# HW09_zilinw3

Zilin Wang (zilinw3)

2024-10-30

## Contents



## 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.

**Answer:**

```
library(randomForest)
```

```
## randomForest 4.7-1.2
```

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

```r
# Generate data
set.seed(123)
n_train <- 200
n_test <- 300

# Generate training data
X1_train <- rnorm(n_train)
X2_train <- rnorm(n_train)
epsilon_train <- rnorm(n_train)
Y_train <- X1_train + X2_train + epsilon_train
train_data <- data.frame(X1 = X1_train, X2 = X2_train, Y = Y_train)

# Generate test data
X1_test <- rnorm(n_test)
X2_test <- rnorm(n_test)
epsilon_test <- rnorm(n_test)
Y_test <- X1_test + X2_test + epsilon_test
test_data <- data.frame(X1 = X1_test, X2 = X2_test, Y = Y_test)

# Fit random forest model with default parameters
rf_model <- randomForest(Y ~ X1 + X2, data = train_data)
predictions <- predict(rf_model, newdata = test_data)
mse <- mean((Y_test - predictions)^2)

# Output the MSE
cat("The MSE: ", mse, "\n")
```

```
## The MSE:  1.258612
```

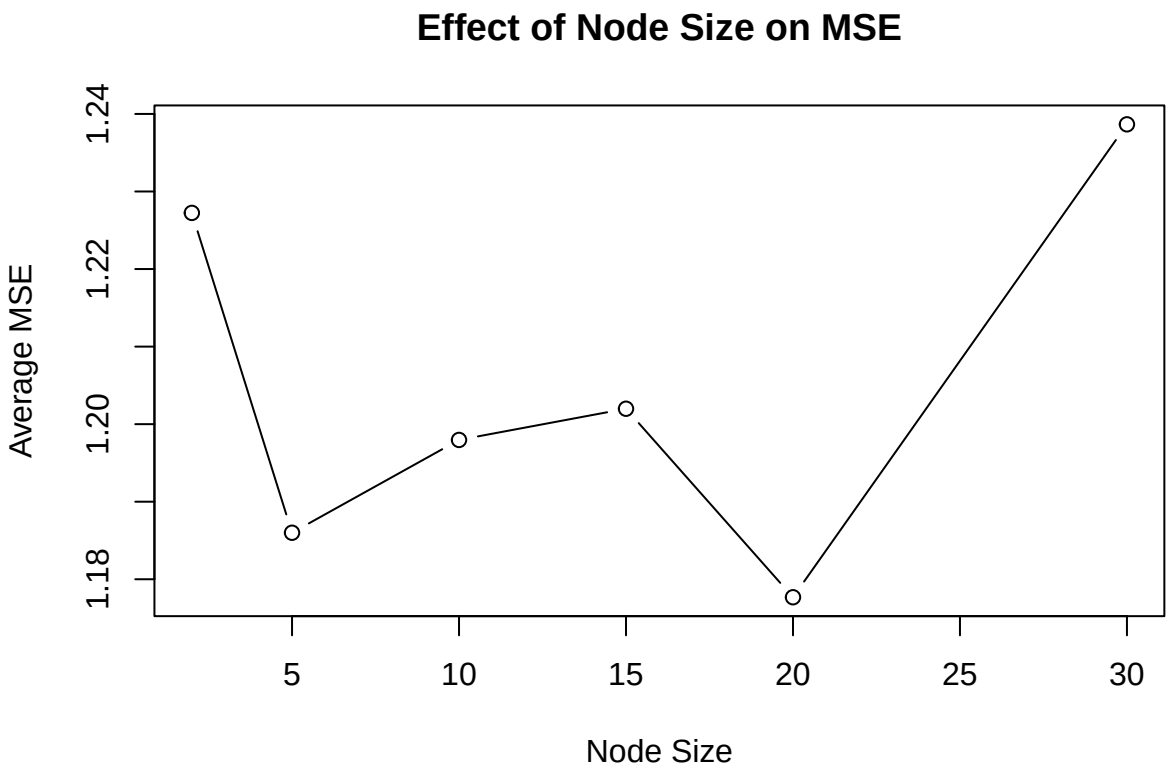
b. [15 pts] Let's analyze the effect of the terminal node size `nodesize`. We will consider the following values for `nodesize`: 2, 5, 10, 15, 20 and 30. Set `mtry` as 1 and the bootstrap sample size as 150. For each value of `nodesize`, fit a random forest model to the training set and record the MSE 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?

