Stat 432 Homework 5

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

Contents

Instruction

Please remove this section when submitting your homework.

Students are encouraged to work together on homework and/or utilize advanced AI tools. However, **sharing, copying, or providing any part of a homework solution or code to others** is an infraction of the University's rules on Academic Integrity. Any violation will be punished as severely as possible. Final submissions must be uploaded to Gradescope. No email or hard copy will be accepted. For **late submission policy and grading rubrics**, please refer to the course website.

- You are required to submit the rendered file `HWx_yourNetID.pdf`. For example, `HW01_rqzhu.pdf`. Please note that this must be a `.pdf` file. `.html` format **cannot** be accepted. Make all of your `R` code chunks visible for grading.
- Include your Name and NetID in the report.
- If you use this file or the example homework `.Rmd` file as a template, be sure to **remove this instruction** section.
- Make sure that you **set seed** properly so that the results can be replicated if needed.
- For some questions, there will be restrictions on what packages/functions you can use. Please read the requirements carefully. As long as the question does not specify such restrictions, you can use anything.
- **When using AI tools**, you are encouraged to document your comment on your experience with AI tools especially when it's difficult for them to grasp the idea of the question.
- **On random seed and reproducibility**: Make sure the version of your `R` is $\geq$ 4.0.0. This will ensure your random seed generation is the same as everyone else. Please note that updating the `R` version may require you to reinstall all of your packages.

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.

```
set.seed(432)
beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)

# Calculate the true mean
true_mean <- exp(sum(beta * x0))
true_mean
```

```
## [1] 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.

```
set.seed(432)
library(FNN)
beta <- c(0.5, -0.5, 0)
n <- 100
k <- 21


X_train <- matrix(runif(n * 3, 0, 1), ncol = 3)
epsilon <- rnorm(n, mean = 0, sd = 1)
Y_train <- exp(X_train %*% beta) + epsilon

# Define test point x0
x0 <- c(1, -0.75, -0.7)

# Function to compute Euclidean distance
euclidean_distance <- function(x1, x2) {
  sqrt(sum((x1 - x2)^2))
}

# Compute distances from x0 to each point in X_train
distances <- apply(X_train, 1, function(row) euclidean_distance(row, x0))

# Find indices of the k nearest neighbors
nearest_neighbors_idx <- order(distances)[1:k]

# Calculate the KNN prediction (average Y of the nearest neighbors)
knn_prediction <- mean(Y_train[nearest_neighbors_idx])
knn_prediction
```

```
## [1] 1.285897
```

```
validation <- knn.reg(train = X_train, test = matrix(x0, nrow = 1), y = Y_train, k = k)
validation$pred
```

```
## [1] 1.285897
```

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.

```
set.seed(432)
beta <- c(0.5, -0.5, 0)
n <- 100
k <- 21
n_sim <- 1000

x0 <- c(1, -0.75, -0.7)

f_x0 <- exp(sum(beta * x0))

euclidean_distance <- function(x1, x2) {
  sqrt(sum((x1 - x2)^2))
}

knn_predictions <- numeric(n_sim)

# Perform simulation
for (i in 1:n_sim) {

  # Generate training data
  X_train <- matrix(runif(n * 3, 0, 1), ncol = 3)  # 100 rows, 3 columns
  epsilon <- rnorm(n, mean = 0, sd = 1)
  Y_train <- exp(X_train %*% beta) + epsilon

  # Compute distances from x0 to each point in X_train
  distances <- apply(X_train, 1, function(row) euclidean_distance(row, x0))

  # Find indices of the k nearest neighbors
  nearest_neighbors_idx <- order(distances)[1:k]

  # Calculate KNN prediction (average Y of the nearest neighbors)
  knn_predictions[i] <- mean(Y_train[nearest_neighbors_idx])
}

# Estimate the bias
estimated_bias <- mean(knn_predictions) - f_x0
estimated_bias
```

```
## [1] -1.172899
```

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.

