STAT542 Statistical Learning Homework 4

Huamin Zhang

Nov 14, 2017

Name: Huamin Zhang (huaminz2@illinois.edu)

Question 1

a) [15 points]

Answer:

We genetate a data example with 7200 observation in training data, and 1000 observation in test data.

```r
set.seed(1)
# Generate some data
n = 7200; testn = 1000; p = 1
x = matrix(runif(n*p), n, p); y = sin(2*pi*x) + 0.2*rnorm(n)
# Generate testing data points
test.x = matrix(runif(testn*p), testn, p); test.y = sin(2*pi*test.x) + 0.2*rnorm(testn)
```

Then, we write two functions. `NW_kernel` perform Nadaraya-Watson kernel regression estimator for a one dimensional problem and `cv.NW_kernel` perform cross-validation to tuning parameter for the bandwidth. We use the mean squared error as the loss function in the algorithm. Here is the code.

```r
# train_x/test_x: A matrix, each row is an observation vector in training/test data
# train_y/test_y: A matrix, response variable in training/test data
# bandwidth: The parameter which defines the width of the neighborhood
# Output: pred.y: A vector, the predict value on test data
#         test.error: The mean squared error of the prediction
NW_kernel<-function(train_x,train_y,test_x,test_y,bandwidth){
  train_x = matrix(train_x); train_y = matrix(train_y)
  test_x = matrix(test_x); test_y = matrix(test_y)
  # Any constant will be absorbed into the bandwidth, 1/sqrt(2pi) will not matter
  # either since in the NW estimator, you will have to divide by the normalizing
  # constant. Thus, we didn't consider the constant
  pred_y = apply(test_x,1,function(m) weighted.mean(train_y,
               exp(-(abs(train_x - m)/bandwidth)^2)))
  test_error = mean((test_y - pred_y)^2,na.rm=TRUE)
  return(list(pred.y = pred_y, test.error = test_error))
}
# X/test_x: A matrix, each row is an observation vector in training/test data
# Y/test_y: A matrix, response variable in training/test data
# nfold: the fold of cross validation
# bandwidth: a sequence of tuning parameters for bandwidth
# Output:  CV.error: A data frame of the sequence of tuning parameters and
#                    the corresponding cross validation accurracy
#          best.bandwidth: The best bandwidth value according to CV.error
#          pred.y: The predict value on test data based on best.bandwidth
#          test.error: The mean squared error of the prediction pred,y
cv.NW_kernel<-function(X,Y,test_x,test_y,nfold,bandwidth){
  # Reorder the data
  random_index = sample(length(X))
  X = X[random_index]; Y = Y[random_index]
  CV_loss_function = rep(NA,length(bandwidth))
  for(i in 1:length(bandwidth)){
    loss_function = rep(NA,nfold); size = floor(length(X)/nfold)
    # Do the cross validation
    for(j in 1:nfold){
      # Split the data into train and validation part
      index = ((j-1)*size+1):(j*size)
      train_x = X[-index]; train_y = Y[-index]
      validation_x = X[index]; validation_y = Y[index]
      KW_solution = NW_kernel(train_x,train_y,validation_x,validation_y, bandwidth[i])
      loss_function[j] = KW_solution$test.error
    }
    CV_loss_function[i] = mean(loss_function)
  }
  CV_error = rbind(bandwidth,CV_loss_function)
  # Choose the best bandwidth
  best_bandwidth = bandwidth[which.min(CV_loss_function)]
  solution = NW_kernel(X,Y,test_x,test_y, best_bandwidth)
  return(list(CV.error = CV_error, pred.y = solution$pred.y,
              test.error = solution$test.error, best.bandwidth = best_bandwidth))
}
```

Here we use a sequence of bandwidth $(0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.5, 1, 1.06\sigma^2 n^{-\frac{1}{5}})$ and we will do 5-fold cross validation on the training data to select the bandwidth.

```
c = 1.06*sd(x)*n^{-1/5}; bandwidth = c(0.001,0.002,0.005,0.01,0.02,0.05,0.1,0.5,1,c)
CV_result = cv.NW_kernel(x,y,test.x,test.y,5,bandwidth)
CV_result$CV.error
```

```
##                        [,1]       [,2]       [,3]       [,4]       [,5]
## bandwidth        0.00100000 0.00200000 0.00500000 0.01000000 0.02000000
## CV_loss_function 0.04207569 0.04072711 0.03977304 0.03941628 0.03935958
##                       [,6]       [,7]      [,8]      [,9]      [,10]
## bandwidth        0.0500000 0.10000000 0.5000000 1.0000000 0.05231421
## CV_loss_function 0.0406723 0.04954102 0.2840723 0.4466424 0.04086645
```

```
CV_result$best.bandwidth
```

