Stat 432 Homework 9

Assigned: Oct 21, 2024; Due: 11:59 PM CT, Oct 31, 2024

- Question 1: A Simulation Study for Random Forests [50 pts]
- Question 2: Parameter Tuning with OOB Prediction [20 pts]
- Question 3: Using `xgboost` [30 pts]

Question 1: A Simulation Study for Random Forests [50 pts]

We learned that random forests have several key parameters and some of them are also involved in trading the bias and variance. To confirm some of our understandings, we will conduct a simulation study to investigate each of them:

1. The terminal node size `nodesize`
2. The number of variables randomly sampled as candidates at each split `mtry`
3. The number of trees in the forest `ntree`

For this question, we will use the `randomForest` package. This package is quite slow, so you may want to try smaller amount of simulations first to make sure your code is correct.

a. [5 pts] Generate the data using the following model:

$$Y = X_1 + X_2 + \epsilon,$$

where the two covariates X_1 and X_2 are independently from standard normal distribution and $\epsilon \sim N(0, 1)$. Generate a training set of size 200 and a test set of size 300 using this model. Fit a random forest model to the training set with the default parameters. Report the MSE on the test set.

Solution:

```
# set seed
set.seed(432)

# packages
library(randomForest)
```

```
## randomForest 4.7-1.1
```

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

```r
# simulation setting
n_train <- 200
n_test <- 300
p <- 2

# generate covariates
X_train <- matrix(rnorm(p * n_train), nrow = n_train)
X_test <- matrix(rnorm(p * n_test), nrow = n_test)

# define the true model
linkfun <- function(x){
  true_Y_expectation <- 1 * x[, 1] + 1 * x[, 2]
  return(true_Y_expectation)
}

# generate Y's
Y_train <- linkfun(X_train) + rnorm(n_train)
Y_test <- linkfun(X_test) + rnorm(n_test)

# fit random forest model
rf <- randomForest(X_train, Y_train)

# report MSE
mse <- mean((Y_test - predict(rf, X_test))^2)
print(paste0("MSE on test set: ", round(mse, digits = 3)))
```

