STAT542 HW4

Name: Chun Yin Ricky Chue (chue2@illinois.edu)

Question 1.

Part a.

We design our own code for a Nadaraya-Watson kernel regression estimator for a 1D problem. We consider just for the Gaussian kernel function. We firstly generate an example with 10,000 data points, uniformly distributed across $(0, 2\pi)$. The corresponding response $y = 2 \sin x + \epsilon$, where ϵ is standard normal error.

```r
set.seed(1)
x <- runif(10000, 0, 2*pi)
y <- 2*sin(x) + rnorm(length(x))
```

We define a function for our own Nadaraya-Watson kernel regression estimator with Gaussian kernel function for 1D. Inputs are the testing points and bandwidth of kernel. The 0.37 factor is to match up with the result from `ksmooth`, which defines `bandwidth` in an alternative way.

```r
fitloc1 = function(x0, bandwidth, X, Y) {
  w <- dnorm((X-x0), sd = 0.3706506 * bandwidth)
  ym <- weighted.mean(Y, w)
  return(ym)
}
```

We then generate some testing data points, and predict their y's given the kernel regression.

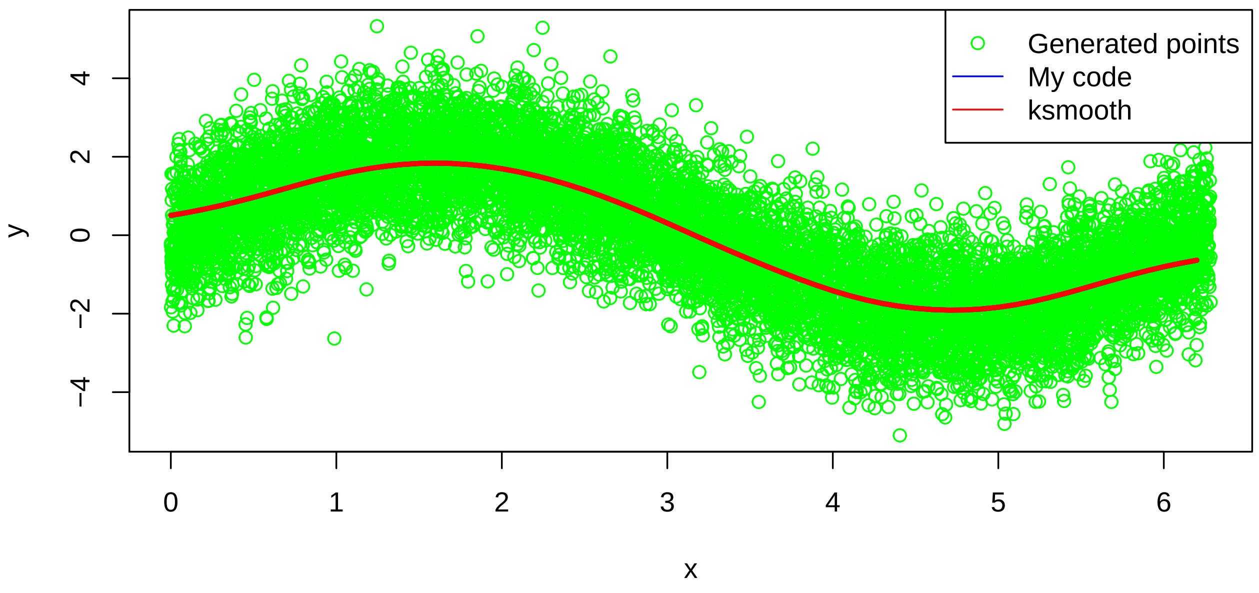
```r
test_x = seq(0, 2*pi, by=0.1)
fit_bd = 1   # Fitting bandwidth
fit_y = mapply(fitloc1, test_x,  fit_bd, list(x), list(y))
```

We compare the `ksmooth` package to check the validity of our own code.

```r
kersmooth <- ksmooth(x, y, kernel = "normal", bandwidth = fit_bd, x.points = test_x)
```

We plot the results generated by the two methods. We choose `bandwidth` $= 0.5$, which fits the data points just right, and it has a sinusoidal form. For illustrative purpose, we show that the two methods are identical. At the boundaries the estimates are biased, because we are implementing an averaged kernel estimator, so the training data used at the boundaries are biased at one direction.

```r
plot(x, y, col = "green")
lines(test_x, fit_y, col = "blue", lwd = 3)
lines(kersmooth$x, kersmooth$y, col = "red", lwd = 3)
legend("topright", bg = "white", c("Generated points","My code","ksmooth"),
       col=c("green", "blue", "red"), pch=c(1, NA, NA), lty = c(NA, 1, 1))
```



Part b.

We first define a function for $\log(1 + X)$ transformation.

```r
logtrans <- function(X) {
  return(log(1 + X))
}
```

We test our Kernel regression code with a Kaggle dataset “Video Game Sales with Ratings”. We perform estimation on `log(1 + Global_Sales)` using each of the three scores: `Critic_Score`, `Critic_Count` and `User_Score`. We replace the values that are blanked and `tbd` by `NA`.

```r
Game_df <- read.csv("Video_Games_Sales_as_at_22_Dec_2016.csv", na.strings=c("", "tbd", "NA"),
                    header = TRUE, sep = ",")[c("Global_Sales","Critic_Score",
                                                "Critic_Count","User_Score")]
Game_df <- na.omit(Game_df)  # Eliminate the entries with missing values.
Game_df["Global_Sales"] <- lapply(Game_df["Global_Sales"], logtrans) # Log transformation.
```

