mybag[[j]], newdata=x.test, type='class')
  misclass.3 = sum(x.test$y != pred.3.test)
  error_rate.3[j]=misclass.3/length(pred.3.test)
}
avg_error_rate = sum(error_rate.3)/N
```

The average error rate for bagging 100 trees is 0.0325293.

d)

I simulate 100 trees which can be finished in 3 minutes. The random forest performs best, which has the lowest error rate about 0.001. Bagging and a single tree have similar error rate.

Problem Extra 61

a)

```
# Load data
library(readxl)
concrete=read_excel('~/Desktop/HW/Concrete_Data.xls')
names(concrete)=c('Cement', 'Slag', 'Fly', 'Water', 'Super', 'Coarse', 'Fine', 'Age', 'Strength')
```

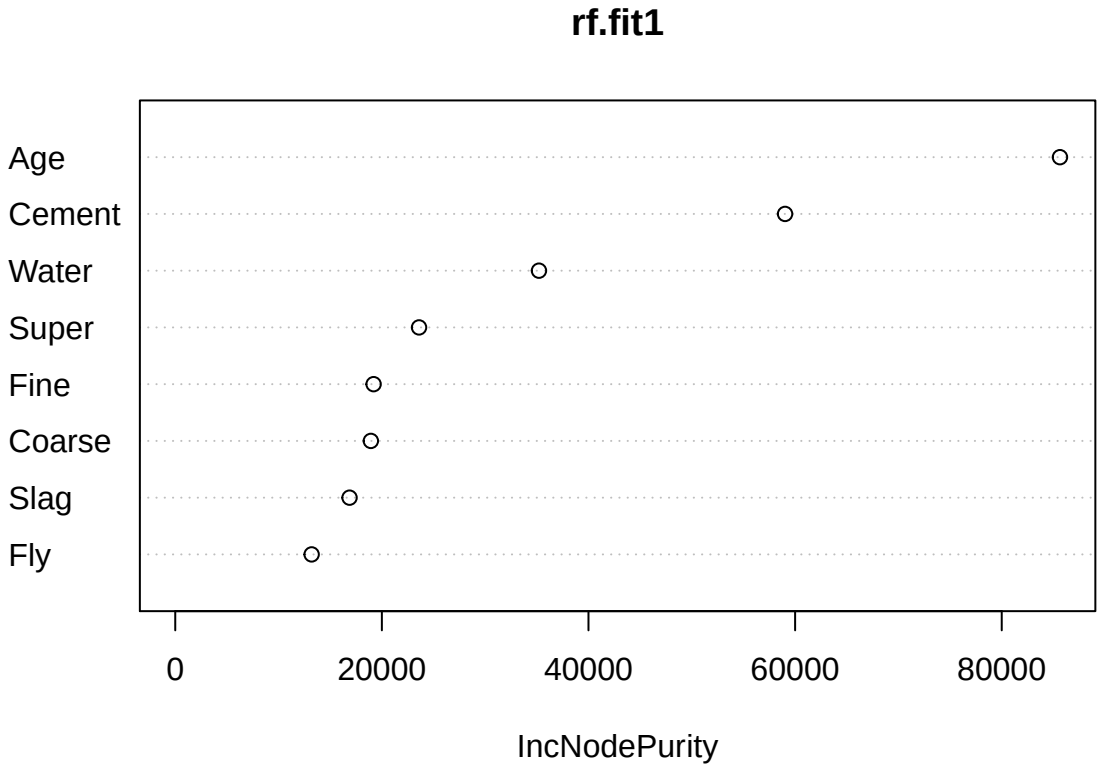
```r
# Build a random forest
rf.fit1=randomForest(Strength~., data=concrete)
```

```r
# 10-fold cross validation
n_data <- dim(concrete)[1]
n_folds <- 10
folds_i <- sample(rep(1:n_folds, length.out = n_data))

error=rep(NA, 10)

for (k in 1:n_folds) {
    # Split train and test data
    test_i <- which(folds_i == k)
    train <- concrete[-test_i, ]
    test <- concrete[test_i, ]

    # Fit model and make prediction
    fitted_models <- randomForest(Strength~., data=train)
    pred <- predict(fitted_models, newdata=test)

    # Calculate rms
    error[k] <- sqrt(mean((test$Strength-pred)^2))
}

avg_error=sum(error)/10
```

The rms prediction error using 10-fold cross validation is 5.1321533.

b)

```r
varImpPlot(rf.fit1)
```



c)

