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.

```
# Load the randomForest package
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 for reproducibility
set.seed(123)

# Generate training data
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(Y = Y_train, X1 = X1_train, X2 = X2_train)

# Generate test data
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(Y = Y_test, X1 = X1_test, X2 = X2_test)

# Fit a random forest model with default parameters
rf_model <- randomForest(Y ~ X1 + X2, data = train_data)

# Predict on the test set
predictions <- predict(rf_model, newdata = test_data)

# Calculate the Mean Squared Error (MSE)
mse <- mean((test_data$Y - predictions)^2)
cat("MSE on the test set:", mse, "\n")
```

```
## MSE on the test set: 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:

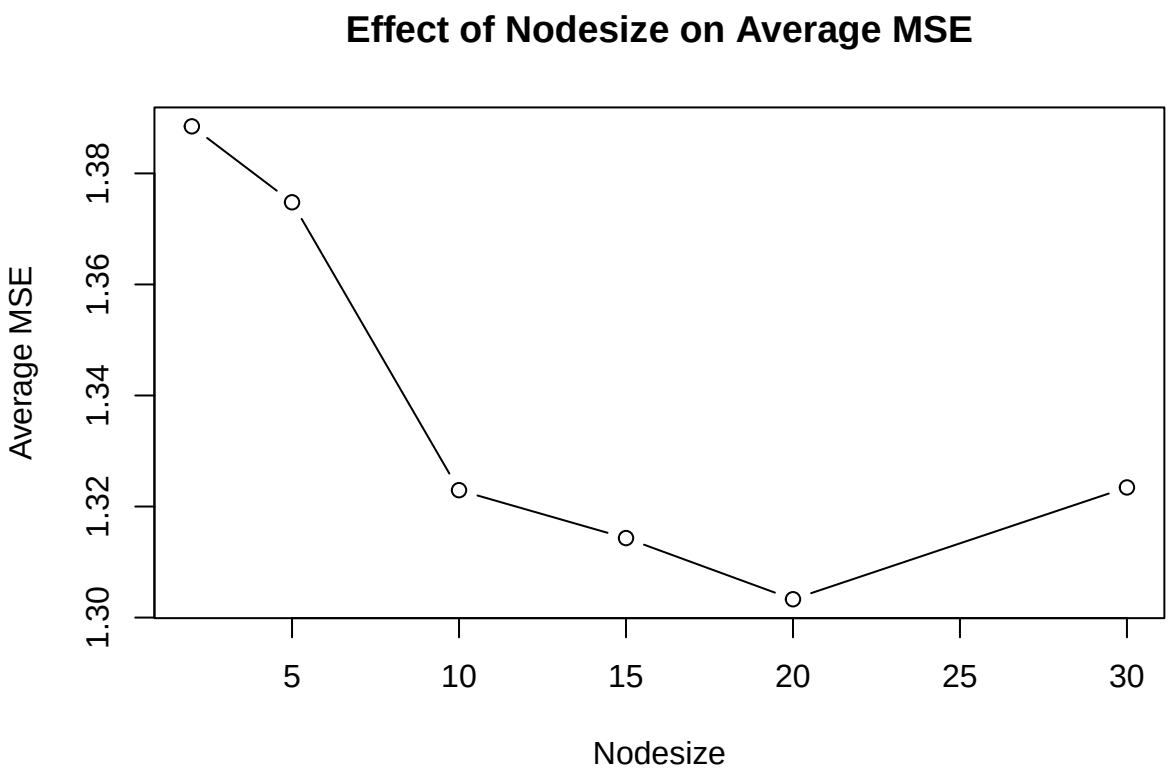
- Do you think our choice of the `nodesize` parameter is reasonable? What is the optimal node size you obtained? If you don't think the choice is reasonable, re-define your range of tuning and report your results and the optimal node size.
- What is the effect of `nodesize` on the bias-variance trade-off?