We divide the dataset into 10-folds for cross-validation (CV).

```r
set.seed(30)
n <- dim(Game_df)[1]
K <- 10
bw_par <- c(0.1,0.2,0.4,0.8,1,2,4,8,10,12,14,18,20,25,30,35,40)
bw_len <- length(bw_par)
Game_df <- Game_df[sample(n),]    # Shuffle the data
Game_df["CVfold"] <- cut(seq(1,n), breaks = K, labels = FALSE) # Create 10 equally sized folds
# Divide the three estimating features and responses into three distinct dataframes.
Game_df_CS <- Game_df[c("Global_Sales","Critic_Score")]
Game_df_CC <- Game_df[c("Global_Sales","Critic_Count")]
Game_df_US <- Game_df[c("Global_Sales","User_Score")]

CStest_err <- matrix(0., nrow = bw_len, ncol = K)   # A matrix to store the test errors
CCtest_err <- matrix(0., nrow = bw_len, ncol = K)   # A matrix to store the test errors
UStest_err <- matrix(0., nrow = bw_len, ncol = K)   # A matrix to store the test errors
```

We perform 10-fold CV on the dataset to choose the optimal bandwidth. We store up the mean squared prediction error (MSPE) for each CV fold and bandwidth.

```r
for (j in 1:bw_len) {  #bw_len
  print(j)
  for (i in 1:K) {
    train_ind <- which(Game_df["CVfold"] != i)
    test_ind <- which(Game_df["CVfold"] == i)
    # Critical Score prediction
    CS_train <- data.matrix(Game_df_CS["Critic_Score"])[train_ind,]
    CS_test <- data.matrix(Game_df_CS["Critic_Score"])[test_ind,]
    # Critical Count prediction
    CC_train <- data.matrix(Game_df_CC["Critic_Count"])[train_ind,]
    CC_test <- data.matrix(Game_df_CC["Critic_Count"])[test_ind,]
    # User Score prediction
    US_train <- data.matrix(Game_df_US["User_Score"])[train_ind,]
    US_test <- data.matrix(Game_df_US["User_Score"])[test_ind,]
    # Responses
    y_train <- data.matrix(Game_df_CS["Global_Sales"])[train_ind,]
    y_test <- data.matrix(Game_df_CS["Global_Sales"])[test_ind,]

    # Number of testing data.
    n_test <- length(y_test)

    fit_CS_y = mapply(fitloc1, CS_test, bw_par[j], list(CS_train), list(y_train))
    fit_CC_y = mapply(fitloc1, CC_test, bw_par[j], list(CC_train), list(y_train))
    fit_US_y = mapply(fitloc1, US_test, bw_par[j], list(US_train), list(y_train))

    # Mean squared prediction error.
    CStest_err[j,i] <- mean((fit_CS_y - y_test)^2)
    CCtest_err[j,i] <- mean((fit_CC_y - y_test)^2)
    UStest_err[j,i] <- mean((fit_US_y - y_test)^2)
  }
}
```

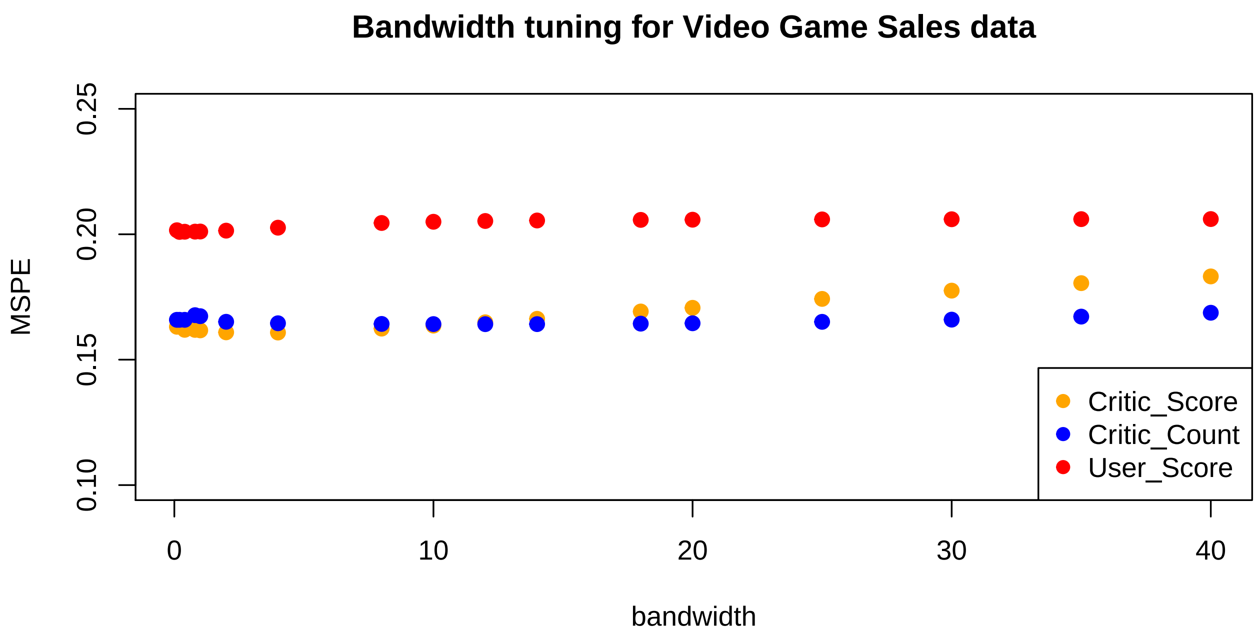
We average the MSPE in all CV folds.

```r
CS_mspemean <- rowMeans(CStest_err, na.rm = TRUE)
CC_mspemean <- rowMeans(CCtest_err, na.rm = TRUE)
US_mspemean <- rowMeans(UStest_err, na.rm = TRUE)
print(c(min(CS_mspemean),min(CC_mspemean),min(US_mspemean)))
```

```
## [1] 0.1608248 0.1641593 0.2009402
```

