HW08_zilinw3

Zilin Wang (zilinw3)

2024-10-21

Contents

Question 1: Discriminant Analysis (60 points)

We will be using the first 2500 observations of the MNIST dataset. You can use the following code, or the saved data from our previous homework.

```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. [10 pts] Write you own code to fit a Linear Discriminant Analysis (LDA) model to the MNIST dataset. Use the first 1250 observations as the training set and the remaining observations as the test set. An issue with this dataset is that some pixels display little or no variation across all observations. This zero variance issue poses a problem when inverting the estimated covariance matrix. To address this issue, take digits 1, 7, and 9 from the training data, and perform a screening on the marginal variance of all 784 pixels. Take the top 300 pixels with the largest variance and use them to fit the LDA model. Remove the remaining ones from the training and test data.

Answer:

```r
library(MASS)
# Split data into training and testing
mnist_train <- mnist[1:1250, ]
mnist_test <- mnist[1251:2500, ]

# Filter for digits 1, 7, and 9
mnist_train <- mnist_train[mnist_train$Digit %in% c(1, 7, 9), ]
mnist_test <- mnist_test[mnist_test$Digit %in% c(1, 7, 9), ]

# Calculate variances of each pixel
variances <- apply(mnist_train[, -1], 2, var)  # Excluding the 'Digit' column

# Get the indices of the top 300 pixels based on variance
top_300_indices <- order(variances, decreasing = TRUE)[1:300]

# Filter both training and testing data to include only these pixels
mnist_train_filtered <- mnist_train[, c(1, top_300_indices + 1)]
mnist_test_filtered <- mnist_test[, c(1, top_300_indices + 1)]

# Fit the LDA model
lda_model <- lda(Digit ~ ., data = mnist_train_filtered)
```

b. [30 pts] Write your own code to implement the LDA model. Remember that LDA requires the estimation of several parameters: Σ, μ_k, and π_k. Estimate these parameters and calculate the decision scores δ_k on the testing data to predict the class label. Report the accuracy and the confusion matrix based on the testing data.

Answer:

```r
# Function to calculate class means and prior probabilities
calculate_class_parameters <- function(X, y) {
  classes <- unique(y)
  n <- length(y)
  class_means <- list()
  prior_probs <- numeric(length(classes))

  for(i in seq_along(classes)) {
    class_idx <- y == classes[i]
    class_means[[i]] <- colMeans(X[class_idx, ])
    prior_probs[i] <- sum(class_idx) / n
  }

  return(list(means = class_means, priors = prior_probs, classes = classes))
}

# Function to calculate pooled covariance matrix
calculate_pooled_covariance <- function(X, y, class_means) {
  classes <- unique(y)
  p <- ncol(X)
  S <- matrix(0, p, p)
  n <- length(y)

  for(i in seq_along(classes)) {
    class_idx <- y == classes[i]
    centered <- scale(X[class_idx, ], center = class_means[[i]], scale = FALSE)
    S <- S + (t(centered) %*% centered)
  }

  S <- S / (n - length(classes))
  return(S)
}

# Function to calculate LDA decision scores - FIXED VERSION
calculate_decision_scores <- function(X, class_means, pooled_cov, prior_probs) {
  scores <- matrix(0, nrow(X), length(class_means))
  inv_cov <- solve(pooled_cov)

  for(i in seq_along(class_means)) {
    mu <- matrix(class_means[[i]], ncol = 1)
    term1 <- X %*% (inv_cov %*% mu)
    term2 <- 0.5 * as.numeric(t(mu) %*% inv_cov %*% mu)
    term3 <- log(prior_probs[i])
    scores[, i] <- term1 - term2 + term3
  }

  return(scores)
}

# Main LDA function
custom_lda <- function(X_train, y_train, X_test) {
  # Calculate class parameters
  params <- calculate_class_parameters(X_train, y_train)

  # Calculate pooled covariance matrix
  pooled_cov <- calculate_pooled_covariance(X_train, y_train, params$means)

  # Calculate decision scores for test data
  scores <- calculate_decision_scores(X_test, params$means, pooled_cov, params$priors)

  # Predict classes
  predictions <- params$classes[max.col(scores)]
  return(predictions)
}

# Apply custom LDA to the filtered MNIST data
X_train <- as.matrix(mnist_train_filtered[, -1])  # Remove Digit column
y_train <- mnist_train_filtered$Digit
X_test <- as.matrix(mnist_test_filtered[, -1])
y_test <- mnist_test_filtered$Digit

# Make predictions
predictions <- custom_lda(X_train, y_train, X_test)
```

```r
# Calculate accuracy
accuracy <- mean(predictions == y_test)
print(paste("Accuracy:", round(accuracy * 100, 2), "%"))
```

