Stat 432 Homework 9

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

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.

```
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 4.3.3
```

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

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

```r
set.seed(42)

# Step 1: Generate Training and Test Data
# Training set
n_train <- 200
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)

# Test set
n_test <- 300
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)

# Step 2: Fit the Random Forest Model
rf_model <- randomForest(Y ~ X1 + X2, data = train_data)

# Step 3: Predict on the Test Set and Calculate MSE
predictions <- predict(rf_model, newdata = test_data)
mse <- mean((Y_test - predictions)^2)

# Output the MSE
mse
```

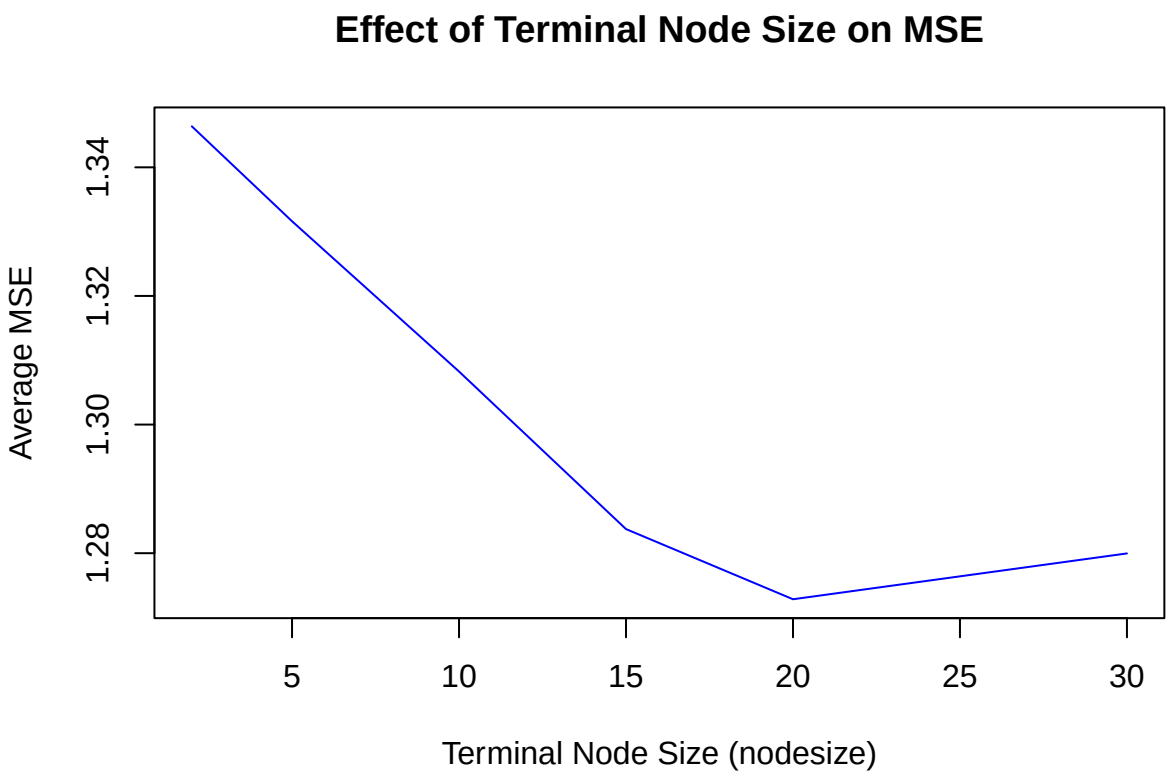
```
## [1] 1.212378
```

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?

```r
nodesize_values <- c(2, 5, 10, 15, 20, 30)
mtry_value <- 1
bootstrap_size <- 150
num_simulations <- 100

set.seed(42)

# Storage for average MSEs
avg_mse <- numeric(length(nodesize_values))
```

```r
# Loop over each nodesize value
for (i in seq_along(nodesize_values)) {
  nodesize <- nodesize_values[i]
  mse_values <- numeric(num_simulations)

  # Repeat simulation for each nodesize
  for (j in 1:num_simulations) {
    # Bootstrap sample from the training set