We visualize how bandwidth tuning changes the prediction accuracy of `Global_Sales`. Among the three scores, `Critic_Score` is doing the best job (MSPE = 0.16), and the best tuning model is `bandwidth` = 4.

```r
plot(x = bw_par, y = CS_mspemean, pch = 19, lwd = 2, col = "orange", xlab = "bandwidth",
     ylab = "MSPE", ylim = c(0.10,0.25))
points(bw_par, CC_mspemean, pch = 19, col = "blue", lwd = 2)
points(bw_par, US_mspemean, pch = 19, col = "red", lwd = 2)
title("Bandwidth tuning for Video Game Sales data")
legend("bottomright", bg = "white", c("Critic_Score","Critic_Count","User_Score"),
       col=c("orange","blue", "red"), pch=c(19, 19, 19))
```



Question 2.

Part a.

We generate a dataset with independent standard normal distribution. The true model is $f(X) = 1 + 0.5\sum_{j=1}^{4} X_j + \epsilon$, where ϵ is standard norm error.

```r
library(randomForest)
set.seed(1)
n <- 200
p <- 20
X <- matrix(rnorm(n*p, mean = 0, sd = 1), n, p)
# Model without error noise. Just to avoid doing it iteratively in the loop.
y_model <- 1 + 0.5 * (X[,1] + X[,2] + X[,3] + X[,4])
```

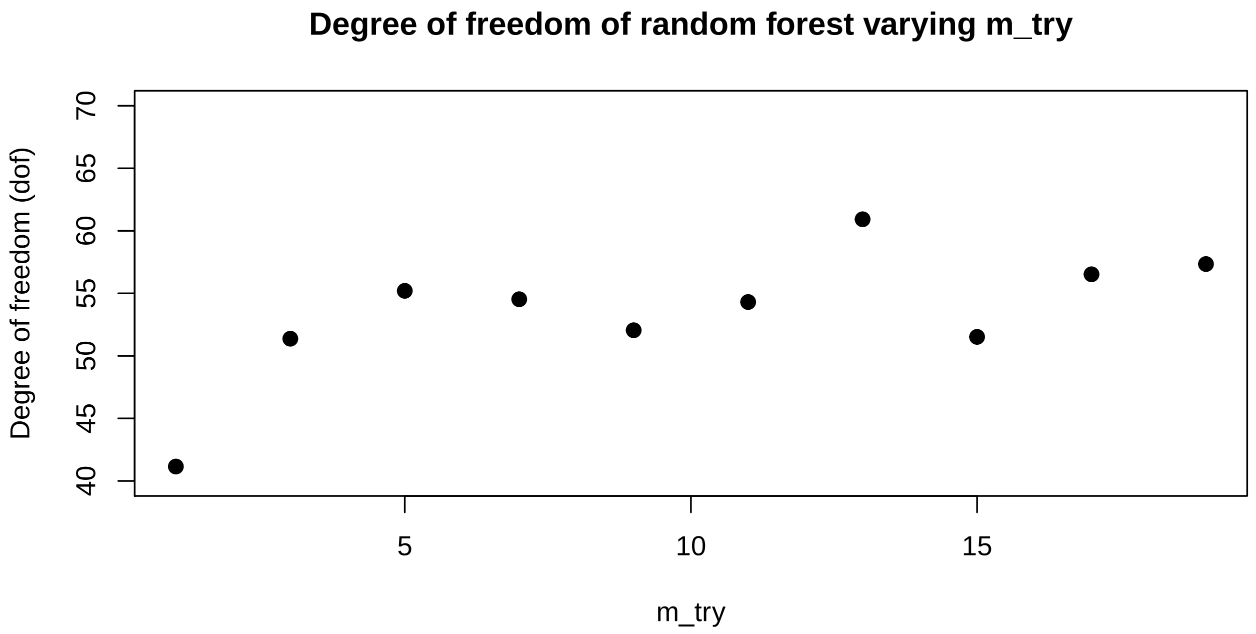
We then leverage the `randomForest` algorithm to fit the dataset, and estimate the dof for the Random Forest algorithm. We evaluate the dof using $\sum_{j=1}^{n} \mathrm{Cov}(\widehat{\mathrm{y}}_\mathrm{i}, \mathrm{y}_\mathrm{i})/\sigma^2$. y_i and $\widehat{y_i}$ are the true and model predicted responses respectively. As the error term ϵ is i.i.d., $\sigma^2 = 1$. The dof is averaged over 20 iterations for each `m_try` value.

We fix the default `ntree` value (500), and tune `mtry` and `nodesize` to see how dof vary with these two parameters. We first vary against `mtry`.

```r
set.seed(Sys.time())
n_iter <- 10   # Number of simulations. 10
mtry_arr <- seq(1,p,2)     # The array for different m_try
mtry_arrlen <- length(mtry_arr)
mtry_dof <- matrix(0, mtry_arrlen, 1)
for (k in 1:mtry_arrlen) {
  print(k)
  # Matrices to store the predicted and true responses.
  y_true <- matrix(0, n_iter, n)
  y_pred <- matrix(0, n_iter, n)
  n_cov <- matrix(0, n_iter, 1)
  for (i in 1:n_iter) {
    y_true[i,] <- y_model + rnorm(n, mean = 0, sd = 1)
    rf.fit = randomForest(X, y_true[i,], mtry = mtry_arr[k], nodesize = 5, importance = TRUE)
    y_pred[i,] <- rf.fit$predicted    # Prediction of y given X by random forest.
    n_cov[i,] <- sum(diag(cov(y_true,y_pred)))  # This covariance is an unbiased estimate.
  }
  mtry_dof[k,] <- mean(n_cov)
}
```

We have the estimate of dof at each `m_try`, we visualize the result as follows.

```
plot(x = mtry_arr, y = mtry_dof, pch = 19, lwd = 2, ylim = c(40,70),
     xlab = "m_try", ylab = "Degree of freedom (dof)")
title("Degree of freedom of random forest varying m_try")
```



Dof increases as `m_try` increases from 1 to about half the number of features (~10). It is because there are more combinations of features that could be chosen from random forest as `m_try` increases in this range. However, it plateaus at larger `m_try`, since now the the number of combination of choices decreases again, and hence the dof decreases.