**Answer:**

```r
set.seed(123)

# Generate data
nodesize_values <- c(2, 5, 10, 15, 20, 30)
num_simulations <- 100
mse_results <- matrix(NA, nrow = num_simulations, ncol = length(nodesize_values))
```

```r
generate_data <- function(n) {
  X1 <- rnorm(n)
  X2 <- rnorm(n)
  epsilon <- rnorm(n)
  Y <- X1 + X2 + epsilon
  data <- data.frame(X1 = X1, X2 = X2, Y = Y)
  return(data)
}

# Run simulation
for (j in 1:length(nodesize_values)) {
  for (i in 1:num_simulations) {
    train_data <- generate_data(200)
    test_data <- generate_data(300)
    rf_model <- randomForest(Y ~ X1 + X2,
                             data = train_data,
                             nodesize = nodesize_values[j],
                             mtry = 1,
                             sampsize = 150)
    predictions <- predict(rf_model, newdata = test_data)
    mse_results[i, j] <- mean((test_data$Y - predictions)^2)
  }
}

# Calculate average MSE for each nodesize
avg_mse <- colMeans(mse_results)

# Plotting
plot(nodesize_values, avg_mse, type = "b",
     xlab = "Node Size", ylab = "Average MSE",
     main = "Effect of Node Size on MSE")
```




```
# Identify the optimal nodesize
# optimal_mse <- min(avg_mse)
optimal_nodesize <- nodesize_values[which.min(avg_mse)]

# Display the results
cat("Optimal Nodesize:", optimal_nodesize, "\n")
```

```
## Optimal Nodesize: 20
```

```
# cat("Minimum Average MSE:", optimal_mse, "\n")
```

I think our choice of the nodesize parameter is reasonable. Because the results show a clear trend that allows us to see where the model performs best. The lowest MSE occurs when nodesize is 20. This suggests that nodesize = 20 is the optimal choice among the tested values, balancing the model's fit to the training data without overfitting or underfitting.

The nodesize parameter controls the depth of each tree in the random forest, which directly affects the bias-variance trade-off:

Smaller Nodesize (Higher Variance, Lower Bias): When nodesize is small (e.g., 2), each tree can grow deeper and fit the training data more precisely. This reduces bias because the model has more flexibility to capture the complexities in the data. However, this increased flexibility can lead to higher variance, as each individual tree may overfit to its particular bootstrap sample.

Larger Nodesize (Lower Variance, Higher Bias): As nodesize increases (e.g., 30), the trees become shallower, reducing their flexibility. This increases bias because the model becomes less capable of capturing detailed

patterns, but it also reduces variance because each tree is less sensitive to the specific variations in its bootstrap sample.

The lowest MSE at nodesize = 20 suggests this is the point where the random forest achieves the best balance between bias and variance.