    bootstrap_indices <- sample(1:n_train, bootstrap_size, replace = TRUE)
    bootstrap_train <- train_data[bootstrap_indices, ]

    # Fit the random forest model
    rf_model <- randomForest(Y ~ X1 + X2, data = bootstrap_train,
                             mtry = mtry_value, nodesize = nodesize)

    # Calculate MSE on test set
    predictions <- predict(rf_model, newdata = test_data)
    mse_values[j] <- mean((Y_test - predictions)^2)
  }

  # Store average MSE for current nodesize
  avg_mse[i] <- mean(mse_values)
}

# Plot average MSE against nodesize
plot(nodesize_values, avg_mse, type = "l", pch = 16, col = "blue",
     xlab = "Terminal Node Size (nodesize)", ylab = "Average MSE",
     main = "Effect of Terminal Node Size on MSE")
```




```
# Output average MSE for analysis
avg_mse
```

```
## [1] 1.346370 1.331613 1.308251 1.283750 1.272843 1.279963
```

The range of node size seems to be reasonable, and the best node size appears to be 20 since it has the smallest mse. For nodesizes,Smaller nodesize decreases bias but increases variance. Larger nodesize increases bias but decreases variance. Optimal nodesize balances the trade-off, resulting in a model that performs well on both training and test data.

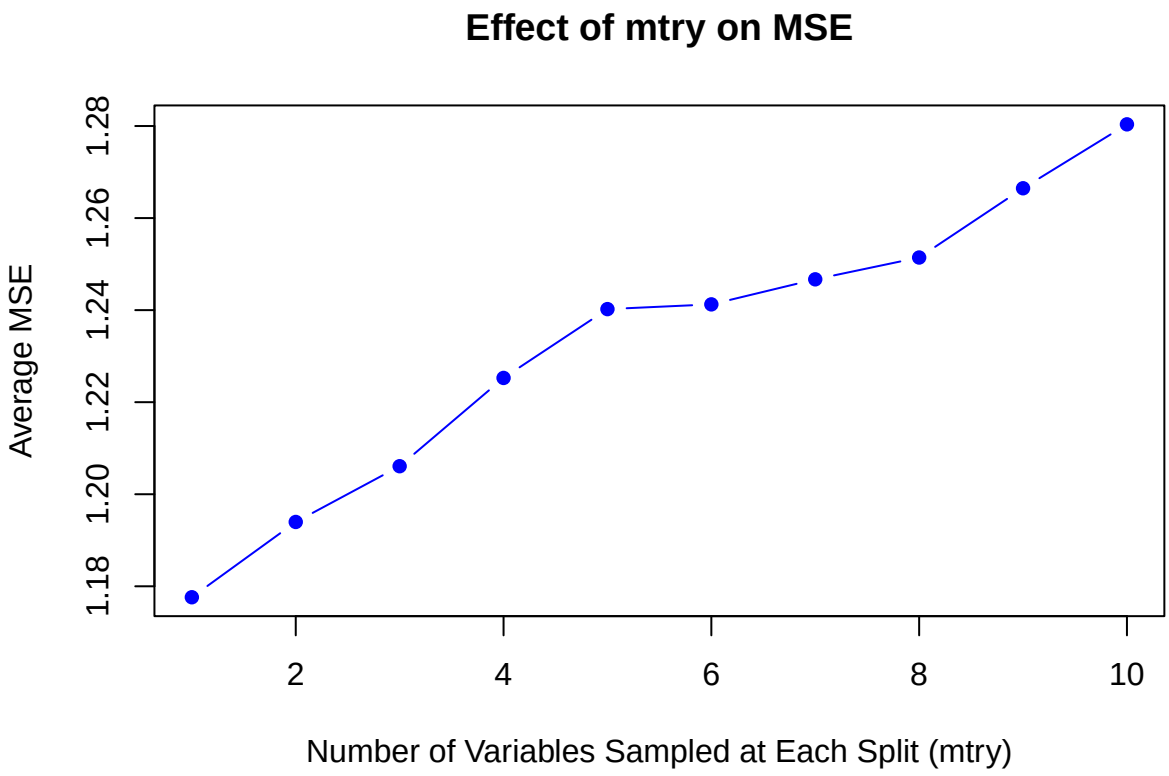
c. [15 pts] In this question, let's analyze the effect of `mtry`. We will consider a new data generator:

