Stat 432 Homework 5

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

Contents

Instruction

Please remove this section when submitting your homework.

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

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

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

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

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

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

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

```r
beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)
beta_t_x0 <- sum(beta * x0)
true_mean <- exp(beta_t_x0)
true_mean
```

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

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

```r
set.seed(123)

beta <- c(0.5, -0.5, 0)
n_train <- 100
k <- 21

# Generate training data
X_train <- matrix(runif(n_train * 3, min = 0, max = 1), ncol = 3)
epsilon <- rnorm(n_train, mean = 0, sd = 1)
Y_train <- exp(X_train %*% beta) + epsilon

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

# Compute Euclidean distances
distances <- sqrt(rowSums((X_train - matrix(x0, nrow = n_train, ncol = 3, byrow = TRUE))^2))

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

# Manual KNN prediction
knn_pred_manual <- mean(Y_train[nearest_indices])
print("Manual KNN Prediction:")
```

```
## [1] "Manual KNN Prediction:"
```

```r
print(knn_pred_manual)
```

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

```r
library(FNN)

knn_result <- knn.reg(train = X_train, test = matrix(x0, nrow = 1), y = Y_train, k = k)
knn_pred_FNN <- knn_result$pred
print("KNN Prediction using knn.reg from FNN package:")
```

```
## [1] "KNN Prediction using knn.reg from FNN package:"
```

```r
print(knn_pred_FNN)
```

```
## [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)

beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)
n_train <- 100
k <- 21
N <- 1000

# Compute the true mean at x0
beta_t_x0 <- sum(beta * x0)
true_mean <- exp(beta_t_x0)

# Initialize a vector to store KNN predictions
knn_predictions <- numeric(N)

# Simulation loop
for (i in 1:N) {
  # Generate training data
  X_train <- matrix(runif(n_train * 3, min = 0, max = 1), ncol = 3)
  epsilon <- rnorm(n_train, mean = 0, sd = 1)
  Y_train <- exp(X_train %*% beta) + epsilon

  # Compute Euclidean distances from x0 to all training points
  distances <- sqrt(rowSums((X_train - matrix(x0, nrow = n_train, ncol = 3, byrow = TRUE))^2))

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

  # Compute the KNN prediction
  knn_pred <- mean(Y_train[nearest_indices])

  # Store the prediction
  knn_predictions[i] <- knn_pred
}

# Calculate the estimated bias
mean_pred <- mean(knn_predictions)
estimated_bias <- mean_pred - true_mean

# Output the results
cat("Estimated Bias of the KNN model at x0:\n")
```

```
## Estimated Bias of the KNN model at x0:
```

```
cat("Mean of KNN predictions over", N, "simulations:", mean_pred, "\n")
```

```
## Mean of KNN predictions over 1000 simulations: 1.226658
```

```
cat("True mean at x0:", true_mean, "\n")
```

```
## True mean at x0: 2.398875
```

```
cat("Estimated Bias:", estimated_bias, "\n")
```

```
## Estimated 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.

```
estimated_variance <- var(knn_predictions)

# Output the estimated variance
cat("Estimated Variance of the KNN model at x0:\n")
```

```
## Estimated Variance of the KNN model at x0:
```

```
cat("Estimated Variance:", estimated_variance, "\n")
```

```
## Estimated 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?

```
mean_pred <- 1.226658
true_mean <- 2.398875
estimated_bias <- mean_pred - true_mean

bias_squared <- estimated_bias^2

cat("Bias squared:\n")
```

```
## Bias squared:
```

```r
cat("Bias^2 =", bias_squared, "\n\n")
```

```
## Bias^2 = 1.374093
```

```r
estimated_variance <- var(knn_predictions)

cat("Estimated Variance:\n")
```

```
## Estimated Variance:
```

```r
cat("Variance =", estimated_variance, "\n\n")
```

```
## Variance = 0.05032568
```

```r
sigma_squared <- 1

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

```
## Irreducible Error:
```

```r
cat("Sigma^2 =", sigma_squared, "\n\n")
```

```
## Sigma^2 = 1
```

```r
prediction_error <- bias_squared + estimated_variance + sigma_squared

cat("Calculated Prediction Error:\n")
```

```
## Calculated Prediction Error:
```

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