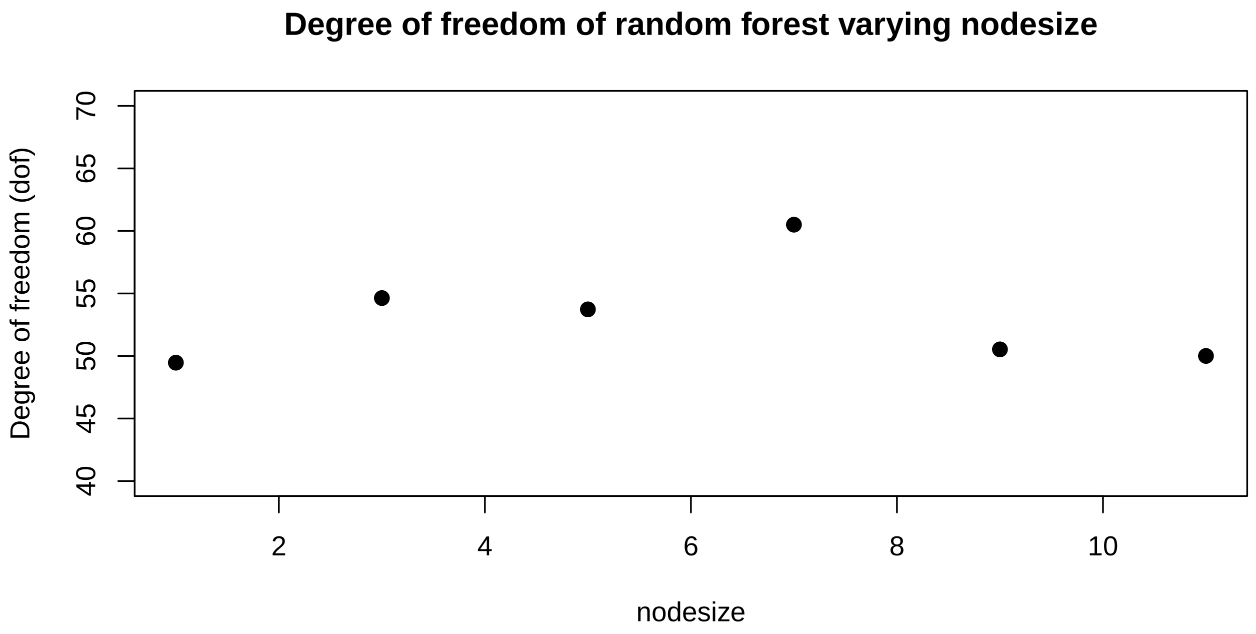
We now vary `nodesize`, we fix the value of `mtry` at the recommended value for regression, i.e. $p/3$, in this case we set `m_try` $= 7$.

```
set.seed(Sys.time())
n_iter <- 10   # Number of simulations. 10
nodesize_arr <- seq(1,12,2)    # The array for different m_try
nodesize_arrlen <- length(nodesize_arr)
nodesize_dof <- matrix(0, nodesize_arrlen, 1)
for (k in 1:nodesize_arrlen) {
  print(k)
  # Matrices to store the predicted and true responses.
  y_true <- matrix(0, n_iter, n)
  y_pred <- matrix(0, n_iter, n)
  n_cov <- matrix(0, n_iter, 1)
```

```r
  for (i in 1:n_iter) {
    y_true[i,] <- y_model + rnorm(n, mean = 0, sd = 1)
    rf.fit = randomForest(X, y_true[i,], mtry = 7, nodesize = nodesize_arr[k])
    y_pred[i,] <- rf.fit$predicted    # Prediction of y given X by random forest.
    n_cov[i,] <- sum(diag(cov(y_true,y_pred)))  # This covariance is an unbiased estimate.
  }
  nodesize_dof[k,] <- mean(n_cov)
}
```

We illustrate how dof varies with `nodesize`

```r
plot(x = nodesize_arr, y = nodesize_dof, pch = 19, lwd = 2, ylim = c(40,70),
     xlab = "nodesize", ylab = "Degree of freedom (dof)")
title("Degree of freedom of random forest varying nodesize")
```



The dof does not seem to vary much with `nodesize` at fixed `m_try`, when `nodesize` is increased from 1 to 11.

Part b.

In this part, we fix `m_try` and `nodesize` as the default value, and tune `ntree`. We compare the variance of the estimator $\sum_{i=1}^{n}\left(\widehat{f}(x_i) - E\left[\widehat{f}(x_i)\right]\right)^2$ for different value of `ntree`. As a note, we have already defined `y_model` in part (a), which is $E\left[\widehat{f}(x_i)\right]$.

