Lab Report: Computer lab 1 block 1

Group_A10

2024-11-24

Assignment 1: Handwritten Digit Recognition with K-Nearest Neighbors

- **Training Misclassification Error**: 0.0429095
- **Test Misclassification Error**: 0.0324607

Confusion Matrix: Test Data

```
      Predicted
Actual  0  1  2  3  4  5  6  7  8  9
     0 97  0  0  0  0  0  0  0  0  0
     1  0 97  0  0  0  0  0  0  0  0
     2  0  0 98  0  1  0  0  0  0  0
     3  0  0  1 99  0  2  0  1  0  0
     4  0  0  0  0 88  0  0  2  0  0
     5  0  0  0  1  1 82  0  1  0  2
     6  0  1  0  0  0  0 95  0  0  0
     7  0  0  0  0  1  0  0 83  0  1
     8  0  9  0  0  0  0  0  0 94  1
     9  0  0  0  3  1  1  0  1  0 91
```

Confusion Matrix: Training Data

```
      Predicted
Actual   0   1   2   3   4   5   6   7   8   9
     0 196   0   0   0   0   0   0   0   0   0
     1   0 175   8   0   0   0   0   0   1   4
     2   1   0 185   0   0   0   0   1   0   0
     3   0   0   0 179   0   1   0   3   1   0
     4   0   0   0   0 189   0   2   4   2   4
     5   1   0   0   0   0 188   0   1   0   7
     6   1   1   0   0   0   1 178   0   0   0
     7   0   1   1   1   0   0   0 198   0   0
     8   0   9   0   3   0   1   2   0 165   0
     9   1   2   0   5   1   1   0   5   5 176
```

Summary of Prediction Quality

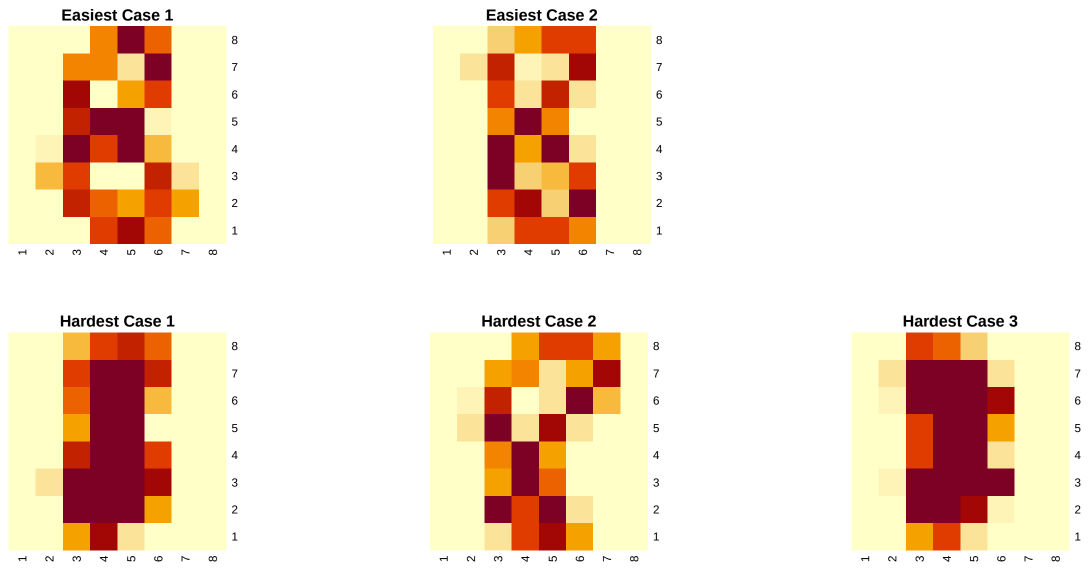
Key Results:

- **Training Error**: 4.36%
- **Test Error**: 4.62%
- Strong generalization across datasets.