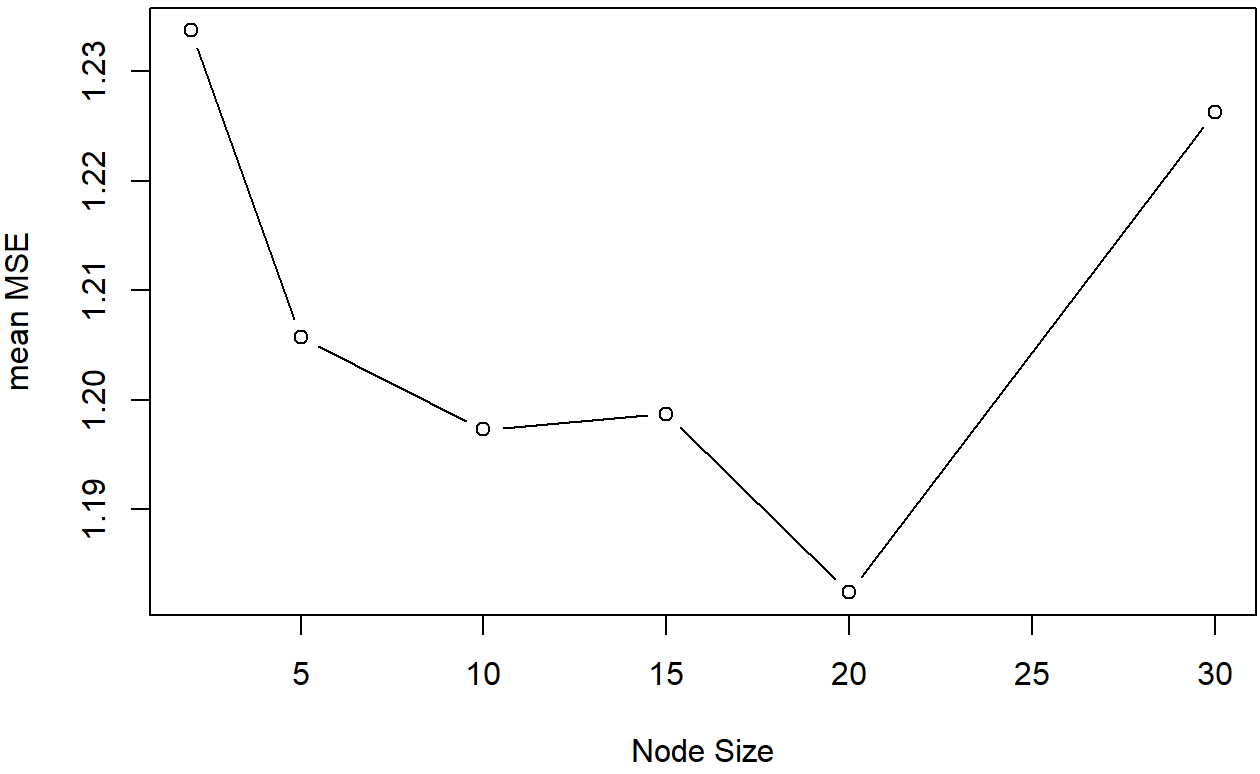
```
## [1] "MSE on test set: 1.28"
```

b. [15 pts] Let's analyze the effect of the terminal node size `nodesize`. We will consider the following values for `nodesize`: 2, 5, 10, 15, 20 and 30. Set `mtry` as 1 and the bootstrap sample size as 150. For each value of `nodesize`, fit a random forest model to the training set and record the MSE on the test set. Then repeat this process 100 times and report (plot) the average MSE against the `nodesize`. Same idea of the simulation has been considered before when we worked on the KNN model. After getting the results, answer the following questions:

- Do you think our choice of the `nodesize` parameter is reasonable? What is the optimal node size you obtained? If you don't think the choice is reasonable, re-define your range of tuning and report your results and the optimal node size.
- What is the effect of `nodesize` on the bias-variance trade-off?

Solution:

```r
# simulation setting
nsim <- 100
allnodesize <- c(2, 5, 10, 15, 20, 30)

# storage for mse values
all_mse <- matrix(NA, nrow = nsim, ncol = length(allnodesize))
rownames(all_mse) <- paste0("Sim_", 1:nsim)
colnames(all_mse) <- paste0("Node_Size_", allnodesize)

# loop through each simulation
for (i in 1:nsim) {
  # loop through each node size
  for (j in 1:length(allnodesize)) {
    # simulate X
    X_train <- matrix(rnorm(p * n_train), nrow = n_train)
    X_test <- matrix(rnorm(p * n_test), nrow = n_test)

    # simulate Y
    Y_train <- linkfun(X_train) + rnorm(n_train)
    Y_test <- linkfun(X_test) + rnorm(n_test)

    # fit random forest model
    rf <- randomForest(X_train, Y_train, nodesize = allnodesize[j], mtry = 1,
                       sampsize = 150)

    # store mse
    all_mse[i, j] <- mean((Y_test - predict(rf, X_test))^2)
  }
}

# calculate mean mse for all simulations for each nodesize value
mean_mse <- colMeans(all_mse)

# plot
plot(allnodesize, mean_mse, type = "b", xlab = "Node Size", ylab = "mean MSE")
```



The optimal node size is `20`. It looks like our choice of the node size parameter is reasonable as it not one of the boundary values within the set of values we tested, which would have indicated we need to try more values. In addition, an U shape is observed within the plot of mean MSE vs Node Size. The effect of `nodesize` on the bias-variance trade-off is that the bias increases and the variance decreases as the node size increases. It behaves similarly to the KNN model when we increase the number of neighbors.

c. [15 pts] In this question, let's analyze the effect of `mtry`. We will consider a new data generator:

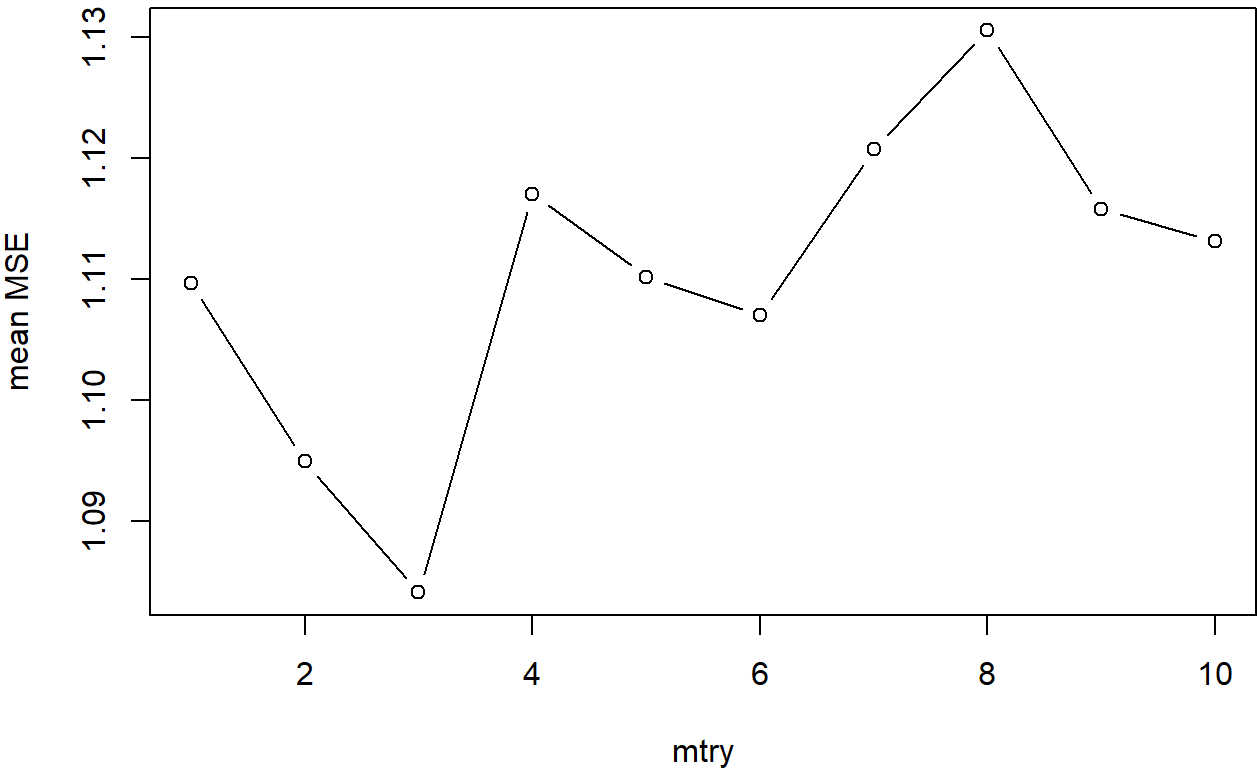
$$Y = 0.2 \times \sum_{j=1}^{5} X_j + \epsilon,$$

where we generate a total of 10 covariates independently from standard normal distribution and $\epsilon \sim N(0, 1)$. Generate a training set of size 200 and a test set of size 300 using the model above. Fix the node size as 3, the bootstrap sample size as 150, and consider `mtry` to be all integers from 1 to 10. Perform the simulation study with 100 runs, report your results using a plot, and answer the following questions:

- What is the optimal value of `mtry` you obtained?
- What is the effect of `mtry` on the bias-variance trade-off?

Solution:

```r
# simulation setting
nsim <- 100
allmtry <- 1:10
p <- 10

# data generator
linkfun <- function(x){
  true_y_expectation <- rowSums(x[, 1:5]) * 0.2
  return(true_y_expectation)
}

# storage for mse values
all_mse <- matrix(NA, nrow = nsim, ncol = length(allmtry))
rownames(all_mse) <- paste0("Sim_", 1:nsim)
colnames(all_mse) <- paste0("mtry_", allmtry)

# loop through each simulation
for (i in 1:nsim) {
  # loop through each mtry value
  for (j in 1:length(allmtry)) {
    # simulate X
    X_train <- matrix(rnorm(p * n_train), nrow = n_train)
    X_test <- matrix(rnorm(p * n_test), nrow = n_test)

    # simulate Y
    Y_train <- linkfun(X_train) + rnorm(n_train)
    Y_test <- linkfun(X_test) + rnorm(n_test)

    # fit random forest model
    rf <- randomForest(X_train, Y_train, nodesize = 3, mtry = allmtry[j], sampsize = 150)
    all_mse[i, j] <- mean((Y_test - predict(rf, X_test))^2)
  }
}

# calculate mean mse for all simulations for each mtry value
mean_mse <- colMeans(all_mse)

# plot
plot(allmtry, mean_mse, type = "b", xlab = "mtry", ylab = "mean MSE")
```