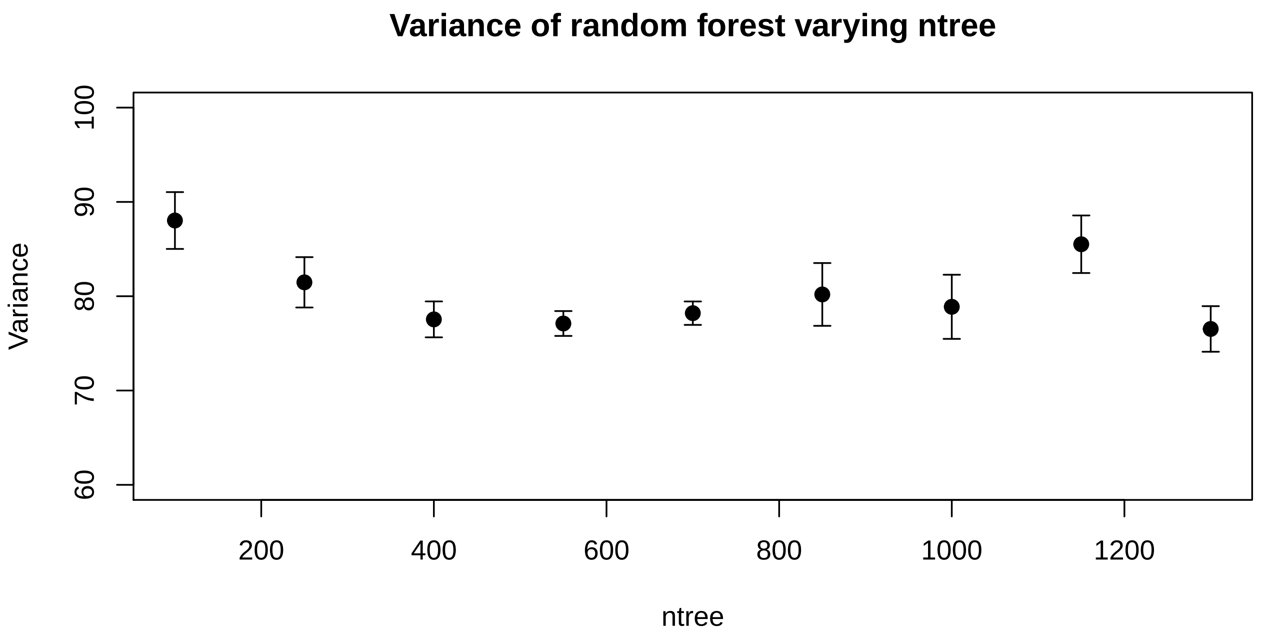
```r
set.seed(Sys.time())
ntree_arr <- seq(100,1300,150)    # The array for different m_try, 100 min, 1500 max
ntree_arrlen <- length(ntree_arr)
n_iter <- 10
y_sqdiff <- matrix(0, n_iter, ntree_arrlen)
# Matrices to store the predicted and true responses.
for (k in 1:ntree_arrlen) {
  print(k)
  y_true <- matrix(0, n_iter, n)
  y_pred <- matrix(0, n_iter, n)
  for (i in 1:n_iter) {
    y_true[i,] <- y_model + rnorm(n, mean = 0, sd = 1)
    rf.fit = randomForest(X, y_true[i,], ntree = ntree_arr[k])
    y_pred[i,] <- rf.fit$predicted    # Prediction of y given X by random forest.
    y_sqdiff[i,k] = sum((y_model - y_pred[i,])^2)
  }
}
```

We then calculate the mean and standard errors of variance estimates for each `n_tree`.

```
mf <- function(x){sd(y_sqdiff[,x])/sqrt(length(y_sqdiff[,x]))}
y_varserr <- sapply(1:length(y_sqdiff[1,]),mf)
y_varmean <- colMeans(y_sqdiff)
```

We illustrate how variance varies with `ntree`

```
plot(x = ntree_arr, y = y_varmean, pch = 19, lwd = 2, xlab = "ntree", ylab = "Variance",
     ylim=c(60,100))
arrows(ntree_arr, y_varmean-y_varserr, ntree_arr, y_varmean+y_varserr,
       length=0.05, angle=90, code = 3)
title("Variance of random forest varying ntree")
```



As can see in the figure, the variance estimate converges at `ntree` = 1000. The variance is about 80. The variance is large when `ntree` is small, and it decreases and converges when `ntree` increases.

Question 3.

Part a.

```r
X <- matrix(c(3.6,1.7,3.7,3.9,3.0,7.5,9.0,7.4,10.1,6.6), 10,1)
y1 <- matrix(c(1,-1,-1,-1,-1,1,-1,1,1,1), 10,1)
w <- matrix(c(0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1), 10,1)
numseq <- matrix(seq(1,length(X),1), 10, 1)
db <- cbind(numseq, X, y1, w)
```

We write a function to split the testing dataset, determined on a value.

```r
test_split <- function(value, db) {
  left_ind <- c()
  right_ind <- c()
  db_row <- nrow(db)
  # Store up the indices for the left and right nodes.
  for (i in 1:db_row) {
    if (db[,2][i] < value) {left_ind <- c(left_ind, i)}
    else {right_ind <- c(right_ind, i)}
  }
  left_len <- length(left_ind)
  right_len <- length(right_ind)
  # Left and right nodes.
  left_node <- matrix(db[left_ind,], left_len, 4)
  right_node <- matrix(db[right_ind,], right_len, 4)
  return(list(left_node, right_node))
}
```

Function for calculating the Gini impurity.

```r
gini_index <- function(groups, classes, db_row) {
  gini <- 0.0
  tot_weight <- sum(groups[[1]][,4]) + sum(groups[[2]][,4])
  for (i in 1:length(groups)) {
    size <- nrow(groups[[i]])
    nodetot_weight <- sum(groups[[i]][,4]) # Total weight in the node.
    # Find the response y = 1.
    pos <- which(groups[[i]][,3] == 1)
    score <- 0.0
    # If the node is pure.
    if (length(pos) == 0) {score <- 0}
    else {
      p <- sum(groups[[i]][,3][pos] * groups[[i]][,4][pos]) /
        nodetot_weight
      score <- p * (1 - p)
    }
    gini <- gini + ((nodetot_weight + 0.) / tot_weight) * score
  }
  return(gini)
}
```