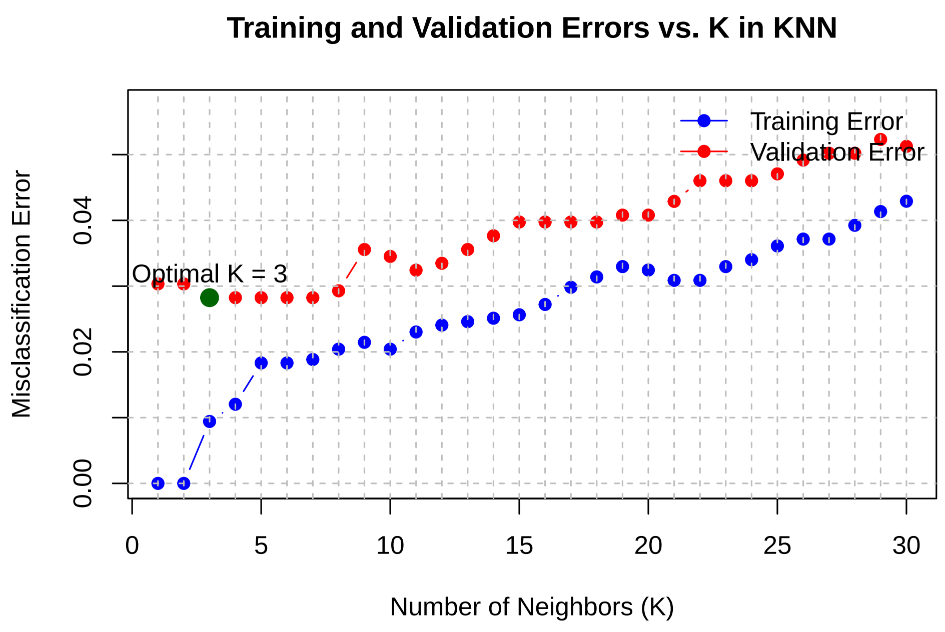
Class-Specific Observations:

- **Best Performance**: Digit `0` (perfect classification).
- **Strong Performance**: Digits `1`, `2`, `6`, `7` (minimal errors).
- **Challenging Cases**:
  - Digits `3`, `4`, `5`: Overlap with similar digits.
  - Digits `8`, `9`: High confusion with `1` and `3`.

Visualizations of Easiest and Hardest Cases for Digit '8'



The hardest 3 cases are visually ambiguous and challenging to identify as "8," likely due to overlapping features or incomplete digit representation. In contrast, the easiest 2 cases are visually distinct and clearly recognizable as "8," making them straightforward to classify.