```
## [1] "Accuracy: 89.04 %"
```

```r
# Create confusion matrix
conf_matrix <- table(Predicted = predictions, Actual = y_test)
print("Confusion Matrix:")
```

```
## [1] "Confusion Matrix:"
```

```r
print(conf_matrix)
```

```
##          Actual
## Predicted   1   7   9
##         1 125   3   2
##         7   2 111  21
##         9   1  12  97
```

c. [10 pts] Use the `lda()` function from MASS package to fit LDA. Report the accuracy and the confusion matrix based on the testing data. Compare your results with part b.

Answer:

```r
library(MASS)

# Fit LDA
mass_lda <- lda(Digit ~ ., data = mnist_train_filtered)

# Make predictions
mass_predictions <- predict(mass_lda, mnist_test_filtered)$class

# Calculate accuracy
mass_accuracy <- mean(mass_predictions == mnist_test_filtered$Digit)
print(paste("MASS LDA Accuracy:", round(mass_accuracy * 100, 2), "%"))
```

```
## [1] "MASS LDA Accuracy: 89.04 %"
```

```r
# Create confusion matrix
mass_conf_matrix <- table(Predicted = mass_predictions, Actual = mnist_test_filtered$Digit)
print("MASS LDA Confusion Matrix:")
```

```
## [1] "MASS LDA Confusion Matrix:"
```

```r
print(mass_conf_matrix)
```

```
##           Actual
## Predicted   1   7   9
##         1 125   3   2
##         7   2 111  21
##         9   1  12  97
```

The results are the same with part b.

d. [10 pts] Use the `qda()` function from MASS package to fit QDA. Does the code work directly? Why? If you are asked to modify your own code to perform QDA, what would you do? Discuss this issue and propose at least two solutions to address it. If relavent, provide mathematical reasoning (in latex) of your solution. You **do not** need to implement that with code.

Answer:

```r
# Fit QDA
tryCatch({
    qda_model <- qda(Digit ~ ., data = mnist_train_filtered)
}, error = function(e) {
    print("QDA failed with error:")
    print(e)
})
```

```
## [1] "QDA failed with error:"
## <simpleError in qda.default(x, grouping, ...): some group is too small for 'qda'>
```

The code does not work directly. The error message indicates that "some group is too small for 'qda'." This typically means that the number of samples in at least one of the classes in the dataset is less than the number of features (300), which is required to estimate the class-specific covariance matrices reliably. QDA requires estimating a separate covariance matrix for each class. If any class has fewer samples than the number of features, the covariance matrix for that class cannot be estimated accurately because the matrix would be singular (i.e., not invertible).

Solution 1: Before applying QDA, use PCA to reduce the dimensionality of the data. By reducing the number of features to less than the smallest class size, PCA can help make the problem tractable. PCA transforms the data into a new coordinate system, reducing the dimension while retaining the variance that is most explanatory. The transformed dataset, with fewer dimensions, would allow each class-specific covariance matrix to be estimable even with fewer samples. Mathematical reasoning: If X is your data matrix and V are the eigenvectors corresponding to the largest eigenvalues derived from the covariance matrix of X, then $X_{PCA} = XV$ will have reduced dimensions where each class can potentially have more samples than features.

Solution 2: One solution is regularization, using shrinkage techniques. In this method, we add a small regularization term to the diagonal of each class's covariance matrix, making it more stable and invertible even when the number of samples in class is small. This adjust the covariance matrices by pulling them towards a central matrix, such as the identity matrix or the pooled covariance matrix, thus avoiding the singularity problem.

Mathematical reasoning: This can be expressed as

$$S_k^* = \lambda S_k + (1 - \lambda)S,$$

where S_k is the original covariance matrix for class k, S is the pooled covariance, and λ is a parameter between 0 and 1. This makes S_k^* more stable and invertible.

Question 2: Regression Trees (40 points)

Load data `Carseats` from the `ISLR` package. Use the following code to define the training and test sets.

