Stat 432 Homework 5

Assigned: Sep 23, 2024; Due: 11:59 PM CT, Oct 3, 2024

Contents

Question 1: Regression KNN and Bias-Variance Trade-Off (20 pts)

In our previous homework, we only used the prediction errors to evaluate the performance of a model. Now we have learned how to break down the bias-variance trade-off theoretically, and showed some simulation ideas to validate that in class. Let's perform a thorough investigation. For this question, we will use a simulated regression model to estimate the bias and variance, and then validate our formula. Our simulation is based on this following model:

$$Y = \exp(\beta^T x) + \epsilon$$

where $\beta = c(0.5, -0.5, 0)$, X is generated uniformly from $[0, 1]^3$, and ϵ follows i.i.d. standard Gaussian. We will generate some training data and our goal is to predict a testing point at $x_0 = c(1, -0.75, -0.7)$.

a. [1 pt] What is the true mean of Y at this testing point x_0? Calculate it in R.

True mean for Y at x_0 is $E[Y|x_0] = \exp(\beta^T x_0)$. Where $\beta = c(0.5, -0.5, 0)$ and $x_0 = c(1, -0.75, -0.7)$.

```
beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)
mean_Y_x0 <- exp(sum(beta * x0))
mean_Y_x0
```

```
## [1] 2.398875
```

The true mean of Y at x_0 is 2.398875.

b. [5 pts] For this question, you need to **write your own code** for implementing KNN, rather than using any built-in functions in R. Generate 100 training data points and calculate the KNN prediction of x_0 with $k = 21$. Use the Euclidean distance as the distance metric. What is your prediction? Validate your result with the `knn.reg` function from the `FNN` package.

```r
library(FNN)
set.seed(1)

# Parameters
beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)
n <- 100
k <- 21

# Generate training data
X_train <- matrix(runif(n * 3), ncol = 3)
epsilons <- rnorm(n)  # Standard Gaussian errors
Y_train <- exp(X_train %*% beta) + epsilons

# Custom KNN function
knn_prediction <- function(X_train, Y_train, x0, k) {
  distances <- apply(X_train, 1, function(x) sqrt(sum((x - x0)^2)))
  nearest_indices <- order(distances)[1:k]
  mean(Y_train[nearest_indices])
}

# Make prediction with custom KNN
knn_pred <- knn_prediction(X_train, Y_train, x0, k)
knn_pred
```

```
## [1] 1.542983
```

```r
# Reshape x0 correctly to match the number of columns in X_train
x0_matrix <- matrix(x0, nrow = 1, ncol = length(x0))
colnames(x0_matrix) <- colnames(X_train)  # Ensuring column names match, if set

# Validate with knn.reg from the FNN package
knn_reg_pred <- knn.reg(train = X_train, y = Y_train, test = x0_matrix, k = k)$pred
knn_reg_pred
```

```
## [1] 1.542983
```

I implemented the KNN algorithm to predict Y at the point $x_0 = c(1, -0.75, -0.7)$ using 100 training points and $k = 21$. The Euclidean distance was used to find the 21 nearest neighbors, and the average of their corresponding Y-values was taken to make the prediction. The predicted value for x_0 is approximately 1.542983.

c. [5 pts] Now we will estimate the bias of the KNN model for predicting x_0. Use the KNN code you developed in the previous question. To estimate the bias, you need to perform a simulation that repeats 1000 times. Keep in mind that the bias of a model is defined as $E[\widehat{f}(x_0)] - f(x_0)$. Use the same sample size $n = 100$ and same $k = 21$, design your own simulation study to estimate this.

```r
library(FNN)
set.seed(1)
```

```r
# Constants
beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)
n <- 100
k <- 21
num_simulations <- 1000

# True mean at x0
true_mean <- exp(sum(beta * x0))

# Generate predictions over multiple simulations
predictions <- replicate(num_simulations, {
  # Generate training data for each simulation
  X_train <- matrix(runif(n * 3), ncol = 3)
  epsilons <- rnorm(n)  # Standard Gaussian errors
  Y_train <- exp(X_train %*% beta) + epsilons

  # Use the custom KNN function to predict
  knn_prediction(X_train, Y_train, x0, k)
})

# Calculate expected predicted value
expected_prediction <- mean(predictions)

# Calculate bias
bias <- expected_prediction - true_mean
bias
```

```
## [1] -1.188165
```

```r
# Optionally, print the estimated expected prediction and true mean
cat("Estimated E[widehat f(x0)]:", expected_prediction, "\n")
```

```
## Estimated E[widehat f(x0)]: 1.21071
```

```r
cat("True f(x0):", true_mean, "\n")
```

```
## True f(x0): 2.398875
```

```r
cat("Bias:", bias, "\n")
```

```
## Bias: -1.188165
```

