Stat 432 Homework 8

Assigned: Oct 14, 2024; Due: 11:59 PM CT, Oct 24, 2024

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.

```
# 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.

```
library(MASS)
# Split data into training and test sets
train_data <- mnist[1:1250, ]
test_data <- mnist[1251:2500, ]
```

```r
# Filter training and test data for digits 1, 7, and 9
train_data_filtered <- train_data[train_data$Digit %in% c(1, 7, 9), ]
test_data_filtered <- test_data[test_data$Digit %in% c(1, 7, 9), ]

# Compute the variance for each pixel in the training set
pixel_variances <- apply(train_data_filtered[, -1], 2, var)

# Select the top 300 pixels based on the highest variance
top_pixels <- order(pixel_variances, decreasing = TRUE)[1:300]

# Subset the training and test data to keep only the top 300 pixels
mnist_train_filtered <- train_data_filtered[, c(1, top_pixels + 1)]  # +1 for the Digit column
mnist_test_filtered <- test_data_filtered[, c(1, top_pixels + 1)]

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

# Print the LDA model summary
summary(lda_model)
```

```
##         Length Class  Mode
## prior      3    -none- numeric
## counts     3    -none- numeric
## means    900    -none- numeric
## scaling  600    -none- numeric
## lev        3    -none- character
## svd        2    -none- numeric
## N          1    -none- numeric
## call       3    -none- call
## terms      3    terms  call
## xlevels    0    -none- list
```

```r
# Make predictions on the test data
lda_predictions <- predict(lda_model, newdata = mnist_test_filtered)


# Generate confusion matrix to evaluate predictions
confusion_matrix <- table(mnist_test_filtered$Digit, lda_predictions$class)
#accuracy <-mean(lda_predictions$class==mnist_test_filtered$Digit)
# Calculate and print accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Test Accuracy:", accuracy * 100, "%"))
```

```
## [1] "Test Accuracy: 89.0374331550802 %"
```

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

1. **Estimate the parameters:**
   - μ_k: the mean vector for each class k.
   - Σ: the shared covariance matrix across all classes.
   - π_k: the prior probability for each class k.
2. **Calculate the decision scores $\delta_k(x)$ for each class on the testing data:**

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log(\pi_k)$$

   where x is the input data point and $\delta_k(x)$ is the score for class k.

```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)
  n <- length(y)
  S <- matrix(0, nrow = p, ncol = p)  # Initialize the covariance matrix
  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  # Update covariance matrix
  }
  S <- S / (n - length(classes))  # Pooled covariance matrix
  return(S)
}

# Function to calculate LDA decision scores
calculate_decision_scores <- function(X, class_means, pooled_cov, prior_probs) {
  scores <- matrix(0, nrow = nrow(X), ncol = length(class_means))
  inv_cov <- solve(pooled_cov)  # Inverse of the covariance matrix
  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
  }
```

```r
  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$priors)
  # Predict classes based on maximum score
  predictions <- params$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)

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

```
## [1] "Accuracy: 89.0374331550802 %"
```

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

```
## 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.

```r
# Load the MASS package
library(MASS)

# Fit the LDA model using the MASS lda() function
lda_model_mass <- lda(Digit ~ ., data = mnist_train_filtered)

# Make predictions on the test set
lda_predictions_mass <- predict(lda_model_mass, newdata = mnist_test_filtered)

# Generate confusion matrix to evaluate predictions
confusion_matrix_mass <- table(Predicted = mnist_test_filtered$Digit, Actual = lda_predictions_mass$cla

# Calculate accuracy
accuracy_mass <- sum(diag(confusion_matrix_mass)) / sum(confusion_matrix_mass)

# Print the results
cat("Test Accuracy (MASS lda()):", accuracy_mass * 100, "%")
```

```
## Test Accuracy (MASS lda()): 89.03743 %
```

```r
cat("Confusion Matrix (MASS lda()): ", "\n")
```

```
## Confusion Matrix (MASS lda()):
```

```r
print(confusion_matrix_mass)
```

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

```r
# Compare with part (b)
cat("\nComparison with Part (b):\n")
```

```
##
## Comparison with Part (b):
```

```r
cat("Accuracy in Part (b):", accuracy * 100, "%")
```

```
## Accuracy in Part (b): 89.03743 %
```

```r
cat("Confusion Matrix in Part (b):\n")
```

```
## Confusion Matrix in Part (b):
```

```r
print(conf_matrix)
```

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

The accuracy of the predictions in part c was 89.03743 %, which is same to the accuracy obtained from our manual implementation in part b. The confusion matrix in part c shows similar results to part b, with most predictions being accurate, though there were some misclassifications between digits 7 and 9. Specifically, digit 7 was misclassified as 9 in 12 cases, and digit 9 was misclassified as 7 in 21 cases.

Overall, the performance of the LDA model using the lda() function matches that of the manual implementation. This demonstrates that both approaches are effective in classifying the digits with similar accuracy and confusion patterns. However, using the lda() function is more straightforward and computationally efficient compared to the manual implementation, which required estimating all the parameters explicitly.

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.

