Stat 432 Homework 5

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

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 (https://studentcode.illinois.edu/article1/part4/1-401/). Any violation will be punished as severely as possible. Final submissions must be uploaded to Gradescope (https://www.gradescope.com/courses/570816). No email or hard copy will be accepted. For **late submission policy and grading rubrics** (https://teazrq.github.io/stat432/syllabus.html), 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.

Solution:

The true mean of Y at the testing point x_0 is

$$\exp(\beta^T x_0) \approx 0.852$$

```r
# true beta
b = c(0.5, -0.5, 0)

# testing point
x0 = c(1, -0.75, -0.7)

# true mean
f0 = exp(sum(b * x0))
f0
```

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

The true mean of Y at the testing point x_0 is 2.3988753.

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.

Solution:

```r
# set seed
set.seed(432)

# generate training data
n = 100
p = 3
X = matrix(runif(3 * n), ncol = 3)
Y = exp(X %*% b) + rnorm(n)

# find the closest k points
k = 21
distance = rowSums(sweep(X, 2, x0, "-")^2)
index = order(distance)[1:k]

# prediction
mean(Y[index])
```

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

```r
# validate that with the kknn package
library(FNN)
knn.fit = knn.reg(train = X, test = data.frame(x = t(x0)), y = Y, k = k, algorithm = "brute")
knn.fit$pred
```

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

The prediction is 1.2858967. This matches the result from the `knn.reg()` function.

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.

Solution:

```
# simulation
nsim = 1000
fhat = rep(NA, nsim)

for (i in 1:nsim) {
  # generate training data
  X = matrix(runif(3 * n), ncol = 3)
  Y = exp(X %*% b) + rnorm(n)

  # find the closest k points
  distance = rowSums(sweep(X, 2, x0, "-")^2)
  index = order(distance)[1:k]

  # predicted label
  fhat[i] = mean(Y[index])
}

# bias
bias = mean(fhat) - f0
bias
```

```
## [1] -1.17298
```

The estimated bias of the KNN model for predicting x_0 is -1.17298.

d. [1 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.

Solution:

```
# variance
variance = var(fhat)
variance
```

```
## [1] 0.04869883
```

The estimated variance of the KNN model for predicting x_0 is 0.0486988.

e. [3 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?

Solution:

Based on the bias-variance decomposition, the irreducible error is the variance of the noise term, which is 1. Based on our previous simulation, the Bias^2 error is 1.375882 and the variance error is 0.0486988. Therefore, the prediction error is

```
# prediction error
1 + bias^2 + variance
```

```
## [1] 2.424581
```

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.

Solution:

```
# simulation
error = rep(NA, nsim)

for (i in 1:nsim) {
  # generate training data
  X = matrix(runif(3 * n), ncol = 3)
  Y = exp(X %*% b) + rnorm(n)

  # find the closest k points
  distance = rowSums(sweep(X, 2, x0, "-")^2)
  index = order(distance)[1:k]

  # predicted label
  y_pred = mean(Y[index])

  # generate testing data
  Y0 = exp(x0 %*% b) + rnorm(1)

  # prediction error
  error[i] = (Y0 - y_pred)^2
}

# prediction error
mean(error)
```

```
## [1] 2.489355
```

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?

Solution:

```
  # encode default so "No" -> and "Yes" -> 1
  Default_train$default <- ifelse(Default_train$default == "Yes", 1, 0)
  Default_test$default <- ifelse(Default_test$default == "Yes", 1, 0)

  # fit logistic regression model for training dataset
  logistic.Default <- glm(default ~ balance + income,
                          data = Default_train,
                          family = binomial(link = "logit"))

  # Function Name: loglikelihood
  # Function Usage: to calculate the log likelihood
  # Input 1: beta, a vector of length p
  # Input 2: X, a matrix of dimension n x p
  # Input 3: Y, a vector of length n
  # Output: output_loglik, a numeric
  loglikelihood <- function(beta, X, Y){
    beta <- as.matrix(beta)
    Xbeta <- X %*% beta
    output_loglik <- sum(Y*Xbeta - log(1 + exp(Xbeta)))
    return(output_loglik)
  }

  # calculate the maximum log likelihood
  Default_train_X <- as.matrix(cbind(1, Default_train[, 3:4]))
  Default_train_Y <- Default_train[,1]
  Default_train_maximumLikelihood <- loglikelihood(beta = logistic.Default$coef,
                                                   X = Default_train_X,
                                                   Y = Default_train_Y)
  print(paste0("Maximum Log Likelihood: ", Default_train_maximumLikelihood))
```

```
## [1] "Maximum Log Likelihood: -712.29808123702"
```

```
  # deviance
  Default_train_deviance <- logistic.Default$deviance
  print(paste0("Deviance: ", Default_train_deviance))
```

```
## [1] "Deviance: 1424.59616247404"
```

```
  # relationship between deviance and log likelihood
  print(-2*loglikelihood(logistic.Default$coef, Default_train_X, Default_train_Y))
```