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

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?

```r
mtry_values <- 1:10
nodesize <- 3
bootstrap_size <- 150
num_simulations <- 100

# Set seed for reproducibility
set.seed(42)

# Generate data
n_train <- 200
n_test <- 300
X_train <- replicate(10, rnorm(n_train))
X_test <- replicate(10, rnorm(n_test))
epsilon_train <- rnorm(n_train)
epsilon_test <- rnorm(n_test)
Y_train <- 0.2 * rowSums(X_train[, 1:5]) + epsilon_train
Y_test <- 0.2 * rowSums(X_test[, 1:5]) + epsilon_test

train_data <- as.data.frame(cbind(X_train, Y = Y_train))
test_data <- as.data.frame(cbind(X_test, Y = Y_test))

# Storage for average MSEs
avg_mse <- numeric(length(mtry_values))

# Loop over each mtry value
for (i in seq_along(mtry_values)) {
  mtry <- mtry_values[i]
  mse_values <- numeric(num_simulations)

  # Repeat simulation for each mtry
  for (j in 1:num_simulations) {
    # Bootstrap sample from the training set
    bootstrap_indices <- sample(1:n_train, bootstrap_size, replace = TRUE)
    bootstrap_train <- train_data[bootstrap_indices, ]

    # Fit the random forest model
    rf_model <- randomForest(Y ~ ., data = bootstrap_train,
                             mtry = mtry, nodesize = nodesize)

    # Calculate MSE on the test set
    predictions <- predict(rf_model, newdata = test_data)
    mse_values[j] <- mean((Y_test - predictions)^2)
  }

  # Store average MSE for current mtry
  avg_mse[i] <- mean(mse_values)
}

# Plot average MSE against mtry
plot(mtry_values, avg_mse, type = "b", pch = 16, col = "blue",
     xlab = "Number of Variables Sampled at Each Split (mtry)", ylab = "Average MSE",
     main = "Effect of mtry on MSE")
```




```
# Output average MSE for analysis
avg_mse
```

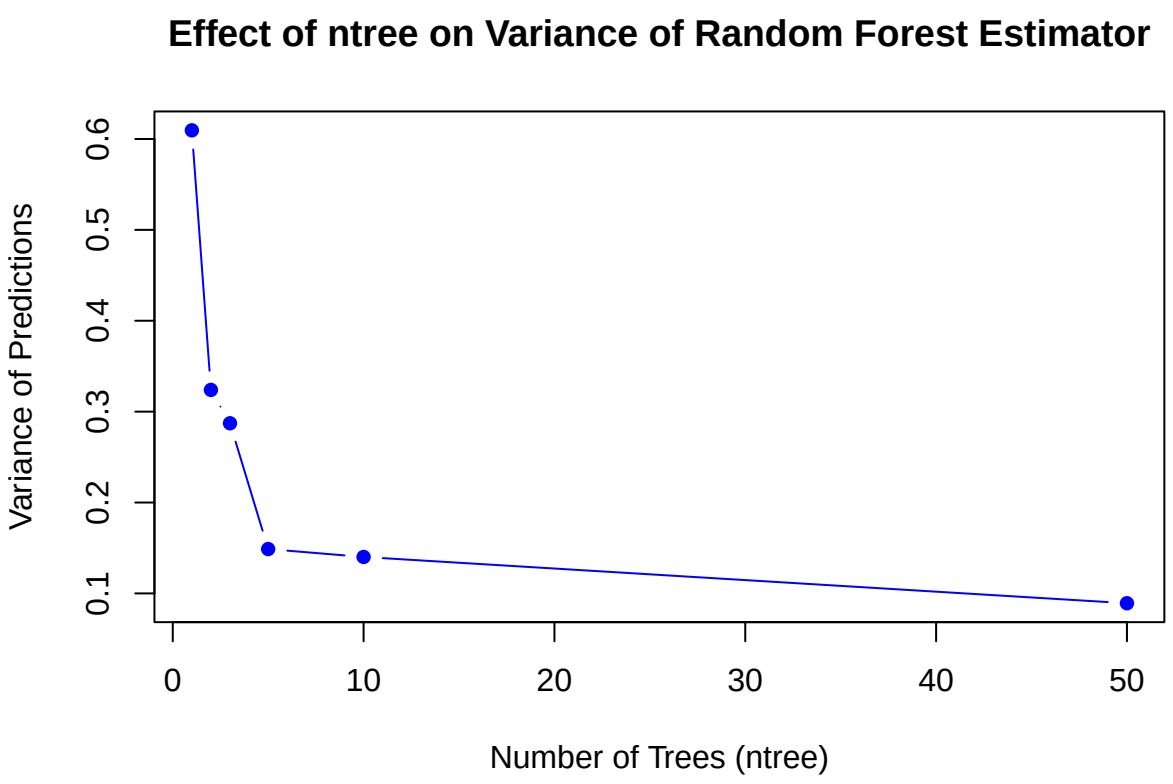
```
##  [1] 1.177618 1.193969 1.206086 1.225265 1.240217 1.241247 1.246691 1.251431
##  [9] 1.266474 1.280361
```