```
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'>
```

QDA using the `qda()` function:

When attempting to fit a Quadratic Discriminant Analysis (QDA) model using the `qda()` function from the `MASS` package, the code does not always work directly. This is because QDA estimates a separate covariance matrix for each class. If the covariance matrix for any class is singular (non-invertible), it leads to an error during matrix inversion.

Why does this happen?

The covariance matrix can become singular or near-singular due to two main reasons:

- **High-dimensionality (p > n problem):** When the number of features (pixels) p exceeds the number of observations n, the covariance matrix is not of full rank. This is because there are not enough data points to estimate all the parameters of the covariance matrix. Mathematically, a covariance matrix $\Sigma_k \in \mathbb{R}^{p \times p}$ is invertible if and only if its rank is p. However, if $p > n$, the matrix becomes rank-deficient, leading to singularity.
- **Near-zero variance features:** If certain features (pixels) have very little or no variation within a class, the covariance matrix becomes nearly singular. These near-zero variance features create redundancy, leading to a covariance matrix with nearly linearly dependent rows or columns, which again results in a matrix that cannot be inverted.

How to address this issue?

Solution 1: Regularization (Shrinkage QDA)

One solution is to apply **regularization** to the covariance matrix. In Shrinkage QDA, a regularization term is added to the covariance matrix to prevent it from becoming singular:

$$\Sigma_k^{\text{reg}} = (1 - \lambda)\Sigma_k + \lambda I$$

Where:

- Σ_k is the estimated covariance matrix for class k,
- λ is the regularization parameter (typically a small positive constant),
- I is the identity matrix of size $p \times p$.

By adding λI, we ensure that the eigenvalues of Σ_k^{reg} are strictly positive, making the covariance matrix invertible. This shrinks the covariance matrix towards a diagonal matrix, which stabilizes the inversion process and avoids singularity.

Solution 2: Dimensionality Reduction (PCA)

Another approach is to reduce the dimensionality of the feature space using **Principal Component Analysis (PCA)**. In this method, we project the data onto a lower-dimensional subspace spanned by the top k principal components, where $k < n$. This reduces the number of features p, making it more likely that the covariance matrix will be invertible.

The transformation is given by:

$$X_{\text{PCA}} = XV_k$$

Where:

- X is the original data matrix,
- V_k is a matrix containing the top k eigenvectors of the covariance matrix of X,
- X_{PCA} is the transformed dataset in the lower-dimensional space.

By selecting the principal components that capture the most variance, we eliminate the redundant or low-variance features that could lead to a singular covariance matrix. This approach helps reduce the chance of singularity, especially in high-dimensional settings like image data.

Other Potential Solutions:

- **Feature Selection**: Instead of reducing the dimensionality through PCA, another approach is to use feature selection techniques to retain only the most important features. This reduces the likelihood of singular matrices while keeping the most relevant features for classification.
- **Handling Near-zero Variance Features**: Automatically removing or penalizing features with near-zero variance before fitting the QDA model can also help avoid the singular covariance matrix problem.

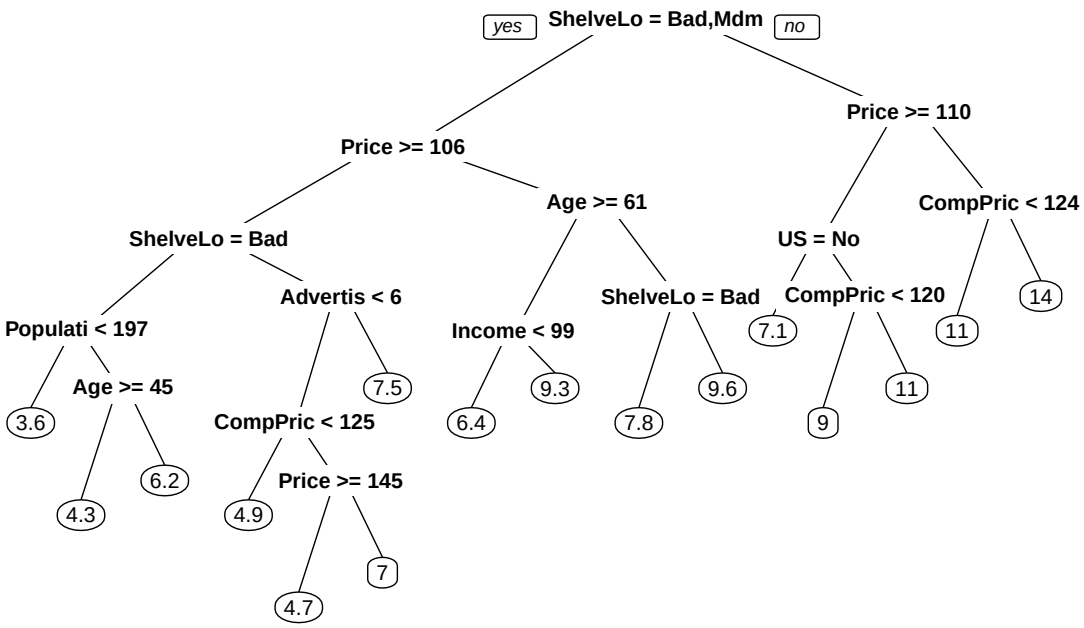
Question 2: Regression Trees (40 points)

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