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

The relationship is $\text{Deviance} = -2 * \text{Maximum Log Likelihood}$.

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?

Solution:

```r
# predict the probability of default for test dataset
pred.Default <- predict(logistic.Default, newdata = Default_test[,3:4], type = "response")

# cutoff values
cutoffValues <- c(0.3, 0.5, 0.7)
numCutoffValues <- length(cutoffValues)

# loop through each cutoff value
# calculate test error, sensitivity, specificity, precision
for(i in 1:numCutoffValues){
  # particular cutoff value
  cutoffValueTemp <- cutoffValues[i]
  print(paste0("Cutoff: ", cutoffValueTemp))

  # apply cutoff
  pred.Default_class1 <- ifelse(pred.Default > cutoffValueTemp, "Yes", "No")
  confusionMatTemp <- table(Predicted = pred.Default_class1,
                            Actual = Default_test$default)

  # print confusion matrix
  print(confusionMatTemp)

  # extract values from confusion matrix
  trueNegative <- confusionMatTemp[1, 1]
  falseNegative <- confusionMatTemp[1, 2]
  falsePositive <- confusionMatTemp[2, 1]
  truePositive <- confusionMatTemp[2, 2]

  # calculate
  testError <- (falseNegative + falsePositive)/defaultTestNumRows
  testSensitivity <- truePositive/(truePositive + falseNegative)
  testSpecificity <- trueNegative/(trueNegative + falsePositive)
  testPrecision <- truePositive/(truePositive + falsePositive)

  # report
  print(paste0("Test Error: ", testError))
  print(paste0("Test Sensitivity: ", testSensitivity))
  print(paste0("Test Specificity: ", testSpecificity))
  print(paste0("Test Precision: ", testPrecision, "\n"))
}
```

```
## [1] "Cutoff: 0.3"
##          Actual
## Predicted   0   1
##       No  954  14
##       Yes  13  19
## [1] "Test Error: 0.027"
## [1] "Test Sensitivity: 0.575757575757576"
## [1] "Test Specificity: 0.986556359875905"
## [1] "Test Precision: 0.59375\n"
## [1] "Cutoff: 0.5"
##          Actual
## Predicted   0   1
##       No  963  22
##       Yes   4  11
## [1] "Test Error: 0.026"
## [1] "Test Sensitivity: 0.333333333333333"
## [1] "Test Specificity: 0.995863495346432"
## [1] "Test Precision: 0.733333333333333\n"
## [1] "Cutoff: 0.7"
##          Actual
## Predicted   0   1
##       No  964  29
##       Yes   3   4
## [1] "Test Error: 0.032"
## [1] "Test Sensitivity: 0.121212121212121"
## [1] "Test Specificity: 0.996897621509824"
## [1] "Test Precision: 0.571428571428571\n"
```