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

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

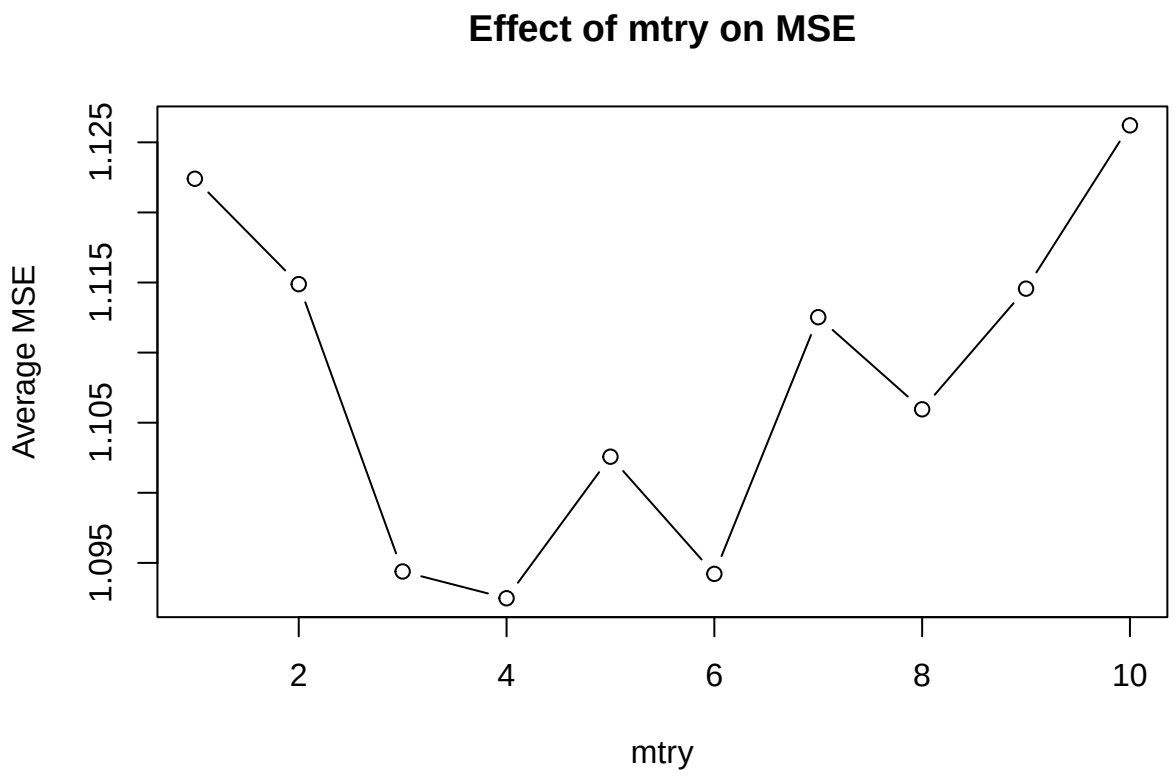
where we generate a total of 10 covariates independently from standard normal distribution and $\epsilon \sim N(0, 1)$. Generate a training set of size 200 and a test set of size 300 using the model above. Fix the node size as 3, the bootstrap sample size as 150, and consider `mtry` to be all integers 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?
```

**Answer:**

```r
# Generate data
set.seed(123)
num_simulations <- 100
mtry_values <- 1:10
mse_results <- matrix(NA, nrow = num_simulations, ncol = length(mtry_values))

# Generate new training and test data for 10 covariates
generate_data_2 <- function(n) {
  X <- matrix(rnorm(n * 10), ncol = 10)
  Y <- 0.2 * rowSums(X[, 1:5]) + rnorm(n)
  data <- data.frame(X = X, Y = Y)
  return(data)
}

# Run simulation
for (j in 1:length(mtry_values)) {
  for (i in 1:num_simulations) {
    train_data_2 <- generate_data_2(200)
    test_data_2 <- generate_data_2(300)
    rf_model <- randomForest(Y ~ .,
                             data = train_data_2,
                             nodesize = 3,
                             mtry = mtry_values[j],
                             sampsize = 150)
    predictions <- predict(rf_model, newdata = test_data_2)
    mse_results[i, j] <- mean((test_data_2$Y - predictions)^2)
  }
}

# Calculate average MSE for each mtry
```

```r
avg_mse <- colMeans(mse_results)

# Plotting
plot(mtry_values, avg_mse, type = "b",
     xlab = "mtry", ylab = "Average MSE",
     main = "Effect of mtry on MSE")
```




```r
# Identify the optimal mtry
# optimal_mse <- min(avg_mse)
optimal_mtry <- mtry_values[which.min(avg_mse)]

# Display the results
cat("Optimal mtry:", optimal_mtry, "\n")
```

```
## Optimal mtry: 4
```

```r
# cat("Minimum Average MSE:", optimal_mse, "\n")
```

The lowest MSE occurs when mtry is 4. This suggests that the optimal value of `mtry` is 4.

The mtry parameter in random forests determines the number of features randomly selected as candidates at each split. This parameter plays a crucial role in controlling the model's bias-variance trade-off:

Lower mtry (Higher Variance, Lower Bias): When mtry is small (e.g., 1 or 2), each split in a tree considers fewer features, leading to more variation in the trees. This introduces more randomness and independence

among the trees, as each tree will likely use different features for its splits. Variance increases because each tree becomes more unique and more responsive to individual variations in the data. However, this can lower bias, as the ensemble of trees has more flexibility to fit the training data, potentially capturing diverse patterns.

Higher mtry (Lower Variance, Higher Bias): As mtry increases (e.g., 9 or 10), more features are considered at each split, making the trees in the random forest more similar to each other. Bias increases as the trees become more similar and less flexible, fitting fewer unique patterns. However, variance is reduced because the model is more stable, with trees producing similar predictions due to reduced randomness in feature selection.

At mtry = 4, the model appears to achieve the best balance between bias and variance. The MSE is minimized, indicating that the trees have sufficient flexibility to capture relevant patterns without overfitting. This value allows enough randomness to reduce overfitting while maintaining a level of consistency across trees.

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.

**Answer:**