```
## Prediction Error = 2.424418
```

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)

beta <- c(0.5, -0.5, 0)
x0 <- c(1, -0.75, -0.7)
n_train <- 100
k <- 21
N <- 1000
```

```r
# Compute the true mean at x0
beta_t_x0 <- sum(beta * x0)
true_mean <- exp(beta_t_x0)

# Initialize vectors to store results
knn_predictions <- numeric(N)
squared_errors <- numeric(N)

# Simulation loop
for (i in 1:N) {
  # Generate training data
  X_train <- matrix(runif(n_train * 3, min = 0, max = 1), ncol = 3)
  epsilon <- rnorm(n_train, mean = 0, sd = 1)
  Y_train <- exp(X_train %*% beta) + epsilon

  # Compute Euclidean distances from x0 to all training points
  distances <- sqrt(rowSums((X_train - matrix(x0, nrow = n_train, ncol = 3, byrow = TRUE))^2))

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

  # Compute the KNN prediction
  knn_pred <- mean(Y_train[nearest_indices])

  # Store the prediction
  knn_predictions[i] <- knn_pred

  # Generate Y0 using x0
  epsilon_0 <- rnorm(1, mean = 0, sd = 1)
  Y0 <- true_mean + epsilon_0

  # Compute squared error
  squared_errors[i] <- (Y0 - knn_pred)^2
}

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

# Output the results
cat("Empirical Average Prediction Error over", N, "simulations:", empirical_prediction_error, "\n")
```

```
## Empirical Average Prediction Error over 1000 simulations: 2.546811
```

```r
cat("Theoretical Prediction Error from part (e):", 3.321914, "\n")
```

```
## Theoretical Prediction Error from part (e): 3.321914
```

Question 2: Logistic Regression (30 points)

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

```r
  # load library
  library(ISLR2)

  # load data
  data(Default)

  # set seed
  set.seed(7)

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

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

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

```r
glm_fit <- glm(default ~ balance + income, data = Default_train, family = binomial)

# Prepare the input data matrix X and response vector Y
X <- model.matrix(default ~ balance + income, data = Default_train)
Y <- as.numeric(Default_train$default == "Yes")  # Convert 'Yes' to 1 and 'No' to 0

# Define the log-likelihood function
loglikelihood <- function(beta, X, Y) {
  eta <- X %*% beta          # Linear predictor
  p <- 1 / (1 + exp(-eta))   # Logistic function
  # Avoid numerical issues by bounding p away from 0 and 1
  epsilon <- 1e-15
  p <- pmax(pmin(p, 1 - epsilon), epsilon)
  # Compute the log-likelihood
  loglik <- sum(Y * log(p) + (1 - Y) * log(1 - p))
  return(loglik)
}

# Extract the estimated coefficients from the glm() model
beta_hat <- coef(glm_fit)

# Calculate the maximum log-likelihood using the estimated coefficients
max_loglik <- loglikelihood(beta_hat, X, Y)
print(paste("Maximum Log-Likelihood:", max_loglik))
```

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

```r
# Get the deviance value from the glm() object
deviance <- glm_fit$deviance
print(paste("Deviance:", deviance))
```

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

```r
# Relationship between deviance and maximum log-likelihood
# Deviance = -2 * (Log-Likelihood - Log-Likelihood of Saturated Model)
# For the saturated model, the log-likelihood is zero in this context
# Therefore, Deviance = -2 * (Log-Likelihood - 0) => Log-Likelihood = -Deviance / 2

# Verify the relationship
calculated_loglik <- -deviance / 2
print(paste("Calculated Log-Likelihood from Deviance:", calculated_loglik))
```

```
## [1] "Calculated Log-Likelihood from Deviance: -712.29808123702"
```

```r
# Check if both log-likelihoods are approximately equal
all.equal(max_loglik, calculated_loglik)
```