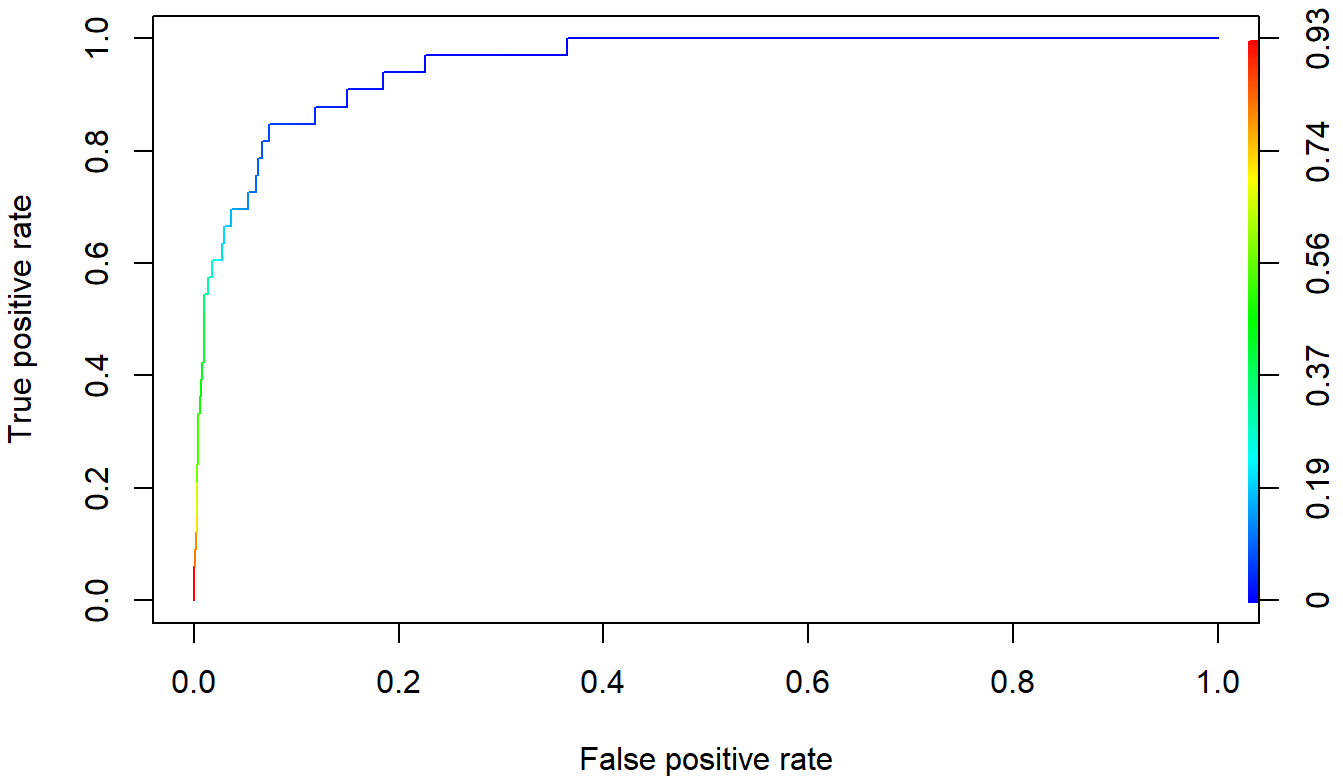
The best option depends on our goal. For our case, since we care the most about classifying people who will default correctly, the metric we should look at is sensitivity. The cutoff value of 0.3 gives the highest sensitivity, so we prefer that one.

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.

Solution:

```
# load library
library(ROCR)

# calculating the ROC curve
roc.Default <- prediction(pred.Default, Default_test$default)
perf.Default <- performance(roc.Default, "tpr", "fpr")
plot(perf.Default, colorize = TRUE)
```




```
# calculate the AUC
auc.Default <- performance(roc.Default, measure = "auc")
auc.Default <- auc.Default@y.values[[1]]
print(paste0("AUC: ", auc.Default))
```

```
## [1] "AUC: 0.952304847858107"
```

There are two types of classification mistakes, misclassifying people as `Default=Yes` if they will not default, and misclassifying people as "Default=No" if they will default. We are less concerned with the first type of classification mistake, so we are fine with the false positive rate being on the higher side. However, we do not want to classify people as `Default=No` if they will indeed default, so we seek to maximize the true positive rate. With this reasoning, a cutoff value of around 0.2 could be appropriate.

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.

Solution:

```
# load library
library(glmnet)
```

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

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

```
# parameters for cv.glmnet function
q1_measure <- "auc"
q1_numFolds <- 20
q1_alpha <- 0 # "ridge" -> alpha = 0
q1_family <- "binomial"

# divide data
Default_train_X <- as.matrix(Default_train[,c("balance", "income")])
Default_train_Y <- as.matrix(Default_train["default"])
colnames(Default_train_Y) <- "default"

# cv.glmnet
train_default_ridge_cv <- cv.glmnet(x = Default_train_X,
                                    y = Default_train_Y,
                                    type.measure = q1_measure,
                                    nfolds = q1_numFolds,
                                    alpha = q1_alpha,
                                    family = q1_family)
print(train_default_ridge_cv)
```

```
##
## Call:  cv.glmnet(x = Default_train_X, y = Default_train_Y, type.measure = q1_measure,      nfo
lds = q1_numFolds, alpha = q1_alpha, family = q1_family)
##
## Measure: AUC
##
##     Lambda Index Measure       SE Nonzero
## min   0.01   100  0.9510 0.005999       2
## 1se  57.15     2  0.9482 0.006273       2
```

We can set λ to be either $\lambda_{\text{min}} = 0.01$ or $\lambda_{\text{1se}} = 57.15$.