```r
# Load the randomForest package
library(randomForest)

# Set seed for reproducibility
set.seed(123)

# Parameters
nodesize_values <- c(2, 5, 10, 15, 20, 30)
```

```r
num_simulations <- 100
mtry_val <- 1
bootstrap_size <- 150

# Initialize storage for average MSE results
avg_mse_results <- numeric(length(nodesize_values))

# Loop over each nodesize value
for (i in seq_along(nodesize_values)) {

  # Set current nodesize
  nodesize_val <- nodesize_values[i]

  # Collect MSE for current nodesize across simulations
  mse_list <- numeric(num_simulations)

  for (j in 1:num_simulations) {

    # Bootstrap sample from training data
    train_indices <- sample(1:nrow(train_data), bootstrap_size, replace = TRUE)
    train_bootstrap <- train_data[train_indices, ]

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

    # Predict on the test set and calculate MSE
    predictions <- predict(rf_model, newdata = test_data)
    mse_list[j] <- mean((test_data$Y - predictions)^2)
  }

  # Store the average MSE for the current nodesize
  avg_mse_results[i] <- mean(mse_list)
}

avg_mse_results
```

```
## [1] 1.388473 1.374796 1.322944 1.314318 1.303302 1.323437
```

```r
# Plot the results
plot(nodesize_values, avg_mse_results, type = "b",
     xlab = "Nodesize", ylab = "Average MSE",
     main = "Effect of Nodesize on Average MSE")
```




```
# Analysis Questions:
# - What is the optimal `nodesize`?
optimal_nodesize <- nodesize_values[which.min(avg_mse_results)]
optimal_nodesize
```

```
## [1] 20
```

- The choice of the `nodesize` parameters reasonable. The optimal node size is 20 since it yields the lowest average MSE and the trend is very clear. This indicates that a nodesize of 20 is indeed a reasonable choice for this model, as it minimizes the prediction error on the test set, balancing the model's fit to the training data without obvious overfitting or underfitting.
- The effect of `nodesize` on the bias-variance trade-off:
  - Smaller Nodesize (Higher Variance, Lower Bias): When nodesize is small (e.g., 2 or 5), each tree can grow deeper and capture more intricate details of the data, reducing bias by allowing the model greater flexibility to fit complex patterns. However, this increased flexibility also raises variance, as each tree may overfit its specific bootstrap sample, resulting in higher Mean Squared Error (MSE).
  - Larger Nodesize (Lower Variance, Higher Bias): As nodesize increases (e.g., 25 or 30), the trees become shallower, limiting their flexibility. This increased bias means the model becomes less capable of capturing finer details, potentially leading to underfitting. However, the reduced flexibility also lowers variance, making each tree less sensitive to the unique characteristics of its bootstrap sample, which results in a slight increase in MSE.
  - Optimal Nodesize (Balance Between Bias and Variance): The lowest MSE observed at nodesize = 20 suggests this as the optimal point for balancing bias and variance, where the random

forest model achieves the best predictive performance by maintaining sufficient flexibility while minimizing overfitting.

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:

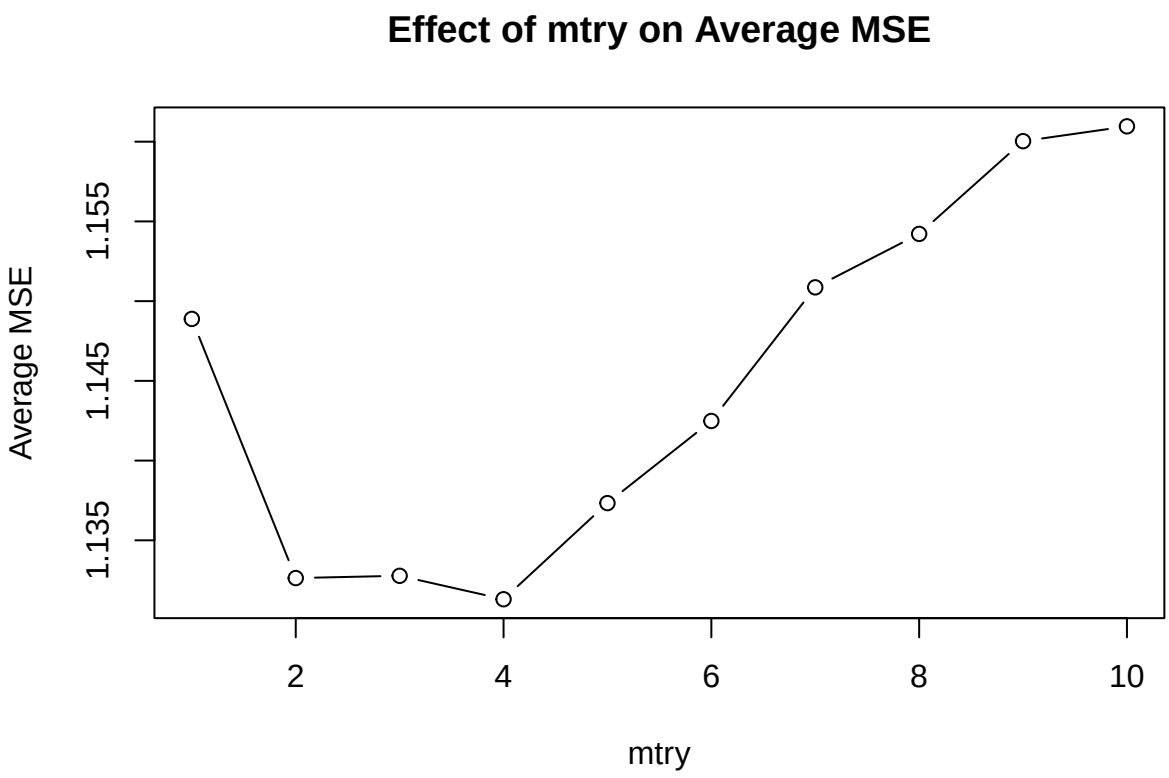
```
    * What is the optimal value of `mtry` you obtained?
    * What is the effect of `mtry` on the bias-variance trade-off?
```