```
set.seed(432)
mean_knn_prediction <- mean(knn_predictions)

# Step 2: Compute the squared deviations from the mean
squared_deviations <- (knn_predictions - mean_knn_prediction)^2

# Step 3: Estimate the variance
estimated_variance <- mean(squared_deviations)
estimated_variance
```

```
## [1] 0.04865328
```

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?

```
set.seed(432)
# Irreducible error (variance of epsilon)
irreducible_error <- 1

# Bias was previously calculated
bias_squared <- estimated_bias^2

# Variance was previously calculated
variance <- estimated_variance

# Total prediction error
prediction_error <- irreducible_error + bias_squared + variance
prediction_error
```

```
## [1] 2.424345
```

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.

```
set.seed(432)
# Define true function value at x0
f_x0 <- exp(sum(beta * x0))

# Initialize a vector to store the empirical prediction errors
empirical_errors <- numeric(n_sim)

# Perform the simulation and calculate empirical prediction error
for (i in 1:n_sim) {
  # Generate a new Y_0 using the true model
```

```
  epsilon_test <- rnorm(1, mean = 0, sd = 1)  # Noise term for testing data
  Y_0 <- f_x0 + epsilon_test  # True value of Y_0

  # Calculate the squared prediction error
  empirical_errors[i] <- (knn_predictions[i] - Y_0)^2
}

# Calculate the average empirical prediction error
empirical_prediction_error <- mean(empirical_errors)
empirical_prediction_error
```

```
## [1] 2.336539
```

```
prediction_error-empirical_prediction_error
```

```
## [1] 0.08780595
```

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.

```
  # 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
glm_model <- glm(default ~ balance + income, data = Default_train, family = "binomial")
summary(glm_model)
```

```
##
## Call:
## glm(formula = default ~ balance + income, family = "binomial",
##     data = Default_train)
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.161e+01  4.608e-01 -25.197  < 2e-16 ***
## balance      5.654e-03  2.401e-04  23.547  < 2e-16 ***
## income       2.275e-05  5.291e-06   4.299 1.71e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2630.6  on 8999  degrees of freedom
## Residual deviance: 1424.6  on 8997  degrees of freedom
## AIC: 1430.6
##
## Number of Fisher Scoring iterations: 8
```

```r
loglikelihood <- function(beta, X, Y) {
  # Calculate predicted probabilities using logistic function
  p <- 1 / (1 + exp(-X %*% beta))

  # Calculate log-likelihood
  log_lik <- sum(Y * log(p) + (1 - Y) * log(1 - p))

  return(log_lik)
}

# Prepare data for log-likelihood calculation
X_train <- as.matrix(cbind(1, Default_train$balance, Default_train$income))  # Add intercept term
Y_train <- as.numeric(Default_train$default == "Yes")  # Convert to binary (0, 1)

# Extract estimated coefficients from the glm model
beta_estimates <- coef(glm_model)

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

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

```r
deviance_glm <- glm_model$deviance
deviance_glm
```

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

```r
calculated_deviance <- -2 * max_loglik
calculated_deviance
```