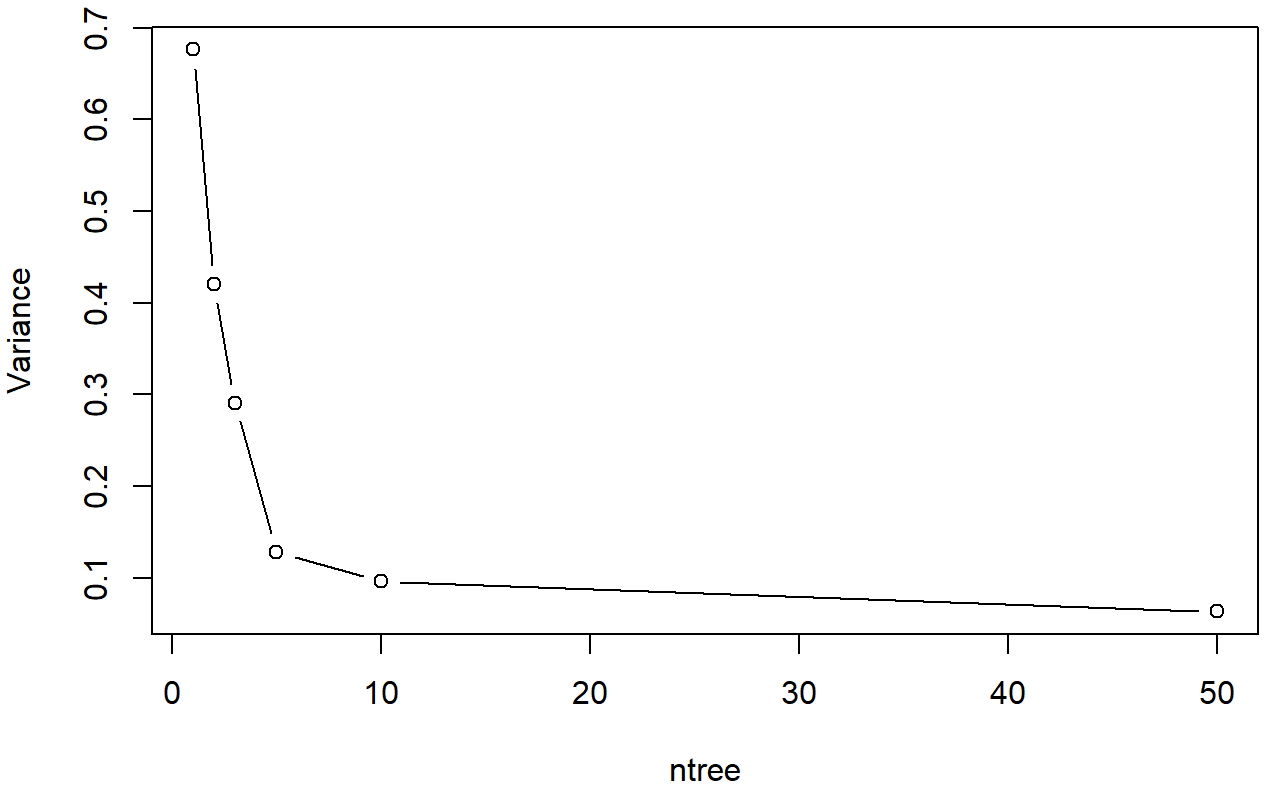


The optimal value of `mtry` is `3`. As `mtry` increases, the bias decreases as we are increasing the chance to split the true variable. As `mtry` increases, the variance increases as the trees are more correlated with each other.

d. [15 pts] In this question, let's analyze the effect of `ntree`. We will consider the same data generator as in part (c). Fix the node size as 10, the bootstrap sample size as 150, and `mtry` as 3. Consider the following values for `ntree`: 1, 2, 3, 5, 10, 50. Perform the simulation study with 100 runs. For this question, we do not need to calculate the prediction of all subjects. Instead, calculate just the prediction on a target point that all the covariate values are 0. After obtaining the simulation results, calculate the variance of the random forest estimator under different `ntree` values (for the definition of variance of an estimator, see our previous homework on the bias-variance simulation). Comment on your findings.