```r
# Load the randomForest package
library(randomForest)

# Set seed for reproducibility
set.seed(123)

# Parameters
mtry_values <- 1:10
num_simulations <- 100
nodesize_val <- 3
bootstrap_size <- 150

# Initialize storage for average MSE results
avg_mse_results <- numeric(length(mtry_values))

# Generate training data
n_train <- 200
X_train <- matrix(rnorm(n_train * 10), n_train, 10)
epsilon_train <- rnorm(n_train)
Y_train <- 0.2 * rowSums(X_train[, 1:5]) + epsilon_train
train_data <- data.frame(Y = Y_train, X_train)

# Generate test data
n_test <- 300
X_test <- matrix(rnorm(n_test * 10), n_test, 10)
epsilon_test <- rnorm(n_test)
Y_test <- 0.2 * rowSums(X_test[, 1:5]) + epsilon_test
test_data <- data.frame(Y = Y_test, X_test)

# Loop over each mtry value
for (i in seq_along(mtry_values)) {

  # Set current mtry
  mtry_val <- mtry_values[i]
```

```r
  # Collect MSE for current mtry across simulations
  mse_list <- numeric(num_simulations)

  for (j in 1:num_simulations) {

    # Bootstrap sample from training data
    train_indices <- sample(1:nrow(train_data), bootstrap_size, replace = TRUE)
    train_bootstrap <- train_data[train_indices, ]

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

    # Predict on the test set and calculate MSE
    predictions <- predict(rf_model, newdata = test_data)
    mse_list[j] <- mean((test_data$Y - predictions)^2)
  }

  # Store the average MSE for the current mtry
  avg_mse_results[i] <- mean(mse_list)
}

mtry_values
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
# Plot the results
plot(mtry_values, avg_mse_results, type = "b",
     xlab = "mtry", ylab = "Average MSE",
     main = "Effect of mtry on Average MSE")
```




```
# Analysis Questions:
# - What is the optimal `mtry`?
optimal_mtry <- mtry_values[which.min(avg_mse_results)]
optimal_mtry
```