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

The relationship is shown above 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?

```r
probabilities <- predict(glm_model, newdata = Default_test, type = "response")
calculate_metrics <- function(probabilities, Y_test, cutoff) {
  # Classify based on cutoff
  predictions <- ifelse(probabilities > cutoff, "Yes", "No")

  # Convert true values to binary
  Y_test_binary <- as.numeric(Default_test$default == "Yes")
  predictions_binary <- as.numeric(predictions == "Yes")

  # Calculate confusion matrix components
  TP <- sum(predictions_binary == 1 & Y_test_binary == 1)  # True Positives
  TN <- sum(predictions_binary == 0 & Y_test_binary == 0)  # True Negatives
  FP <- sum(predictions_binary == 1 & Y_test_binary == 0)  # False Positives
  FN <- sum(predictions_binary == 0 & Y_test_binary == 1)  # False Negatives

  # Calculate metrics
  test_error <- (FP + FN) / length(Y_test_binary)  # Test error
  sensitivity <- TP / (TP + FN)  # Sensitivity
  specificity <- TN / (TN + FP)  # Specificity
  precision <- TP / (TP + FP)  # Precision

  # Print confusion matrix
  print(paste("Cutoff:", cutoff))
  print(paste("Confusion Matrix:"))
  print(matrix(c(TN, FP, FN, TP), 2, 2, byrow = TRUE, dimnames = list(c("Actual No", "Actual Yes"), c("

  # Return calculated values
  return(list(test_error = test_error, sensitivity = sensitivity, specificity = specificity, precision =
}

# Apply the function for each cutoff
metrics_0.3 <- calculate_metrics(probabilities, Default_test$default, 0.3)
```

```
## [1] "Cutoff: 0.3"
## [1] "Confusion Matrix:"
##            Pred No Pred Yes
## Actual No      954       13
## Actual Yes      14       19
```

```r
metrics_0.5 <- calculate_metrics(probabilities, Default_test$default, 0.5)
```

```
## [1] "Cutoff: 0.5"
## [1] "Confusion Matrix:"
##            Pred No Pred Yes
## Actual No      963        4
## Actual Yes      22       11
```

```r
metrics_0.7 <- calculate_metrics(probabilities, Default_test$default, 0.7)
```

```
## [1] "Cutoff: 0.7"
## [1] "Confusion Matrix:"
##            Pred No Pred Yes
## Actual No      964        3
## Actual Yes      29        4
```

```r
# Display the calculated metrics
metrics_0.3
```

```
## $test_error
## [1] 0.027
##
## $sensitivity
## [1] 0.5757576
##
## $specificity
## [1] 0.9865564
##
## $precision
## [1] 0.59375
```

```r
metrics_0.5
```

```
## $test_error
## [1] 0.026
##
## $sensitivity
## [1] 0.3333333
##
## $specificity
## [1] 0.9958635
##
## $precision
## [1] 0.7333333
```

```r
metrics_0.7
```

```
## $test_error
## [1] 0.032
##
```

```
## $sensitivity
## [1] 0.1212121
##
## $specificity
## [1] 0.9968976
##
## $precision
## [1] 0.5714286
```

I personally prefer the one with highest precision, which is the 0.5 cutoff. If our goal is to capture as many people who will default as possible (high sensitivity), without worrying too much about false positives (misclassifying people as Default=Yes even if they will not default), we should use cutoff = 0.3.