Solution:

```r
# simulation setting
nsim <- 100
allntree <- c(1, 2, 3, 5, 10, 50)
p <- 10

# data generator
linkfun <- function(x){
  true_y_expectation <- rowSums(x[, 1:5, drop = FALSE]) * 0.2
  return(true_y_expectation)
}

# storage for all predictions
allpred <- matrix(NA, nsim, length(allntree))
rownames(allpred) <- paste0("Sim_", 1:nsim)
colnames(allpred) <- paste0("ntree_", allntree)

# loop through each simulation
for (i in 1:nsim) {
  # loop through each ntree value
  for (j in 1:length(allntree)) {
    # simulate X
    X_train <- matrix(rnorm(p * n_train), nrow = n_train)
    X_test <- matrix(rep(0, p), nrow = 1)

    # simulate Y
    Y_train <- linkfun(X_train) + rnorm(n_train)
    Y_test <- linkfun(X_test) + rnorm(n_test)

    # fit random forest model
    rf <- randomForest(X_train, Y_train, nodesize = 10, mtry = 3, ntree = allntree[j],
                       sampsize = 150)

    # store predictions
    allpred[i, j] <- predict(rf, X_test)
  }
}

# calculate variance across all simulation runs for a particular ntree value
all_var <- apply(allpred, 2, var)

# plot
plot(allntree, all_var, type = "b", xlab = "ntree", ylab = "Variance")
```



The number of trees works to reduce the variance of the random forest estimator. As the number of trees increases, the variance of the random forest estimator decreases. The more independent each tree is, the more effective this parameter is since averaging independent variables will lead to variance in the scale of 1/ntree.

Question 2: Parameter Tuning with OOB Prediction [20 pts]

We will again use the `MNIST` dataset. We will use the first 2600 observations of it:

```
# inputs to download file
fileLocation <- "https://pjreddie.com/media/files/mnist_train.csv"
numRowsToDownload <- 2600
localFileName <- paste0("mnist_first", numRowsToDownload, ".RData")

# download the data and add column names
mnist2600 <- read.csv(fileLocation, nrows = numRowsToDownload)
numColsMnist <- dim(mnist2600)[2]
colnames(mnist2600) <- c("Digit", paste("Pixel", seq(1:(numColsMnist - 1)), sep = ""))

# save file
# in the future we can read in from the local copy instead of having to redownload
save(mnist2600, file = localFileName)
```

```
# you can load the data with the following code
load(file = "mnist_first2600.RData")
dim(mnist2600)
```

```
## [1] 2600  785
```

a. [5 pts] Similar to what we have done before, split the data into a training set of size 1300 and a test set of the remaining data. Then keep only the digits 2, 4 and 8. After this screen the data and only keep the top 250 variables with the highest variance.

Solution:

```
# get train and test data
train <- mnist2600[1:1300, ]
train <- train[train$Digit == 2 | train$Digit == 4 | train$Digit == 8, ]
test <- mnist2600[1300:2600, ]
test <- test[test$Digit == 2 | test$Digit == 4 | test$Digit == 8, ]

# perform marginal screening
var <- apply(train[, -1], 2, var)
varuse <- order(var, decreasing = TRUE)[1:250]
train <- train[, c(1, varuse + 1)]
test <- test[, c(1, varuse + 1)]
```

b. [15 pts] Fit classification random forests to the training set and tune parameters `mtry` and `nodesize`. Choose 4 values for each of the parameters. Use `ntree` = 1000 and keep all other parameters as default. To perform the tuning, you must use the OOB prediction. Report your results for each tuning and the optimal choice. After this, use the random forest corresponds to the optimal tuning to predict the testing data, and report the confusion matrix and the accuracy.