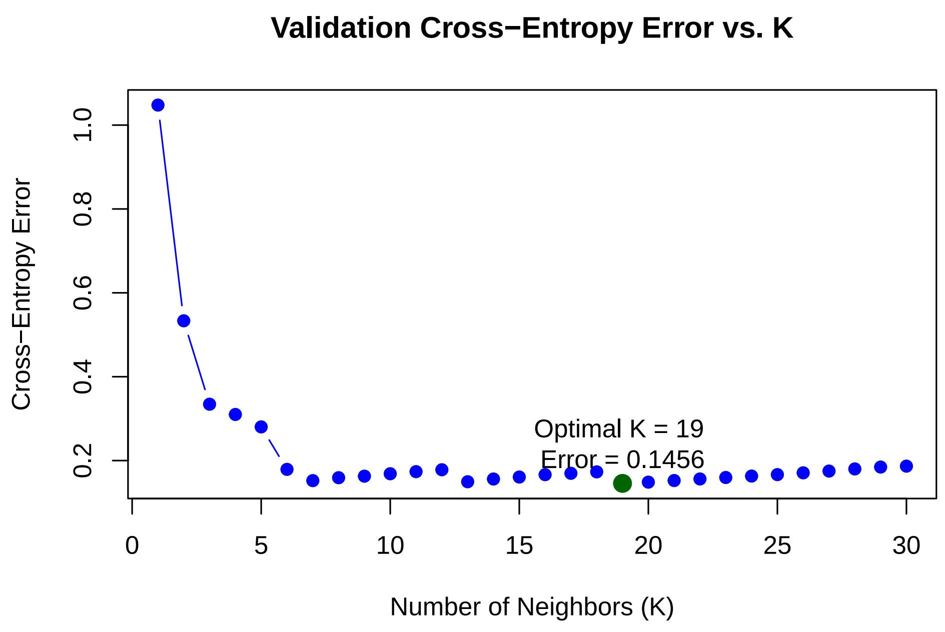
- **Model Complexity**:
  - As K increases, the model becomes less complex because it considers more neighbors for classification. This effectively smooths the decision boundaries.
  - For **small** K (e.g., $K = 1$), the model is highly complex as it relies only on the nearest neighbor, leading to potential overfitting.
- **Effect on Errors**:
  - **Training Error**:
    - For small K, the training error is very low (almost 0) due to overfitting.
    - As K increases, the training error gradually rises because the model becomes less sensitive to individual training points.
  - **Validation Error**:
    - Validation error initially decreases as K increases, reducing overfitting.
    - Beyond a certain point, validation error starts increasing due to underfitting when the model becomes overly smoothed.

Optimal K According to the Plot

- From the plot, the **optimal** K corresponds to the minimum validation error.
- **Optimal** $K = 3$ in this case, as the validation error is lowest at this point.

Comparison of Test, Training, and Validation Errors

The test misclassification error for $K = 3$ is **0.0325**, which is slightly higher than the validation error (**0.0282**) and significantly higher than the training error (**0.0094**). This indicates that the model generalizes well to unseen data while maintaining a low error on the training data.



Optimal K Value and Suitability of Cross-Entropy Error

- **Optimal** K:
  - From the graph, the **optimal** K is **19**, as the validation cross-entropy error is minimized at this point (**Error = 0.1456**).
- **Why Cross-Entropy is More Suitable**:
  - Cross-entropy considers the confidence of predictions by penalizing incorrect or low-probability predictions more heavily.
  - Unlike misclassification error, which treats all incorrect predictions equally, cross-entropy error differentiates between predictions with varying levels of certainty.
  - In a multinomial distribution (as in this case with multiple classes), cross-entropy provides a probabilistic perspective, which is essential for understanding model confidence and calibration, making it more informative for fine-tuning models.

Assignment 2: Linear regression and ridge regression

- **Training MSE**: 6.0799301
- **Test MSE**: 6.5498889

The linear regression model shows that age, sex, total_UPDRS, Jitter.Abs., Shimmer.APQ5, and Shimmer.APQ11 are significant predictors of motor_UPDRS based on their low p-values (< 0.05). The total_UPDRS variable has the strongest influence, while others contribute moderately. The training MSE and test MSE indicate the model's accuracy on respective datasets.

Optimal theta parameters for different lambda values

- **lambda = 1**

- **Optimized Theta (coefficients):**: 21.1882596, -0.1217787, -0.21849, 0.3530682, -0.0099003, 7.7113778, 0.5164646, -0.6079357, -0.1722544, 0.1518057, -0.1631611, 0.481972, 0.1094098, -0.0787833, -1.2877144, 0.9182178, -0.0801917, 0.1322423, 0.0305513, -0.2051966, -0.0628982, 0.418351
- **lambda = 100**
- **Optimized Theta (coefficients):**: 21.1898045, 0.4885099, 0.5076442, -0.1029531, 0.1492863, 2.9360379, 0.0308151, -0.0974856, 0.0059887, 0.0123506, 0.0060308, 0.0233771, 0.046099, -0.0308132, -0.0016512, 0.1462391, -0.0308086, -0.0230071, -0.161175, 0.0867249, -0.3214638, 0.2401156
- **lambda = 1000**
- **Optimized Theta (coefficients):**: 21.1882596, -0.1217787, -0.21849, 0.3530682, -0.0099003, 7.7113778, 0.5164646, -0.6079357, -0.1722544, 0.1518057, -0.1631611, 0.481972, 0.1094098, -0.0787833, -1.2877144, 0.9182178, -0.0801917, 0.1322423, 0.0305513, -0.2051966, -0.0628982, 0.418351