```r
set.seed(123)
ntree_values <- c(1, 2, 3, 5, 10, 50)
variance_results <- numeric(length(ntree_values))

# Ensure target_point has matching column names as train_data
target_point <- as.data.frame(matrix(0, nrow = 1, ncol = 10))
colnames(target_point) <- paste0("X.", 1:10)

for (j in 1:length(ntree_values)) {
  for (i in 1:num_simulations) {
    predictions <- numeric(num_simulations)
    train_data_3 <- generate_data_2(200)
    test_data_3 <- generate_data_2(300)
    rf_model <- randomForest(Y ~ .,
                             data = train_data_3,
                             nodesize = 10,
                             mtry = 3,
                             ntree = ntree_values[j],
                             sampsize = 150)
    predictions[i] <- predict(rf_model, newdata = target_point)
  }
  variance_results[j] <- var(predictions)
}

# Optional: display or plot variance_results
print(variance_results)
```

```
## [1] 3.639403e-03 9.337884e-03 6.959562e-06 3.826108e-05 3.889178e-04
## [6] 3.460202e-03
```

Comment on your findings.

## 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 = localFileName)
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.

**Answer:**

```
set.seed(123)

# Split data into training and testing
mnist_train <- mnist2600[1:1300, ]
mnist_test <- mnist2600[1301:2600, ]

# Subset data to include only digits 2, 4, and 8
mnist_train <- mnist_train[mnist_train$Digit %in% c(2, 4, 8), ]
mnist_test <- mnist_test[mnist_test$Digit %in% c(2, 4, 8), ]

# Calculate variance for each pixel column
pixel_vars <- apply(mnist_train[ , -1], 2, var)

# Select top 250 pixel columns with the highest variance
top_pixels <- names(sort(pixel_vars, decreasing = TRUE))[1:250]

# Subset training and test sets to only include these top 250 pixels
mnist_train_subset <- mnist_train[, c("Digit", top_pixels)]
mnist_test_subset <- mnist_test[, c("Digit", top_pixels)]
```

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.

**Answer:**

```r
library(randomForest)

# Example values for mtry and nodesize
mtry_values <- c(5, 10, 15, 20)
nodesize_values <- c(1, 5, 10, 15)

# Ensure the response variable is treated as a factor
mnist_train_subset$Digit <- as.factor(mnist_train_subset$Digit)

# Initialize results data frame
results <- data.frame(mtry = integer(),
                      nodesize = integer(),
                      OOB_Error = numeric())

for (mtry_val in mtry_values) {
  for (nodesize_val in nodesize_values) {
    # Train Random Forest model
    rf_model <- randomForest(Digit ~ .,
                             data = mnist_train_subset,
                             ntree = 1000,
                             mtry = mtry_val,
                             nodesize = nodesize_val)

    # Check if rf_model$err.rate exists and get the last OOB error
    oob_error <- if (!is.null(rf_model$err.rate)) {
      rf_model$err.rate[nrow(rf_model$err.rate), "OOB"]
    } else {
      NA
    }

    # Append the results
    results <- rbind(results, data.frame(mtry = mtry_val,
                                         nodesize = nodesize_val,
                                         OOB_Error = oob_error))
  }
}

# Display results
print(results)
```

```
##       mtry nodesize  OOB_Error
## OOB      5        1 0.06701031
## OOB1     5        5 0.05927835
## OOB2     5       10 0.06185567
## OOB3     5       15 0.06185567
## OOB4    10        1 0.06443299
```

```
## OOB5    10        5 0.06701031
## OOB6    10       10 0.06185567
## OOB7    10       15 0.07216495
## OOB8    15        1 0.06701031
## OOB9    15        5 0.06958763
## OOB10   15       10 0.06185567
## OOB11   15       15 0.07731959
## OOB12   20        1 0.06958763
## OOB13   20        5 0.07216495
## OOB14   20       10 0.06185567
## OOB15   20       15 0.06701031
```