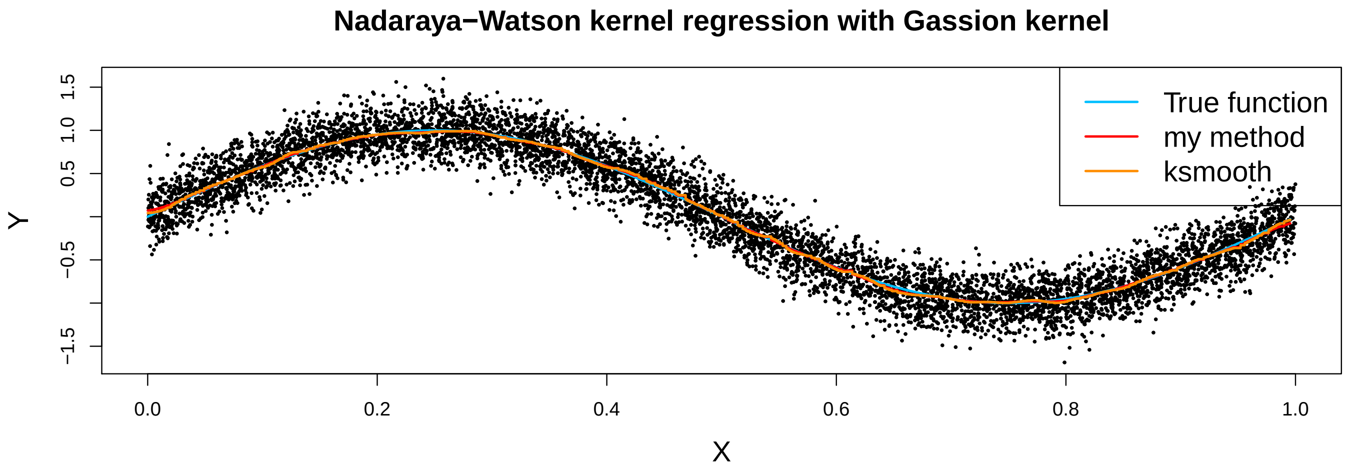
```
## [1] 0.02
```

```
CV_result$test.error
```

```
## [1] 0.0392201
```

Here we see the best bandwidth is 0.02. Then we use Nadaraya-Watson kernel regression with this best bandwidth to fit all the train data again, and the corresponding error(MSE) on test data is 0.0392201. **We also make a plot of the prediction of our method versus the true value, as well as the result from ksmooth package. We can see our model fit the data well and almost the same as the true value and ksmooth.**

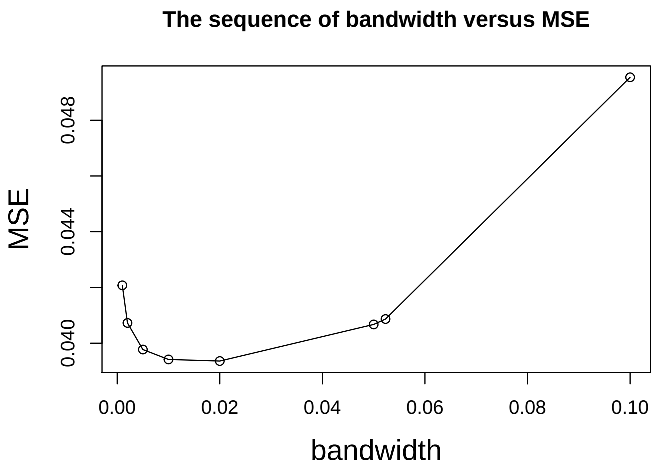
```
# plot the prediction of our method versus the true value and the result from  ksmooth
pred.y = CV_result$pred.y[order(test.x)]
test.x = test.x[order(test.x)]; test.y = test.y[order(test.x)]
ks = ksmooth(x, y, kernel = "normal", bandwidth = CV_result$best.bandwidth, x.points = test.
plot(x, y, xlim = c(0, 1), pch =19, cex = 0.3, xlab = "X", ylab = "Y", cex.lab = 1.5)
title(main=paste("Nadaraya-Watson kernel regression with Gassion kernel"), cex.main = 1.5)
lines(test.x, sin(2*pi*test.x), col = "deepskyblue", lwd = 2)
lines(test.x, pred.y, type = "s", col = "red", lwd = 2)
lines(test.x, ks$y, type = "s", col = "darkorange", lwd = 2)
legend("topright", c("True function","my method", "ksmooth"), col =
         c("deepskyblue","red","darkorange"), lty = 1, cex = 1.5, lwd =2)
```