```
## [1] TRUE
```

The relationship is Deviance= -2 $\times$ (Log-Likelihood - Log-Likelihood of Saturated Model)

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?

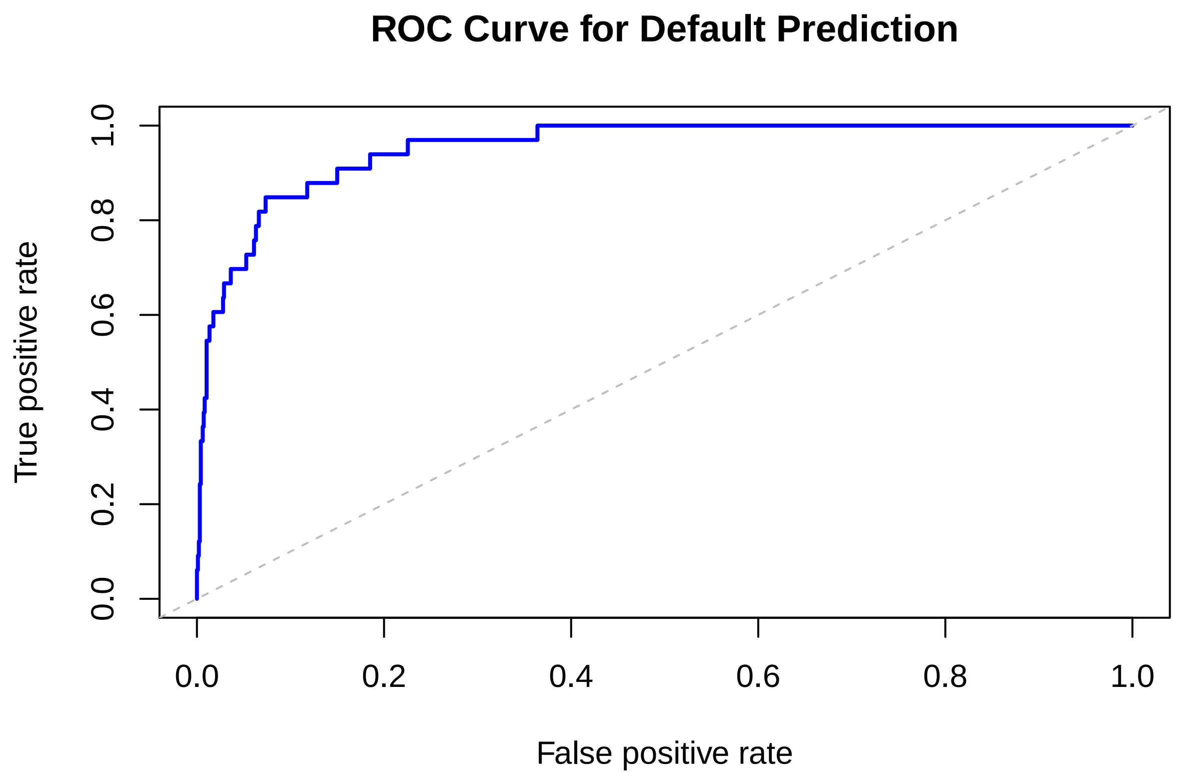
```r
# Predict probabilities on test dataset
probabilities <- predict(glm_fit, newdata = Default_test, type = "response")

# Convert actual defaults to numeric (1 for 'Yes', 0 for 'No')
Y_true_num <- ifelse(Default_test$default == "Yes", 1, 0)

# Define cutoff values
cutoffs <- c(0.3, 0.5, 0.7)

# Loop over each cutoff value
for (cutoff in cutoffs) {
  # Predict classes based on cutoff
  Y_pred <- ifelse(probabilities >= cutoff, 1, 0)

  # Confusion matrix components
  TP <- sum((Y_pred == 1) & (Y_true_num == 1))
  TN <- sum((Y_pred == 0) & (Y_true_num == 0))
  FP <- sum((Y_pred == 1) & (Y_true_num == 0))
  FN <- sum((Y_pred == 0) & (Y_true_num == 1))
  # Print confusion matrix
  confusion_matrix <- matrix(c(TN, FP, FN, TP), nrow = 2, byrow = TRUE)
  rownames(confusion_matrix) <- c("Actual No", "Actual Yes")
  colnames(confusion_matrix) <- c("Predicted No", "Predicted Yes")
  cat("\nConfusion Matrix for cutoff =", cutoff, ":\n")
  print(confusion_matrix)

  # Calculate metrics without using any R functions beyond basic operators
  total_predictions <- length(Y_true_num)
  test_error <- (FP + FN) / total_predictions
  sensitivity <- TP / (TP + FN)
  specificity <- TN / (TN + FP)
  precision <- TP / (TP + FP)