```
# Find the optimal combination with the lowest OOB error
best_params <- results[which.min(results$OOB_Error), ]
print(best_params)
```

```
##      mtry nodesize  OOB_Error
## OOB1    5        5 0.05927835
```

```
# Train the Final Model with Optimal Parameters
optimal_rf <- randomForest(Digit ~ .,
                           data = mnist_train_subset,
                           ntree = 1000,
                           mtry = best_params$mtry,
                           nodesize = best_params$nodesize)

# Predict on test data
predictions <- predict(optimal_rf, mnist_test_subset)

# Confusion matrix
confusion_matrix <- table(predictions, mnist_test_subset$Digit)
print(confusion_matrix)
```

```
##
## predictions   2   4   8
##           2 122   3   6
##           4   5 141   4
##           8   2   2 100
```

```
# Accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Accuracy:", round(accuracy, 4)))
```

```
## [1] "Accuracy: 0.9429"
```

## 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.

**Answer:**

```
library(xgboost)
library(caret)
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:randomForest':
##
##     margin
```

```
## Loading required package: lattice
```

```
# Prepare data for XGBoost
# Convert the Digit column to zero-based indexing (assuming 2 -> 0, 4 -> 1, 8 -> 2)
mnist_train_subset$Digit <- as.numeric(factor(mnist_train_subset$Digit)) - 1
mnist_test_subset$Digit <- as.numeric(factor(mnist_test_subset$Digit)) - 1

# Convert data to DMatrix format
train_matrix <- xgb.DMatrix(data = as.matrix(mnist_train_subset[, -1]),
                            label = mnist_train_subset$Digit)
test_matrix <- xgb.DMatrix(data = as.matrix(mnist_test_subset[, -1]),
                           label = mnist_test_subset$Digit)

# Set parameters for XGBoost
params <- list(
  objective = "multi:softmax",
  num_class = 3,
  booster = "gbtree",
  eta = 0.5,
  max_depth = 2
)
```

```r
# Train the model
nrounds <- 100
xgb_model <- xgboost(params = params,
                     data = train_matrix,
                     nrounds = nrounds,
                     verbose = 0)

