# HW05_zilinw3

Zilin Wang (zilinw3)

2024-09-30

## 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 $\epsilon$ 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.

**Answer:**

```
beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)

# Calculate beta^T x0
beta_x0 <- sum(beta * x0)

# Calculate the true mean of Y
true_mean_Y <- exp(beta_x0)
cat("The True Mean of Y:", true_mean_Y, "\n")
```

```
## The True Mean of Y: 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.

**Answer:**

```
set.seed(123)
n <- 100
k <- 21

# Generate training data
x_train <- matrix(runif(n * 3), ncol = 3)
epsilon <- rnorm(n)
y_train <- exp(x_train %*% beta) + epsilon

# Define the 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])
}

# Predicting for x0 using custom KNN
knn_result <- knn_prediction(x_train, y_train, x0, k)

# Validate with FNN package
if (!requireNamespace("FNN", quietly = TRUE)) install.packages("FNN")
library(FNN)
knn_fnn_result <- knn.reg(train = x_train, test = matrix(x0, nrow = 1), y = y_train, k = k)$pred

# Output results
cat("The KNN Prediction:", knn_result, "\n")
```

```
## The KNN Prediction: 1.001475
```

```
cat("The FNN KNN Prediction:", knn_fnn_result, "\n")
```

```
## The FNN KNN Prediction: 1.001475
```

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 $E[\hat{f}(x_0)] - f(x_0)$. Use the same sample size $n = 100$ and same $k = 21$, design your own simulation study to estimate this.

**Answer:**

```r
set.seed(123)

# Simulation to estimate the bias
n_simulations <- 1000
knn_predictions <- numeric(n_simulations)

# Run the simulation 1000 times
for (i in 1:n_simulations) {
  X <- matrix(runif(n * 3), ncol = 3)
  epsilon <- rnorm(n, mean = 0, sd = 1)
  Y <- exp(X %*% beta) + epsilon
  knn_predictions[i] <- knn_prediction(X, Y, x0, k)
}

# Calculate bias
bias <- mean(knn_predictions) - true_mean_Y
cat("The Bias:", bias, "\n")
```

```
## The Bias: -1.172217
```

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 $E[(\hat{f}(x_0) - E[\hat{f}(x_0)])^2]$. Calculate and report the value.

**Answer:**

```r
variance <- var(knn_predictions)
cat("The Variance:", variance, "\n")
```

```
## The Variance: 0.05032568
```

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?

**Answer:**

```r
prediction_error <- bias^2 + variance + 1  # Adding irreducible error, which is the variance of epsilon
cat("The Calculated Prediction Error:", prediction_error, "\n")
```

```
## The Calculated Prediction Error: 2.424419
```

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.

**Answer:**

```r
set.seed(123)

# Generate testing data and calculate the empirical prediction error
test_epsilon <- rnorm(1000)
test_y0 <- exp(beta_x0) + test_epsilon

# Empirical prediction error
empirical_error <- mean((knn_predictions - test_y0)^2)
cat("The Empirical Prediction Error:", empirical_error, "\n")
```

```
## The Empirical Prediction Error: 2.442789
```

```r
# Compare empirical error to theoretical prediction error
list(theoretical = prediction_error, empirical = empirical_error)
```

```
## $theoretical
## [1] 2.424419
##
## $empirical
## [1] 2.442789
```

The empirical prediction error is slightly larger, but very close to the theoretical prediction error.

## 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)
```

```
## Warning:   'ISLR2' R 4.3.3
```

```
  # 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?

**Answer:**

```
library(ISLR2)
data(Default)
set.seed(7)

defaultNumRows <- dim(Default)[1]
defaultTestNumRows <- 1000
test_idx <- sample(x = 1:defaultNumRows, size = defaultTestNumRows)
Default_train <- Default[-test_idx,]
Default_test <- Default[test_idx,]