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

```
##
## Confusion Matrix for cutoff = 0.3 :
##            Predicted No Predicted Yes
## Actual No           954            13
## Actual Yes           14            19
## Test Error: 0.027
## Sensitivity: 0.5757576
## Specificity: 0.9865564
## Precision: 0.59375
##
## Confusion Matrix for cutoff = 0.5 :
##            Predicted No Predicted Yes
## Actual No           963             4
## Actual Yes           22            11
## Test Error: 0.026
## Sensitivity: 0.3333333
## Specificity: 0.9958635
## Precision: 0.7333333
##
## Confusion Matrix for cutoff = 0.7 :
##            Predicted No Predicted Yes
## Actual No           964             3
## Actual Yes           29             4
## Test Error: 0.032
## Sensitivity: 0.1212121
## Specificity: 0.9968976
## Precision: 0.5714286
```

Cutoff = 0.5 might be preferred as it balances sensitivity and specificity, providing moderate test error with acceptable sensitivity and specificity. If our goal is to capture as many people, then we prefer cutoff of 0.3 since it has highest Sensitivity, which aligns with the goal of capturing as many people who will default as possible.

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

```
library(ROCR)

# Predict probabilities on the test dataset
probabilities <- predict(glm_fit, newdata = Default_test, type = "response")

# Actual class labels (0 and 1)
Y_true_num <- ifelse(Default_test$default == "Yes", 1, 0)

# Create a prediction object
pred <- prediction(probabilities, Y_true_num)

# Calculate performance metrics
perf <- performance(pred, "tpr", "fpr")

# Plot the ROC curve
plot(perf, main = "ROC Curve for Default Prediction", col = "blue", lwd = 2)
abline(a=0, b=1, lty=2, col="gray")  # Diagonal line
```



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

```
## AUC: 0.9523048
```

```
# Use the ROC curve to determine a cutoff value
# Find the cutoff corresponding to the point closest to the top-left corner
cutoffs <- pred@cutoffs[[1]]
fpr <- perf@x.values[[1]]
tpr <- perf@y.values[[1]]
distances <- sqrt((fpr)^2 + (1 - tpr)^2)  # Euclidean distance to (0,1)

# Find the index of the minimum distance
min_index <- which.min(distances)
optimal_cutoff <- cutoffs[min_index]
cat("Optimal cutoff value based on ROC curve:", optimal_cutoff, "\n")
```

```
## Optimal cutoff value based on ROC curve: 0.06527171
```

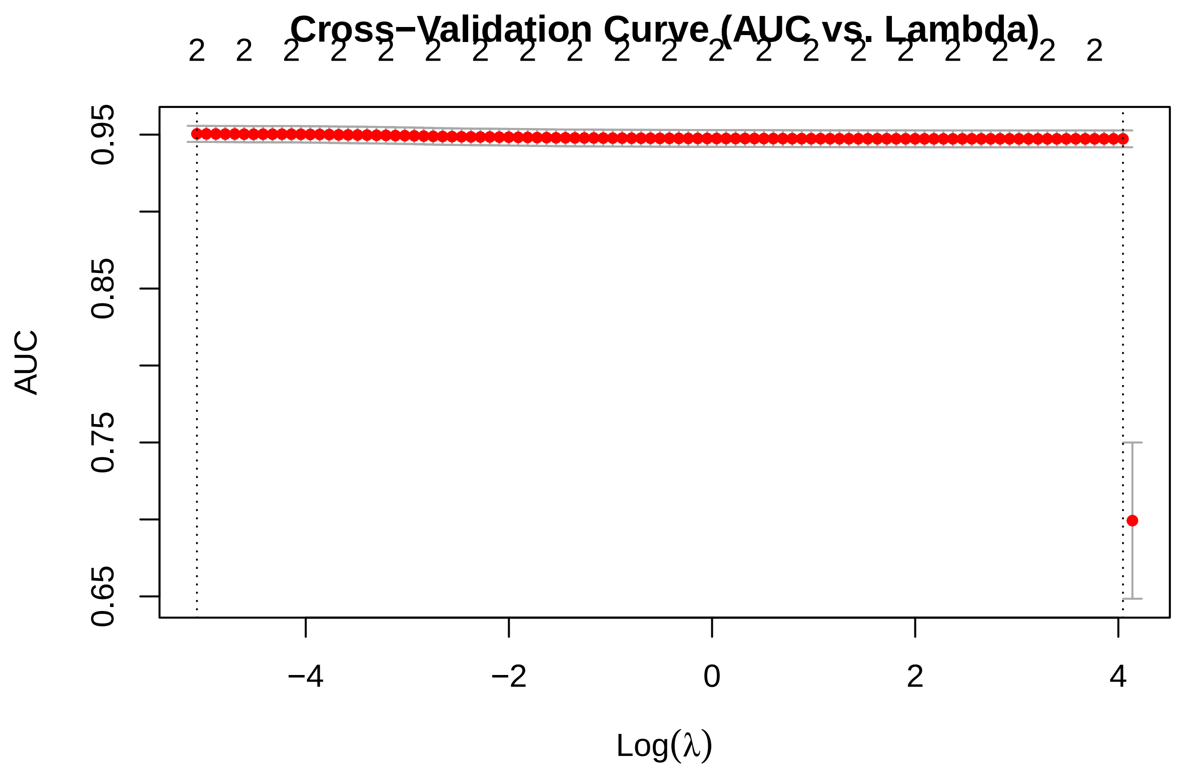
d. [5 pts] Load the library `glmnet`. Using the `cv.glmnet()` function, do 20-fold cross-validation on the training dataset to determine the optimal penalty coefficient, λ, in the logistic regression with ridge penalty. In order to choose the best penalty coefficient use AUC as the Cross-Validation metric.

```
library(glmnet)
```