# Predict on the test set
test_preds <- predict(xgb_model, test_matrix)

# Calculate confusion matrix and error rate
conf_matrix <- confusionMatrix(as.factor(test_preds), as.factor(mnist_test_subset$Digit))
print(conf_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2
##          0 119   3   1
##          1   5 137   4
##          2   5   6 105
##
## Overall Statistics
##
##                Accuracy : 0.9377
##                  95% CI : (0.9087, 0.9597)
##     No Information Rate : 0.3792
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.906
##
##  Mcnemar's Test P-Value : 0.3122
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2
## Sensitivity            0.9225   0.9384   0.9545
## Specificity            0.9844   0.9623   0.9600
## Pos Pred Value         0.9675   0.9384   0.9052
## Neg Pred Value         0.9618   0.9623   0.9814
## Prevalence             0.3351   0.3792   0.2857
## Detection Rate         0.3091   0.3558   0.2727
## Detection Prevalence   0.3195   0.3792   0.3013
## Balanced Accuracy      0.9534   0.9503   0.9573
```

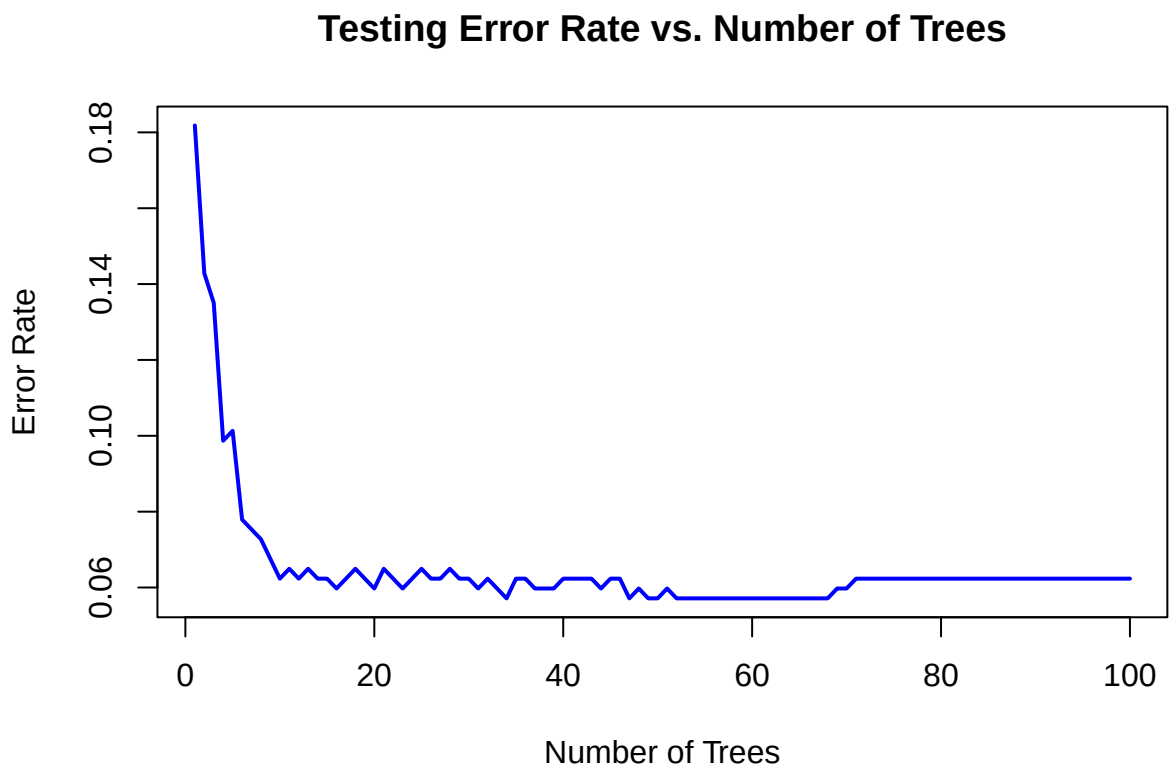
```r
error_rate <- 1 - conf_matrix$overall["Accuracy"]
print(paste("Testing error rate:", round(error_rate, 4)))
```

```
## [1] "Testing error rate: 0.0623"
```

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 iteration_range argument in the `predict()` function. Plot your prediction error vs. number of trees. Comment on your results.

**Answer:**

```r
# Define a vector to store error rates for different numbers of trees
error_rates <- numeric(nrounds)

# Loop over different numbers of trees for prediction
for (num_trees in 1:nrounds) {
  # Predict with a limited number of trees
  test_preds_limited <- predict(xgb_model,
                                test_matrix,
                                iterationrange = c(1, num_trees + 1))

  # Calculate error rate
  error_rates[num_trees] <- mean(test_preds_limited != mnist_test_subset$Digit)
}

# Plot error rate vs. number of trees
plot(1:nrounds, error_rates, type = "l", col = "blue", lwd = 2,
     xlab = "Number of Trees", ylab = "Error Rate",
     main = "Testing Error Rate vs. Number of Trees")
```




The error rate starts quite high with a small number of trees, as expected, since only a few weak learners are available, and their combined predictions are not yet very accurate. As more trees are added, the error rate drops rapidly, indicating that the model is effectively learning from the data and that boosting is improving accuracy. By around 10 trees, the error rate has significantly decreased and is close to its lowest point.

After around 10 trees, the error rate stabilizes and fluctuates within a narrow range, maintaining an error rate around 0.06. This suggests that the model has reached its optimal level of performance by this point, and adding additional trees is not leading to further improvement, indicating that XGBoost has effectively captured the patterns in the data without needing a large number of trees. There's a slight increase in error rate toward the end of the plot, around 80–100 trees. This can be an indication of overfitting, where the model starts to fit the training data too closely and loses generalizability on the test set.

The plot suggests that around 10–20 trees might have smallest error rate and be sufficient for this model to achieve its best performance on the test set. Stopping early in this range could reduce computational costs while preserving accuracy. Using fewer trees can also help avoid the mild overfitting observed at higher numbers of trees, as the error rate starts to plateau and slightly increase around 80–100 trees. The relatively fast convergence of the error rate to a stable level indicates that the learning rate (eta = 0.5) is set high enough for rapid learning but not so high as to cause instability or poor generalization.