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

```r
# Function to calculate log-likelihood
loglikelihood <- function(beta, X, Y) {
  p <- exp(X %*% beta) / (1 + exp(X %*% beta))
  output_loglik <- sum(Y * log(p) + (1 - Y) * log(1 - p))
  return(output_loglik)
}

# Calculate log-likelihood for the fitted model
X <- model.matrix(~ balance + income, Default_train)
Y <- Default_train$default
beta <- coef(model)
max_log_lik <- loglikelihood(beta, X, as.numeric(Y) - 1)

# Calculate the deviance value
deviance_value <- model$deviance

cat("The maximum log likelihood:", max_log_lik, "\n")
```

```
## The maximum log likelihood: -712.2981
```

```r
cat("The deviance value:", deviance_value, "\n")
```

```
## The deviance value: 1424.596
```

Deviance = -2 * Maximum log Likelihood

Both deviance and log likelihood are measures of how well a model fits the data, but while log likelihood gives a direct measure, deviance provides a relative measure that's useful for comparing models.

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?

**Answer:**

```r
# Predict probabilities
predicted_probs <- predict(model, Default_test, type = "response")

# Function to calculate metrics based on cutoff
get_metrics <- function(cutoff) {
  predicted_classes <- ifelse(predicted_probs > cutoff, "Yes", "No")
  table <- table(Predicted = predicted_classes, Actual = Default_test$default)
  test_error <- mean(predicted_classes != Default_test$default)
  sensitivity <- table[2, 2] / sum(Default_test$default == "Yes")
  specificity <- table[1, 1] / sum(Default_test$default == "No")
  precision <- table[2, 2] / sum(predicted_classes == "Yes")
```

```
  list(confusion_matrix = table, test_error = test_error,
       sensitivity = sensitivity, specificity = specificity, precision = precision)
}

# Evaluate at cutoffs 0.3, 0.5, 0.7
results_0.3 <- get_metrics(0.3)
results_0.5 <- get_metrics(0.5)
results_0.7 <- get_metrics(0.7)
results_0.3
```

```
## $confusion_matrix
##          Actual
## Predicted  No Yes
##       No  954  14
##       Yes  13  19
##
## $test_error
## [1] 0.027
##
## $sensitivity
## [1] 0.5757576
##
## $specificity
## [1] 0.9865564
##
## $precision
## [1] 0.59375
```

```
results_0.5
```

```
## $confusion_matrix
##          Actual
## Predicted  No Yes
##       No  963  22
##       Yes   4  11
##
## $test_error
## [1] 0.026
##
## $sensitivity
## [1] 0.3333333
##
## $specificity
## [1] 0.9958635
##
## $precision
## [1] 0.7333333
```

```
results_0.7
```

```
## $confusion_matrix
##          Actual
```

```
## Predicted  No Yes
##       No  964  29
##       Yes   3   4
##
## $test_error
## [1] 0.032
##
## $sensitivity
## [1] 0.1212121
##
## $specificity
## [1] 0.9968976
##
## $precision
## [1] 0.5714286
```

I prefer the cutoff of 0.5. Because it balances both sensitivity and precision. It has a reasonable sensitivity and significantly better precision than at 0.3, meaning it maintains a balance between identifying defaults and minimizing false positives

Given the goal to capture as many people who will default as possible, we should use a cutoff of 0.3. This cutoff offers the highest sensitivity, detecting 57.57% of all true defaulters. This means it identifies the most actual defaults, ensuring that the largest number of defaulters are identified.