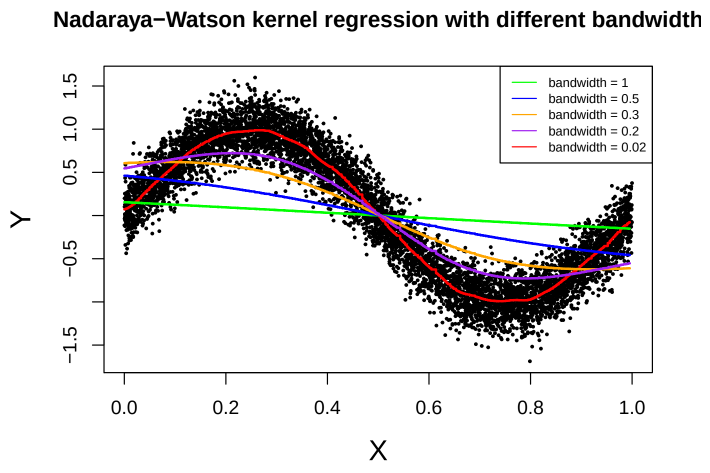


To show the effect of the bandwidth, we plot the cross-validation error versus the different bandwidth(left) and plot the prediction with different bandwidth(right).

```
par(mfrow=c(1,2))
plot(bandwidth[order(bandwidth)][1:8],CV_result$CV.error[2,][order(bandwidth)][1:8],
     xlab = "bandwidth", ylab = "MSE", cex.lab = 1.5,type = "l")
points(bandwidth[order(bandwidth)][1:8],CV_result$CV.error[2,][order(bandwidth)][1:8])
title(main=paste("The sequence of bandwidth versus MSE"))
KW1 = NW_kernel(x,y,test.x,test.y, CV_result$best.bandwidth)
KW2 = NW_kernel(x,y,test.x,test.y, 1); KW3 = NW_kernel(x,y,test.x,test.y, 0.5)
KW4 = NW_kernel(x,y,test.x,test.y, 0.3); KW5 = NW_kernel(x,y,test.x,test.y, 0.2)
plot(x, y, xlim = c(0, 1), pch =19, cex = 0.3, xlab = "X", ylab = "Y", cex.lab = 1.5)
title(main=paste("Nadaraya-Watson kernel regression with different bandwidth"))
lines(test.x, KW1$pred.y, type = "s", col = "red", lwd = 2)
```

```r
lines(test.x, KW2$pred.y, type = "s", col = "green", lwd = 2)
lines(test.x, KW3$pred.y, type = "s", col = "blue", lwd = 2)
lines(test.x, KW4$pred.y, type = "s", col = "orange", lwd = 2)
lines(test.x, KW5$pred.y, type = "s", col = "purple", lwd = 2)
legend("topright", c("bandwidth = 1","bandwidth = 0.5","bandwidth = 0.3","bandwidth = 0.2",
  "bandwidth = 0.02"), col = c("green","blue","orange","purple","red"), cex = 0.7, lty = 1)
```





According to the plot and the previous result, we think our code is correct and bandwidth plays an important role on the performance of the model. Finally, we show the running time of our code on 7200 training data and 1000 test data, which seems fast enough.

```r
system.time(NW_kernel(x,y,test.x,test.y, CV_result$best.bandwidth))
```

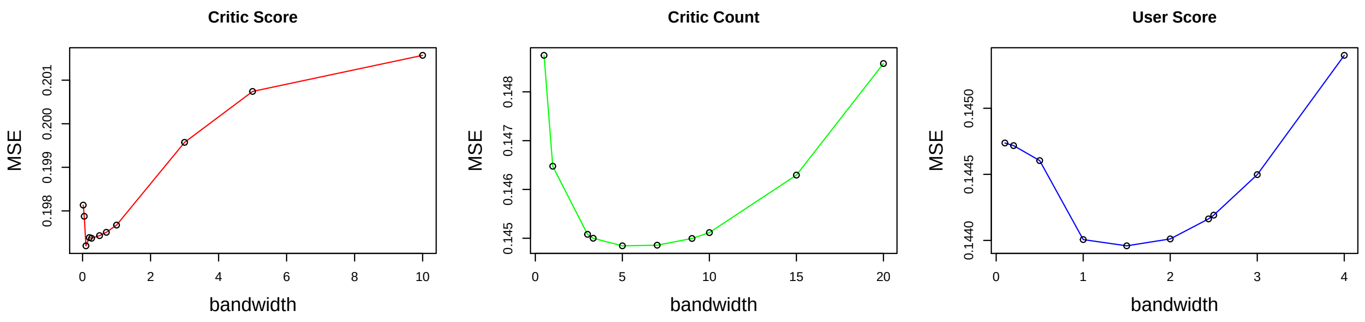
```
##    user  system elapsed
##    1.75    0.00    1.84
```

b) [15 points]

Answer:

We write a function `NW_data_preprocess` to split the train and test data, then perform 10-fold cross-validation to find the best bandwidth, finally return the result including the MSE on test data.

```r
# Read the data and extract the attribute
set.seed(0)
data = read.csv("Video_Games_Sales_as_at_22_Dec_2016.csv")
y = log(1 + data$Global_Sales);
Critic_Score = cbind(data$Critic_Score,y);
Critic_Count = cbind(data$Critic_Count,y);
User_Score = as.array(as.matrix(data$User_Score)); User_Score[User_Score == 'tbd'] = NA;
User_Score = cbind(as.numeric(User_Score),y)
# df: The dataframe of all data, including the response variable
# bandwidth: a sequence of tuning parameters for bandwidth
# Output: CV.error: A data frame of the sequence of tuning parameters and
#                       the corresponding cross validation accurracy
#           best.bandwidth: The best bandwidth value according to CV.error
#           pred.y: The predict value on test data based on best.bandwidth
#           test.error: The mean squared error of the prediction pred,y
NW_data_preprocess<-function(df, bandwidth){
  # Deal with the NA and Reorder the data
  df = na.omit(df); n = dim(df)[1]; random_index = sample(n)
  # Split into 80% train and 20% test part
  size = floor(n*0.8)
  train = df[random_index,][1:size,]; test = df[random_index,][(size+1):dim(df)[1],]
  train_x = matrix(train[,1]); train_y = matrix(train[,2]);
  test_x = matrix(test[,1]); test_y = matrix(test[,2]);
  # We add a auto bandwidth from (Silverman 1986) to the sequence of bandwidth
  bandwidth = c(1.06*sd(df[,1])*n^{-1/5},bandwidth)
  # Do the cross-validation and fit model with all the train data
  CV_result = cv.NW_kernel(train_x,train_y,test_x,test_y,10,bandwidth)
  return(CV_result)
}
# Set the bandwidth sequence
bandwidth_CS = c(0.02,0.05,0.1,0.2,0.5,0.7,1,3,5,10)
bandwidth_CC = c(0.5,1,3,5,7,9,10,15,20)
bandwidth_US = c(0.1,0.2,0.5,1,1.5,2,2.5,3,4)
# Tune the bandwidth using 10-fold cross-validation and fit the model
NW_CS = NW_data_preprocess(User_Score,bandwidth_CS)
NW_CC = NW_data_preprocess(Critic_Count,bandwidth_CC)
NW_US = NW_data_preprocess(Critic_Score,bandwidth_US)
par(mfrow=c(1,3))
plot(NW_CS$CV.error[1,][order(NW_CS$CV.error[1,])],NW_CS$CV.error[2,][order(NW_CS$CV.error[1
     xlab = "bandwidth", ylab = "MSE", cex.lab = 1.5,type = "l",col = "red",main = "Critic S
points(NW_CS$CV.error[1,][order(NW_CS$CV.error[1,])],NW_CS$CV.error[2,][order(NW_CS$CV.error
plot(NW_CC$CV.error[1,][order(NW_CC$CV.error[1,])],NW_CC$CV.error[2,][order(NW_CC$CV.error[1
     xlab = "bandwidth", ylab = "MSE", cex.lab = 1.5,type = "l",col="green",main = "Critic C
points(NW_CC$CV.error[1,][order(NW_CC$CV.error[1,])],NW_CC$CV.error[2,][order(NW_CC$CV.error
plot(NW_US$CV.error[1,][order(NW_US$CV.error[1,])],NW_US$CV.error[2,][order(NW_US$CV.error[1
     xlab = "bandwidth", ylab = "MSE", cex.lab = 1.5,type = "l",col = "blue",main = "User Sc
points(NW_US$CV.error[1,][order(NW_US$CV.error[1,])],NW_US$CV.error[2,][order(NW_US$CV.error
```



According to the plots, we think we have tuned the bandwidth well. The following is the MSE on test data.

```r
result = rbind(c(NW_CS$best.bandwidth,NW_CS$test.error),c(NW_CC$best.bandwidth,
                 NW_CC$test.error),c(NW_US$best.bandwidth,NW_US$test.error))
rownames(result) = c("Critic_Score","Critic_Count","User_Score")
colnames(result) = c("best.bandwidth","MSE")
result
```

```
##              best.bandwidth       MSE
## Critic_Score            0.1 0.1813225
## Critic_Count            5.0 0.1604420
## User_Score              1.5 0.1518890
```

Conclusion: Accoding to the MSE of each model, we think User Score gives the best model with test error = 0.1518890.

Question 2

a) [20 points]

Answer:

To estimate the degree of freedom for each tree, we use the formula:

$$df(\hat{f}) = \frac{1}{\sigma^2}\sum_{i=1}^{n}\text{Cov}(\hat{y}_i, y_i).$$

To estimating $\text{Cov}(\hat{y}_i, y_i)$, we fix X and do 20 times simulation.(Generate Y, fit the model, and predict $\hat{Y}$. Then use the sample covariance to estimate the degree of freedom.