```r
# load library
library(ISLR)
```

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

```r
# load data
data(Carseats)

# set seed
set.seed(7)

# number of rows in entire dataset
n_Carseats <- dim(Carseats)[1]

# training set parameters
train_percentage <- 0.75
train_size <- floor(train_percentage*n_Carseats)
train_indices <- sample(x = 1:n_Carseats, size = train_size)

# separate dataset into train and test
train_Carseats <- Carseats[train_indices,]
test_Carseats <- Carseats[-train_indices,]
```

a. [20 pts] We seek to predict the variable `Sales` using a regression tree. Load the library `rpart`. Fit a regression tree to the training set using the `rpart()` function, all hyperparameter arguments should be left as default. Load the library `rpart.plot()`. Plot the tree using the `prp()` function. Based on this model, what type of observations has the highest or lowest sales? Predict using the tree onto the test set, calculate and report the MSE on the testing data.

Answer:

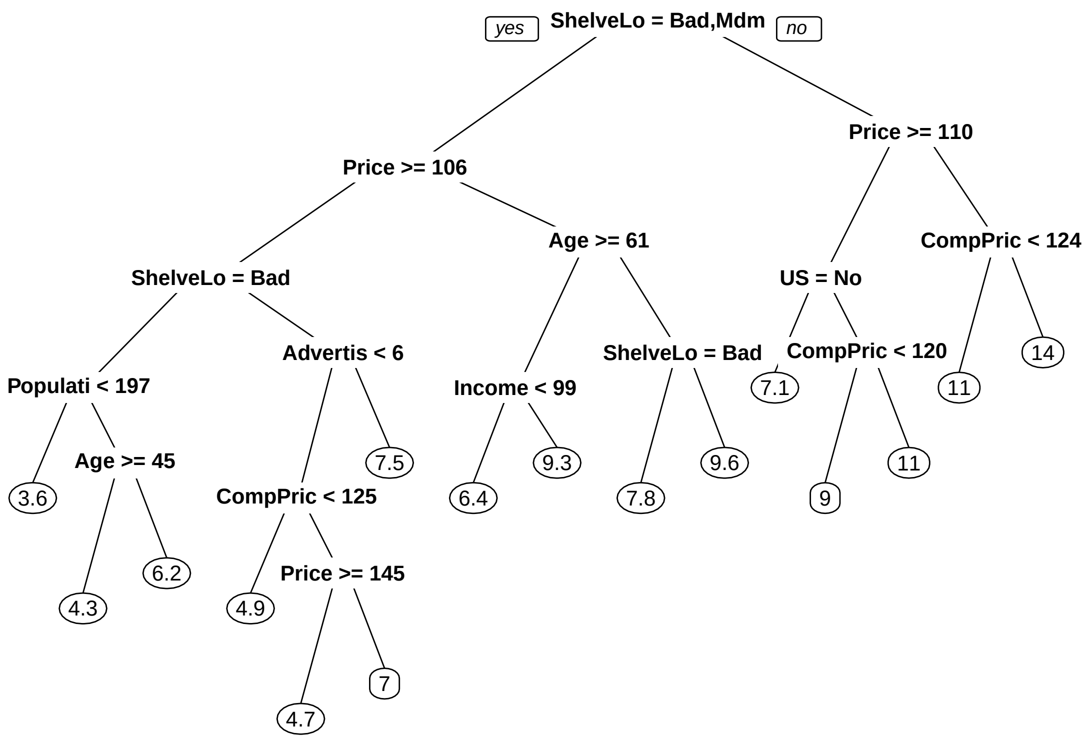
```r
library(rpart)
```

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

```r
library(rpart.plot)
```

```
## Warning: package 'rpart.plot' was built under R version 4.3.3
```

```r
# Fit regression tree model on training data
tree_model <- rpart(Sales ~ ., data = train_Carseats)

# Plot the tree
prp(tree_model)
```



```
# Predict on test data using the fitted tree
test_predictions <- predict(tree_model, newdata = test_Carseats)

# Calculate Mean Squared Error
mse <- mean((test_Carseats$Sales - test_predictions)^2)
cat("MSE on the testing data: ", mse, "\n")
```