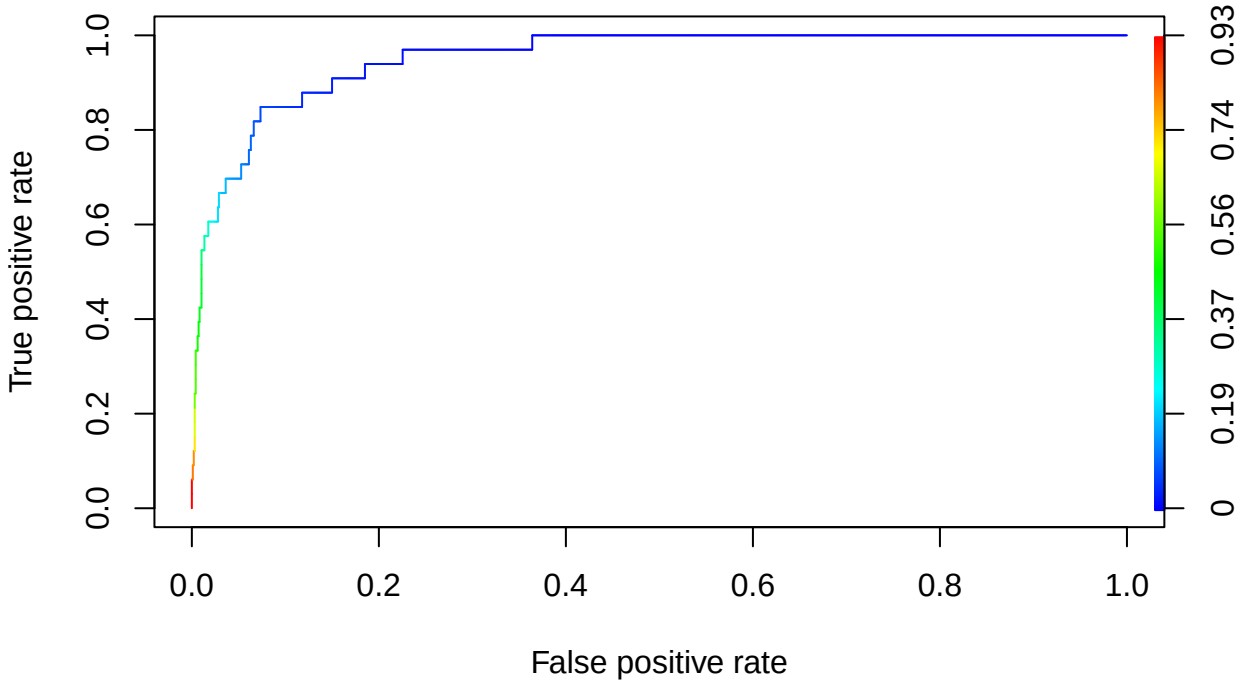
c. [5 pts] Load the library `ROCR`. Using the functions in that library, plot the ROC curve and calculate the AUC. Use the ROC curve to determine a cutoff value and comment on your reasoning.

**Answer:**

```r
# install.packages("ROCR")
library(ROCR)

# Predictions object
pred <- prediction(predicted_probs, Default_test$default)

# Plot ROC curve
roc_performance <- performance(pred, measure = "tpr", x.measure = "fpr")
plot(roc_performance, colorize = TRUE)
```




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

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

I choose the cutoff between 0.3 and 0.4. The point along the curve that rises sharply before leveling off typically offers a good balance between catching true positives and avoiding false positives. From the ROC curve, the transition from green to blue appears to be a significant change in slope, when the True Positive Rate is high without the False Positive Rate becoming too large. Therefore, a cutoff around 0.3 to 0.4 might be suitable.

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, $\lambda$, in the logistic regression with ridge penalty. In order to choose the best penalty coefficient use AUC as the Cross-Validation metric.