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

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

```
# Prepare the data for glmnet
# Exclude the intercept term from X
X_train <- model.matrix(default ~ balance + income, data = Default_train)[, -1]
Y_train <- ifelse(Default_train$default == "Yes", 1, 0)

# Set seed for reproducibility
set.seed(7)

# Perform 20-fold cross-validation using cv.glmnet()
cv_fit <- cv.glmnet(
  x = X_train,
  y = Y_train,
  family = "binomial",
  alpha = 0,
  nfolds = 20,
  type.measure = "auc"
)

# Plot cross-validation curve
plot(cv_fit)
title("Cross-Validation Curve (AUC vs. Lambda)")
```



```r
# Get the optimal lambda (penalty coefficient)
optimal_lambda <- cv_fit$lambda.min
cat("Optimal lambda (penalty coefficient):", optimal_lambda, "\n")
```

```
## Optimal lambda (penalty coefficient): 0.006272533
```

```r
# Get lambda within one standard error
lambda_1se <- cv_fit$lambda.1se
cat("Lambda at 1 standard error (simpler model):", lambda_1se, "\n")
```

```
## Lambda at 1 standard error (simpler model): 57.15298
```

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

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

```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 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?

```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) {
  if (euclDistUsed) {
    distVec <- euclidean_distance_all(vec1, mat1_X)
  } else {
    distVec <- manhattan_distance_all(vec1, mat1_X)
  }

  # Find the indices of the K nearest neighbors
  idx <- order(distVec)[1:K]

  # Get the class labels of the K nearest neighbors
  nearest_labels <- mat1_Y[idx]

  # Compute the majority vote
  majority_vote <- as.numeric(names(sort(table(nearest_labels), decreasing = TRUE)[1]))

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

train_X <- mnist[1:100, -1]  # Exclude the 'Digit' column
train_Y <- mnist[1:100, 1]   # 'Digit' column as labels

# Extract the 123rd observation as the test vector
test_vec <- mnist[123, -1]   # Exclude the 'Digit' column
actual_label <- mnist[123, 1]  # Actual label of the test observation

# Set K = 10
K <- 10

# Predict using Euclidean distance
result_eucl <- my_KNN(test_vec, train_X, train_Y, K, euclDistUsed = TRUE)
```

```r
# Extract the predicted label
predicted_label_eucl <- result_eucl[[2]]