Table 1: Ridge Regression Results for Different Lambda Values

Lambda	Train_MSE	Test_MSE	Degrees_of_Freedom	Optimized_Sigma
1	6.085893	6.560836	19.84347	2.467315
100	25.886156	26.947515	15.59068	5.088174
1000	61.360932	62.806605	10.04924	7.833403

From the table above, the choice of the penalty parameter significantly affects the model's performance and complexity. Here's the detailed analysis:

- **lambda = 1**:
  - Achieves the lowest **Train MSE** (6.085893) and **Test MSE** (6.560836).
  - Maintains a **Degrees of Freedom (DF)** of 19.84347, meaning more predictors contribute effectively to the model.
  - Indicates a balance between model complexity and generalization.
- **lambda = 100**:
  - Results in higher **Train MSE** (25.88616) and **Test MSE** (26.94751) compared to **lambda = 1**.
  - Reduces **DF** to 15.59068, indicating a stronger penalty and increased regularization.
  - May lead to underfitting as the model becomes overly simplified.
- **lambda = 1000**:
  - Has the highest **Train MSE** (61.36093) and **Test MSE** (62.8066), demonstrating significant underfitting.
  - Reduces **DF** further to 10.04924, suggesting excessive shrinkage of predictor coefficients.
  - Results in a simpler but less accurate model.

Based on the results and analysis:

- **lambda = 1** is the most appropriate penalty parameter. It minimizes both **Train MSE** and **Test MSE**, while maintaining a high **Degrees of Freedom**, ensuring that the model is neither over-regularized nor under-regularized.
- As **lambda** increases, the model becomes overly regularized, leading to underfitting (higher MSE and lower DF).