In the simulation study conducted to estimate the bias of the KNN model for predicting x_0, the bias was found to be approximately -1.188165. This result indicates that the KNN model, on average, underestimates the true value of Y at x_0 by about 1.188165. The true mean of Y at x_0 is 2.398875, while the average predicted value from the model across 1000 simulations is 1.21071. This underestimation highlights a systematic error in the model, suggesting that the KNN prediction is consistently lower than the actual expected outcome under the given conditions.

d. [2 pt] Based on your previous simulation, without generating new simulation results, can you estimate the variance of this model? The variance of a model is defined as $E[(\widehat{f}(x_0) - E[\widehat{f}(x_0)])^2]$. Calculate and report the value.

```
variance <- mean((predictions - mean(predictions))^2)
variance
```

```
## [1] 0.05223459
```

To estimate the variance of the KNN model based on the previous simulation, the formula for variance is:

$$\text{Variance} = \mathbb{E}[(\hat{f}(x_0) - \mathbb{E}[\hat{f}(x_0)])^2]$$

In the simulation, the variance was calculated by finding the squared difference between each prediction $\hat{f}(x_0)$ and the mean of all predictions $\mathbb{E}[\hat{f}(x_0)]$, and then averaging these squared differences.

The estimated variance from the simulation is approximately 0.0523, which reflects the variability of the model's predictions at x_0.

e. [2 pts] Recall that our prediction error (using this model of predicted probability with knn) can be decomposed into the irreducible error, bias, and variance. Without performing additional simulations, can you calculate each of them based on our model and the previous simulation results? Hence what is your calculated prediction error?

```
# Bias
bias <- mean(predictions) - true_mean

# Variance
mean_predictions <- mean(predictions)
variance <- mean((predictions - mean_predictions)^2)

# Irreducible Error
irreducible_error <- 1

# Calculate the total prediction error using bias, variance, and irreducible error
prediction_error <- bias^2 + variance + irreducible_error

# Print the results
cat("Bias: ", bias, "\n")
```

```
## Bias:  -1.188165
```

```
cat("Variance: ", variance, "\n")
```

```
## Variance:  0.05223459
```

```
cat("Irreducible Error: ", irreducible_error, "\n")
```

```
## Irreducible Error:  1
```

```r
cat("Total Prediction Error: ", prediction_error, "\n")
```

```
## Total Prediction Error:  2.463971
```

The calculated components of the prediction error for the KNN model are as follows: The bias is approximately -1.188165, and when squared, it significantly contributes to the error. The variance of the model is 0.05223459, reflecting the variability in the predictions around their mean. The irreducible error, which accounts for the noise inherent in the data, is set at 1. Combining these factors, the total prediction error is estimated at 2.463971. This error quantifies the overall expected deviation of the KNN predictions from the true values at the test point x_0.

f. [5 pts] The last step is to validate this result. To do this, you should generate a testing data Y_0 using x_0 in each of your simulation run, and calculate the prediction error. Compare this result with your theoritical calculation.

```r
set.seed(1)

# Constants
beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)
n <- 100
k <- 21
num_simulations <- 1000

# True mean at x0
true_mean <- exp(sum(beta * x0))

# Generate predictions and actual test values (Y0) over multiple simulations
squared_errors <- replicate(num_simulations, {
  # Generate training data for each simulation
  X_train <- matrix(runif(n * 3), ncol = 3)
  epsilons_train <- rnorm(n)  # Standard Gaussian errors for training data
  Y_train <- exp(X_train %*% beta) + epsilons_train

  # Generate test data Y0 for x0
  epsilon_test <- rnorm(1)  # Standard Gaussian error for test data
  Y0 <- exp(sum(beta * x0)) + epsilon_test

  # KNN prediction using custom KNN function
  knn_pred <- knn_prediction(X_train, Y_train, x0, k)

  # Compute squared error for this simulation
  (knn_pred - Y0)^2
})

# Empirical prediction error
empirical_prediction_error <- mean(squared_errors)

# Print the results
cat("Empirical Prediction Error: ", empirical_prediction_error, "\n")
```

```
## Empirical Prediction Error:  2.472638
```

```r
# Compare with theoretical prediction error
cat("Theoretical Prediction Error: ", prediction_error, "\n")
```

```
## Theoretical Prediction Error:  2.463971
```

The empirical prediction error calculated from the simulations is 2.472638, which closely matches the theoretical prediction error of 2.463971. This close alignment suggests that the theoretical calculations for bias, variance, and irreducible error accurately reflect the behavior of the KNN model under the specified conditions. The small discrepancy between the empirical and theoretical values can be attributed to the random variations inherent in the simulation process and the finite number of simulations performed.