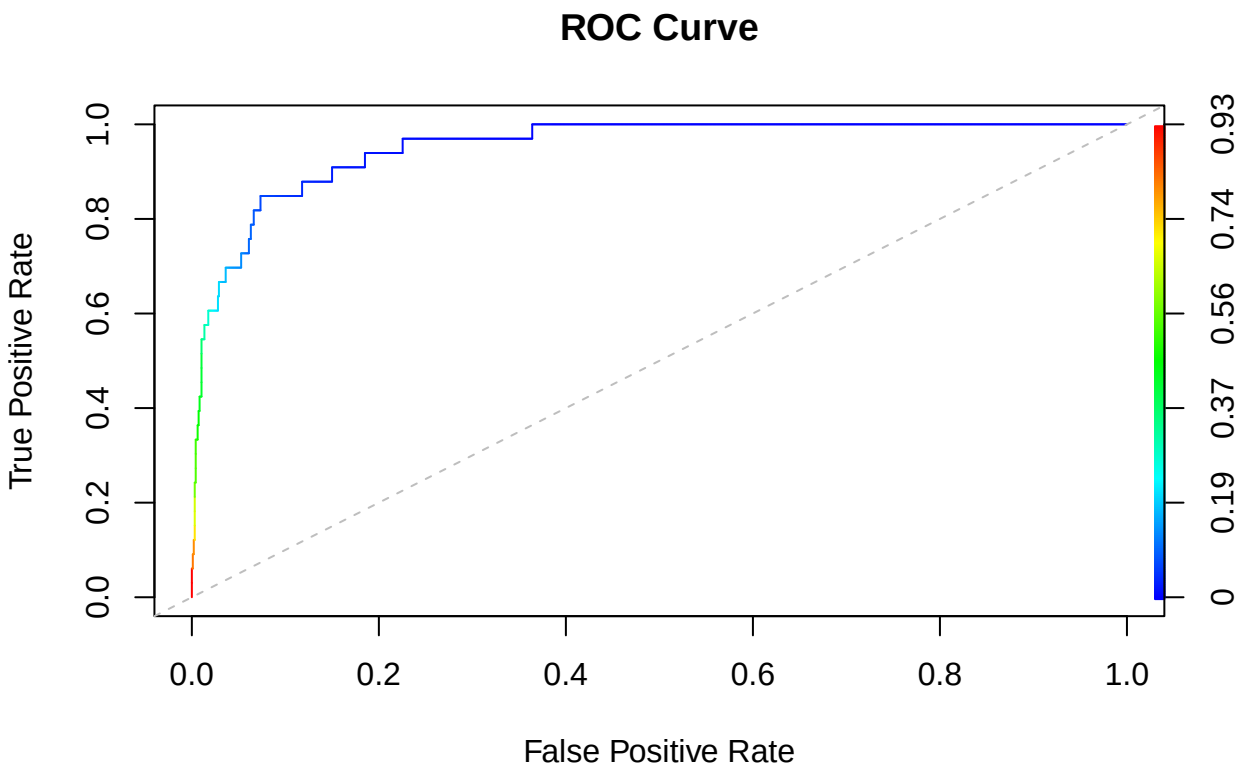
It has the highest sensitivity (0.5758), meaning it correctly identifies the highest number of true defaults.

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.

```r
library(ROCR)
probabilities <- predict(glm_model, newdata = Default_test, type = "response")
pred <- prediction(probabilities, Default_test$default)

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

# Plot ROC curve
plot(perf, colorize = TRUE, main = "ROC Curve", xlab = "False Positive Rate", ylab = "True Positive Rat
abline(a = 0, b = 1, col = "gray", lty = 2)  # Diagonal reference line
```




```
# Calculate AUC
auc_perf <- performance(pred, measure = "auc")
auc_value <- auc_perf@y.values[[1]]
auc_value
```

```
## [1] 0.9523048
```

Looking at the color gradient on the ROC curve, the cutoff value that corresponds to a point where the True Positive Rate is high (close to 1) and the False Positive Rate is still relatively low (close to 0) seems to be around 0.37 - 0.56 (green section of the curve). This would likely be a good cutoff value, since it offers a good trade-off between correctly identifying true positives while minimizing false positives.

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.