```r
# Build a random forest without the least important predictor
n_data <- dim(concrete)[1]
n_folds <- 10
folds_i <- sample(rep(1:n_folds, length.out = n_data))

error2=rep(NA, 10)

for (k in 1:n_folds) {
    # Split train and test data
    test_i <- which(folds_i == k)
    train <- concrete[-test_i, ]
    test <- concrete[test_i, ]

    # Fit model and make prediction
    fitted_models <- randomForest(Strength~.-Fly, data=train)
    pred <- predict(fitted_models, newdata=test)

    # Calculate rms
    error2[k] <- sqrt(mean((test$Strength-pred)^2))
}

avg_error2=sum(error2)/10
```

The estimated rms prediction error decreases slightly, which means excluding the least important predictor makes the model performs better.

Problem Extra 62

```
set.seed(1234)
# Split the train(70%) and test(30%) dataset
n=dim(concrete)[1]
train.index=sample(n, 0.7*n, replace=FALSE)
train.concrete=concrete[train.index, ]
test.concrete=concrete[-train.index,]
```

```
# Write a function to calculate rms taking n.trees, interaction.depth and shrinkage as arguments for di
rms <- function(t, d, s)
{
# Build model
model <- gbm(Strength ~ ., data = train.concrete, distribution = "gaussian", n.trees = t,
               interaction.depth = d, shrinkage = s)

# Predict
pred.train <- predict(model, newdata=train.concrete, n.trees = t)
pred.test <- predict(model, newdata=test.concrete, n.trees = t)

# Calculate rms
error.train <- sqrt(mean((train.concrete$Strength-pred.train)^2))
error.test <- sqrt(mean((test.concrete$Strength-pred.test)^2))

return(c(error.train, error.test))
}

#### Try different n.trees and fix others
# Model.1 with 500, 4, 0.1
print(rms(500, 4, 0.1))
```

```
## [1] 2.694437 4.432223
```

```
# Model.2 with 1000, 4, 0.1
print(rms(1000, 4, 0.1))
```

```
## [1] 2.093628 4.387216
```

```
# Model.3 with 2000, 4, 0.1
print(rms(2000, 4, 0.1))
```

```
## [1] 1.757227 4.347354
```

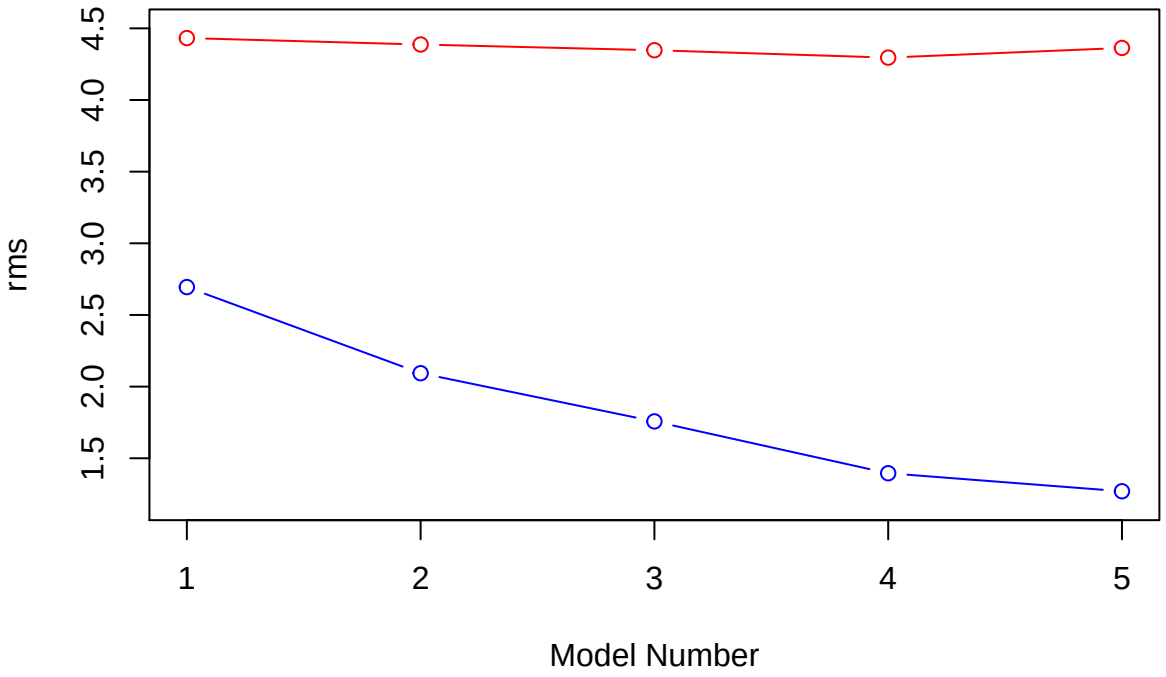
```
# Model.4 with 5000, 4, 0.1
print(rms(5000, 4, 0.1))
```