We select the best split point for the dataset. The output of the model would be: `[[1]]` the cutting point c, `[[2]]` and `[[3]]` represent the elements put in the f_L and f_R nodes respectively. The prediction of the child nodes is determined by weighed average of the classes in that node. Hence, the last column `[[4]]` represent the predicted response of that class.

```r
get_split <- function(db) {
  class_values <- c(-1, 1)
  # Initializing the parameters.
  cut_pt <- 999; b_score <- 999; left_gp <- NA; right_gp <- NA
  db_row <- nrow(db)
  for (i in 1:db_row) {
    # Getting the left and right nodes.
    groups <- test_split(db[,2][i], db)
    gini <- gini_index(groups, class_values, db_row)
    # Store up the best split information.
    if (gini < b_score) {
      cut_pt <- db[i,][2]    # Cutting point
      b_score <- gini
      if (nrow(groups[[1]]) != 0) {
        # Storing the X's in the predicting nodes.
        left_gp <- groups[[1]]
        totweight <- sum(left_gp[,4])
        negpos <- which(left_gp[,3] == -1)
        negweight <- sum(left_gp[,4][negpos])
        negprop <- negweight / totweight
        if (negprop > 0.5) {
          left_gp <- cbind(left_gp, -1)
        }
        else {
          left_gp <- cbind(left_gp, 1)
        }
      }
      if (nrow(groups[[2]]) != 0) {
        right_gp <- groups[[2]]
        totweight <- sum(right_gp[,4])
        negpos <- which(right_gp[,3] == -1)
        negweight <- sum(right_gp[,4][negpos])
        negprop <- negweight / totweight
        if (negprop > 0.5) {
          right_gp <- cbind(right_gp, -1)
        }
        else {
          right_gp <- cbind(right_gp, 1)
        }
      }
    }
  }
  return(list(cut_pt, left_gp, right_gp))
}
```

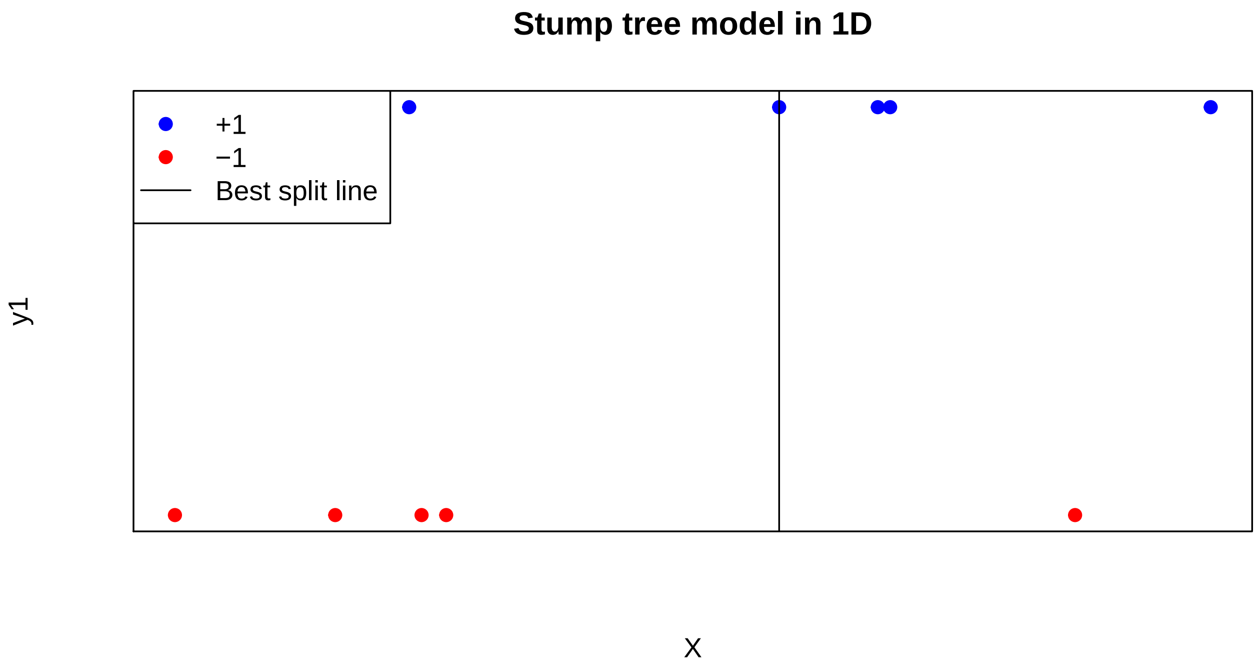
We try out the code and get the best split line for the stump model.

```r
stump <- get_split(db)
```

We visualize the splitting result as the following. The best split line is represented by the solid black vertical line in the figure.

```r
plot(X, y1, col = ifelse(y1 == 1, "blue", "red"), pch = 19, yaxt="n", xaxt = "n")
abline(v=stump[[1]])
```

```r
legend("topleft", bg = "white", c("+1","-1","Best split line"),
       col=c("blue", "red", "black"), pch=c(19, 19, NA), lty = c(NA,NA,1))
title("Stump tree model in 1D")
```



Part b.