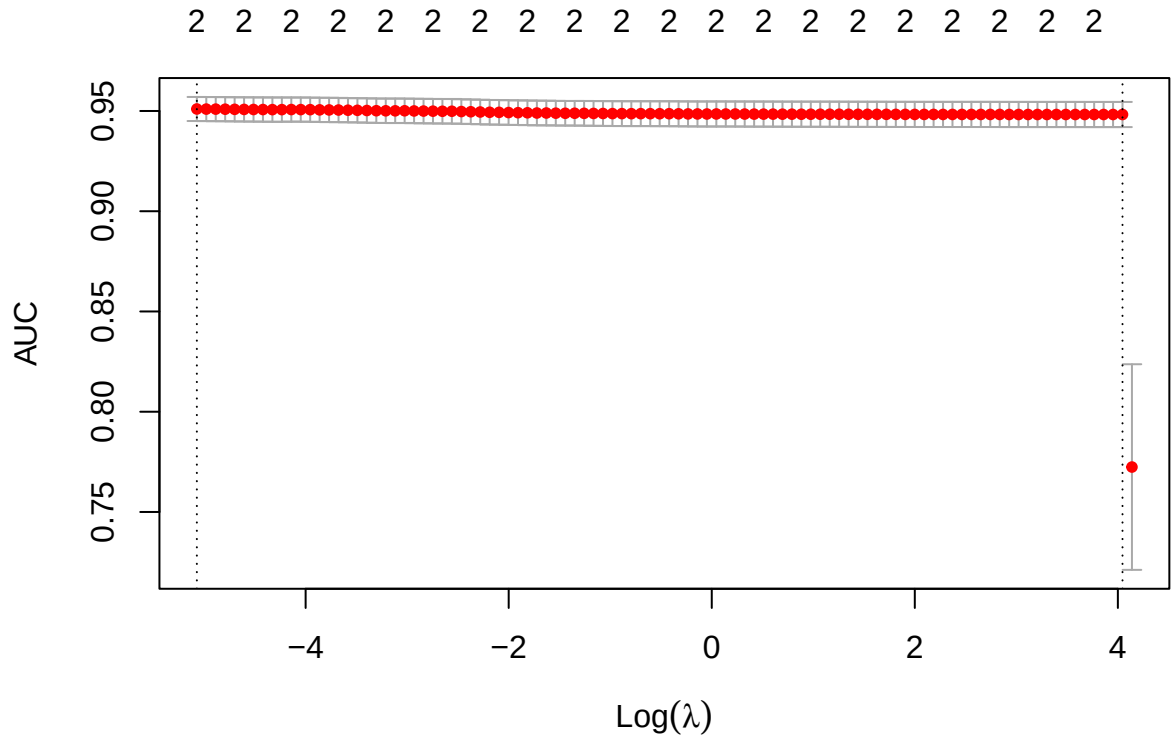
```
library(glmnet)
```

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

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

```
X_train_glmnet <- as.matrix(Default_train[, c("balance", "income")])
Y_train_glmnet <- as.numeric(Default_train$default == "Yes")
cv_model <- cv.glmnet(
  X_train_glmnet,
  Y_train_glmnet,
  family = "binomial",
  alpha = 0,
  type.measure = "auc",
  nfolds = 20
)

# Plot the cross-validation results
plot(cv_model)
```




```
lambda_min <- cv_model$lambda.min
lambda_min
```

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

```
# Lambda within 1 standard error of the optimal lambda
lambda_1se <- cv_model$lambda.1se
lambda_1se
```

```
## [1] 57.15298
```

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
  # 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`).

```
euclidean_distance <- function(vec1, vec2) {
  euclDist <- sqrt(sum((vec1 - vec2)^2))
  return(euclDist)
}
```

```
  - Write a function called ‘manhattan_distance‘ that calculates the Manhattan distance between two vec
```

```
manhattan_distance <- function(vec1, vec2) {
  manhDist <- sum(abs(vec1 - vec2))
  return(manhDist)
}
```

```
  - Write a function called ‘euclidean_distance_all‘ that calculates the Euclidean distance between a v
```

```
euclidean_distance_all <- function(vec1, mat1_X) {
  output_euclDistVec <- apply(mat1_X, 1, function(row) euclidean_distance(vec1, row))
  return(output_euclDistVec)
}
```

- Write a function called ‘manhattan_distance_all‘ that calculates the Manhattan distance between a v

```
manhattan_distance_all <- function(vec1, mat1_X) {
  output_manhattanDistVec <- apply(mat1_X, 1, function(row) manhattan_distance(vec1, row))
  return(output_manhattanDistVec)
}
```