```r
# load library
library(ISLR)

# 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.

```r
# Set the seed for reproducibility
set.seed(7)

# Load necessary libraries
library(rpart)
library(rpart.plot)

# Fit the regression tree on the training set
tree_model <- rpart(Sales ~ ., data = train_Carseats)

# Plot the regression tree using prp()
prp(tree_model, main = "Regression Tree for Carseats Sales")
```




```
# Predict sales on the test set using the fitted tree model
pred_test <- predict(tree_model, newdata = test_Carseats)

# Calculate the Mean Squared Error (MSE) for the test set predictions
mse_test <- mean((pred_test - test_Carseats$Sales)^2)

# Print the test MSE
print(paste("Test MSE for unpruned tree:", mse_test))
```

```
## [1] "Test MSE for unpruned tree: 4.5134702708434"
```

Based on the regression tree model, the observations with the highest predicted sales (14 units) occur when the shelf location is not bad or medium (ShelveLo $\neq$ Bad, Mdm), the product price is relatively high (Price $\geq$ 110), and the competitor's price is relatively low (CompPrice $<$ 124). This suggests that sales are maximized when the product is positioned as a premium offering with a higher price, especially when competitors are offering similar products at lower prices.

Conversely, the lowest predicted sales (3.6 units) occur when the shelf location is poor (ShelveLo = Bad), the population is smaller (Population $<$ 197), and the average customer Age $\geq$ 45. These conditions likely result in reduced product visibility in stores, a limited customer base, and an older demographic that may be less inclined to purchase the product.

When applying the tree model to the test set, the MSE was calculated to be 4.51347, indicating that while the model demonstrates reasonable predictive accuracy, there is potential for improvement through further tuning or the exploration of alternative models.