**Answer:**

```
library(glmnet)
```

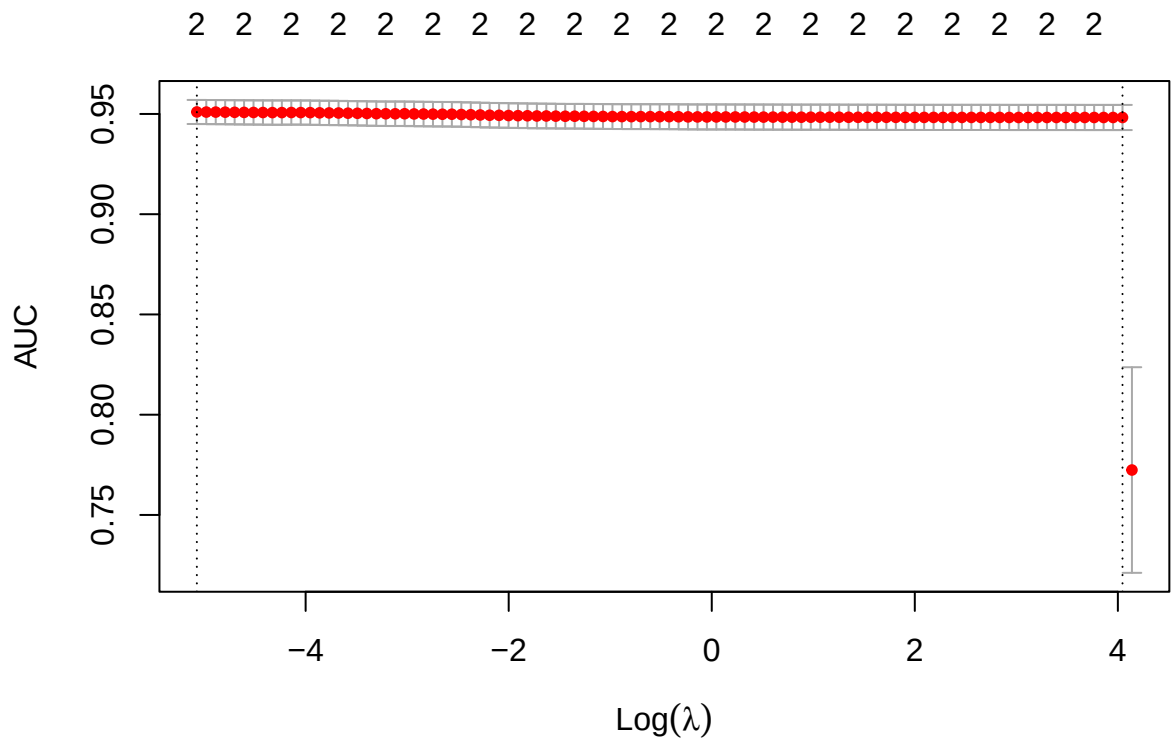
```
##      Matrix
```

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

```r
# Prepare data
x_matrix <- model.matrix(~ balance + income, Default_train)[,-1]
y_vector <- ifelse(Default_train$default == "Yes", 1, 0)

# Apply cross-validation
cv_fit <- cv.glmnet(x_matrix, y_vector, family = "binomial", alpha = 0, nfolds = 20, type.measure = "au

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




```r
opt_lambda <- cv_fit$lambda.min

# Output the optimal lambda
cat("The optimal penalty coefficient:", opt_lambda, "\n")
```

```
## The optimal penalty coefficient: 0.006272533
```

## 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 $123^{\text{rd}}$ 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?

**Answer:**

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

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

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

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

my_KNN <- function(vec1, mat1_X, mat1_Y, K, euclDistUsed) {
  if (euclDistUsed == TRUE) {
    distances <- euclidean_distance_all(vec1, mat1_X)
  } else {
    distances <- manhattan_distance_all(vec1, mat1_X)
  }

  sorted_indices <- order(distances)
  nearest_indices <- sorted_indices[1:K]
  nearest_labels <- mat1_Y[nearest_indices]

  majority_vote <- names(which.max(table(nearest_labels)))

  output_knnMajorityVote <- list(neighbors_labels = nearest_labels, majority_vote = majority_vote)
  return(output_knnMajorityVote)
}

# Use the first 100 observations for training
```

```r
train_data <- mnist[1:100, -1]
train_labels <- mnist[1:100, 1]

# The 123rd observation to predict
test_observation <- mnist[123, -1]

# Applying KNN with K = 10
results_euclidean <- my_KNN(test_observation, train_data, train_labels, 10, euclDistUsed = TRUE)
results_manhattan <- my_KNN(test_observation, train_data, train_labels, 10, euclDistUsed = FALSE)

# The actual label of the 123rd observation
actual_label <- mnist[123, 1]

# Output results
cat("The predicted label using Euclidean distance:", results_euclidean$majority_vote, "\n")
```

```
## The predicted label using Euclidean distance: 7
```

```r
cat("The predicted label using Manhattan distance:", results_manhattan$majority_vote, "\n")
```

```
## The predicted label using Manhattan distance: 7
```

```r
cat("The actual label of the 123rd observation:", actual_label, "\n")
```

```
## The actual label of the 123rd observation: 7
```

Since the predicted label using Euclidean distance and the predicted label using Manhattan distance are both 7, which is the same as the actual label of the 123rd observation, both 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.