We write the `AdaBoost` algorithm, which is based on the stump model. As a remark, we store the cutting points and create a vector `alpha_arr` to store up the corresponding α values for each iteration. We also store the predictions of the left and right nodes at each iteration.

```r
AdaBoostDS <- function(db, numIter) {
  db_nrow <- nrow(db)
  pred_mat <- matrix(NA, db_nrow, numIter) # Matrix to store predictions.
  alpha_arr <- c() # Array for storing the alpha in each iteration.
  cutpt_arr <- c()
  nodepred_arr <- matrix(NA, 2, numIter) # Matrix storing the prediction of nodes.
  for (i in 1:numIter) {
    if (i %% 100 == 0) {
      print("Total number of iterations")
      print(numIter)
      print("Iteration #")
      print(i)
    }
    stump <- get_split(db)
    cutpt_arr <- c(cutpt_arr, stump[[1]])
    # Updated database, 2nd column is the true response, and 4th column
    # is the predicted response.
    # Left [[2]] and right [[3]] node prediction
    nodepred_arr[1,i] <- stump[[2]][,5][1]
    nodepred_arr[2,i] <- stump[[3]][,5][1]
    db <- rbind(stump[[2]], stump[[3]])
    db <- db[order(db[,1]),]
    pred_mat[,i] <- db[,5]
    fpos <- which(db[,3] != db[,5]) # Elements with wrong prediction.
    err <- sum(db[,4][fpos])
    # The max function is to prevent error raised from division with 0 (pure node)
    alpha <- 0.5 * log(max((1.0 - err), 1e-16)/max(err, 1e-16))
    alpha_arr <- c(alpha_arr,alpha)
    w_t1 <- db[,4] * exp(-alpha * (db[,3] * db[,5]))
    # Update weights
    db[,4] <- w_t1 / sum(w_t1)
    db <- db[,c(1,2,3,4)] # Drop the last column for correct format to stump function.
  }
  # Best split point, alpha and classified array.
  return(list(cutpt_arr, alpha_arr, nodepred_arr))
}
```

This is a function to classify the testing data, given the trained model. The prediction at each iteration $f_t(x_i)$ is stored in a matrix form. The ultimate prediction $F_T(x_i)$ is made by taking the dot product between the $f_T(x_i)$ matrix and the α vector.

```
AdaBoostClassify <- function(trained_adb, db_test, numIter) {
  test_row <- nrow(db_test)
  fit_fx <- matrix(NA, test_row, numIter)
  for (k in 1:test_row) { # test_row
    for (i in 1:numIter) { # numIter
      if (db_test[,2][k] < trained_adb[[1]][i]) {
        fit_fx[k,i] <- trained_adb[[3]][1,i]
      }
      else { fit_fx[k,i] <- trained_adb[[3]][2,i] }
    }
  }
  fit_y <- sign(fit_fx %*% trained_adb[[2]]) # Fitting the test data
  return(list(fit_fx, trained_adb[[2]], fit_y))
}
```

We generate the following model to test if our own-designed `AdaBoost` code is correct. If it is, the error rate should decrease as the number of iteration increases, and it should stabilize beyond some number of iterations.

```
set.seed(80)
n <- 300
x <- matrix(runif(n), n, 1)
y <- matrix((rbinom(n , 1 , ( sin (4*pi*x)+1)/2) - 0.5) * 2, n, 1)
w <- matrix(rep(1/n,n),n,1)
numseq <- matrix(seq(1,n,1), n, 1)
db <- cbind(numseq, x, y, w)
```

We train the data in the following.

```
adaiter <- 1500
trained_adb <- AdaBoostDS(db, adaiter)
```

After the iterations, we can calculate the misclassification errors at some number of iterations. We write a function to incorporate it.

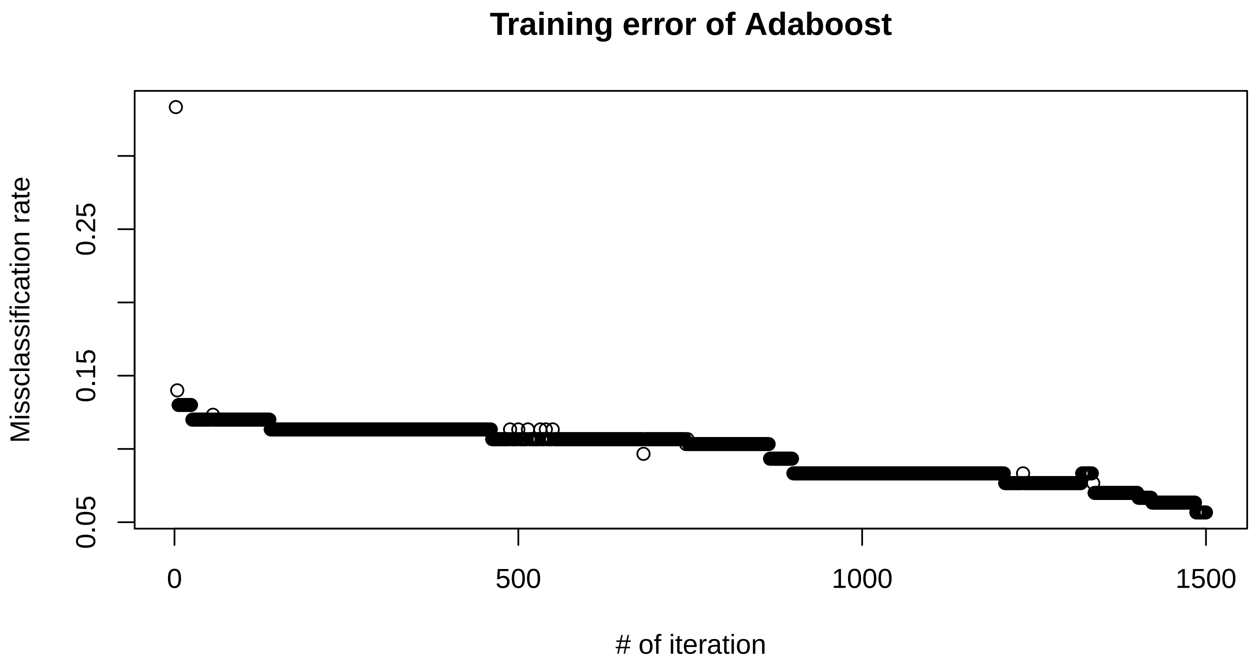
```
mis_err <- function(adb, numIter) {
  alpha <- adb[[2]]
  predfx <- adb[[1]]
  adaiter_arr <- seq(2,numIter,2)
  adaiter_num <- length(adaiter_arr)
  adapred <- matrix(NA, adaiter_num, 1)

  for (i in 1:adaiter_num) {
    alpha_sub <- alpha[1:adaiter_arr[i]]
    predfx_sub <- predfx[,1:adaiter_arr[i]]
    prediction_ada <- sign(predfx_sub %*% alpha_sub)
    adapred[i,] <- length(which(prediction_ada != y)) / n
  }
  return(list(adaiter_arr, adapred))
}
```

