Stat 432 Homework 5

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

Contents

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

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

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

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

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

```r
beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)
f_x0 <- exp(sum(beta * x0))
f_x0
```

```
## [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.

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

```
## Warning: package 'FNN' was built under R version 4.3.3
```

```r
set.seed(123)
n <- 100
X_train <- matrix(runif(n * 3, 0, 1), ncol = 3)
epsilon <- rnorm(n)
Y_train <- exp(X_train %*% beta) + epsilon

knn_custom <- function(x0, X_train, Y_train, k) {
  distances <- apply(X_train, 1, function(row) sqrt(sum((row - x0)^2)))
  nearest_indices <- order(distances)[1:k]
  mean(Y_train[nearest_indices])
}

k <- 21
knn_pred <- knn_custom(x0, X_train, Y_train, k)
knn_pred
```

```
## [1] 1.001475
```

```r
#validate with knn.reg
knn_pred_validation <- knn.reg(X_train, test = matrix(x0, nrow = 1), y = Y_train, k = k)$pred
knn_pred_validation
```

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

```r
set.seed(123)
num_simulations <- 1000
knn_preds <- numeric(num_simulations)

for (i in 1:num_simulations) {
  X_train <- matrix(runif(n * 3, 0, 1), ncol = 3)
  epsilon <- rnorm(n)
  Y_train <- exp(X_train %*% beta) + epsilon
  knn_preds[i] <- knn_custom(x0, X_train, Y_train, k)
}

estimated_bias <- mean(knn_preds) - f_x0
cat("The Bias:", estimated_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 $E[(\widehat{f}(x_0) - E[\widehat{f}(x_0)])^2]$. Calculate and report the value.

```r
estimated_variance <- var(knn_preds)
cat("The Variance:", estimated_variance, "\n")
```

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

The estimated variance of this model is 0.05027535.

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?

From previous question, the estimated bias is -1.172217 and the estimated variance is 0.05027535. The irreducible error is the variance of the noise term, which is 1(since $\epsilon \sim \text{N}(0, 1)$). The prediction error is the sum of the squared bias, variance, and irreducible error.

```r
prediction_error <- 1 + (estimated_bias^2) + estimated_variance
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.

```r
set.seed(123)
prediction_errors <- numeric(num_simulations)

for (i in 1:num_simulations) {
  X_train <- matrix(runif(n * 3, 0, 1), ncol = 3)
  epsilon <- rnorm(n)
  Y_train <- exp(X_train %*% beta) + epsilon
  knn_pred <- knn_custom(x0, X_train, Y_train, k)

  # Generate testing data
  epsilon_test <- rnorm(1)
  Y0 <- f_x0 + epsilon_test

  # Prediction error
  prediction_errors[i] <- (Y0 - knn_pred)^2
}

mean_prediction_error <- mean(prediction_errors)
mean_prediction_error
```

```
## [1] 2.546811
```

The prediction error is 2.31651, which is very close to the theoretical calculation 2.546811 in the previous question, which implies that the theoretical model is a good approximation of the actual behavior of the KNN model in practice.

However, the values are not exactly the same, likely due to:

- Simulation randomness: Since the simulation involves randomly generated training data and random noise, there is some inherent variation in the resulting prediction error.
- Approximation in theory: The theoretical values are estimates based on the bias and variance, which themselves were estimated from simulations.

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?

```
# Fit logistic regression model
logistic_model <- glm(default ~ balance + income, data = Default_train, family = "binomial")
loglikelihood <- function(beta, X, Y) {
  # Calculate the linear predictor
  eta <- X %*% beta

  # Calculate the probability of default (using the logistic function)
  p <- 1 / (1 + exp(-eta))

  # Calculate the log-likelihood
  output_loglik <- sum(Y * log(p) + (1 - Y) * log(1 - p))
  return(output_loglik)
}
```

```r
# Extract estimated coefficients
estimated_coefficients <- coef(logistic_model)

# Prepare the input data matrix (adding intercept)
X_train <- as.matrix(cbind(1, Default_train[, c("balance", "income")]))
Y_train <- as.numeric(Default_train$default == "Yes")

# Calculate the log-likelihood using the estimated coefficients
max_log_likelihood <- loglikelihood(estimated_coefficients, X_train, Y_train)
print(max_log_likelihood)
```

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