- Write a function called ‘my_KNN‘ that compares a vector to a matrix and finds its K-nearest neighbo

```
my_KNN <- function(vec1, mat1_X, mat1_Y, K, euclDistUsed = TRUE) {

  # Calculate distances
  if (euclDistUsed) {
    distances <- euclidean_distance_all(vec1, mat1_X)
  } else {
    distances <- manhattan_distance_all(vec1, mat1_X)
  }

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

  # Get the class labels of the K-nearest neighbors
  neighbor_labels <- mat1_Y[neighbor_indices]

  # Majority vote
  majority_vote <- names(sort(table(neighbor_labels), decreasing = TRUE))[1]

  # Return a list: 1st element is the class labels of K neighbors, 2nd element is majority vote
  output_knnMajorityVote <- list(neighbor_labels, majority_vote)

  return(output_knnMajorityVote)
}
```

Apply this function to predict the label of the 123^{rd} observation using the first 100 o

```
train_data <- mnist[1:100, -1]  # Pixel values only (columns 2 to end)
train_labels <- mnist$Digit[1:100]  # Corresponding digit labels

# Test data: 123rd observation
test_data <- mnist[123, -1]  # Pixel values for 123rd observation
test_label <- mnist$Digit[123]  # True label for 123rd observation

# K = 10
K <- 10

# Predict with Euclidean distance
knn_euclidean <- my_KNN(test_data, train_data, train_labels, K, euclDistUsed = TRUE)
print(paste("Predicted label (Euclidean):", knn_euclidean[[2]]))
```

```
## [1] "Predicted label (Euclidean): 7"
```

```r
print(paste("True label:", test_label))
```

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

```r
# Predict with Manhattan distance
knn_manhattan <- my_KNN(test_data, train_data, train_labels, K, euclDistUsed = FALSE)
print(paste("Predicted label (Manhattan):", knn_manhattan[[2]]))
```

```
## [1] "Predicted label (Manhattan): 7"
```

```r
print(paste("True label:", test_label))
```

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

They produce the same prediction and they are both correct.

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
set.seed(7)
library(caret)
```

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

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

```r
train_data <- mnist[1:1250, -1]  # Pixel values (remove 'Digit' column)
train_labels <- as.factor(mnist$Digit[1:1250])  # Class labels

# Set up 20-fold cross-validation
train_control <- trainControl(method = "cv", number = 20)

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

# Train KNN model with 20-fold CV for different K values
knn_model <- train(
  x = train_data,
  y = train_labels,
  method = "knn",
  trControl = train_control,
```

```
  tuneGrid = expand.grid(k = k_values)
)

# Check the results
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:", best_k))
```

```
## [1] "Best K: 1"
```

```
test_data <- mnist[1251:2500, -1]  # Pixel values
test_labels <- as.factor(mnist$Digit[1251:2500])  # True labels

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

# Check the confusion matrix
confusion_matrix <- table(Predicted = final_knn_model, Actual = test_labels)
print(confusion_matrix)
```

```
##           Actual
## Predicted   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
```

```
test_error <- sum(final_knn_model != test_labels) / length(test_labels)
print(paste("Test Error:", test_error))
```

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

```
# Calculate sensitivity for each class
sensitivity <- function(conf_matrix, class_label) {
  TP <- conf_matrix[class_label, class_label]
  FN <- sum(conf_matrix[class_label, ]) - TP
  sensitivity_value <- TP / (TP + FN)
  return(sensitivity_value)
}

# Apply sensitivity calculation for each class in the confusion matrix
for (class_label in rownames(confusion_matrix)) {
  sens_value <- sensitivity(confusion_matrix, class_label)
  print(paste("Sensitivity for class", class_label, ":", sens_value))
}
```