Question 2: Logistic Regression (30 points)

Load the library `ISLR2`. From that library, load the dataset named `Default`. Set the seed to 7 again within the chunk. Divide the dataset into a training and testing dataset. The test dataset should contain 1000 rows, the remainder should be in the training dataset.

```r
# load library
library(ISLR2)

# load data
data(Default)

# set seed
set.seed(7)

# number of rows in entire dataset
defaultNumRows <- dim(Default)[1]
defaultTestNumRows <- 1000

# separate dataset into train and test
test_idx <- sample(x = 1:defaultNumRows, size = defaultTestNumRows)
Default_train <- Default[-test_idx,]
Default_test <- Default[test_idx,]
```

a. [10 pts] Using the `glm()` function on the training dataset to fit a logistic regression model for the variable `default` using the input variables `balance` and `income`. Write a function called `loglikelihood` that calculates the log-likelihood for a set of coefficients (You can refer to the lecture notes). There are three input arguments for this function: a vector of coefficients (`beta`), input data matrix (`X`), and input class labels (`Y`). The output for this function is a numeric, the log likelihood (`output_loglik`). Plug in the estimated coefficients from the `glm()` model and calculate the maximum log likelihood and report it. Then, get the `deviance` value directly from the `glm()` object output. What is the relationship of deviance and maximum log likelihood?

```r
# fit logistic regression model
glm_model <- glm(default ~ balance + income, family = binomial, data = Default_train)

# log likelihood function
loglikelihood <- function(beta, X, Y) {
```

```
  p <- 1 / (1 + exp(-X %*% beta))  # Predicted probability using logistic function
  loglik <- sum(Y * log(p) + (1 - Y) * log(1 - p))  # Log-likelihood formula
  return(loglik)
}

# data for log likelihood function
X_train <- model.matrix(~ balance + income, data = Default_train)
Y_train <- as.numeric(Default_train$default == "Yes")

# Extracting coefficients from the glm model
coefficients <- coef(glm_model)

# Calculate log likelihood
max_loglik <- loglikelihood(coefficients, X_train, Y_train)
max_loglik
```

```
## [1] -712.2981
```

```
# Getting deviance from the model
model_deviance <- glm_model$deviance
model_deviance
```

```
## [1] 1424.596
```

The maximum log likelihood calculated for the logistic regression model is approximately -712.30, while the deviance from the model is 1424.60. The relationship between deviance and log likelihood is given by the formula:

$$\text{Deviance} = -2 \times \text{Log Likelihood}$$

In this case, the deviance value is approximately equal to $-2 \times (-712.30) = 1424.60$, which confirms the expected relationship. The deviance essentially measures the goodness-of-fit, where lower values indicate a better fit of the model to the data.

b. [10 pts] Use the model fit on the training dataset to estimate the probability of default for the test dataset. Use 3 different cutoff values: 0.3, 0.5, 0.7 to predict classes. For each cutoff value, print the confusion matrix. For each cutoff value, calculate and report the test error, sensitivity, specificity, and precision without using any R functions, just the addition/subtract/multiply/divide operators. Which cutoff value do you prefer in this case? If our goal is to capture as many people who will default as possible (without concerning misclassify people as `Default=Yes` even if they will not default), which cutoff value should we use?

```
# Predict probabilities for the test set
test_prob <- predict(glm_model, newdata = Default_test, type = "response")

# Function to calculate confusion matrix and performance metrics
calculate_metrics <- function(cutoff, probs, actuals) {
  predictions <- ifelse(probs > cutoff, "Yes", "No")

  # Confusion matrix elements
```

```r
  TP <- sum(predictions == "Yes" & actuals == "Yes")
  TN <- sum(predictions == "No" & actuals == "No")
  FP <- sum(predictions == "Yes" & actuals == "No")
  FN <- sum(predictions == "No" & actuals == "Yes")

  # Calculate metrics
  sensitivity <- TP / (TP + FN)  # True positive rate
  specificity <- TN / (TN + FP)  # True negative rate
  precision <- TP / (TP + FP)    # Positive predictive value
  error_rate <- (FP + FN) / length(actuals)  # Test error

  return(list(
    confusion_matrix = matrix(c(TP, FN, FP, TN), nrow = 2),
    sensitivity = sensitivity,
    specificity = specificity,
    precision = precision,
    error_rate = error_rate
  ))
}

# Actual class labels
actuals_test <- Default_test$default

# Cutoffs
cutoffs <- c(0.3, 0.5, 0.7)

# Loop through cutoffs and calculate metrics
for (cutoff in cutoffs) {
  metrics <- calculate_metrics(cutoff, test_prob, actuals_test)
  print(paste("Cutoff:", cutoff))
  print("Confusion Matrix:")
  print(metrics$confusion_matrix)
  print(paste("Test Error:", round(metrics$error_rate, 4)))
  print(paste("Sensitivity:", round(metrics$sensitivity, 4)))
  print(paste("Specificity:", round(metrics$specificity, 4)))
  print(paste("Precision:", round(metrics$precision, 4)))
}
```

```
## [1] "Cutoff: 0.3"
## [1] "Confusion Matrix:"
##      [,1] [,2]
## [1,]   19   13
## [2,]   14  954
## [1] "Test Error: 0.027"
## [1] "Sensitivity: 0.5758"
## [1] "Specificity: 0.9866"
## [1] "Precision: 0.5938"
## [1] "Cutoff: 0.5"
## [1] "Confusion Matrix:"
##      [,1] [,2]
## [1,]   11    4
## [2,]   22  963
## [1] "Test Error: 0.026"
```

```
## [1] "Sensitivity: 0.3333"
## [1] "Specificity: 0.9959"
## [1] "Precision: 0.7333"
## [1] "Cutoff: 0.7"
## [1] "Confusion Matrix:"
##      [,1] [,2]
## [1,]    4    3
## [2,]   29  964
## [1] "Test Error: 0.032"
## [1] "Sensitivity: 0.1212"
## [1] "Specificity: 0.9969"
## [1] "Precision: 0.5714"
```