```
## [1] 1.395429 4.295716
```

```
# Model.5 with 10000, 4, 0.1
print(rms(10000, 4, 0.1))
```

```
## [1] 1.269634 4.363343
```

```
# Make a plot
plot(seq(1:5), c(4.432223, 4.387216, 4.347354, 4.295716, 4.363343), type='b', col='red', ylim = c(1.2, 4
     xlab='Model Number', ylab='rms', main='Different n.trees')
lines(seq(1:5), c(2.694437, 2.093628, 1.757227, 1.395429, 1.269634), type='b', col='blue')
```



From the graph we can see that the training rms keep decreasing, while the test rms decreases first and then increases when we change the n.trees from 5000 to 10000, which indicates that too large n.trees can make the model overfitting.

```
#### Try different depth and fix others
# Model.1 with 2000, 4, 0.1
print(rms(2000, 4, 0.1))
```

```
## [1] 1.710697 4.062444
```

```
# Model.2 with 2000, 8, 0.1
print(rms(2000, 8, 0.1))
```

```
## [1] 1.331882 4.107078
```

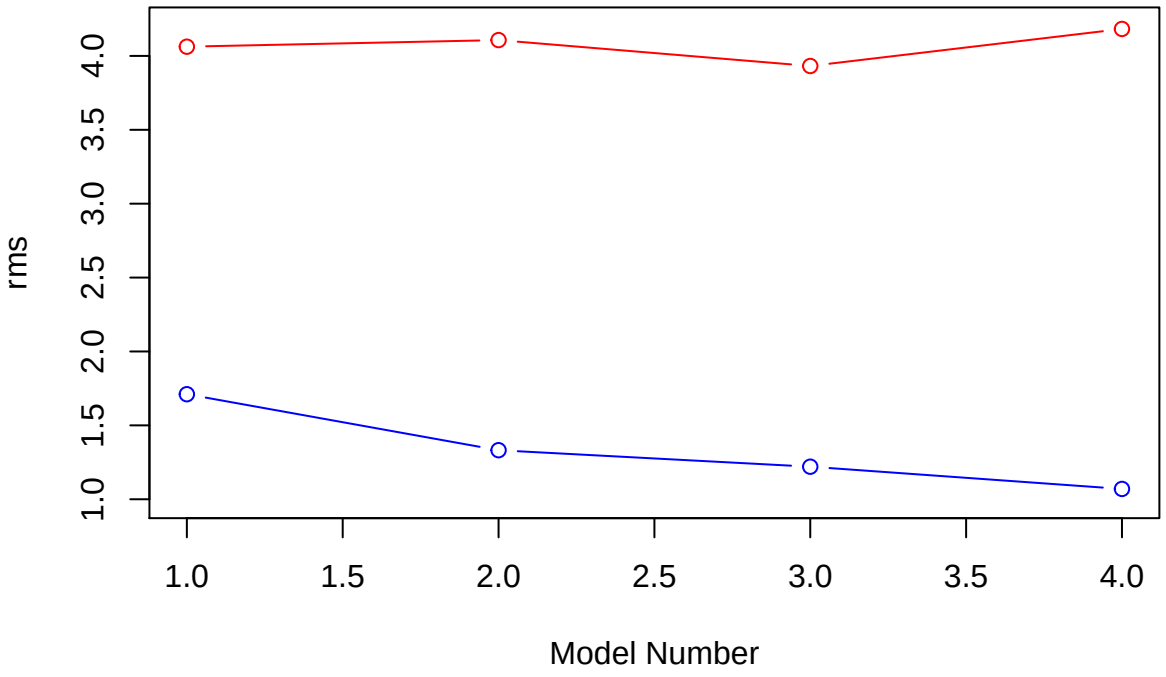
```
# Model.3 with 2000, 10, 0.1
print(rms(2000, 10, 0.1))
```

```
## [1] 1.220268 3.931792
```

```
# Model.4 with 5000, 12, 0.1
print(rms(5000, 12, 0.1))
```

```
## [1] 1.069756 4.182382
```

```
# Make a plot
plot(seq(1:4), c(4.062444, 4.107078, 3.931792, 4.182382), type='b', col='red', ylim = c(1, 4.2),
     xlab='Model Number', ylab='rms', main='Different interaction depth')
lines(seq(1:4), c(1.710697, 1.331882, 1.220268, 1.069756), type='b', col='blue')
```



From the graph we can see that the training rms keep decreasing, while the test rms decreases first and then increases when we change the depth from 4 to 8 and 10 to 12(dramatically), which indicates that too large depth can make the model overfitting.