Appendix

Code for Asssignment 1

```r
#Load and Preprocess Data
# Load dataset
file_path <- "data/optdigits.csv"
optdigits_data <- read.csv(file_path)
colnames(optdigits_data) <- c(paste0("pixel_", 1:64), "label")
optdigits_data$label <- as.factor(optdigits_data$label)

# Split into training and test sets
set.seed(123)
n <- nrow(optdigits_data)
# Generate indices
train_indices <- sample(1:n, size = 0.5 * n)  # 50% for training
remaining_indices <- setdiff(1:n, train_indices)  # Remaining 50%
test_indices <- sample(remaining_indices, size = 0.25 * n)  # 25% for testing
validation_indices <- setdiff(remaining_indices, test_indices)  # Remaining 25% for validation

# Subset the data
train_data <- optdigits_data[train_indices, ]
test_data <- optdigits_data[test_indices, ]
validation_data <- optdigits_data[validation_indices, ]

# Train the KNN classifier
knn_model <- kknn(label ~ ., train = train_data, test = train_data, k = 30, kernel = "rectangular")


# Predictions and performance
train_predictions <- fitted(knn_model)
conf_matrix_train <- table(Actual = train_data$label, Predicted = train_predictions)
print("Confusion Matrix (Training):")
print(conf_matrix_train)

# Predict on test data
test_predictions <- fitted(kknn(label ~ ., train = train_data, test = test_data, k = 30, kernel = "recta
conf_matrix_test <- table(Actual = test_data$label, Predicted = test_predictions)
print("Confusion Matrix (Test):")
print(conf_matrix_test)


# Misclassification error for training data
train_total <- sum(conf_matrix_train)  # Total instances in training data
train_correct <- sum(diag(conf_matrix_train))  # Correctly classified instances (sum of diagonal)
train_misclassification_error <- 1 - train_correct / train_total

cat("Training Misclassification Error:", train_misclassification_error, "\n")

# Misclassification error for test data
test_total <- sum(conf_matrix_test)  # Total instances in training data
test_correct <- sum(diag(conf_matrix_test))  # Correctly classified instances (sum of diagonal)
test_misclassification_error <- 1 - test_correct / test_total
```

```r
cat("Test Misclassification Error:", test_misclassification_error, "\n")


# Get predicted probabilities for training data
train_probabilities <- knn_model$prob  # Matrix of probabilities
train_actual <- train_data$label  # Actual labels for training data

# Extract rows corresponding to digit "8" in the training data
digit_8_indices <- which(train_actual == "8")
digit_8_probs <- train_probabilities[digit_8_indices, ]  # Probabilities for digit "8"

digit_8_predicted_probs <- digit_8_probs[, "8"]

easiest_indices <- digit_8_indices[order(digit_8_predicted_probs, decreasing = TRUE)[1:2]]
hardest_indices <- digit_8_indices[order(digit_8_predicted_probs, decreasing = FALSE)[1:3]]

# Extract feature data (pixels) for easiest and hardest cases
easiest_features <- train_data[easiest_indices, 1:64]  # Features for easiest cases
hardest_features <- train_data[hardest_indices, 1:64]  # Features for hardest cases

# Reshape into 8x8 matrices
# Reshape and store as numeric matrices
easiest_matrices <- lapply(1:nrow(easiest_features), function(i) {
  matrix(as.numeric(easiest_features[i, ]), nrow = 8, byrow = TRUE)
})

hardest_matrices <- lapply(1:nrow(hardest_features), function(i) {
  matrix(as.numeric(hardest_features[i, ]), nrow = 8, byrow = TRUE)
})

# Visualize easiest cases
heatmap(easiest_matrices[[1]], Colv = NA, Rowv = NA, scale = "none", main = paste("Easiest Case", 1), xl
heatmap(easiest_matrices[[2]], Colv = NA, Rowv = NA, scale = "none", main = paste("Easiest Case", 2), xl

# Visualize hardest cases
heatmap(hardest_matrices[[1]], Colv = NA, Rowv = NA, scale = "none", main = paste("Hardest Case", 1), xl