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`).
- 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?

Solution:

```r
# Function Name: euclidean_distance
# Function Usage: to calculate the Euclidean distance between two vectors
# Input 1: vec1, a vector of length p
# Input 2: vec2, a vector of length p
# Output: euclDist, a numeric
euclidean_distance <- function(vec1, vec2){
  # calculate Euclidean distance
  euclDist <- sqrt(sum((vec1 - vec2)^2))

  # return Euclidean distance
  return(euclDist)
}

# Function Name: manhattan_distance
# Function Usage: to calculate the Manhattan distance between two vectors
# Input 1: vec1, a vector of length p
# Input 2: vec2, a vector of length p
# Output: manhDist, a numeric
manhattan_distance <- function(vec1, vec2){
  # calculate Manhattan distance
  manhDist <- sum(abs(vec1 - vec2))

  # return Manhattan distance
  return(manhDist)
}

# Function Name: euclidean_distance_all
# Function Usage: to calculate the Euclidean distance between a vectors and
# all the row vectors in a matrix
# Input 1: vec1, a vector of length p
# Input 2: mat1_X, a matrix of dimension n x p
# Output: output_euclDistVec, a vector of length n
euclidean_distance_all <- function(vec1, mat1_X){
  # number of rows
  numRows <- dim(mat1_X)[1]
  output_euclDistVec <- rep(NA, times = numRows)

  # calculate Euclidean distance between vec1 and all row vectors of mat1_X
  for(i in 1:numRows){
    output_euclDistVec[i] <- euclidean_distance(vec1 = vec1, vec2 = mat1_X[i,])
  }

  # return Euclidean distances
  return(output_euclDistVec)
}

# Function Name: manhattan_distance_all
# Function Usage: to calculate the Manhattan distance between a vectors and
# all the row vectors in a matrix
# Input 1: vec1, a vector of length p
# Input 2: mat1_X, a matrix of dimension n x p
# Output: output_manhattanDistVec, a vector of length n
manhattan_distance_all <- function(vec1, mat1_X){
  # number of rows
  numRows <- dim(mat1_X)[1]
  output_manhattanDistVec <- rep(NA, times = numRows)
```

```
  # calculate Manhattan distance between vec1 and all row vectors of mat1_X
  for(i in 1:numRows){
    output_manhattanDistVec[i] <- manhattan_distance(vec1 = vec1, vec2 = mat1_X[i,])
  }

  # return Manhattan distances
  return(output_manhattanDistVec)
}

# Function Name: my_KNN
# Function Usage: to calculate the K nearest neighbors for a vector,
# the potential neighbors are all the row vectors in a matrix
# Input 1: vec1, a vector of length p
# Input 2: mat1_X, a matrix of dimension n x p
# Input 3: mat1_Y, a vector of length n
# Input 4: K, a positive integer
# Input 5: euclDistUsed, a Boolean
# Output: output_knnMajorityVote, a list of length 2
my_KNN <- function(vec1, mat1_X, mat1_Y, K, euclDistUsed){
  # number of row vectors
  numRows <- dim(mat1_X)[1]

  # check if we are calculating Euclidean distances or Manhattan distances
  if(euclDistUsed){
    distancesToVecs <- euclidean_distance_all(vec1 = vec1, mat1_X = mat1_X)
  } else{
    distancesToVecs <- manhattan_distance_all(vec1 = vec1, mat1_X = mat1_X)
  }

  # sort the distances
  distancesToVecsSorted <- sort(distancesToVecs, decreasing = FALSE)

  # extract K-smallest value
  K_th_smallest_distance <- distancesToVecsSorted[K]

  # extract indices from unsorted vector where value is <= K_th_smallest_distance
  K_closest_neighbor_indices <- (1:numRows)[which(distancesToVecs <= K_th_smallest_distance)]

  # extract labels corresponding to those indices
  K_closest_neighbors_labels <- mat1_Y[K_closest_neighbor_indices]

  # majority vote
   K_closest_neighbors_labels_majority_vote <- as.numeric(names(sort(table(K_closest_neighbors_lab
els),
                                                                    decreasing = TRUE)[1]))

  # store results in list
  output_knnMajorityVote <- list("K Nearest Neigbor Labels" = K_closest_neighbors_labels,
                                 "Majority Vote" = K_closest_neighbors_labels_majority_vote)

  # return list
  return(output_knnMajorityVote)
}

# divide dataset into training dataset
```

```
q2_1_training_data_size <- 100
q2_1_train <- mnist[1:q2_1_training_data_size,]
q2_1_train_X <- as.matrix(q2_1_train[,which(colnames(q2_1_train) != "Digit")])
q2_1_train_Y <- as.matrix(q2_1_train$Digit)
colnames(q2_1_train_Y) <- "Digit"

# parameters for KNN
q2_1_K <- 10

# extract vector for 123rd observation
obs_123_X <- mnist[123, which(colnames(mnist) != "Digit")]
obs_123_Y <- mnist[123, "Digit"]

# run KNN for 123rd observation with Euclidean distance
obs_123_eucl <- my_KNN(vec1 = obs_123_X,
                       mat1_X = q2_1_train_X,
                       mat1_Y = q2_1_train_Y,
                       K = q2_1_K,
                       euclDistUsed = TRUE)
print(obs_123_eucl)
```

```
## $`K Nearest Neigbor Labels`
##  [1] 7 9 7 9 9 7 7 9 7 7
##
## $`Majority Vote`
## [1] 7
```

```
# run KNN for 123rd observation with Manhattan distance
obs_123_manhattan <- my_KNN(vec1 = obs_123_X,
                            mat1_X = q2_1_train_X,
                            mat1_Y = q2_1_train_Y,
                            K = q2_1_K,
                            euclDistUsed = FALSE)
print(obs_123_manhattan)
```

```
## $`K Nearest Neigbor Labels`
##  [1] 7 9 7 9 9 7 7 9 7 7
##
## $`Majority Vote`
## [1] 7
```

```
# true label
print(paste0("True label for 123rd observation: ", obs_123_Y))
```

```
## [1] "True label for 123rd observation: 7"
```