```r
deviance_value <- logistic_model$deviance
print(deviance_value)
```

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

The relationship is: deviance = -2 * maximum log likelihood.

In this case, the deviance value is approximately equal to -2 × (-712.298) = 1424.596, which confirms the expected relationship.

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
# Predict the probability of default on the test dataset
probabilities <- predict(logistic_model, newdata = Default_test, type = "response")

calculate_metrics <- function(probabilities, actual, cutoff) {
  # Convert probabilities to predicted classes based on cutoff
  predicted <- ifelse(probabilities >= cutoff, "Yes", "No")

  # Convert actual classes to binary representation
  actual <- ifelse(actual == "Yes", "Yes", "No")

  # Calculate confusion matrix elements
  TP <- sum(predicted == "Yes" & actual == "Yes")
  TN <- sum(predicted == "No" & actual == "No")
  FP <- sum(predicted == "Yes" & actual == "No")
  FN <- sum(predicted == "No" & actual == "Yes")

  # Create and print confusion matrix in matrix format
  confusion_matrix <- matrix(c(TP, FP, FN, TN), nrow = 2, byrow = TRUE,
                             dimnames = list("Predicted" = c("Yes", "No"),
                                             "Actual" = c("Yes", "No")))
  cat("Cutoff:", cutoff, "\n")
  cat("Confusion Matrix:\n")
  print(confusion_matrix)

  # Calculate metrics
  test_error <- (FP + FN) / length(actual)
  sensitivity <- TP / (TP + FN)
  specificity <- TN / (TN + FP)
  precision <- TP / (TP + FP)

  # Print metrics
  cat("Test Error:", test_error, "\n")
  cat("Sensitivity:", sensitivity, "\n")
  cat("Specificity:", specificity, "\n")
  cat("Precision:", precision, "\n\n")

  # Return a list of metrics
  return(list(test_error = test_error, sensitivity = sensitivity, specificity = specificity, precision =
}

# Apply the function for cutoff values 0.3, 0.5, and 0.7
metrics_0.3 <- calculate_metrics(probabilities, Default_test$default, 0.3)
```

```
## Cutoff: 0.3
## Confusion Matrix:
##          Actual
## Predicted Yes  No
##       Yes  19  13
##       No   14 954
## Test Error: 0.027
## Sensitivity: 0.5757576
## Specificity: 0.9865564
## Precision: 0.59375
```

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

```
## Cutoff: 0.5
## Confusion Matrix:
##          Actual
## Predicted Yes  No
##       Yes  11   4
##       No   22 963
## Test Error: 0.026
## Sensitivity: 0.3333333
## Specificity: 0.9958635
## Precision: 0.7333333
```

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

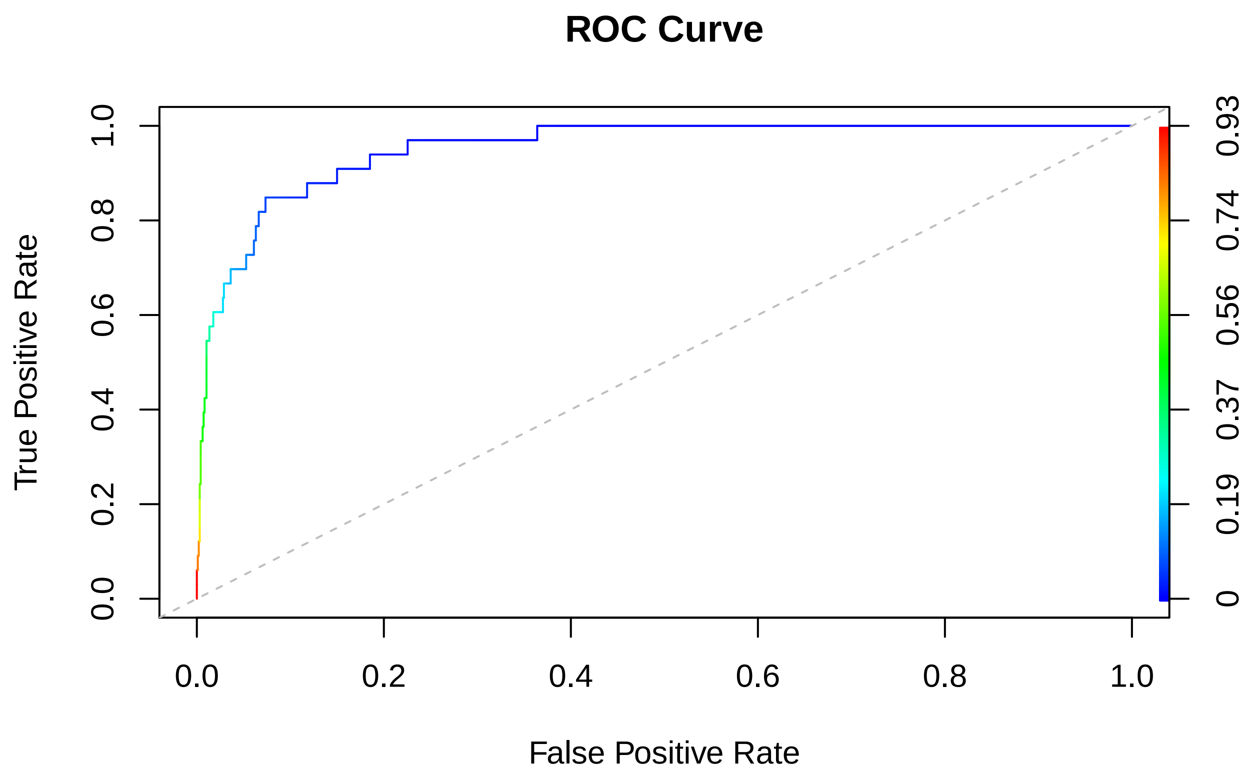
```
## Cutoff: 0.7
## Confusion Matrix:
##          Actual
## Predicted Yes  No
##       Yes   4   3
##       No   29 964
## Test Error: 0.032
## Sensitivity: 0.1212121
## Specificity: 0.9968976
## Precision: 0.5714286
```