The optimal value i fitted is 1. Smaller mtry increases bias but reduces variance. Larger mtry reduces bias but increases variance. Optimal mtry balances bias and variance, resulting in the best generalization performance on unseen data.

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.

```
ntree_values <- c(1, 2, 3, 5, 10, 50)
nodesize <- 10
mtry <- 3
bootstrap_size <- 150
num_simulations <- 100
```

```r
set.seed(42)

target_point <- as.data.frame(matrix(0, nrow = 1, ncol = 10))
names(target_point) <- paste0("X", 1:10)
variance_results <- numeric(length(ntree_values))

for (i in seq_along(ntree_values)) {
  ntree <- ntree_values[i]
  predictions <- numeric(num_simulations)

  for (j in 1:num_simulations) {
   X_train <- as.data.frame(replicate(10, rnorm(200)))
names(X_train) <- paste0("X", 1:10)
epsilon_train <- rnorm(200)
Y_train <- 0.2 * rowSums(X_train[, 1:5]) + epsilon_train
train_data <- cbind(X_train, Y = Y_train)

# Define target point with matching column names
target_point <- as.data.frame(matrix(0, nrow = 1, ncol = 10))
names(target_point) <- names(X_train)
    bootstrap_indices <- sample(1:200, bootstrap_size, replace = TRUE)
    bootstrap_train <- train_data[bootstrap_indices, ]
    rf_model <- randomForest(Y ~ ., data = bootstrap_train,
                             mtry = mtry, nodesize = nodesize, ntree = ntree)
    predictions[j] <- predict(rf_model, newdata = target_point)
  }

  variance_results[i] <- var(predictions)
}

# Plot variance of predictions against ntree
plot(ntree_values, variance_results, type = "b", pch = 16, col = "blue",
     xlab = "Number of Trees (ntree)", ylab = "Variance of Predictions",
     main = "Effect of ntree on Variance of Random Forest Estimator")
```




```
# Output variance results for analysis
variance_results
```

```
## [1] 0.60948930 0.32390646 0.28720515 0.14878621 0.14010206 0.08919135
```

Higher ntree reduce variance and increase bias in terms of tradeoffs. # 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.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::combine()  masks randomForest::combine()
## x dplyr::filter()   masks stats::filter()
## x dplyr::lag()      masks stats::lag()
## x ggplot2::margin() masks randomForest::margin()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
numColsMnist <- dim(mnist2600)[2]
colnames(mnist2600) <- c("Digit", paste("Pixel", seq(1:(numColsMnist - 1)), sep = ""))

# Split data into training and test sets
train_data <- mnist2600[1:1300, ]
test_data <- mnist2600[1301:2600, ]

# Filter to keep only digits 2, 4, and 8
train_data <- train_data %>% filter(Digit %in% c(2, 4, 8))
test_data <- test_data %>% filter(Digit %in% c(2, 4, 8))

# Calculate the variance for each pixel column
pixel_variances <- sapply(train_data[ , -1], var)

# Select the top 250 pixel columns with the highest variance
top_250_pixels <- names(sort(pixel_variances, decreasing = TRUE))[1:250]

# Keep only the top 250 pixels and the Digit column in the training and test sets
train_data <- train_data %>% select(Digit, all_of(top_250_pixels))
test_data <- test_data %>% select(Digit, all_of(top_250_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.

```r
library(caret)
```

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

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```r
# Convert the response variable to a factor
train_data$Digit <- as.factor(train_data$Digit)
test_data$Digit <- as.factor(test_data$Digit)

# Define the tuning grid for mtry and nodesize
mtry_values <- c(10, 30, 50, 70)  # Example values, can be adjusted
nodesize_values <- c(1, 5, 10, 15)  # Example values, can be adjusted
ntree <- 1000