The design of the code is that it stores the prediction $f_t(x_i)$ for each iteration, and the corresponding α's. Hence, if we want to figure out the prediction after i iterations, we simply take the first i columns from the prediction matrix in the algorithm, and dot it to the first i components in the α array. We classify the training data.

```
adb_trained_perform <- AdaBoostClassify(trained_adb, db, adaiter)
trained_miserr <- mis_err(adb_trained_perform, adaiter)
```

We visualize the training error of `Adaboost`, it decreases as number of iteration increases. We check the fitted model with testing data to see when we see the overfitting problem.

```
plot(trained_miserr[[1]], trained_miserr[[2]], ylab = "Missclassification rate",
     xlab = "# of iteration", pch = 1)
title("Training error of Adaboost")
```



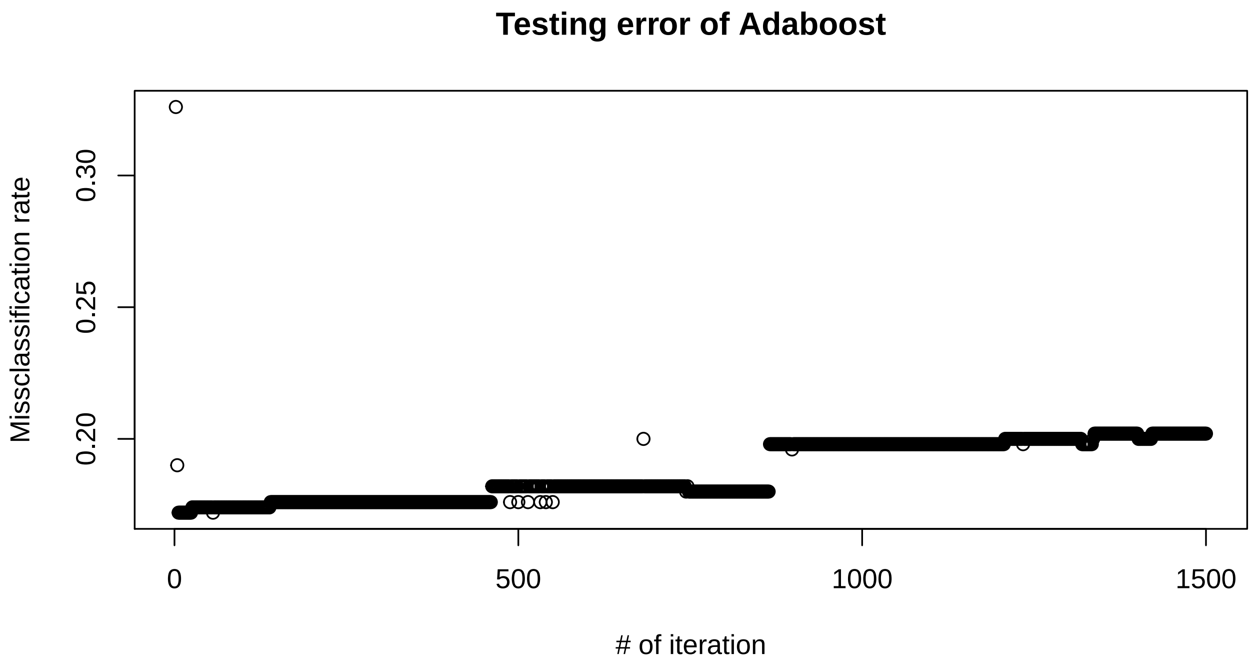
We test the trained model with some testing data.

```
set.seed(60)
n <- 500
x <- matrix(runif(n), n, 1)
y <- matrix((rbinom(n , 1 , ( sin (4*pi*x)+1)/2) - 0.5) * 2, n, 1)
w <- matrix(rep(1/n,n),n,1)
```

```r
numseq <- matrix(seq(1,n,1), n, 1)
test_db <- cbind(numseq, x, y, w)

adb_test_perform <- AdaBoostClassify(trained_adb, test_db, adaiter)
test_miserr <- mis_err(adb_test_perform, adaiter)
```

We visualize the testing error of the same `Adaboost` model. The model performs the best when the number of iteration is about 100. Then the testing error increases, indicating beyond 100 iterations, there is overfitting problem.

```r
plot(test_miserr[[1]], test_miserr[[2]], ylab = "Missclassification rate",
     xlab = "# of iteration", pch = 1)
title("Testing error of Adaboost")
```


Testing error of Adaboost
Missclassification rate
# of iteration