The predicted label when we do KNN with K = 10 and a majority vote is 7 when we use Euclidean or Manhattan distance. Since the true label is 7, the predictions are 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.

```
# set seed
set.seed(7)

# load library
library(caret)
```

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

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

```
# parameters for cross-validation
q2_2_resamplingMethod <- "cv"
q2_2_numFolds <- 20
q2_2_classificationMethod <- "knn"
q2_2_performanceMetric <- "Accuracy"

# sequence of K values
q2_2_K_seq <- data.frame(k = c(1, 5, 10, 20, 50, 100))

# get training dataset
q2_2_training_data_size <- 1250
q2_2_train <- mnist[1:q2_2_training_data_size,]
q2_2_train_X <- as.matrix(q2_2_train[,which(colnames(q2_2_train) != "Digit")])
q2_2_train_Y <- as.factor(q2_2_train$Digit)

# setup the cross-validation options
knn_cv_train_control <- trainControl(method = q2_2_resamplingMethod,
                                     number = q2_2_numFolds)

# train the model
knn_cv_train <- train(x = q2_2_train_X,
                      y = q2_2_train_Y,
                      method = q2_2_classificationMethod,
                      metric = q2_2_performanceMetric,
                      trControl = knn_cv_train_control,
                      tuneGrid = q2_2_K_seq)
print(knn_cv_train)
```

```
## 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.
```