# Storage for OOB error results
tuning_results <- expand.grid(mtry = mtry_values, nodesize = nodesize_values)
tuning_results$OOB_Error <- NA  # To store OOB errors

# Loop through each combination of mtry and nodesize
for (i in 1:nrow(tuning_results)) {
  mtry_val <- tuning_results$mtry[i]
  nodesize_val <- tuning_results$nodesize[i]

  # Fit random forest with specified mtry and nodesize
  rf_model <- randomForest(Digit ~ ., data = train_data,
                           mtry = mtry_val, nodesize = nodesize_val,
                           ntree = ntree)

  # Check if err.rate exists and record the OOB error if it does
  if (!is.null(rf_model$err.rate)) {
    tuning_results$OOB_Error[i] <- rf_model$err.rate[ntree, "OOB"]
  } else {
    tuning_results$OOB_Error[i] <- NA  # Assign NA if no error rate is available
  }
}

# Find the optimal parameters (lowest OOB error)
optimal_params <- tuning_results[which.min(tuning_results$OOB_Error), ]

# Print tuning results
print(tuning_results)
```

```
##    mtry nodesize  OOB_Error
## 1    10        1 0.06443299
## 2    30        1 0.06701031
## 3    50        1 0.06701031
```

```
## 4    70        1 0.06443299
## 5    10        5 0.06443299
## 6    30        5 0.07731959
## 7    50        5 0.06958763
## 8    70        5 0.06958763
## 9    10       10 0.06958763
## 10   30       10 0.06701031
## 11   50       10 0.08247423
## 12   70       10 0.06443299
## 13   10       15 0.06958763
## 14   30       15 0.06443299
## 15   50       15 0.08505155
## 16   70       15 0.07989691
```

```
print(optimal_params)
```

```
##   mtry nodesize  OOB_Error
## 1   10        1 0.06443299
```

```
# Fit final random forest model with optimal parameters
final_rf_model <- randomForest(Digit ~ ., data = train_data,
                               mtry = optimal_params$mtry,
                               nodesize = optimal_params$nodesize,
                               ntree = ntree)

# Predict on test data
test_predictions <- predict(final_rf_model, newdata = test_data)

# Calculate confusion matrix and accuracy
confusion_mat <- confusionMatrix(test_predictions, test_data$Digit)
print(confusion_mat)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   2   4   8
##          2 122   3   2
##          4   4 141   3
##          8   3   2 105
##
## Overall Statistics
##
##                Accuracy : 0.9558
##                  95% CI : (0.9302, 0.9741)
##     No Information Rate : 0.3792
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9333
##
##  Mcnemar's Test P-Value : 0.9094
##
## Statistics by Class:
##
```

```
##                      Class: 2 Class: 4 Class: 8
## Sensitivity            0.9457   0.9658   0.9545
## Specificity            0.9805   0.9707   0.9818
## Pos Pred Value         0.9606   0.9527   0.9545
## Neg Pred Value         0.9729   0.9789   0.9818
## Prevalence             0.3351   0.3792   0.2857
## Detection Rate         0.3169   0.3662   0.2727
## Detection Prevalence   0.3299   0.3844   0.2857
## Balanced Accuracy      0.9631   0.9682   0.9682
```

```r
# Output accuracy
accuracy <- confusion_mat$overall['Accuracy']
print(accuracy)
```

```
##  Accuracy
## 0.9558442
```

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.

```r
library(xgboost)
```

```
## Warning: package 'xgboost' was built under R version 4.3.3
```

```
##
## Attaching package: 'xgboost'
```

```
## The following object is masked from 'package:dplyr':
##
##     slice
```

```r
# Convert training and test data to matrix format required by xgboost
train_matrix <- as.matrix(train_data[ , -1])  # Remove Digit column for features
train_labels <- as.numeric(as.factor(train_data$Digit)) - 1  # Convert labels to 0, 1, 2
test_matrix <- as.matrix(test_data[ , -1])
test_labels <- as.numeric(as.factor(test_data$Digit)) - 1

# Define xgboost parameters
```

```r
params <- list(
  objective = "multi:softmax",
  num_class = 3,
  booster = "gbtree",
  eta = 0.5,
  max_depth = 2
)