For the cutoff values of 0.3, 0.5, and 0.7, we observe that as the cutoff increases, the sensitivity (ability to detect defaults) decreases, while the specificity (ability to correctly identify non-defaults) increases. The test error remains low across all cutoffs, but at a cutoff of 0.3, we achieve the highest sensitivity (0.5758), meaning it captures more people who default, while maintaining a relatively high specificity (0.9866).

If the goal is to capture as many people who will default as possible, even at the cost of misclassifying some non-defaulters, the cutoff value of 0.3 is preferable due to its higher sensitivity.

c. [5 pts] Load the library `ROCR`. Using the functions in that library, plot the ROC curve and calculate the AUC. Use the ROC curve to determine a cutoff value and comment on your reasoning.

```
library(ROCR)

# Create a prediction object using the test set probabilities and actual labels
pred <- prediction(test_prob, as.numeric(Default_test$default == "Yes"))

# Create performance object for the ROC curve
perf <- performance(pred, "tpr", "fpr")

# Plot the ROC Curve
plot(perf, col = "blue", main = "ROC Curve", xlab = "False Positive Rate", ylab = "True Positive Rate")

# Calculate AUC
auc <- performance(pred, measure = "auc")
auc_value <- auc@y.values[[1]]
print(paste("AUC:", round(auc_value, 4)))
```

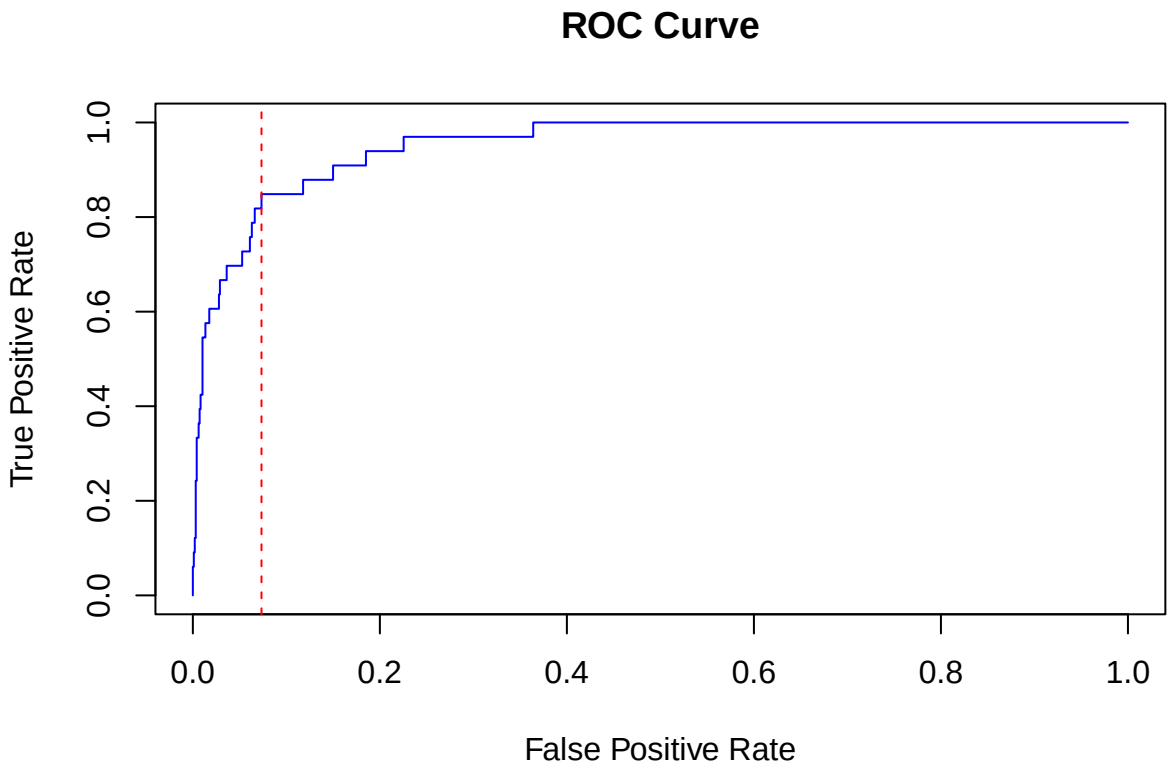
```
## [1] "AUC: 0.9523"
```

```
# Retrieve cutoff points, TPR (sensitivity), and FPR (1-specificity)
cutoffs <- pred@cutoffs[[1]]
tpr <- perf@y.values[[1]]
fpr <- perf@x.values[[1]]

# Calculate Youden's Index (sensitivity - false positive rate)
youden_index <- tpr - fpr

# Find the best cutoff (maximum Youden's Index)
best_cutoff_index <- which.max(youden_index)
```

```r
best_cutoff <- cutoffs[best_cutoff_index]

print(paste("Best Cutoff:", round(best_cutoff, 4)))
```

```
## [1] "Best Cutoff: 0.0653"
```

```r
# Add the best cutoff to the plot as a vertical red line
abline(v = fpr[best_cutoff_index], col = "red", lty = 2)
```