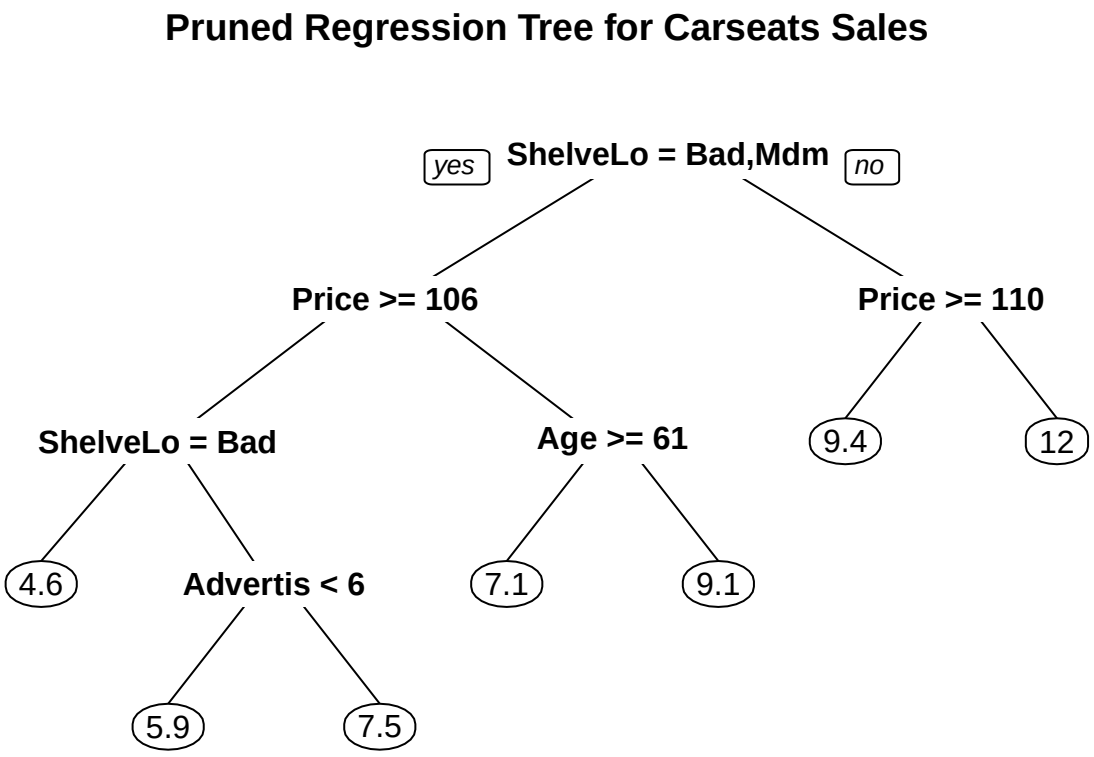
b. [20 pts] Set the seed to 7 at the beginning of the chunk and do this question in a single chunk so the seed doesn't get switched. Find the largest complexity parameter value of the tree you grew in part a) that will ensure that the cross-validation error < min(cross-validation error) + cross-validation 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.

```
# Set the seed for reproducibility
set.seed(7)

# Load necessary libraries
library(rpart)
library(rpart.plot)

# Fit the regression tree on the training set
tree_model <- rpart(Sales ~ ., data = train_Carseats)

# Print the complexity parameter table (cp table)
printcp(tree_model)
```

```
##
## 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 1.00885 0.080902
## 2  0.108414      1   0.73829 0.74614 0.060710
## 3  0.055183      2   0.62987 0.64085 0.050502
## 4  0.041433      3   0.57469 0.64468 0.052282
## 5  0.035313      4   0.53326 0.63833 0.052327
## 6  0.030281      5   0.49794 0.61103 0.051070
## 7  0.029581      6   0.46766 0.57357 0.046772
## 8  0.021730      7   0.43808 0.56317 0.046245
## 9  0.015299      8   0.41635 0.58863 0.049423
## 10 0.014790     10   0.38575 0.58107 0.047325
## 11 0.011753     11   0.37096 0.57054 0.046653
## 12 0.011561     12   0.35921 0.56889 0.045733
## 13 0.010168     13   0.34765 0.56258 0.045401
## 14 0.010109     14   0.33748 0.55040 0.044131
## 15 0.010000     15   0.32737 0.54886 0.044241
```

```
# Extract the cp table
cp_table <- tree_model$cptable

# Find the smallest cross-validation error (min xerror) and the corresponding standard deviation
min_xerror <- min(cp_table[, "xerror"])
```

```r
min_xerror_std <- cp_table[which.min(cp_table[, "xerror"]), "xstd"]

# Identify the largest cp where xerror < min_xerror + min_xerror_std
optimal_cp <- max(cp_table[cp_table[, "xerror"] < (min_xerror + min_xerror_std), "CP"])

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

# Plot the pruned tree using prp()
prp(pruned_tree, main = "Pruned Regression Tree for Carseats Sales")
```




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

# Calculate and print the MSE for the pruned tree
mse_pruned <- mean((pred_pruned_test - test_Carseats$Sales)^2)
print(paste("Test MSE for pruned tree:", mse_pruned))
```

```
## [1] "Test MSE for pruned tree: 4.36972991404452"
```

```r
# Print the optimal complexity parameter value
print(paste("Optimal complexity parameter (CP):", optimal_cp))
```

```
## [1] "Optimal complexity parameter (CP): 0.0295810985512512"
```

The largest CP value that met this criterion was 0.0295811. Using this value, we pruned the original tree, simplifying the model by reducing the number of splits while retaining predictive power. The pruned tree was then applied to predict sales on the test set, yielding a MSE of 4.36973, an slight improvement over the unpruned tree's MSE of 4.51347. This reduction in MSE indicates the pruned tree performs better on unseen data, reducing overfitting and improving generalization. So the pruning process effectively simplified the model while maintaining strong predictive accuracy.