# Print the results
cat("Euclidean Distance:\n")
```

```
## Euclidean Distance:
```

```r
cat("Predicted Label:", predicted_label_eucl, "\n")
```

```
## Predicted Label: 7
```

```r
cat("Actual Label:", actual_label, "\n")
```

```
## Actual Label: 7
```

```r
cat("Nearest Neighbors' Labels:", result_eucl[[1]], "\n")
```

```
## Nearest Neighbors' Labels: 7 7 7 7 9 9 7 9 7 9
```

```r
# Predict using Manhattan distance
result_manh <- my_KNN(test_vec, train_X, train_Y, K, euclDistUsed = FALSE)

# Extract the predicted label
predicted_label_manh <- result_manh[[2]]

# Print the results
cat("\nManhattan Distance:\n")
```

```
##
## Manhattan Distance:
```

```r
cat("Predicted Label:", predicted_label_manh, "\n")
```

```
## Predicted Label: 7
```

```r
cat("Actual Label:", actual_label, "\n")
```

```
## Actual Label: 7
```

```r
cat("Nearest Neighbors' Labels:", result_manh[[1]], "\n")
```

```
## Nearest Neighbors' Labels: 7 7 7 7 9 7 9 7 9 9
```

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

```r
set.seed(7)

library(caret)
```

```
## Loading required package: ggplot2

## Loading required package: lattice
```

```r
# Training data (first 1250 observations)
train_data <- mnist[1:1250, ]
train_X <- train_data[, -1]  # Exclude 'Digit' column (pixels only)
train_Y <- train_data$Digit  # Labels

# Test data (observations 1251 to 2500)
test_data <- mnist[1251:2500, ]
test_X <- test_data[, -1]    # Pixels only
test_Y <- test_data$Digit    # Actual labels

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

# Define the grid of K values to test
tune_grid <- expand.grid(k = c(1, 5, 10, 20, 50, 100))

# Train the KNN model
knn_fit <- train(x = train_X,
                 y = as.factor(train_Y),
                 method = "knn",
                 trControl = train_control,
                 tuneGrid = tune_grid)

print(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.
```

```
# Best K
best_K <- knn_fit$bestTune$k
cat("The best K according to cross-validation is:", best_K, "\n")
```

```
## The best K according to cross-validation is: 1
```

```
# Predict on the test data
predictions <- predict(knn_fit, newdata = test_X)

# Compute confusion matrix
confusion_mtx <- confusionMatrix(predictions, as.factor(test_Y))
print(confusion_mtx)
```

```
## 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
```

```
# Test error
test_error <- 1 - confusion_mtx$overall['Accuracy']
cat("Test Error:", round(test_error, 4), "\n")
```

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

```
# Sensitivity for each class
sensitivity_per_class <- confusion_mtx$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
```

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)

# Split the data into training and testing sets
train_size <- 1250

# Prepare training data
X_train <- as.matrix(mnist[1:train_size, -1])
y_train <- as.factor(mnist$Digit[1:train_size])

# Prepare testing data
```

```r
X_test <- as.matrix(mnist[(train_size + 1):(2 * train_size), -1])
y_test <- as.factor(mnist$Digit[(train_size + 1):(2 * train_size)])

# Use cv.glmnet with 20 folds to find 1se
cvfit <- cv.glmnet(X_train, y_train, family = "multinomial", nfolds = 20)

# Extract the optimal lambda value
lambda_1se <- cvfit$lambda.1se

# Predict the class labels for the testing data using cvfit
predictions <- predict(cvfit, X_test, s = "lambda.1se", type = "class")

# Report the confusion matrix
confusion_matrix <- table(True = y_test, Predicted = predictions)
print(confusion_matrix)
```

```
##     Predicted
## True   0   1   2   3   4   5   6   7   8   9
##    0 109   0   1   0   1   2   0   0   4   0
##    1   0 115   2   0   0   1   1   1   8   0
##    2   4   3 107   0   4   0   3   2   2   0
##    3   0   4   8  99   0   7   1   1   1   1
##    4   2   2   3   0 115   2   1   3   4  12
##    5   2   0   0   5   2  98   3   3   6   0
##    6   2   1   7   0   2   4 122   1   0   0
##    7   2   1   1   4   2   0   0 109   0   7
##    8   1   4   4   1   1   3   1   2  89   4
##    9   1   0   0   4   8   2   0   6   0  99
```

```r
# Calculate the test error
test_error <- mean(predictions != y_test)
print(paste("Test Error:", round(test_error * 100, 2)))
```

```
## [1] "Test Error: 15.04"
```