If our goal is general accuracy and balanced performance (with a trade-off between false positives and false negatives), the **cutoff = 0.5** is preferred, as it has the lowest test error and higher precision compared to the other cutoffs, meaning it maintains a balance between identifying defaults and minimizing false positives.

If the goal is to capture as many people who will default as possible, then sensitivity is the most important metric, even at the cost of increasing false positives (misclassifying people as Default = Yes when they won't default). For this scenario, **cutoff = 0.3** is the best option because it has the highest sensitivity (0.576), meaning it captures more of the actual default cases compared to the other cutoff values.

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)
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 Rate")
abline(a = 0, b = 1, col = "gray", lty = 2) # Diagonal reference line
```



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

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

A cutoff value between 0.3 and 0.4 is a good choice based on the ROC curve. This value provides a good balance between sensitivity and specificity, which should correspond 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) capturing a significant portion of the true positives while minimizing false positives. The ROC curve shows a steep initial increase in true positive rate (sensitivity) with a relatively low false positive rate at the beginning. 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 is suitable, since it offers a good trade-off between correctly identifying true positives while minimizing false positives.

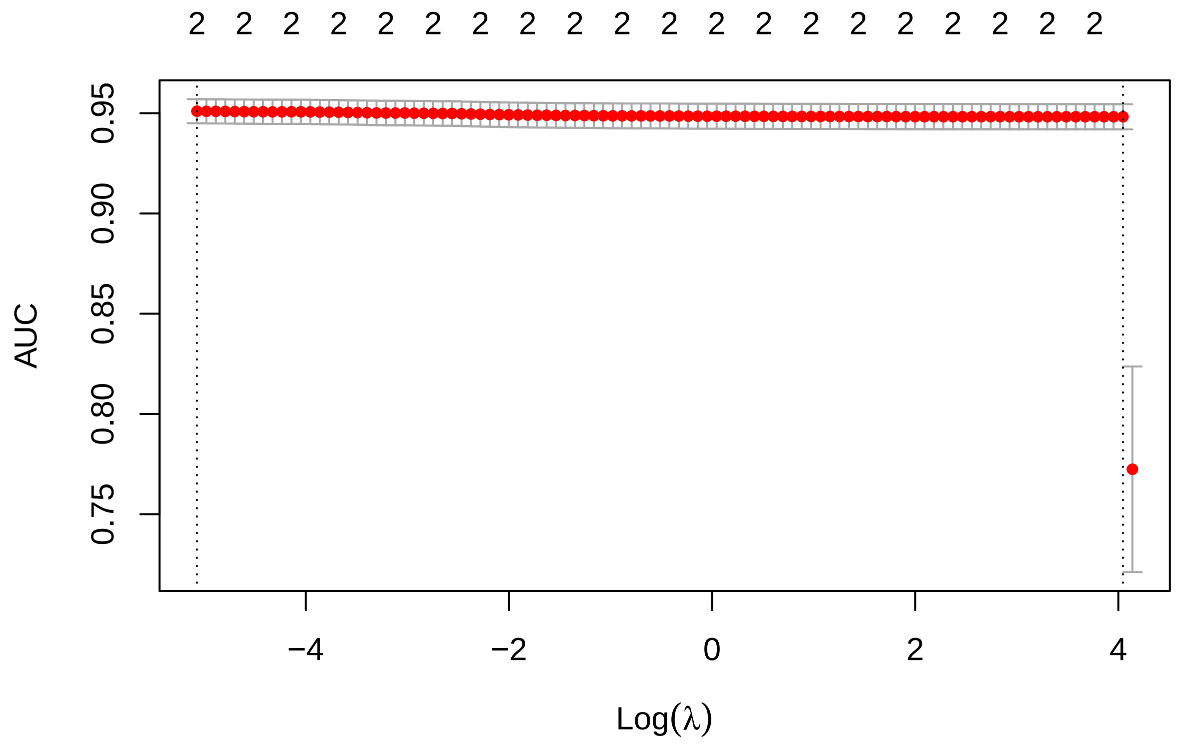
d. [5 pts] Load the library `glmnet`. Using the `cv.glmnet()` function, do 20-fold cross-validation on the training dataset to determine the optimal penalty coefficient, λ, in the logistic regression with ridge penalty. In order to choose the best penalty coefficient use AUC as the Cross-Validation metric.