```
## MSE on the testing data:  4.51347
```

The lowest predicted sales value in the tree is 3.6, which is the left-most branch of the tree, and occurs under the conditions: Shelf Location = Bad, Medium Price >= 106
Shelf Location = Bad
Population < 197

The highest predicted sales value in the tree is 14, which is the right-most branch of the tree, and occurs under the conditions:
Shelf Location = Good Price < 110
Competitive Price >= 124

b. [20 pts] Set the seed to 7 at the beginning of the chunk and do this question in a single chunk so the seed doesn't get switched. Find the largest complexity parameter value of the tree you grew in part a) that will ensure that the cross-validation error < min(cross-validation error) + cross-validation standard deviation. Print that complexity parameter value. Prune the tree using that value. Predict using the pruned tree onto the test set, calculate the test Mean-Squared Error, and print it.

Answer:

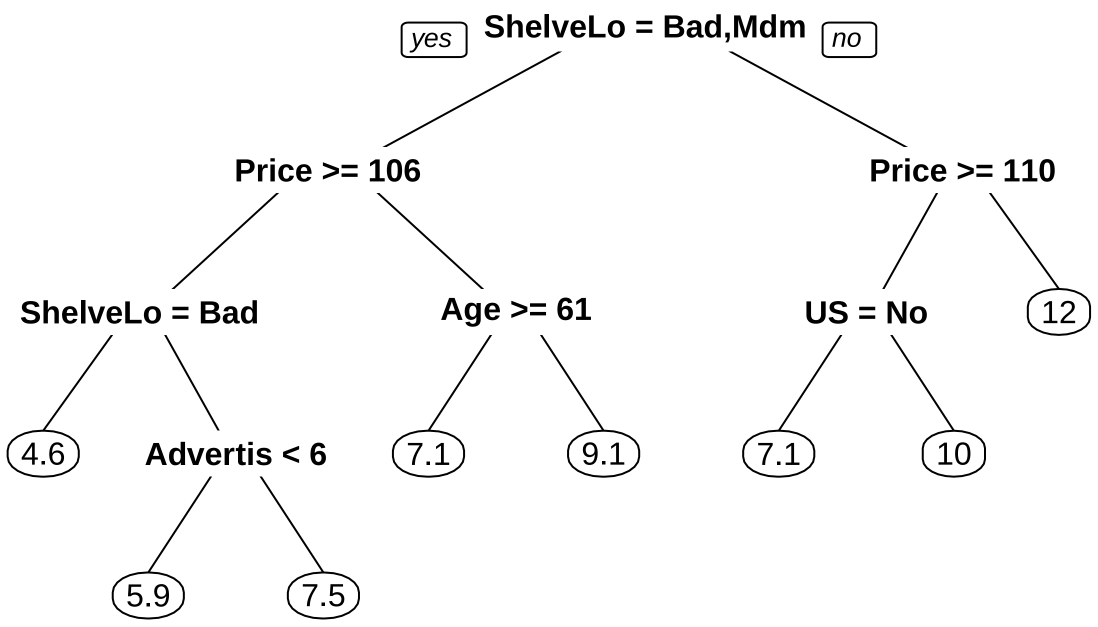
```r
set.seed(7)

# Perform cross-validation and capture the output
cv_results <- printcp(tree_model)  # This prints the CP table to the console
```

```
##
## Regression tree:
## rpart(formula = Sales ~ ., data = train_Carseats)
##
## Variables actually used in tree construction:
## [1] Advertising Age         CompPrice   Income      Population  Price
## [7] ShelveLoc   US
##
## Root node error: 2444.4/300 = 8.1481
##
## n= 300
##
##          CP nsplit rel error  xerror     xstd
## 1  0.261715      0   1.00000 1.00519 0.080889
## 2  0.108414      1   0.73829 0.74837 0.060966
## 3  0.055183      2   0.62987 0.64540 0.050969
## 4  0.041433      3   0.57469 0.60304 0.050812
## 5  0.035313      4   0.53326 0.60202 0.049128
## 6  0.030281      5   0.49794 0.59227 0.047725
## 7  0.029581      6   0.46766 0.56927 0.047203
## 8  0.021730      7   0.43808 0.52865 0.043437
## 9  0.015299      8   0.41635 0.53788 0.042712
## 10 0.014790     10   0.38575 0.53936 0.042342
## 11 0.011753     11   0.37096 0.54096 0.043843
## 12 0.011561     12   0.35921 0.53575 0.042833
## 13 0.010168     13   0.34765 0.52210 0.041029
## 14 0.010109     14   0.33748 0.52258 0.041236
## 15 0.010000     15   0.32737 0.52202 0.041214
```

```r
# Extract the complexity parameter table directly from the model
cptable <- tree_model$cptable

# Find the smallest complexity parameter (CP) that minimizes xerror
min_xerror <- min(cptable[, "xerror"])
min_xstd <- cptable[which.min(cptable[, "xerror"]), "xstd"]

# Compute the threshold for xerror
threshold_xerror <- min_xerror + min_xstd

# Identify the largest CP where xerror is less than the threshold
optimal_cp <- max(cptable[cptable[, "xerror"] < threshold_xerror, "CP"])

# Print the optimal complexity parameter
cat("Optimal Complexity Parameter for Pruning:", optimal_cp, "\n")
```

```
## Optimal Complexity Parameter for Pruning: 0.02173028
```

```r
# Prune the tree using the optimal complexity parameter
pruned_tree <- prune(tree_model, cp = optimal_cp)

# Plot the pruned tree
prp(pruned_tree)
```



```r
# Predict on test data using the pruned tree
test_predictions <- predict(pruned_tree, newdata = test_Carseats)

# Calculate Mean Squared Error
mse <- mean((test_Carseats$Sales - test_predictions)^2)
cat("MSE on the testing data: ", mse, "\n")
```

```
## MSE on the testing data:  4.543565
```