heatmap(hardest_matrices[[2]], Colv = NA, Rowv = NA, scale = "none", main = paste("Hardest Case", 2), xl
heatmap(hardest_matrices[[3]], Colv = NA, Rowv = NA, scale = "none", main = paste("Hardest Case", 3), xl



# Initialize vectors to store errors
k_values <- 1:30  # Values of K to test
train_errors <- numeric(length(k_values))  # Training errors
validation_errors <- numeric(length(k_values))  # Validation errors

# Loop through each value of K
for (k in k_values) {
  # Train KNN model on training data
  knn_model <- kknn(label ~ ., train = train_data, test = train_data, k = k, kernel = "rectangular")
  train_predictions <- fitted(knn_model)

  # Calculate training misclassification error
  conf_matrix_train <- table(Actual = train_data$label, Predicted = train_predictions)
  train_correct <- sum(diag(conf_matrix_train))
  train_total <- sum(conf_matrix_train)
  train_errors[k] <- 1 - train_correct / train_total

  # Validate on validation data
  validation_predictions <- fitted(kknn(label ~ ., train = train_data, test = validation_data, k = k, ke
  conf_matrix_validation <- table(Actual = validation_data$label, Predicted = validation_predictions)
  validation_correct <- sum(diag(conf_matrix_validation))
  validation_total <- sum(conf_matrix_validation)
  validation_errors[k] <- 1 - validation_correct / validation_total

}
#
# Plot training and validation errors
plot(k_values, train_errors, type = "b", col = "blue", pch = 19,
     ylim = c(0, max(c(train_errors, validation_errors)) * 1.1),
     xlab = "Number of Neighbors (K)",
     ylab = "Misclassification Error",
     main = "Training and Validation Errors vs. K in KNN")

lines(k_values, validation_errors, type = "b", col = "red", pch = 19)

# Add gridlines
abline(h = seq(0, max(c(train_errors, validation_errors)), by = 0.01), col = "gray", lty = 2)
abline(v = seq(1, 30, by = 1), col = "gray", lty = 2)

# Highlight optimal K
optimal_k <- which.min(validation_errors)
points(optimal_k, validation_errors[optimal_k], col = "darkgreen", pch = 19, cex = 1.5)
text(optimal_k, validation_errors[optimal_k], labels = paste("Optimal K =", optimal_k), pos = 3)

# Add legend
legend("topright", legend = c("Training Error", "Validation Error"),
       col = c("blue", "red"), pch = 19, lty = 1, bty = "n")



# Train the model with optimal K
optimal_k <- 3
knn_model_optimal <- kknn(label ~ ., train = train_data, test = test_data, k = optimal_k, kernel = "rec

# Predictions and confusion matrix
test_predictions <- fitted(knn_model_optimal)
conf_matrix_test <- table(Actual = test_data$label, Predicted = test_predictions)

# Calculate test misclassification error
test_total <- sum(conf_matrix_test)
test_correct <- sum(diag(conf_matrix_test))
test_misclassification_error <- 1 - test_correct / test_total

cat("Test Misclassification Error for K =", optimal_k, ":", test_misclassification_error, "\n")
```

```
#Plot cross-entropy errors for validation data
plot(k_values, cross_entropy_errors, type = "b", col = "blue", pch = 19,
     xlab = "Number of Neighbors (K)", ylab = "Cross-Entropy Error",
     main = "Validation Cross-Entropy Error vs. K")

#Highlight optimal K
optimal_k_cross_entropy <- which.min(cross_entropy_errors)
optimal_cross_entropy_value <- cross_entropy_errors[optimal_k_cross_entropy]

# Mark the optimal K and cross-entropy value
points(optimal_k_cross_entropy, optimal_cross_entropy_value, col = "darkgreen", pch = 19, cex = 1.5)
text(optimal_k_cross_entropy, optimal_cross_entropy_value,
     labels = paste("Optimal K =", optimal_k_cross_entropy, "\nError =", round(optimal_cross_entropy_va
     pos = 3)

# Add gridlines for better visualization
abline(h = seq(0, max(cross_entropy_errors), by = 0.1), col = "gray", lty = 2)
abline(v = seq(1, 30, by = 1), col = "gray", lty = 2)
```