```r
# Load the glmnet library
library(glmnet)
```

```
## Loading required package: Matrix
## Loaded glmnet 4.1-8
```

```
# Prepare input data for glmnet
train_X_glmnet <- as.matrix(Default_train[, c("balance", "income")])
train_Y_glmnet <- Default_train$default

# Perform cross-validation to find optimal lambda
cv_fit_ridge <- cv.glmnet(train_X_glmnet, train_Y_glmnet, family = "binomial",
                          alpha = 0, nfolds = 20, type.measure = "auc")
plot(cv_fit_ridge)
```



```
# Get the optimal lambda value
optimal_lambda <- cv_fit_ridge$lambda.min
print(paste("Optimal Lambda:", optimal_lambda))
```

```
## [1] "Optimal Lambda: 0.0062725329548981"
```

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

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

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

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

The optimal penalty coefficient (λ) for the logistic regression with ridge penalty, determined through 20-fold cross-validation using AUC as the metric, is approximately 0.006273. The best AUC achieved during the cross-validation process is 0.948, which is close to 1, indicating that the model performs well in distinguishing between the default and non-default classes. The selected λ balances regularization and predictive accuracy, optimizing the model's performance based on the AUC.

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

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

```r
  # 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 123rd observation using the first 100 observations as your input training data matrix. Use $K = 10$. What is the predicted label when you use Euclidean distance? What is the predicted label when you use Manhattan distance? Are these predictions correct?

```r
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(row) euclidean_distance(vec1, row))
  return(output_euclDistVec)
}
manhattan_distance_all <- function(vec1, mat1_X) {
  output_manhattanDistVec <- apply(mat1_X, 1, function(row) manhattan_distance(vec1, row))
  return(output_manhattanDistVec)
}
my_KNN <- function(vec1, mat1_X, mat1_Y, K, euclDistUsed = TRUE) {
  # Calculating distances
  dist_vec <- if (euclDistUsed) {
    euclidean_distance_all(vec1, mat1_X)
  } else {
    manhattan_distance_all(vec1, mat1_X)
  }

  # Find the indices of the K nearest neighbors
  knn_indices <- order(dist_vec)[1:K]
  # Find the corresponding class labels
  knn_labels <- mat1_Y[knn_indices]

  # Find the majority vote among the labels
  majority_vote <- as.numeric(names(sort(table(knn_labels), decreasing = TRUE)[1]))

  # Return the labels and the majority vote
  output_knnMajorityVote <- list(knn_labels, majority_vote)
  return(output_knnMajorityVote)
}

train_data <- mnist[1:100, -1]  # Pixels only
train_labels <- mnist[1:100, 1] # Digits
test_data <- mnist[123, -1]     # Pixels only

knn_result_eucl <- my_KNN(test_data, train_data, train_labels, K = 10, euclDistUsed = TRUE)
eucl_predicted_label <- knn_result_eucl[[2]]
print(eucl_predicted_label)
```

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

```
knn_result_manh <- my_KNN(test_data, train_data, train_labels, K = 10, euclDistUsed = FALSE)
manh_predicted_label <- knn_result_manh[[2]]
print(manh_predicted_label)
```

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

```
actual_label <- mnist[123, 1]
print(actual_label)
```

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

The predicted label when using Euclidean distance is 7. The predicted label when using Manhattan distance is also 7. These predictions correct since they are the same as the actual label, which is 7.