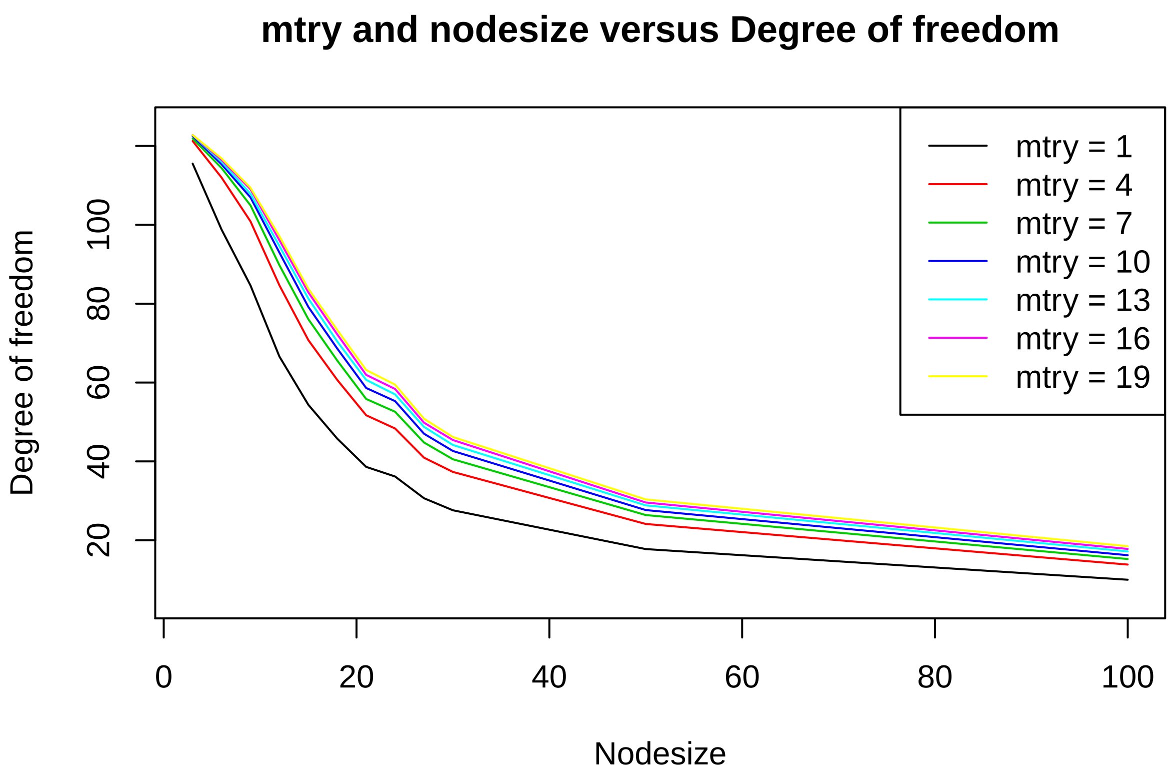
```r
library(MASS); library(randomForest)
# Set the sedd, number of observation and dimension
set.seed(0); P = 20; N = 200
# Function generate_data: Generate the data, input the number of observation N,
# dimension P, and randam seed, return the data X and the response variable Y
# with standard normal errors.
generate_data<-function(N,P,seed_x,seed_y){
  I = diag(nrow = P)
  set.seed(seed_x); X = as.matrix(mvrnorm(N, mu=rep(0,P), Sigma=I))
  set.seed(seed_y); Y = 1 + 0.5 * (X[,1] + X[,2] + X[,3] + X[,4]) +  rnorm(N)
  return(list(X = X, Y = Y))
}
# N: The number of observation
# P: The dimension of the data
# mtry: A seq of mtry parameters to estimate degree of freedom
# nodesize: A seq of nodesize parameters to estimate degree of freedom
# iter: The number of simulations we will perform
# Output: A matrix, the row name is the nodesize, the column name
#         is the mtry, and the value is the estimation of Dof
```

```r
DoF_RF_mtry_nodesize<-function(N,P,mtry,nodesize,iter){
  mtry_n = length(mtry); nodesize_n = length(nodesize)
  result = matrix(NA,nodesize_n,mtry_n)
  rownames(result) = nodesize; colnames(result) = mtry
  for(i in 1:nodesize_n){
    for(j in 1:mtry_n){
      Y.pred = NULL; Y.ture = NULL
      for(m in 1:iter){
        data = generate_data(N,P,0,m); X = data$X; Y = data$Y
        rf.fit = randomForest(X, Y, mtry = mtry[j], nodesize = nodesize[i])
        Y.ture = cbind(Y.ture,Y); Y.pred = cbind(Y.pred, predict(rf.fit, X))
      }# Calculate the degree of freedom
      result[i,j] = sum(sapply(1:N, function(x) cov(Y.ture[x,],Y.pred[x,])))
    }
  }
  return(result)
}
mtry = seq(1,19,3); nodesize =c(seq(3,30,3),50,100)
mtry_nodesize_result = DoF_RF_mtry_nodesize(N,P,mtry,nodesize,20)
mtry_nodesize_result
```

```
##             1         4         7        10        13        16        19
## 3   115.51934 121.25021 121.88871 122.50217 122.77360 122.76103 122.75918
## 6    98.79494 111.98541 114.39870 115.34318 115.96799 116.46944 116.84956
## 9    84.61932 100.86155 104.93131 106.94828 107.86491 109.00974 109.39100
## 12   66.60074  84.65600  89.75726  92.95979  94.64055  96.10129  97.10857
## 15   54.36219  70.76812  76.07349  79.18427  81.30993  82.87040  83.79552
## 18   45.77555  60.67018  65.57971  68.61897  70.42426  72.22953  73.34406
## 21   38.60715  51.70024  55.83294  58.61812  60.67903  61.94778  63.17651
## 24   36.17059  48.34493  52.58692  55.28993  56.99306  58.37114  59.46388
## 27   30.65929  40.94482  44.79441  47.02051  48.75816  49.81399  50.80682
## 30   27.60301  37.33243  40.54553  42.64159  44.16493  45.41675  46.18183
## 50   17.76732  24.13320  26.39450  27.65740  28.85019  29.57759  30.35121
## 100   9.98839  13.83514  15.24167  16.20943  17.17651  17.80109  18.48164
```