Solution:

```
# simulation setting
allmtry = c(10, 20, 50, 100)
allnodesize = c(5, 10, 15, 20)

# storage for oob error
grid.expand <- expand.grid(mtry = allmtry, nodesize = allnodesize)
oob_error <- cbind(grid.expand, "error" = NA)

# loop through each simulation setting
for (i in 1:nrow(grid.expand)) {
  # fit random forest model
  rf <- randomForest(train[, -1], as.factor(train$Digit),
                     mtry = grid.expand$mtry[i],
                     nodesize = grid.expand$nodesize[i],
                     ntree = 1000)

  # store oob error
  oob_error$error[i] <- rf$err.rate[nrow(rf$err.rate), 1]
}

# report results
print(oob_error)
```

```
##    mtry nodesize      error
## 1    10        5 0.06701031
## 2    20        5 0.06958763
## 3    50        5 0.06958763
## 4   100        5 0.06958763
## 5    10       10 0.06443299
## 6    20       10 0.06701031
## 7    50       10 0.07731959
## 8   100       10 0.07474227
## 9    10       15 0.06958763
## 10   20       15 0.07989691
## 11   50       15 0.07474227
## 12  100       15 0.08505155
## 13   10       20 0.06958763
## 14   20       20 0.06958763
## 15   50       20 0.07989691
## 16  100       20 0.08762887
```

```
# determine optimal model
optimal <- oob_error[which.min(oob_error$error), ]
print(optimal)
```

```
##   mtry nodesize      error
## 5   10       10 0.06443299
```

```
# refit optimal model
rf <- randomForest(train[, -1], as.factor(train$Digit),
                   mtry = optimal$mtry,
                   nodesize = optimal$nodesize,
                   ntree = 1000)

# predict on test data set
pred <- predict(rf, test[, -1])

# confusion matrix
confMat <- table(Prediction = pred, Truth = test$Digit)
print(confMat)
```

```
##           Truth
## Prediction   2   4   8
##          2 122   3   6
##          4   4 141   4
##          8   3   2 100
```

```
# accuracy
acc <- mean(pred == test$Digit)
print(acc)
```

```
## [1] 0.9428571
```

Based on our choice, the optimal `mtry` is `10` and the optimal `nodesize` is `10`. The out-of-bag error of the model is `0.064433`. After applying this model to the testing data, we get an accuracy of `0.9428571`.

Question 3: Using `xgboost` [30 pts]

a. [20 pts] We will use the same data as in Question 2. Use the `xgboost` package to fit the MNIST data multi-class classification problem. You should specify the following:

- Use `multi:softmax` as the objective function so that it can handle multi-class classification
- Use `num_class` = 3 to specify the number of classes
- Use `gbtree` as the base learner
- Tune these parameters:
- The learning rate `eta` = 0.5
- The maximum depth of trees `max_depth` = 2
- The number of trees `nrounds` = 100

Report the testing error rate and the confusion matrix.

Solution:

```
# libraries
library(xgboost)
```

```
## Warning: package 'xgboost' was built under R version 4.4.1
```

```
# for xgboost the label must be in [0, num_class)
# digit 2 -> 0
# digit 4 -> 1
# digit 8 -> 2
trainDigitRefactored <- train$Digit
trainDigitRefactored <- ifelse(trainDigitRefactored == 2, 0,
                               ifelse(trainDigitRefactored == 4, 1, 2))

# Train the XGBoost model
xgb_model <- xgboost(data = data.matrix(train[, -1]),
                     label = trainDigitRefactored,
                     params = list(objective = "multi:softmax", # objective function
                                   num_class = 3, # number of classes
                                   booster = "gbtree", # base learner
                                   eta = 0.5, # learning rate,
                                   max_depth = 2 # maximum depth of trees
                                   ),
                     nrounds = 100, # number of trees
                     verbose = 0)

# Evaluate the model on the test set
predictions <- predict(xgb_model, newdata = data.matrix(test[, -1]))

# map the predictions back to the original digits
predictions <- ifelse(predictions == 0, 2, ifelse(predictions == 1, 4, 8))

# testing error rate
testingErrorRate <- 1 - mean(predictions == test$Digit)
print(testingErrorRate)
```

```
## [1] 0.06233766
```