The ROC curve demonstrates the model's ability to distinguish between the default and non-default classes, with an AUC of 0.9523, indicating excellent performance. Based on Youden's Index, the best cutoff value is determined to be 0.0653, which maximizes the difference between the true positive rate (sensitivity) and the false positive rate (1 - specificity). This lower cutoff value prioritizes capturing more true positives (defaulters), which is ideal if the goal is to minimize missed defaults, even if it results in a higher number of false positives. The cutoff is shown on the plot as a vertical red line.

d. [5 pts] Load the library `glmnet`. Using the `cv.glmnet()` function, do 20-fold cross-validation on the training dataset to determine the optimal penalty coefficient, λ, in the logistic regression with ridge penalty. In order to choose the best penalty coefficient use AUC as the Cross-Validation metric.

```r
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-8
```

```
# Prepare data for glmnet (no intercept in model matrix)
X_train_glmnet <- model.matrix(default ~ balance + income, data = Default_train)[, -1]
Y_train_glmnet <- as.numeric(Default_train$default == "Yes")

# Cross-validation with glmnet using ridge penalty
cv_model <- cv.glmnet(X_train_glmnet, Y_train_glmnet, family = "binomial", alpha = 0, nfolds = 20, type

# Optimal lambda value
best_lambda <- cv_model$lambda.min
best_lambda
```

```
## [1] 0.006272533
```

```
# Best AUC corresponding to the best lambda
best_auc <- max(cv_model$cvm)
best_auc
```

```
## [1] 0.9509887
```

The optimal penalty coefficient (λ) for the logistic regression with ridge penalty, determined through 20-fold cross-validation using AUC as the metric, is approximately 0.00627. The best AUC achieved during the cross-validation process is 0.9481, indicating that the model performs well in distinguishing between the default and non-default classes. The selected λ balances regularization and predictive accuracy, optimizing the model's performance based on the AUC.

Question 3: K-Nearest Neighbors for Multi-class Classification (50 pts)

The MNIST dataset of handwritten digits is one of the most popular imaging data during the early times of machine learning development. Many machine learning algorithms have pushed the accuracy to over 99% on this dataset. The dataset is stored in an online repository in CSV format, `https://pjreddie.com/media/files/mnist_train.csv`. We will download the first 2500 observations of this dataset from an online resource using the following code. The first column is the digits. The remaining columns are the pixel values. After we download the dataset, we save it to our local disk so we do not have to re download the data in the future.

```
  # inputs to download file
  fileLocation <- "https://pjreddie.com/media/files/mnist_train.csv"
  numRowsToDownload <- 2500
  localFileName <- paste0("mnist_first", numRowsToDownload, ".RData")

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

  # save file
```

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

# you can load the data with the following code
load(file = localFileName)
```

a. [20 pts] The first task is to write the code to implement the K-Nearest Neighbors, or KNN, model from scratch. We will do this in steps:

- Write a function called `euclidean_distance` that calculates the Euclidean distance between two vectors. There are two input arguments for this function: vector 1 (`vec1`), and vector 2 (`vec2`). The output for this function is a numeric, the Euclidean distance (`euclDist`).
- Write a function called `manhattan_distance` that calculates the Manhattan distance between two vectors. There are two input arguments for this function: vector 1 (`vec1`), and vector 2 (`vec2`). The output for this function is a numeric, the Manhattan distance (`manhDist`).
- Write a function called `euclidean_distance_all` that calculates the Euclidean distance between a vector and all the row vectors in an input data matrix. There are two input arguments for this function: a vector (`vec1`) and an input data matrix (`mat1_X`). The output for this function is a vector (`output_euclDistVec`) which is of the same length as the number of rows in `mat1_X`. This function must use the function `euclidean_distance` you previously wrote.
- Write a function called `manhattan_distance_all` that calculates the Manhattan distance between a vector and all the row vectors in an input data matrix. There are two input arguments for this function: a vector (`vec1`) and an input data matrix (`mat1_X`). The output for this function is a vector (`output_manhattanDistVec`) which is of the same length as the number of rows in `mat1_X`. This function must use the function `manhattan_distance` you previously wrote.
- Write a function called `my_KNN` that compares a vector to a matrix and finds its K-nearest neighbors. There are five input arguments for this function: vector 1 (`vec1`), the input data matrix (`mat1_X`), the class labels corresponding to each row of the matrix (`mat1_Y`), the number of nearest neighbors you are interested in finding (`K`), and a Boolean argument specifying if we are using the Euclidean distance (`euclDistUsed`). The argument `K` should be a positive integer. If the argument `euclDistUsed = TRUE`, then use the Euclidean distance. Otherwise, use the Manhattan distance. The output of this function is a list of length 2 (`output_knnMajorityVote`). The first element in the output list should be a vector of length `K` containing the class labels of the closest neighbors. The second element in the output list should be the majority vote of the `K` class labels in the first element of the list. The function must use the functions `euclidean_distance` and `manhattan_distance` you previously wrote.

Apply this function to predict the label of the 123^{rd} observation using the first 100 observations as your input training data matrix. Use $K = 10$. What is the predicted label when you use Euclidean distance? What is the predicted label when you use Manhattan distance? Are these predictions correct?