```
#### Try different shrinkage and fix others
# Model.1 with 2000, 10, 0.1
print(rms(2000, 10, 0.1))
```

```
## [1] 1.219458 4.526188
```

```
# Model.2 with 2000, 10, 0.2
print(rms(2000, 10, 0.2))
```

```
## [1] 1.124741 4.461782
```

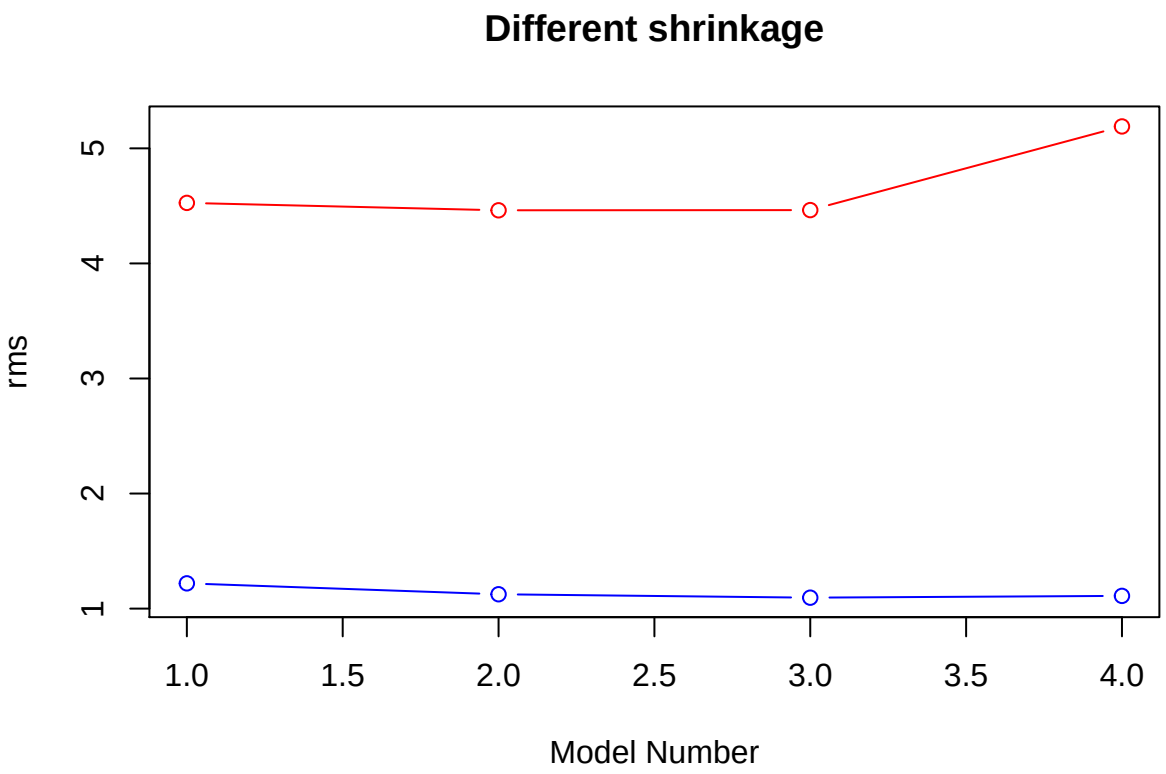
```
# Model.3 with 2000, 10, 0.3
print(rms(2000, 10, 0.3))
```

```
## [1] 1.095132 4.463490
```

```
# Model.3 with 2000, 10, 0.4
print(rms(2000, 10, 0.4))
```

```
## [1] 1.110482 5.190600
```

```
# Make a plot
plot(seq(1:4), c(4.526188, 4.461782, 4.463490, 5.190600), type='b', col='red', ylim = c(1.09, 5.2),
     xlab='Model Number', ylab='rms', main='Different shrinkage')
lines(seq(1:4), c(1.219458, 1.124741, 1.095132, 1.110482), type='b', col='blue')
```



From the graph we can see that the training rms keep decreasing, while the test rms decreases first and then increases when we change the learning rate from 0.2 to 0.3 and 0.3 to 0.4(dramatically), which indicates that too fast learning rate can make the model overfitting.