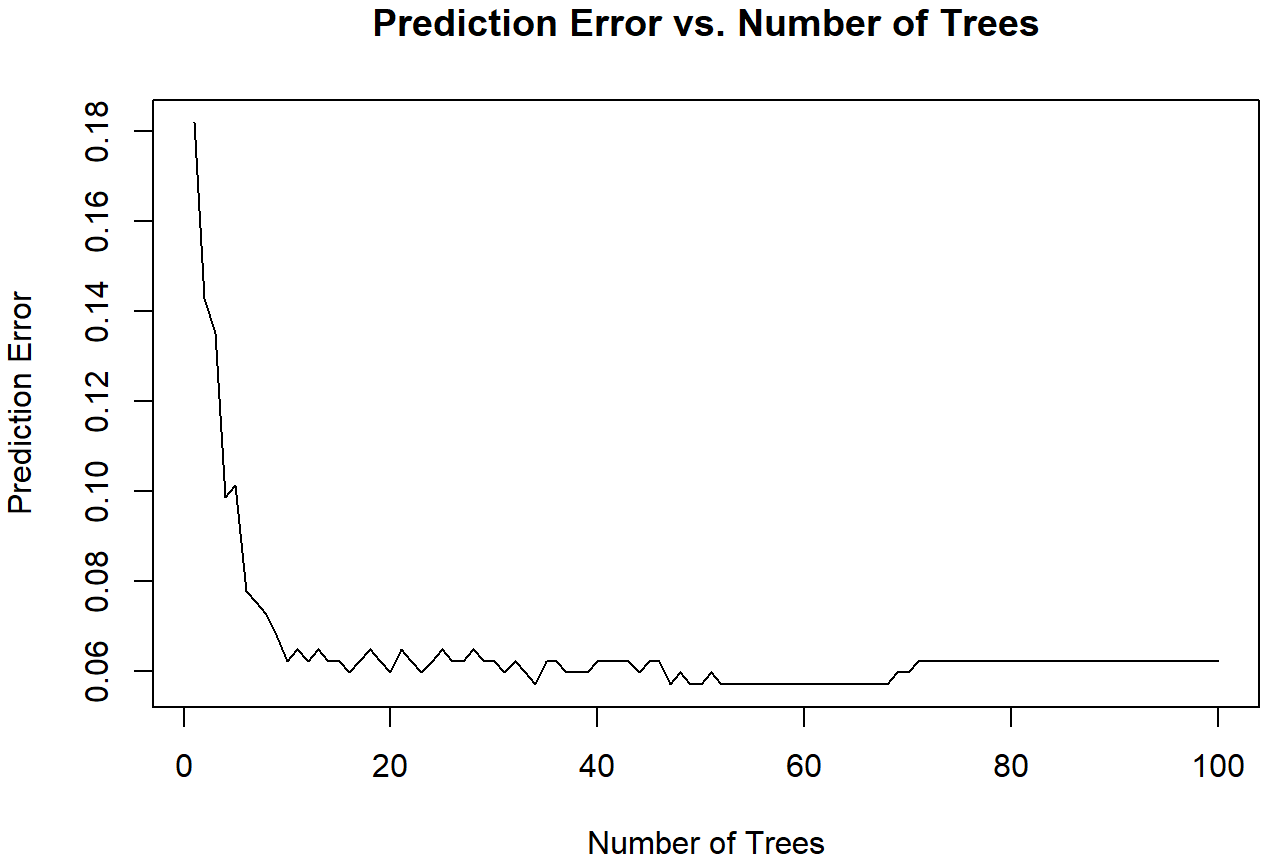
```
# confusion matrix
confMatQ3PartA <- table(Prediction = predictions, Truth = test$Digit)
print(confMatQ3PartA)
```

```
##           Truth
## Prediction   2   4   8
##          2 119   3   1
##          4   5 137   4
##          8   5   6 105
```

b. [10 pts] The model fits with 100 rounds (trees) sequentially. However, you can produce your prediction using just a limited number of trees. This can be controlled using the iterationrange argument in the `predict()` function. Plot your prediction error vs. number of trees. Comment on your results. Note, in the question we posted originally, there was a typo in this question, it should have been iterationrange not iteration_range.

Solution:

```
# storage for prediction errors
errors = rep(NA, 100)

# loop through 1 to 100
for (i in 1:100) {
  # generate predictions using a certain number of trees
  predictions <- predict(xgb_model, newdata = data.matrix(test[, -1]),
                         iterationrange = c(1, i + 1))

  # map the predictions back to the original digits
  predictions <- ifelse(predictions == 0, 2, ifelse(predictions == 1, 4, 8))

  # calculate error
  errors[i] = mean(predictions != test$Digit)
}

# plot
plot(errors, xlab = "Number of Trees", ylab = "Prediction Error",
     main = "Prediction Error vs. Number of Trees", type = "l")
```




```
# comment
paste0("The optimal number of trees to use in terms of prediction error is: ",
       which.min(errors), ".")
```

```
## [1] "The optimal number of trees to use in terms of prediction error is: 34."
```