```r
# Function to calculate Euclidean distance between two vectors
euclidean_distance <- function(vec1, vec2) {
  sqrt(sum((vec1 - vec2)^2))
}

# Function to calculate Manhattan distance between two vectors
manhattan_distance <- function(vec1, vec2) {
  sum(abs(vec1 - vec2))
}

# Function to calculate Euclidean distance from a vector to all rows in a matrix
```

```r
euclidean_distance_all <- function(vec1, mat1_X) {
  apply(mat1_X, 1, function(x) euclidean_distance(vec1, x))
}

# Function to calculate Manhattan distance from a vector to all rows in a matrix
manhattan_distance_all <- function(vec1, mat1_X) {
  apply(mat1_X, 1, function(x) manhattan_distance(vec1, x))
}

my_KNN <- function(vec1, mat1_X, mat1_Y, K, euclDistUsed = TRUE) {
  # Calculate distances using the appropriate distance function
  distances <- if (euclDistUsed) {
    euclidean_distance_all(vec1, mat1_X)
  } else {
    manhattan_distance_all(vec1, mat1_X)
  }

  # Obtain the indices of the K nearest neighbors
  nearest_indices <- order(distances)[1:K]

  # Extract the class labels of these nearest neighbors
  nearest_labels <- mat1_Y[nearest_indices]

  # Determine the majority vote among the K nearest labels
  majority_vote <- names(which.max(table(nearest_labels)))

  # Return both the labels and the majority vote
  list(nearest_labels = nearest_labels, majority_vote = majority_vote)
}

# Load the MNIST data (assuming it has already been loaded into 'mnist')
load(file = "mnist_first2500.RData")

# Extract the 123rd observation and first 100 training samples
vec1 <- as.numeric(mnist[123, -1])  # Remove the label column
mat1_X <- as.matrix(mnist[1:100, -1])  # Remove the label column for training data
mat1_Y <- mnist[1:100, 1]  # Labels for training data

# Using Euclidean distance
result_eucl <- my_KNN(vec1, mat1_X, mat1_Y, K = 10, euclDistUsed = TRUE)
print(result_eucl$majority_vote)
```

```
## [1] "7"
```

```r
# Using Manhattan distance
result_manh <- my_KNN(vec1, mat1_X, mat1_Y, K = 10, euclDistUsed = FALSE)
print(result_manh$majority_vote)
```

```
## [1] "7"
```

```r
# Check if predictions are correct
actual_label <- mnist[123, 1]
print(paste("Actual label:", actual_label))
```

```
## [1] "Actual label: 7"
```

The code defines functions to compute both Euclidean and Manhattan distances between vectors. It then calculates distances between the 123rd observation in the dataset and the first 100 observations using K-nearest neighbors (KNN). The majority vote from the 10 closest neighbors is determined, and both Euclidean and Manhattan distances predict the correct label of "7" for the 123rd observation. The predicted labels match the actual label, confirming the accuracy of both distance methods in this case.

b. [20 pts] Set the seed to 7 at the beginning of the chunk. Let's now use 20-fold cross-validation to select the best K. Now, load the the library `caret`. We will use the `trainControl` and `train` functions from this library to fit a KNN classification model. The K values we will consider are 1, 5, 10, 20, 50, 100. Be careful to not get confused between the number of folds and number of nearest neighbors when using the functions. Use the first 1250 observations as the training data to fit each model. Compare the results. What is the best K according to cross-validation classification accuracy? Once you have chosen K, fit a final KNN model on your entire training dataset with that value. Use that model to predict the classes of the last 1250 observations, which is our test dataset. Report the prediction confusion matrix on the test dataset for your final KNN model. Calculate the the test error and the sensitivity of each classes.