```
## [1] 4
```

- The optimal value of `mtry` is 4, since it yields the lowest average MSE.
- Effect of `mtry` on the bias-variance trade-off:
  - Lower mtry (Higher Variance, Lower Bias): With a smaller mtry (e.g., 1 or 2), fewer features are considered at each split, making individual trees more varied and unique. This increases variance as trees capture different parts of the data structure independently, reducing overall bias and allowing the model to fit diverse patterns.
  - Higher mtry (Lower Variance, Higher Bias): As mtry increases (e.g., 9 or 10), more features are used at each split, making trees more similar to each other. This consistency reduces variance, as trees are less sensitive to individual data variations, but increases bias by limiting the model's flexibility.
  - Optimal mtry (Balanced Bias and Variance): An mtry value around 4 strikes a balance between bias and variance, yielding the lowest MSE. This setting allows enough randomness to avoid overfitting while keeping the model stable and consistent 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

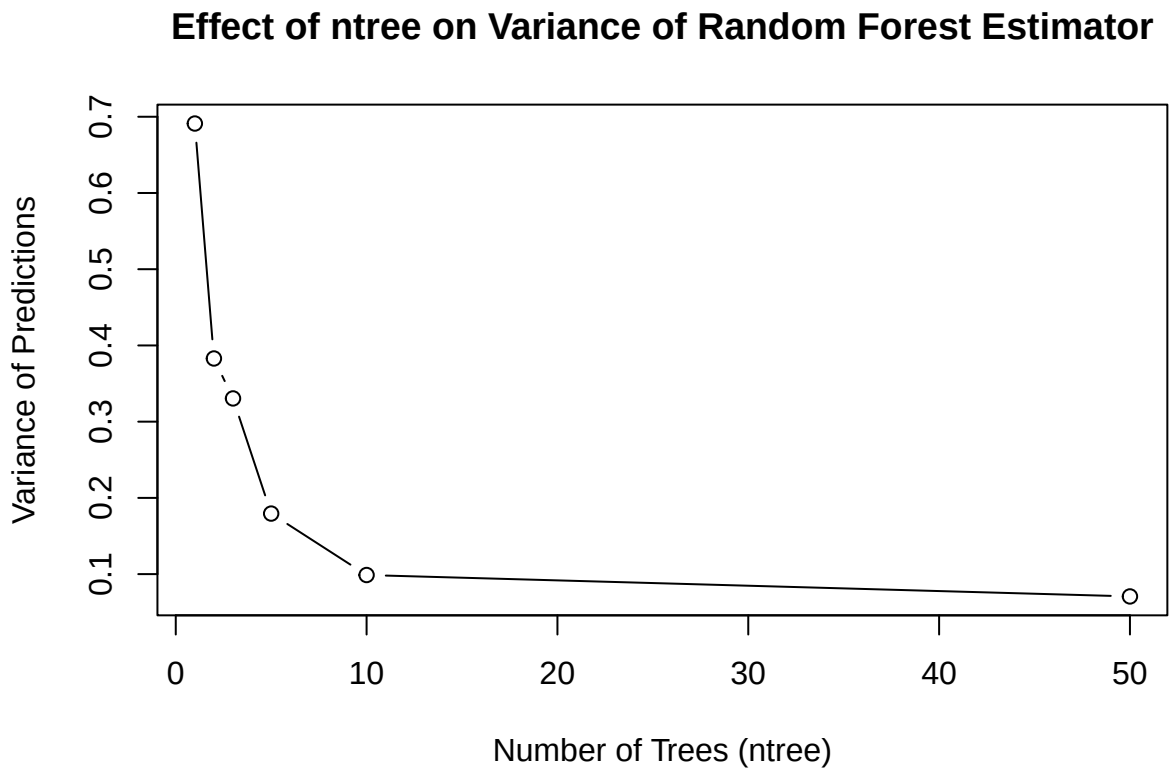
the following values for `ntree`: 1, 2, 3, 5, 10, 50. Perform the simulation study with 100 runs. For this question, we do not need to calculate the prediction of all subjects. Instead, calculate just the prediction on a target point that all the covariate values are 0. After obtaining the simulation 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.

```r
# Load the randomForest package
library(randomForest)

# Set seed for reproducibility
set.seed(123)

# Parameters
ntree_values <- c(1, 2, 3, 5, 10, 50)
num_simulations <- 100
nodesize_val <- 10
mtry_val <- 3
bootstrap_size <- 150

# Initialize storage for variance of predictions
variance_results <- numeric(length(ntree_values))

# Generate training data
n_train <- 200
X_train <- matrix(rnorm(n_train * 10), n_train, 10)
epsilon_train <- rnorm(n_train)
Y_train <- 0.2 * rowSums(X_train[, 1:5]) + epsilon_train
train_data <- data.frame(Y = Y_train, X_train)

# Define the target point with all covariates set to 0
target_point <- as.data.frame(matrix(0, nrow = 1, ncol = 10))
colnames(target_point) <- paste0("X", 1:10)

# Loop over each ntree value
for (i in seq_along(ntree_values)) {

  # Set current ntree
  ntree_val <- ntree_values[i]

  # Collect predictions for the target point across simulations
  predictions <- numeric(num_simulations)

  for (j in 1:num_simulations) {

    # Bootstrap sample from training data
    train_indices <- sample(1:nrow(train_data), bootstrap_size, replace = TRUE)
    train_bootstrap <- train_data[train_indices, ]

    # Fit the random forest model
    rf_model <- randomForest(
      Y ~ ., data = train_bootstrap,
      mtry = mtry_val, nodesize = nodesize_val, ntree = ntree_val
    )
```