In the matrix, the row name is the nodesize, the column name is the mtry, and the value is the estimation of DOF. According to the matrix, we make a plot to summary the relation between Degree of freedom and mtry, nodesize. **We find that when the nodesize parameter increases, the DOF of Random Forest decreses. And when the mty parameter increases, the DOF of Random Forest increases.**

```r
plot(x = NULL, y= NULL,xlim=c(3,100),ylim=c(5,125),xlab = "Nodesize",
     ylab = "Degree of freedom")
for(i in 1:dim(mtry_nodesize_result)[2]){
  lines(rownames(mtry_nodesize_result),mtry_nodesize_result[,i],type='l', col = i)
}
legend("topright", c("mtry = 1","mtry = 4","mtry = 7","mtry = 10", "mtry = 13",
       "mtry = 16","mtry = 19"), col = 1:7, cex = 1, lty = 1)
title(main = "mtry and nodesize versus Degree of freedom")
```



b) [15 points]

Answer:

To estimate the variance of this estimator, we use the formula:

$$\frac{1}{n}\sum_{i}^{n} E_{\hat{f}}(\hat{f}(x_i) - E[\hat{f}(x_i)])^2$$

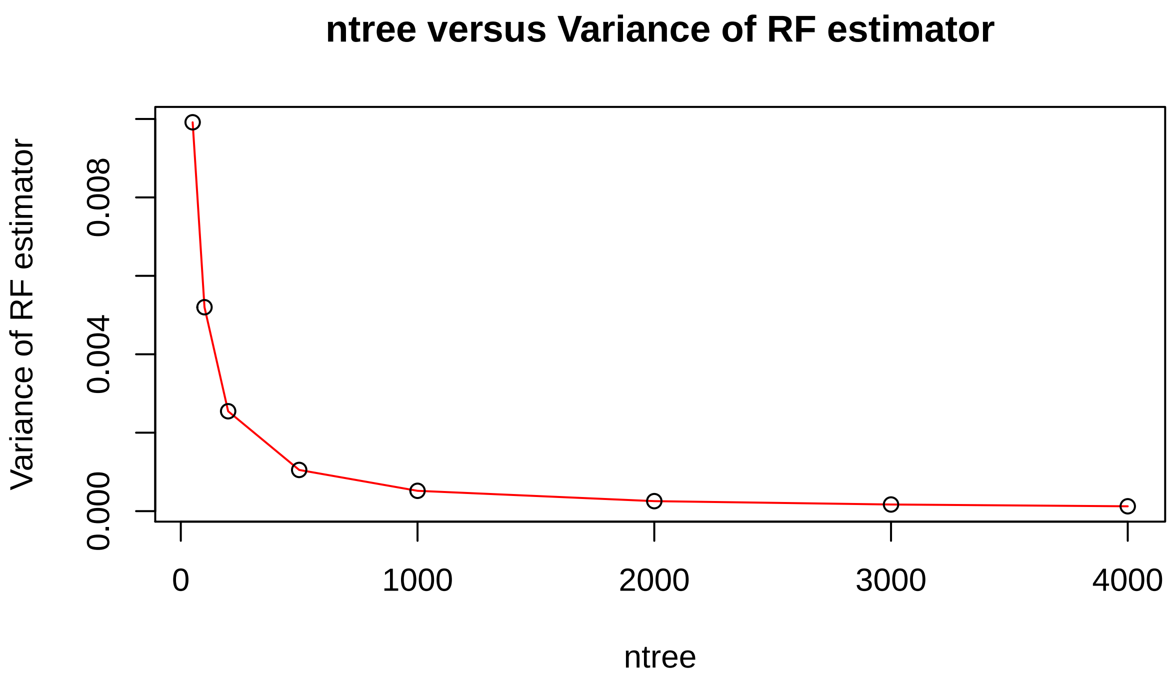
To estimating $E_{\hat{f}}(\hat{f}(x_i) - E[\hat{f}(x_i)])^2$, we fix X and do 20 times simulation.(Generate Y, fit the model, and predict $\hat{Y}$. Then use the predict value to estimate the variance of this estimator.

```
# N: The number of observation
# P: The dimension of the data
# ntree: A seq of ntree parameters to estimate degree of freedom
# iter: The number of simulations we will perform
# Output: A matrix of the ntree parameters and the corresponding degree of freedom
Var_RF_ntree<-function(N,P,ntree,iter){
  ntree_n = length(ntree); var = rep(NA,ntree_n)
  for(i in 1:ntree_n){
    Y.pred = NULL; Y.ture = NULL
    for(m in 1:iter){
      data = generate_data(N,P,0,0)
      X = data$X; Y = data$Y; set.seed(m)
      rf.fit = randomForest(X, Y, ntree = ntree[i])
      Y.ture = cbind(Y.ture,Y); Y.pred = cbind(Y.pred, predict(rf.fit, X))
    }# Calculte the variance
    var[i] = sum(sapply(1:N, function(x) mean((Y.pred[x,] - mean(Y.pred[x,]))^2))) / N
  }
  result = rbind(ntree,var); return(result)
}
ntree = c(5,10,50,100,200,500,1000,2000,3000,4000)
ntree_result = Var_RF_ntree(N,P,ntree,20)
ntree_result
```

```
##              [,1]        [,2]         [,3]         [,4]         [,5]
## ntree 5.00000000 10.00000000 50.000000000 1.000000e+02 2.000000e+02
## var   0.09791961  0.04977433  0.009915496 5.199607e-03 2.547287e-03
##               [,6]         [,7]         [,8]         [,9]        [,10]
## ntree 5.000000e+02 1.000000e+03 2.000000e+03 3.000000e+03 4.000000e+03
## var   1.050329e-03 5.177913e-04 2.545992e-04 1.690471e-04 1.235088e-04
```