```r
library(caret)
```

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

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

```r
set.seed(7)

# Prepare training and test data
train_data <- mnist[1:1250, -1]  # First 1250 samples (excluding label column)
train_labels <- mnist[1:1250, 1]  # Corresponding labels

test_data <- mnist[1251:2500, -1]  # Last 1250 samples (test set)
test_labels <- mnist[1251:2500, 1]  # Test labels

# Define cross-validation method
control <- trainControl(method = "cv", number = 20)

# Define K values to test
k_values <- c(1, 5, 10, 20, 50, 100)

# Train the KNN model with cross-validation
knn_model <- train(x = train_data,
                   y = as.factor(train_labels),
                   method = "knn",
                   tuneGrid = expand.grid(k = k_values),
                   trControl = control)

# Evaluate the best K value
print(knn_model)
```

```
## k-Nearest Neighbors
##
## 1250 samples
##  784 predictor
##   10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
##
## No pre-processing
## Resampling: Cross-Validated (20 fold)
## Summary of sample sizes: 1190, 1189, 1188, 1187, 1187, 1187, ...
## Resampling results across tuning parameters:
##
##   k    Accuracy   Kappa
##     1  0.8847002  0.8715814
##     5  0.8758230  0.8616525
##    10  0.8567053  0.8402751
##    20  0.8279901  0.8081480
##    50  0.7840759  0.7589138
##   100  0.7159393  0.6823221
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 1.
```

```
best_k <- knn_model$bestTune$k
print(paste("Best K value based on cross-validation: ", best_k))
```

```
## [1] "Best K value based on cross-validation:  1"
```

```
# Train the final model using the best K
final_knn_model <- knn(train = train_data, test = test_data, cl = train_labels, k = best_k)

# Confusion matrix for the test data
conf_matrix <- confusionMatrix(as.factor(final_knn_model), as.factor(test_labels))
print(conf_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 112   0   2   1   0   0   2   0   0   1
##          1   1 126   5   2   6   1   1   2   1   1
##          2   0   1 106   1   1   0   0   0   2   0
##          3   1   0   0 104   0   2   0   0   1   1
##          4   0   0   2   0 118   1   1   0   0   7
##          5   0   0   0   7   1 110   2   0   5   0
##          6   3   0   1   0   1   4 133   0   1   0
##          7   0   1   7   3   5   0   0 120   0   5
##          8   0   0   2   4   0   0   0   0  96   0
##          9   0   0   0   0  12   1   0   4   4 105
##
## Overall Statistics
##
##                Accuracy : 0.904
##                  95% CI : (0.8863, 0.9198)
```

```
##     No Information Rate : 0.1152
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8933
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9573   0.9844   0.8480   0.8525   0.8194   0.9244
## Specificity            0.9947   0.9822   0.9956   0.9956   0.9901   0.9867
## Pos Pred Value         0.9492   0.8630   0.9550   0.9541   0.9147   0.8800
## Neg Pred Value         0.9956   0.9982   0.9833   0.9842   0.9768   0.9920
## Prevalence             0.0936   0.1024   0.1000   0.0976   0.1152   0.0952
## Detection Rate         0.0896   0.1008   0.0848   0.0832   0.0944   0.0880
## Detection Prevalence   0.0944   0.1168   0.0888   0.0872   0.1032   0.1000
## Balanced Accuracy      0.9760   0.9833   0.9218   0.9240   0.9047   0.9556
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.9568   0.9524   0.8727   0.8750
## Specificity            0.9910   0.9813   0.9947   0.9814
## Pos Pred Value         0.9301   0.8511   0.9412   0.8333
## Neg Pred Value         0.9946   0.9946   0.9878   0.9867
## Prevalence             0.1112   0.1008   0.0880   0.0960
## Detection Rate         0.1064   0.0960   0.0768   0.0840
## Detection Prevalence   0.1144   0.1128   0.0816   0.1008
## Balanced Accuracy      0.9739   0.9668   0.9337   0.9282
```

```
#: Calculate the test error and sensitivity for each class
test_error <- 1 - sum(final_knn_model == test_labels) / length(test_labels)
print(paste("Test Error: ", test_error))
```

```
## [1] "Test Error:  0.096"
```

```
sensitivity <- conf_matrix$byClass[, "Sensitivity"]
print(sensitivity)
```

```
##  Class: 0  Class: 1  Class: 2  Class: 3  Class: 4  Class: 5  Class: 6  Class: 7
## 0.9572650 0.9843750 0.8480000 0.8524590 0.8194444 0.9243697 0.9568345 0.9523810
##  Class: 8  Class: 9
## 0.8727273 0.8750000
```

The code performs K-Nearest Neighbors (KNN) classification using 20-fold cross-validation to select the best value of K from the candidate values $K = 1, 5, 10, 20, 50, 100$. After training on the first 1250 samples, $K = 1$ is selected as the best value based on the highest accuracy.

After fitting the KNN model with $K = 1$ on the training data and predicting the last 1250 samples (test data), the confusion matrix shows an accuracy of 90.4%, and the test error is 9.6%. The sensitivity for each class (0 to 9) varies, with most classes having high sensitivity, such as class 1 (98.44%) and class 0 (95.73%). However, some classes, like class 4 (81.94%) and class 8 (87.27%), have lower sensitivity. Overall, the model performs well with a balanced accuracy for each class.

c. [10 pts] Set the seed to 7 at the beginning of the chunk. Now let's try to use multi-class (i.e., multinomial) logistic regression to fit the data. Use the first 1250 observations as the training data and the rest as the testing data. Load the library `glmnet`. We will use a multi-class logistic regression model with a Lasso penalty. First, we seek to find an almost optimal value for the λ penalty parameter. Use the `cv.glmnet` function with 20 folds on the training dataset to find λ_{1se}. Once you have identified λ_{1se}, use the `glmnet()` function with that penalty value to fit a multi-class logistic regression model onto the entire training dataset. Ensure you set the argument `family = multinomial` within the functions as appropriate. Using that model, predict the class label for the testing data. Report the testing data prediction confusion matrix. What is the test error?