```
    # Predict on the target point
    predictions[j] <- predict(rf_model, newdata = target_point)
  }

  # Calculate variance of predictions for the current ntree
  variance_results[i] <- var(predictions)
}

ntree_values
```

```
## [1]  1  2  3  5 10 50
```

```
# Plot the results
plot(ntree_values, variance_results, type = "b",
     xlab = "Number of Trees (ntree)", ylab = "Variance of Predictions",
     main = "Effect of ntree on Variance of Random Forest Estimator")
```




```
# Analysis of Findings
variance_results
```

```
## [1] 0.69111039 0.38287963 0.33049965 0.17942072 0.09881207 0.07074808
```

As ntree increases, the prediction variance decreases significantly (and bias increases), which shous the tradeoff betwwen variance and bias.With only a few trees, variance is high due to prediction instability.

However, as ntree grows, variance drops quickly and becomes low by around 10 trees. This shows a key advantage of random forests: adding more trees reduces prediction variance without increasing bias, as each tree helps average out noise, leading to more stable predictions. Setting ntree to 10 or more generally achieves low variance, but higher values can offer minor additional reductions if resources allow.

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.

```
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.

```
library(randomForest)
library(caret)
```

```
## Loading required package: ggplot2

##
## Attaching package: 'ggplot2'

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

## Loading required package: lattice
```

```
# Convert the response variable to a factor
mnist_train_subset$Digit <- as.factor(mnist_train_subset$Digit)
mnist_test_subset$Digit <- as.factor(mnist_test_subset$Digit)

# Define the tuning grid for mtry and nodesize
mtry_values <- c(10, 30, 50, 70)
nodesize_values <- c(1, 5, 10, 15)
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 = mnist_train_subset,
    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
  }
}

# Find the optimal parameters (lowest OOB error)
optimal_params <- tuning_results[which.min(tuning_results$OOB_Error), ]
print("Tuning Results:")
```

```
## [1] "Tuning Results:"
```

```
print(tuning_results)
```

```
##    mtry nodesize  OOB_Error
## 1    10        1 0.06958763
## 2    30        1 0.06958763
## 3    50        1 0.07216495
## 4    70        1 0.07474227
## 5    10        5 0.06443299
## 6    30        5 0.06701031
## 7    50        5 0.06185567
## 8    70        5 0.07216495
## 9    10       10 0.06701031
## 10   30       10 0.07731959
## 11   50       10 0.06701031
## 12   70       10 0.07216495
## 13   10       15 0.06958763
## 14   30       15 0.07474227
## 15   50       15 0.08247423
## 16   70       15 0.07474227
```

```
print("Optimal Parameters:")
```

```
## [1] "Optimal Parameters:"
```

```
print(optimal_params)
```

```
##   mtry nodesize  OOB_Error
## 7   50        5 0.06185567
```

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

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

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

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   2   4   8
##          2 121   2   3
```

```
##          4   5 142   4
##          8   3   2 103
##
## Overall Statistics
##
##                Accuracy : 0.9506
##                  95% CI : (0.924, 0.97)
##     No Information Rate : 0.3792
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9254
##
##  Mcnemar's Test P-Value : 0.5823
##
## Statistics by Class:
##
##                      Class: 2 Class: 4 Class: 8
## Sensitivity            0.9380   0.9726   0.9364
## Specificity            0.9805   0.9623   0.9818
## Pos Pred Value         0.9603   0.9404   0.9537
## Neg Pred Value         0.9691   0.9829   0.9747
## Prevalence             0.3351   0.3792   0.2857
## Detection Rate         0.3143   0.3688   0.2675
## Detection Prevalence   0.3273   0.3922   0.2805
## Balanced Accuracy      0.9592   0.9675   0.9591
```

```r
# Output accuracy
accuracy <- confusion_mat$overall['Accuracy']
cat("Accuracy: ", accuracy)
```

```
## Accuracy:  0.9506494
```

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
# Load required libraries
library(xgboost)
```

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

```
library(caret)
# 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
)
# 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
confusion_matrix <- confusionMatrix(as.factor(test_preds), as.factor(mnist_test_subset$Digit))
print(confusion_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:
##
```