According to the matrix, we make a plot to summary the relation between the variance of this estimator and ntree. **We find that when the ntree parameter increases, the variance of this estimator decreases. We can shrink the estimator's variance using ntree parameter.**

```
plot(x = ntree_result[1,][3:10], y= ntree_result[2,][3:10],xlab = "ntree",
     ylab = "Variance of RF estimator",col = "red", type = 'l')
points(ntree_result[1,][3:10], ntree_result[2,][3:10])
title(main = "ntree versus Variance of RF estimator")
```



Question 3

a) [15 points]

Answer:

```r
# X: Observation(one dimension)
# Y: Response variable
# W: Weight of each observation
# Output: return a stump.model object with the following value.
# cut_point: The cutting point c of this stump model
# left_sign: Left node predictions(The prediction when x <= cut_point)
# right_sign: Right node predictions(The prediction when x > cut_point)
CART_stump<-function(X,Y,W){
  # Calculate the weighted reduction of Gini impurity
  # x: Observation(one dimension)
  # y: Response variable
  # w: Weight of each observation
  # cut_point: cut point we use in the model
  # Output:
  # score: the weighted reduction of Gini impurity
  # left_sign: Left node predictions
  # right_sign: Right node predictions
  cal_score<-function(x,y,weight,cut_point){
    # split data using cut_point
    left = (x <= cut_point); right = (x > cut_point)
    left_y = y[left]; right_y = y[right]
    left_weight = weight[left]; right_weight = weight[right]
    left_p = weighted.mean((left_y == 1),left_weight)
    right_p = weighted.mean((right_y == 1),right_weight)
    left_gini = left_p * (1-left_p)
    right_gini = right_p * (1-right_p)
    # Calculate score
    score = -(sum(left_weight) * left_gini)/sum(weight) -
      (sum(right_weight) * right_gini)/sum(weight)
    # Calculate the sign in each child node
    # If the number of +1 and -1 are the same, we define this prediction as -1
    left_sign = ifelse(sum(left_weight*left_y)>0,1,-1)
    right_sign = ifelse(sum(right_weight*right_y)>0,1,-1)
    return(list(score = score,left_sign = left_sign,right_sign = right_sign))
  }
  # Get the cut points sequence
  split_list = unique(X)
  result = matrix(NA,length(split_list),4)
  # Claculte the weighted reduction of Gini impurity of each cut point
  for(i in 1:length(split_list)){
    split_result = cal_score(X,Y,W,split_list[i])
    result[i,1] = split_list[i]; result[i,2] = split_result$score
    result[i,3] = split_result$left_sign; result[i,4] = split_result$right_sign
  }
  # Choose the point with the maxiumum score as the best cut point
  index = which.max(result[,2])
  result = list(cut_point = result[index,1],left_sign = result[index,3],
                right_sign = result[index,4])
  class(result) <- "stump.model"
  return(result)
}
```

Here we create a small sample data to test our function.

```r
x <- c(1,2,3,4,5,6,7,8,9,10)
y <- c(1,-1,1,-1,-1,1,1,1,1,1)
w <- rep(1/length(x),length(x))
CART_stump(x,y,w)
```

```
## $cut_point
## [1] 5
##
## $left_sign
## [1] -1
##
## $right_sign
## [1] 1
##
## attr(,"class")
## [1] "stump.model"
```

It means when $X <= 5$, $Y = -1$, and when $X > 5$, $Y = 1$. **According to the result, we think our code is correct.**

b) [20 points]

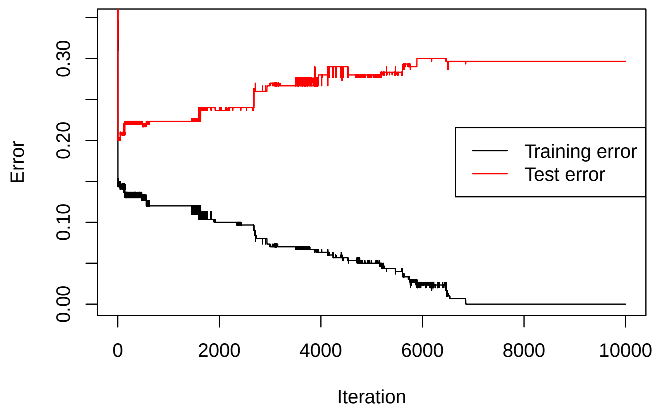
Answer:

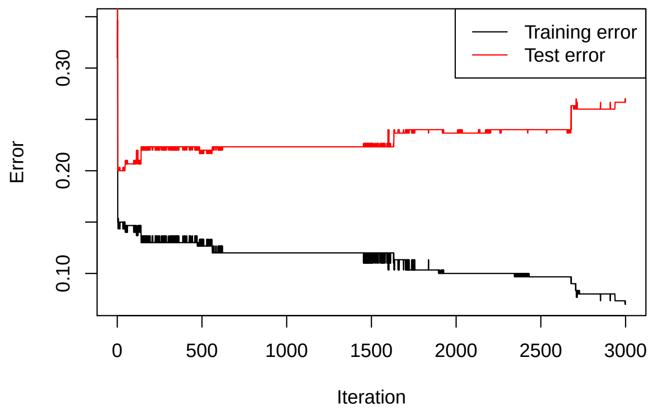
```r
# To make prediction on data X with stump model
# X: Obsearvation data
# model: stump.model object, a stump model.
# Output: pred.y: The prediction value
stump.predict<-function(X,model){
  pred.y = rep(NA,length(X))
  pred.y[X <= model$cut_point] = model$left_sign
  pred.y[X > model$cut_point] = model$right_sign
  return(pred.y)
}
```

```r
# Fit the adaboost model using the stump as the base learner.
# X: The observation data
# Y: The response variable
# iteration: The iteration of Adaboost algoright (the number of base learner)
# Output: A adaboost.stump.model object with the following value
# iteration: The number of base learner in the final adaboost model
# model: A list contain the base learners
# alpha: The weight of base learners in the adaboost model
# epsilon: The error of each base learner
Adaboost_stump<- function(X,Y,iteration = 500){
  weight = rep(1/length(X),length(X))
  epsilon = rep(NA,iteration)
  alpha = rep(NA,iteration)
  model = list()
  # Do the Adaboost
  for(i in 1:iteration){
    # Fit the base learner
    model[[i]] = CART_stump(X,Y,weight)
    pred.y = stump.predict(X,model[[i]])
    epsilon[i] = sum(weight * (Y != pred.y))
    # If error >= 0.5, reverse the model
    if(epsilon[i] >= 0.5){
      model[[i]]$left_sign = model[[i]]$left_sign * -1
      model[[i]]$right_sign = model[[i]]$right_sign * -1
      pred.y = stump.predict(X,model[[i]])
      epsilon[i] = sum(weight * (Y != pred.y))
    }
    # Calculate the alpha
    alpha[i] = 1/2 * log((1-epsilon[i])/max(epsilon[i],1e-10))
    # Update the weight
    w = weight * exp(-alpha[i] * Y * pred.y)
    weight = w / sum(w)
  }
  result = list(iteration = iteration, model = model, alpha = alpha, epsilon = epsilon)
  class(result) <- "adaboost.stump.model"
  return(result)
}

# To make prediction on data X with adaboost model
# X: Obsearvation data
# Y: Response variable
# model: A adaboost.stump.model object, a adaboost model.
# Output:
# pred.y: The final prediction value
# pred.error: The error of the final prediction value
# error.list: The error sequence with the iteration increasing
Adaboost.stump.predict<-function(X,Y,model){
  pred.y = rep(0,length(Y))
  error_list = rep(NA,model$iteration)
  for(i in 1:model$iteration){
    yhat = stump.predict(X,model$model[[i]])
    pred.y = pred.y + yhat * model$alpha[i]
    predict = ifelse(pred.y > 0, 1, -1)
    error_list[i] = sum(predict != Y) / length(Y)
  }
  pred.y = ifelse(pred.y > 0, 1, -1)
  error = sum(pred.y != Y) / length(Y)
  return(list(pred.y = pred.y, pred.error = error, error.list = error_list))
}
```

Here we generate a sample data to test our code and the algorithm.

```
# Generate the data
set.seed(0)
n = 300
x = runif(n)
y = (rbinom(n,1,(sin (4*pi*x)+1)/2)-0.5)*2
test.x = runif(n)
test.y = (rbinom(n,1,(sin (4*pi*test.x)+1)/2)-0.5)*2
w <- rep(1/length(x),length(x))

# Fit a Adaboost model with 10000 base learners.
adaboost.model = Adaboost_stump(x,y,10000)
train_predict = Adaboost.stump.predict(x,y,adaboost.model)
test_predict = Adaboost.stump.predict(test.x,test.y,adaboost.model)
par(mfrow=c(1,2))
plot(train_predict$error.list,type = 'l',xlab="Iteration",ylab = "Error")
lines(test_predict$error.list,col='red')
legend("right", c("Training error","Test error"), col = c("black","red"),
       cex = 1, lty = 1)
plot(train_predict$error.list[1:3000],type = 'l',xlab="Iteration",ylab = "Error")
lines(test_predict$error.list[1:3000],col='red')
legend("topright", c("Training error","Test error"), col = c("black","red"),
       cex = 1, lty = 1)
```





From the left plot, we can see the training error tends to decrease with the increasing of iteration, and if the iteration is large enough, the training error tends to zero. It validates that the training error of AdaBoost decreases the upper bound exponentially. According to the result, we think the code is correct. **Moreover, in the left plot we find that with the iteration increasing, the training error decreases, but the test error decreases first and then we observe an increasing trend, that means the model is not improving anymore and it is overfitting.**

And in the right plot, we foucus on the iteration between 1 to 2000, and we think the testing error start to go up already after just a few hundred iterations. **According to the result, we can say that the Adaboost algorithm will cause overfitting and it is important to choose a reasonable iteration number.**