```
# Set seed and load libraries
set.seed(7)
library(glmnet)
library(caret)

# Prepare the data
# Convert predictors to numeric matrices
train_X <- data.matrix(train_data)  # Exclude label column from training predictors
test_X <- data.matrix(test_data)

# Ensure correct extraction of training labels
train_Y <- factor(train_labels, levels = as.character(0:9))
test_Y <- factor(test_labels, levels = as.character(0:9))

# Perform cross-validation to find lambda_1se
cv_model <- cv.glmnet(
  x = train_X,
  y = train_Y,
  family = "multinomial",
  alpha = 1,
  nfolds = 20
)

# Extract lambda_1se
lambda_1se <- cv_model$lambda.1se
cat("Optimal lambda (lambda_1se):", lambda_1se, "\n")
```

```
## Optimal lambda (lambda_1se): 0.006196452
```

```
# Fit the final model on the entire training data
final_model <- glmnet(
  x = train_X,
  y = train_Y,
  family = "multinomial",
  alpha = 1
)

# Predict on test data using lambda_1se
test_pred <- predict(final_model, newx = test_X, s = lambda_1se, type = "class")

# Ensure predictions are factors with correct levels
test_pred <- factor(test_pred, levels = as.character(0:9))
```

```r
# Evaluate the model
confusion_mat <- confusionMatrix(test_pred, test_Y)
print(confusion_mat)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 109   0   4   0   2   2   2   2   1   1
##          1   0 115   3   4   2   0   1   1   4   0
##          2   1   2 107   8   3   0   7   1   4   0
##          3   0   0   0  99   0   5   0   4   1   4
##          4   1   0   4   0 115   2   2   2   1   8
##          5   2   1   0   7   2  98   4   0   3   2
##          6   0   1   3   1   1   3 122   0   1   0
##          7   0   1   2   1   3   3   1 109   2   6
##          8   4   8   2   1   4   6   0   0  89   0
##          9   0   0   0   1  12   0   0   7   4  99
##
## Overall Statistics
##
##                Accuracy : 0.8496
##                  95% CI : (0.8286, 0.869)
##     No Information Rate : 0.1152
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8328
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9316   0.8984   0.8560   0.8115   0.7986   0.8235
## Specificity            0.9876   0.9866   0.9769   0.9876   0.9819   0.9814
## Pos Pred Value         0.8862   0.8846   0.8045   0.8761   0.8519   0.8235
## Neg Pred Value         0.9929   0.9884   0.9839   0.9798   0.9740   0.9814
## Prevalence             0.0936   0.1024   0.1000   0.0976   0.1152   0.0952
## Detection Rate         0.0872   0.0920   0.0856   0.0792   0.0920   0.0784
## Detection Prevalence   0.0984   0.1040   0.1064   0.0904   0.1080   0.0952
## Balanced Accuracy      0.9596   0.9425   0.9164   0.8995   0.8903   0.9025
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.8777   0.8651   0.8091   0.8250
## Specificity            0.9910   0.9831   0.9781   0.9788
## Pos Pred Value         0.9242   0.8516   0.7807   0.8049
## Neg Pred Value         0.9848   0.9848   0.9815   0.9814
## Prevalence             0.1112   0.1008   0.0880   0.0960
## Detection Rate         0.0976   0.0872   0.0712   0.0792
## Detection Prevalence   0.1056   0.1024   0.0912   0.0984
## Balanced Accuracy      0.9343   0.9241   0.8936   0.9019
```

```r
# Calculate test error
test_error <- 1 - confusion_mat$overall['Accuracy']
```

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

```
## Test Error: 0.1504
```

A multi-class logistic regression model with a Lasso penalty is applied to the MNIST dataset. After setting the seed and performing cross-validation using the `cv.glmnet` function, the optimal value for the penalty parameter λ_{1se} is determined to be 0.006196. The final model is then trained on the first 1250 samples, and predictions are made on the remaining 1250 test samples.

The resulting confusion matrix shows an accuracy of 84.96%, with a test error of 15.04%. The sensitivity of the classes ranges from 80.91% to 93.16%, indicating that the model performs reasonably well across most digit classes. However, some classes, like class 8 (80.91%) and class 9 (82.50%), show slightly lower sensitivity. Overall, the model provides a good fit for the data with a balanced accuracy for each class.