Code for Asssignment 2

```
scale_features <- function(data) {
  data_scaled <- data
  data_scaled[ , -which(names(data) == "motor_UPDRS")] <- scale(data[ , -which(names(data) == "motor_UP
  return(data_scaled)
}

# Define the Loglikelihood function
Loglikelihood <- function(theta, sigma, X, y) {
  n <- nrow(X)
  y_pred <- X %*% theta
  residual_sum_squares <- sum((y - y_pred)^2)

  log_likelihood <- - (n / 2) * log(2 * pi) - (n / 2) * log(sigma^2) -
    (1 / (2 * sigma^2)) * residual_sum_squares
  return(log_likelihood)
}

# Define Ridge function
Ridge <- function(theta, sigma, X, y, lambda) {
  log_likelihood <- Loglikelihood(theta, sigma, X, y)
  ridge_penalty <- lambda * sum(theta[-1]^2)  # Exclude intercept
  return(log_likelihood - ridge_penalty)
}

# Define Ridge optimization function
RidgeOpt <- function(X, y, lambda) {
  initial_theta <- rep(0, ncol(X))  # Initialize theta to zeros
  initial_sigma <- 1                # Initial guess for sigma
  initial_values <- c(initial_theta, initial_sigma)
```

```r
  objective_function <- function(params) {
    theta <- params[1:(length(params) - 1)]
    sigma <- params[length(params)]
    return(-Ridge(theta, sigma, X, y, lambda))
  }

  # Minimize the negative Ridge log-likelihood
  result <- optim(par = initial_values, fn = objective_function, method = "BFGS")

  optimized_theta <- result$par[1:(length(result$par) - 1)]
  optimized_sigma <- result$par[length(result$par)]

  return(list(theta = optimized_theta, sigma = optimized_sigma, value = result$value))
}

# Function to calculate degrees of freedom for Ridge regression
DF <- function(X, lambda) {
  p <- ncol(X)
  I <- diag(p)
  XtX <- t(X) %*% X
  ridge_matrix <- solve(XtX + lambda * I)
  H <- X %*% ridge_matrix %*% t(X)
  return(sum(diag(H)))
}


#Load and Preprocess Data
# Load dataset
file_path <- "data/parkinsons.csv"
parkinson_data <- read.csv(file_path)

# Split into training and test sets
set.seed(123)
n <- nrow(parkinson_data)
train_indices <- sample(1:n, size = 0.6 * n)
train_data <- parkinson_data[train_indices, ]
test_data <- parkinson_data[-train_indices, ]

# Scale features
train_data_scaled <- scale_features(train_data)
test_data_scaled <- scale_features(test_data)


#Train Linear Model
# Fit linear model
linear_model <- lm(motor_UPDRS ~ ., data = train_data_scaled)
# Predictions
train_predictions <- predict(linear_model, train_data_scaled)
test_predictions <- predict(linear_model, test_data_scaled)

# Calculate MSE
train_mse <- mean((train_data_scaled$motor_UPDRS - train_predictions)^2)
test_mse <- mean((test_data_scaled$motor_UPDRS - test_predictions)^2)
```

```r
# Display MSE
cat("Train MSE:", train_mse, "\n")
cat("Test MSE:", test_mse, "\n")



#Ridge Regression
# Prepare data for Ridge regression
X <- as.matrix(cbind(1, train_data_scaled[, -which(names(train_data_scaled) == "motor_UPDRS")]))
y <- train_data_scaled$motor_UPDRS

# Initial log-likelihood
theta_initial <- coef(linear_model)
sigma_initial <- summary(linear_model)$sigma
log_likelihood_value <- Loglikelihood(theta_initial, sigma_initial, X, y)
cat("Log-Likelihood (Linear Model):", log_likelihood_value, "\n")

# Ridge optimization with   = 0.001
lambda <- 0.001
ridge_optimization_result <- RidgeOpt(X, y, lambda)
cat("Optimized Theta (coefficients):\n", ridge_optimization_result$theta, "\n")
cat("Optimized Sigma (residual standard error):\n", ridge_optimization_result$sigma, "\n")
cat("Minimized Negative Log-Likelihood:\n", ridge_optimization_result$value, "\n")


#Degrees of Freedom Analysis
# Degrees of Freedom for Ridge regression
lambda <- 10
degrees_of_freedom <- DF(X, lambda)
#cat("Degrees of Freedom for Ridge Regression (  =", lambda, "):", degrees_of_freedom, "\n")

# Compute predictions and MSE
predict_ridge <- function(X, y, theta) {
  y_pred <- X %*% theta
  mse <- mean((y - y_pred)^2)
  return(list(predictions = y_pred, mse = mse))
}

# Perform Ridge regression and compute metrics for given lambda values


#Display results
for (lambda in names(results)) {
result <- results[[lambda]]
cat("Lambda =", result$lambda, "\n")
cat("Train MSE:", result$train_mse, "\n")
cat("Test MSE:", result$test_mse, "\n")
cat("Degrees of Freedom:", result$degrees_of_freedom, "\n")
cat("Optimized Sigma:", result$optimized_sigma, "\n")
cat("\n")
}
```