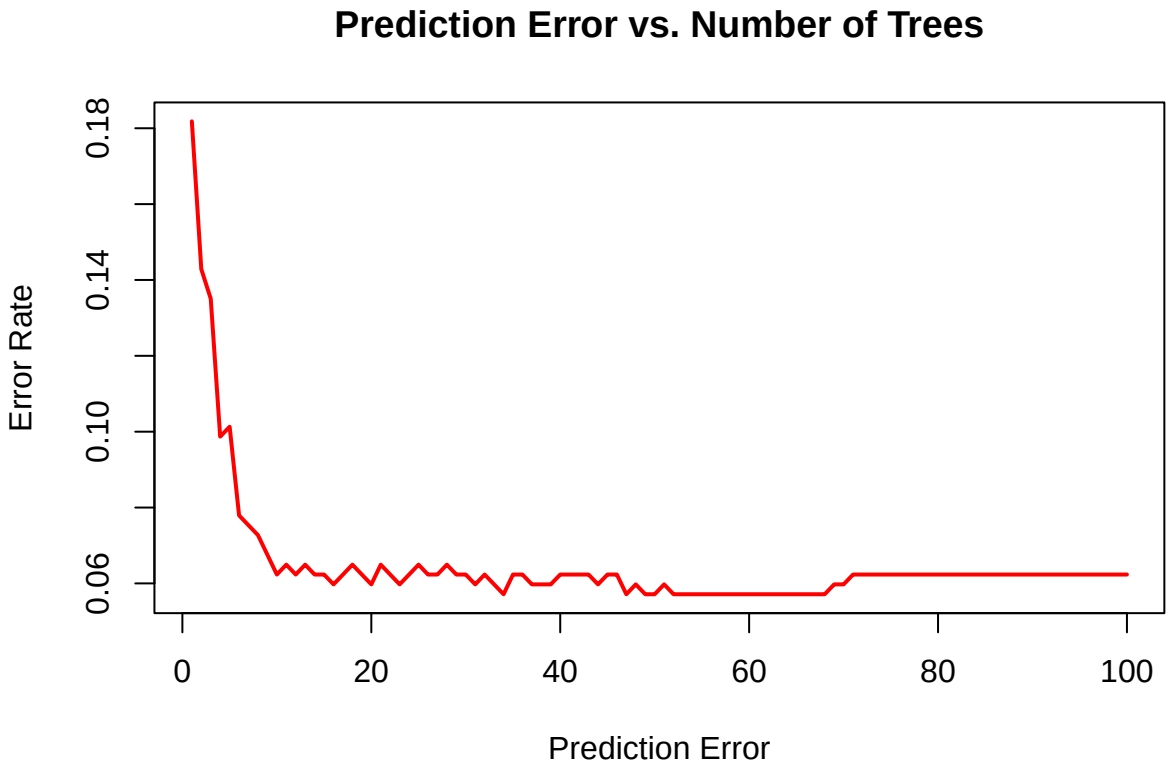
```
##                      Class: 0 Class: 1 Class: 2
## Sensitivity            0.9225   0.9384   0.9545
## Specificity            0.9844   0.9623   0.9600
## Pos Pred Value         0.9675   0.9384   0.9052
## Neg Pred Value         0.9618   0.9623   0.9814
## Prevalence             0.3351   0.3792   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
# Calculate the testing error rate
error_rate <- 1 - confusion_matrix$overall["Accuracy"]
cat("Testing error rate:", error_rate)
```

```
## Testing error rate: 0.06233766
```

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 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 = "red", lwd = 2,
xlab = "Prediction Error", ylab = "Error Rate",
main = "Prediction Error vs. Number of Trees")
```



As the number of trees increases, there is a rapid decrease in error rate, which quickly stabilizes after around 10–20 trees, reaching a plateau near an error rate of 0.06. This indicates that a relatively small number of trees is sufficient to achieve low prediction error, and adding more trees beyond this point yields diminishing returns in terms of error reduction.

Initially, with a few trees, the error rate is high due to the limited number of weak learners, which are not yet accurate enough. As more trees are added, the model effectively learns from the data, and the error rate drops quickly, showing the model's efficiency in capturing patterns.

Beyond approximately 20 trees, the error rate fluctuates within a narrow range, indicating that the model has reached an optimal level of performance. Adding more trees does not significantly improve accuracy and might even lead to a slight increase in error towards the end of the plot, between 80–100 trees. This slight increase could be a sign of overfitting, where the model becomes too tailored to the training data and loses generalizability on the test set.

The plot suggests that using around 10–20 trees provides a good balance between model complexity and prediction accuracy, minimizing computational cost without sacrificing performance. This efficient convergence indicates that the learning rate is appropriately set, allowing the model to learn quickly without compromising generalization.