```
## [1] "Sensitivity for class 0 : 0.949152542372881"
## [1] "Sensitivity for class 1 : 0.863013698630137"
## [1] "Sensitivity for class 2 : 0.954954954954955"
## [1] "Sensitivity for class 3 : 0.954128440366973"
## [1] "Sensitivity for class 4 : 0.914728682170543"
## [1] "Sensitivity for class 5 : 0.88"
## [1] "Sensitivity for class 6 : 0.93006993006993"
## [1] "Sensitivity for class 7 : 0.851063829787234"
## [1] "Sensitivity for class 8 : 0.941176470588235"
## [1] "Sensitivity for class 9 : 0.833333333333333"
```

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(7)

# Load glmnet library
```

```r
library(glmnet)

# Assuming `mnist` dataset is loaded and preprocessed (replace with the correct dataset if needed)
train_data <- as.matrix(mnist[1:1250, -1])  # Pixel values (features)
train_labels <- as.factor(mnist$Digit[1:1250])  # Class labels

test_data <- as.matrix(mnist[1251:nrow(mnist), -1])  # Testing pixel values
test_labels <- as.factor(mnist$Digit[1251:nrow(mnist)])  # Testing class labels

# Perform 20-fold cross-validation to find lambda_1se
cv_model <- cv.glmnet(
  x = train_data,
  y = train_labels,
  family = "multinomial",
  alpha = 1,  # Lasso penalty
  nfolds = 20,
  type = "class"
)

# Extract the optimal lambda (lambda_1se)
lambda_1se <- cv_model$lambda.1se
print(paste("Optimal lambda (1se):", lambda_1se))
```

```
## [1] "Optimal lambda (1se): 0.0068005988239705"
```

```r
lambdas <- cv_model$lambda
# Fit the final model using lambda_1se on the entire training dataset
final_model <- glmnet(
  x = train_data,
  y = train_labels,
  family = "multinomial",
  alpha = 1,
  lambda = lambdas,
)

predicted <- predict(final_model, newx = test_data, s = lambda_1se, type = "class")
predicted <- as.factor(predicted)
confusionMatrix(predicted,test_labels)
```

```
## 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   0 117   3   4   2   0   1   1   4   0
##          2   1   2 108   8   3   0   7   1   4   0
##          3   0   0   0  99   0   6   0   4   1   4
##          4   1   0   3   0 115   1   2   2   1   8
##          5   2   1   0   7   2  97   4   0   4   2
##          6   0   1   3   1   1   3 122   0   1   0
##          7   0   1   2   1   3   3   1 109   2   5
##          8   4   6   2   1   4   7   0   0  88   0
```

```
##          9   0   0   0   1  12   0   0   7   4 100
##
## Overall Statistics
##
##                Accuracy : 0.8512
##                  95% CI : (0.8302, 0.8705)
##     No Information Rate : 0.1152
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8346
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9316   0.9141   0.8640   0.8115   0.7986   0.8151
## Specificity            0.9876   0.9866   0.9769   0.9867   0.9837   0.9805
## Pos Pred Value         0.8862   0.8864   0.8060   0.8684   0.8647   0.8151
## Neg Pred Value         0.9929   0.9902   0.9848   0.9798   0.9740   0.9805
## Prevalence             0.0936   0.1024   0.1000   0.0976   0.1152   0.0952
## Detection Rate         0.0872   0.0936   0.0864   0.0792   0.0920   0.0776
## Detection Prevalence   0.0984   0.1056   0.1072   0.0912   0.1064   0.0952
## Balanced Accuracy      0.9596   0.9503   0.9204   0.8991   0.8912   0.8978
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.8777   0.8651   0.8000   0.8333
## Specificity            0.9910   0.9840   0.9789   0.9788
## Pos Pred Value         0.9242   0.8583   0.7857   0.8065
## Neg Pred Value         0.9848   0.9849   0.9807   0.9822
## Prevalence             0.1112   0.1008   0.0880   0.0960
## Detection Rate         0.0976   0.0872   0.0704   0.0800
## Detection Prevalence   0.1056   0.1016   0.0896   0.0992
## Balanced Accuracy      0.9343   0.9245   0.8895   0.9060
```

```
# Calculate the test error
test_error <- 1 - sum(diag(confusionMatrix(predicted,test_labels)$table))/sum((confusionMatrix(predicte
print(paste("Test Error:", test_error))
```

```
## [1] "Test Error: 0.1488"
```