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

# Predict on the test set
test_predictions <- predict(xgb_model, newdata = test_matrix)

# Calculate testing error rate
error_rate <- mean(test_predictions != test_labels)
print(paste("Testing Error Rate:", round(error_rate * 100, 2), "%"))
```

```
## [1] "Testing Error Rate: 6.23 %"
```

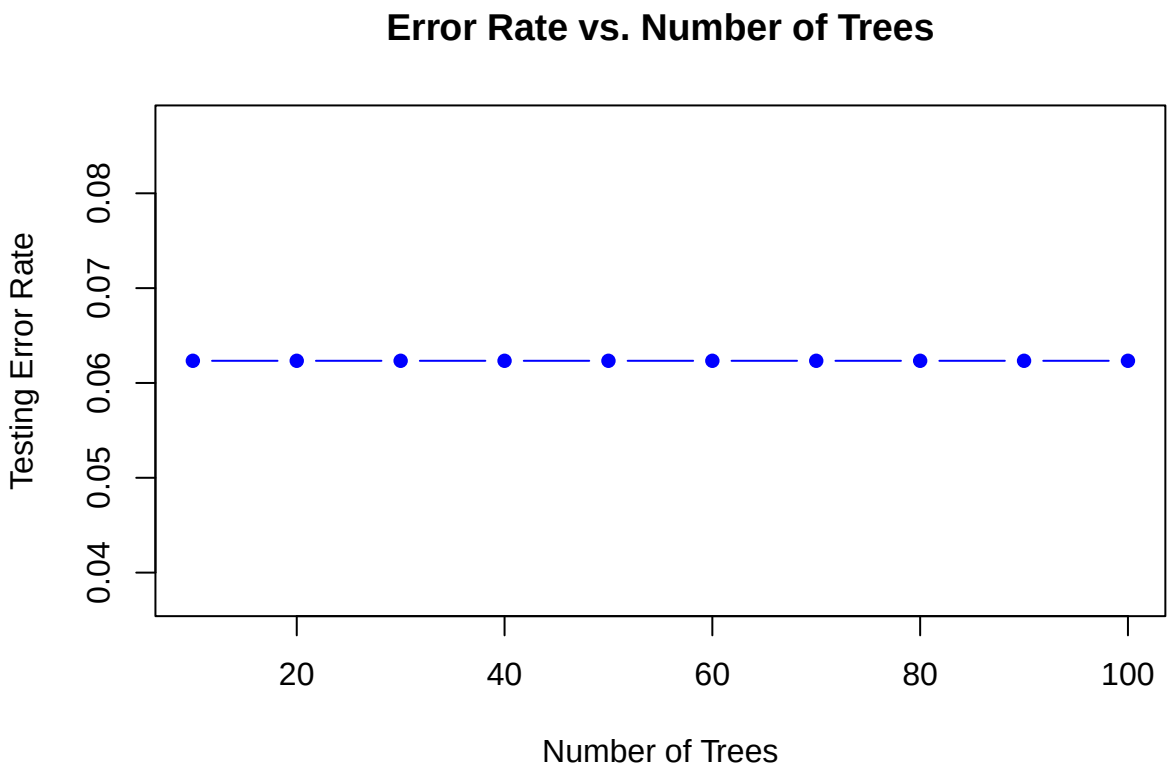
```r
# Confusion matrix
confusion_mat <- confusionMatrix(factor(test_predictions), factor(test_labels))
print(confusion_mat)
```

```
## 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
```

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.

```r
# Define a sequence of tree counts to evaluate
tree_counts <- seq(10, 100, by = 10)
error_rates <- numeric(length(tree_counts))

# Loop over different numbers of trees
for (i in seq_along(tree_counts)) {
  n_trees <- tree_counts[i]

  # Generate predictions using only the first n_trees trees
  predictions <- predict(xgb_model, newdata = test_matrix, iteration_range = c(1, n_trees))

  # Calculate error rate
  error_rates[i] <- mean(predictions != test_labels)
}

# Plot error rate vs number of trees
plot(tree_counts, error_rates, type = "b", col = "blue", pch = 16,
     xlab = "Number of Trees", ylab = "Testing Error Rate",
     main = "Error Rate vs. Number of Trees")
```




```
# Output error rates for analysis
error_rates
```

```
##  [1] 0.06233766 0.06233766 0.06233766 0.06233766 0.06233766 0.06233766
##  [7] 0.06233766 0.06233766 0.06233766 0.06233766
```