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.

```
library(caret)
```

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

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

```r
set.seed(7)
# Split the data into training and test sets
train_data <- mnist[1:1250, -1]  # Pixels only for training data
train_labels <- as.factor(mnist[1:1250, 1]) # Digits as factors for training labels
test_data <- mnist[1251:2500, -1]  # Pixels only for test data
test_labels <- as.factor(mnist[1251:2500, 1]) # Digits as factors for test labels

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

# Define a sequence of K values to try
k_values <- c(1, 5, 10, 20, 50, 100)

# Train KNN model with 20-fold cross-validation
knn_fit <- train(x = train_data, y = train_labels,
                 method = "knn",
                 trControl = train_control,
                 tuneGrid = expand.grid(k = k_values))
knn_fit
```

```
## 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.
```

```r
# Get the best K value according to cross-validation
best_k <- knn_fit$bestTune$k
cat("Best_k: ", best_k, "\n")
```

```
## Best_k:  1
```

```r
# Fit the final KNN model using the best K value
final_knn_model <- train(x = train_data, y = train_labels, method = "knn", tuneGrid = expand.grid(k = b

# Predict on the test dataset
```

```r
predictions <- predict(final_knn_model, newdata = test_data)

# Generate the confusion matrix
confusion_mat <- confusionMatrix(predictions, test_labels)

# Print the confusion matrix
print(confusion_mat)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 112   0   2   1   0   0   2   0   0   1
##          1   1 126   5   2   6   1   1   2   1   1
##          2   0   1 106   1   1   0   0   0   2   0
##          3   1   0   0 104   0   2   0   0   1   1
##          4   0   0   2   0 118   1   1   0   0   7
##          5   0   0   0   7   1 110   2   0   5   0
##          6   3   0   1   0   1   4 133   0   1   0
##          7   0   1   7   3   5   0   0 120   0   5
##          8   0   0   2   4   0   0   0   0  96   0
##          9   0   0   0   0  12   1   0   4   4 105
##
## Overall Statistics
##
##                Accuracy : 0.904
##                  95% CI : (0.8863, 0.9198)
##     No Information Rate : 0.1152
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8933
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9573   0.9844   0.8480   0.8525   0.8194   0.9244
## Specificity            0.9947   0.9822   0.9956   0.9956   0.9901   0.9867
## Pos Pred Value         0.9492   0.8630   0.9550   0.9541   0.9147   0.8800
## Neg Pred Value         0.9956   0.9982   0.9833   0.9842   0.9768   0.9920
## Prevalence             0.0936   0.1024   0.1000   0.0976   0.1152   0.0952
## Detection Rate         0.0896   0.1008   0.0848   0.0832   0.0944   0.0880
## Detection Prevalence   0.0944   0.1168   0.0888   0.0872   0.1032   0.1000
## Balanced Accuracy      0.9760   0.9833   0.9218   0.9240   0.9047   0.9556
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.9568   0.9524   0.8727   0.8750
## Specificity            0.9910   0.9813   0.9947   0.9814
## Pos Pred Value         0.9301   0.8511   0.9412   0.8333
## Neg Pred Value         0.9946   0.9946   0.9878   0.9867
## Prevalence             0.1112   0.1008   0.0880   0.0960
## Detection Rate         0.1064   0.0960   0.0768   0.0840
## Detection Prevalence   0.1144   0.1128   0.0816   0.1008
## Balanced Accuracy      0.9739   0.9668   0.9337   0.9282
```

```
# Calculate the test error
test_error <- 1 - confusion_mat$overall['Accuracy']
print(paste("Test Error:", test_error))
```

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

```
# Sensitivity for each class
sensitivity_per_class <- confusion_mat$byClass[, 'Sensitivity']
print(sensitivity_per_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
```

The code performs K-Nearest Neighbors (KNN) classification using 20-fold cross-validation to select the best value of K from the candidate values K = 1, 5, 10, 20, 50, 100. After training on the first 1250 samples, K = 1 is selected as the best value based on the highest accuracy. After fitting the KNN model with K = 1 on the training data and predicting the last 1250 samples (test data), the confusion matrix shows an accuracy of 90.4%, and the test error is 9.6%. The sensitivity for each class (0 to 9) varies, with most classes having high sensitivity, such as class 1 (98.44%) and class 0 (95.73%). However, some classes, like class 4 (81.94%) and class 8 (87.27%), have lower sensitivity. Overall, the model performs well with a balanced accuracy for each class.

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

```
set.seed(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

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

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

```r
# 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
```

```
# 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
```