The best KNN model according to cross-validation classification accuracy is $K = 1$. Let's fit a KNN model with $K = 1$ onto the entire training dataset.

```
  # the function knn is in the library class
  # load library class
  library(class)
```

```
##
## Attaching package: 'class'
```

```
## The following objects are masked from 'package:FNN':
##
##     knn, knn.cv
```

```r
# best K
q2_2_chosen_K <- 1

# setup test dataset
q2_2_test <- mnist[((q2_2_training_data_size + 1):dim(mnist)[1]),]
q2_2_test_X <- as.matrix(q2_2_test[,which(colnames(q2_2_test) != "Digit")])
q2_2_test_Y <- as.factor(q2_2_test$Digit)

# fit KNN model with K = 1 onto entire training dataset
# predict class labels for test dataset
knn_train <- knn(train = q2_2_train_X,
                 test = q2_2_test_X,
                 cl = q2_2_train_Y,
                 k = q2_2_chosen_K)

# prediction confusion matrix on the test dataset
q2_2_confusionMat <- table(Predicted = knn_train, Actual = q2_2_test_Y)
print(q2_2_confusionMat)
```

```
##          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
```

```r
# calculate test error
q2_2_numCorrectPredictions <- sum(diag(q2_2_confusionMat))
q2_2_numPredictions <- sum(q2_2_confusionMat)
q2_2_testError <- 1 - (q2_2_numCorrectPredictions/q2_2_numPredictions)
print(paste0("The test error is: ", q2_2_testError))
```

```
## [1] "The test error is: 0.096"
```

```
  # calculate the sensitivity of each class
  numClasses <- dim(q2_2_confusionMat)[1]
  for(i in 1:numClasses){
    # class label
    class_label_temp <- dimnames(q2_2_confusionMat)$Actual[i]

    # calculate how many of the true occurrences of each class label were correctly classified
    num_predicted_class_label <- diag(q2_2_confusionMat)[i]

    # calculate true number of occurrences of each class label
    num_true_class_label <- sum(q2_2_confusionMat[,i])

    # calculate sensitivity
    sensitivity_temp <- num_predicted_class_label/num_true_class_label

    # print
     print(paste0("The sensitivity for class label ", class_label_temp, " is: ", sensitivity_tem
p))
  }
```

```
## [1] "The sensitivity for class label 0 is: 0.957264957264957"
## [1] "The sensitivity for class label 1 is: 0.984375"
## [1] "The sensitivity for class label 2 is: 0.848"
## [1] "The sensitivity for class label 3 is: 0.852459016393443"
## [1] "The sensitivity for class label 4 is: 0.819444444444444"
## [1] "The sensitivity for class label 5 is: 0.92436974789916"
## [1] "The sensitivity for class label 6 is: 0.956834532374101"
## [1] "The sensitivity for class label 7 is: 0.952380952380952"
## [1] "The sensitivity for class label 8 is: 0.872727272727273"
## [1] "The sensitivity for class label 9 is: 0.875"
```

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
set.seed(7)

# parameters for cv.glmnet function
q2_3_numFolds <- 20
q2_3_alpha <- 1 # "lasso" -> alpha = 1
q2_3_family <- "multinomial"

# cv.glmnet
q2_3_cv_multinomial <- cv.glmnet(x = q2_2_train_X,
                                 y = q2_2_train_Y,
                                 nfolds = q2_3_numFolds,
                                 alpha = q2_3_alpha,
                                 family = q2_3_family)
print(q2_3_cv_multinomial)
```

```
##
## Call:  cv.glmnet(x = q2_2_train_X, y = q2_2_train_Y, nfolds = q2_3_numFolds,      alpha = q2_3
_alpha, family = q2_3_family)
##
## Measure: Multinomial Deviance
##
##       Lambda Index Measure      SE Nonzero
## min 0.003231    43   1.079 0.06467      62
## 1se 0.006196    36   1.133 0.05605      46
```

The identified value for λ_{1se} is 0.006196. Let's now fit a model onto the entire training dataset and then predict the class labels for the testing dataset,

```
# we fit the model with the entire lambda sequence that cv.glmnet generated
# glmnet does not like it if only a single lambda is specified
# lambda.1se is in that sequence so this is not an issue
q2_3_multinomial <- glmnet(x = q2_2_train_X,
                           y = q2_2_train_Y,
                           family = q2_3_family,
                           alpha = q2_3_alpha,
                           lambda = q2_3_cv_multinomial$lambda)

# predict class labels, ensure you set s = lambda.1se to use that for predictions
q2_3_test_Y_hat <- as.factor(predict(q2_3_multinomial,
                                     newx = q2_2_test_X,
                                     s = q2_3_cv_multinomial$lambda.1se,
                                     type = "class"))

# prediction confusion matrix on the test dataset
q2_3_confusionMat <- table(Predicted = q2_3_test_Y_hat, Actual = q2_2_test_Y)
print(q2_3_confusionMat)
```

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

```
  # calculate test error
  q2_3_numCorrectPredictions <- sum(diag(q2_3_confusionMat))
  q2_3_numPredictions <- sum(q2_3_confusionMat)
  q2_3_testError <- 1 - (q2_3_numCorrectPredictions/q2_3_numPredictions)
  print(paste0("The test error is: ", q2_3_testError))
```

```
## [1] "The test error is: 0.1504"
```