**Answer:**

```r
set.seed(7)

# Split the data into training and test sets
train_data <- mnist[1:1250, -1]
train_labels <- mnist[1:1250, 1]
test_data <- mnist[1251:2500, -1]
test_labels <- mnist[1251:2500, 1]

library(caret)
```

```
##      ggplot2
```

```
##      lattice
```

```
# Define training control
train_control <- trainControl(method = "cv", number = 20)

# Define a grid to search for the best K
k_values <- c(1, 5, 10, 20, 50, 100)
tune_grid <- expand.grid(k = k_values)

# Train the model
knn_model <- train(x = train_data, y = as.factor(train_labels), method = "knn", tuneGrid = tune_grid, t

# Get best K
best_k <- knn_model$bestTune$k
cat("Best K:", best_k, "\n")
```

```
## Best K: 1
```

```
# Fit a final KNN model
knn_model_final <- train(x = train_data, y = as.factor(train_labels), method = "knn", tuneGrid = expand

# Predict on the test data
knn_predictions <- predict(knn_model_final, newdata = test_data)

# Generate the confusion matrix
conf_matrix <- confusionMatrix(knn_predictions, as.factor(test_labels))
cat("The confusion matrix:\n")
```

```
## The confusion matrix:
```

```
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
```

```r
# Calculate the test error
test_error <- 1 - conf_matrix$overall['Accuracy']
cat("Test Error:", test_error, "\n")
```

```
## Test Error: 0.096
```

```r
# Calculate the sensitivity for each classes
sensitivity_each_class <- conf_matrix$byClass[, "Sensitivity"]
cat("Sensitivity for each class:\n")
```

```
## Sensitivity for each class:
```

```r
print(sensitivity_each_class)
```

```
##  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
```

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 $\lambda$ penalty parameter. Use the `cv.glmnet` function with 20 folds on the training dataset to find $\lambda_{1se}$. Once you have identified $\lambda_{1se}$, 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?

**Answer:**

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

# Convert training and testing data to matrices
train_data <- mnist[1:1250, -1]
train_labels <- mnist[1:1250, 1]
test_data <- mnist[1251:2500, -1]
test_labels <- mnist[1251:2500, 1]
x_train <- as.matrix(train_data)
y_train <- as.factor(train_labels)
x_test <- as.matrix(test_data)
y_test <- as.factor(test_labels)

# Perform cross-validation to find the optimal lambda
cv_fit <- cv.glmnet(x_train, y_train, family = "multinomial", type.measure = "class", alpha = 1, nfolds

# Best lambda using 1 standard error rule
lambda_1se <- cv_fit$lambda.1se
cat("Lambda at one standard error:", lambda_1se, "\n")
```

```
## Lambda at one standard error: 0.006800599
```

```
# Get the sequence of lambda
lambda_sequence <- cv_fit$lambda

# Fit the final model using the lambda.1se
final_model <- glmnet(x_train, y_train, family = "multinomial", lambda = lambda_sequence, alpha = 1)

# Predict class labels for the test data
predictions <- predict(final_model, newx = x_test, s = lambda_1se, type = "class")

# Convert predictions to factor to match test labels
predictions <- as.factor(predictions)

# Create the confusion matrix
final_conf_matrix <- confusionMatrix(predictions, y_test)
cat("The confusion matrix:\n")
```

```
## The confusion matrix:
```

```r
print(final_conf_matrix)
```

```
## 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
```

```r
# Calculate the test error
final_test_error <- 1 - sum(diag(final_conf_matrix$table)) / sum(final_conf_matrix$table)
cat("Test Error:", final_test_error, "\n")
```

```
## Test Error: 0.1488
```