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")
# localFileName <- paste0("mnist_first2500.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)  # For LDA function

# Step 1: Split the data into training and test sets
train_data <- mnist[1:1250, ]
```

```r
test_data <- mnist[1251:2500, ]

# Step 2: Filter the digits 1, 7, and 9 from the training set
selected_digits <- train_data[train_data$Digit %in% c(1, 7, 9), ]

# Step 3: Calculate the variance for each pixel (excluding the "Digit" column)
pixel_variance <- apply(selected_digits[, -1], 2, var)

# Step 4: Select the top 300 pixels with the highest variance
top_pixels_indices <- order(pixel_variance, decreasing = TRUE)[1:300]

# Step 5: Subset the training data to include only the top 300 pixels
# Adding 1 to the pixel indices to adjust for the 'Digit' column
train_selected <- selected_digits[, c(1, top_pixels_indices + 1)]
test_selected <- test_data[test_data$Digit %in% c(1, 7, 9), c(1, top_pixels_indices + 1)]

# Step 6: Fit the LDA model using the training set
lda_model <- lda(Digit ~ ., data = train_selected)
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
# Step 7: Make predictions on the test set
lda_predictions <- predict(lda_model, test_selected)

# Output the predicted class labels from the test set
print(lda_predictions$class)
```

```
##   [1] 7 1 9 1 9 9 1 9 7 9 9 1 9 9 1 7 7 9 7 9 7 9 1 7 7 1 7 9 7 7 7 7 7 9 7 1 9
##  [38] 9 1 1 1 9 1 7 7 9 7 1 9 9 1 9 1 9 9 7 7 1 7 1 9 7 9 1 1 7 1 7 1 9 9 1 9 1
##  [75] 7 1 7 1 1 9 9 9 7 9 1 7 7 7 1 1 1 1 7 1 9 9 7 7 9 7 9 1 9 7 7 9 1 1 1 7 9
## [112] 9 7 9 9 7 9 1 9 7 7 1 9 1 7 1 7 1 1 9 1 7 7 1 1 1 7 9 7 9 1 1 1 1 1 1 1 1
## [149] 7 9 9 7 9 7 1 9 7 9 9 7 7 7 9 9 7 7 1 1 7 9 7 1 7 1 1 9 7 1 7 9 1 7 7 1 1
## [186] 7 7 9 9 7 7 1 1 7 9 9 1 9 9 7 9 7 9 9 1 1 7 1 7 9 7 1 9 9 9 1 7 1 7 9 9 7
## [223] 7 9 9 7 7 1 7 7 9 1 9 7 1 1 1 1 7 7 7 9 1 1 9 1 1 7 7 7 9 9 1 1 7 1 7 9 7
## [260] 1 7 1 7 9 9 7 1 1 7 1 1 9 9 9 7 7 1 1 1 7 1 1 7 7 1 7 1 1 1 7 1 9 9 9 9 7
## [297] 1 1 9 7 7 7 9 1 1 9 7 7 9 7 7 7 1 1 1 7 1 1 7 9 1 7 1 1 7 9 7 9 1 7 1 7 1
## [334] 7 9 9 1 1 7 7 7 9 7 7 1 9 1 1 7 1 7 7 1 7 1 7 9 1 7 9 7 1 9 9 9 1 9 9 9 7
## [371] 7 7 1 1
## Levels: 1 7 9
```

```r
# Optional: Calculate the accuracy of the predictions
actual_labels <- test_selected$Digit
predicted_labels <- lda_predictions$class
accuracy <- sum(actual_labels == predicted_labels) / length(actual_labels)
cat("Test Accuracy:", accuracy, "%\n")
```

```
## Test Accuracy: 0.8903743 %
```

we fit a Linear Discriminant Analysis (LDA) model to a subset of the MNIST dataset, focusing on digits 1, 7, and 9. Using the first 1250 observations as the training set and the next 1250 as the test set, we addressed the zero variance issue, which can cause problems when inverting the covariance matrix. To mitigate this, we screened all 784 pixels and selected the top 300 pixels with the highest variance for both training and testing. After fitting the LDA model using the selected features, we made predictions on the test set and calculated the model's accuracy. The model achieved an accuracy of approximately 89.04%, indicating strong classification performance for these three digits using only the most informative pixels. This approach demonstrates how feature selection based on variance can enhance model performance and prevent numerical issues in high-dimensional datasets like MNIST.

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.

```r
library(MASS)  # For matrix operations
library(dplyr) # For data manipulation
```

```
##
## Attaching package: 'dplyr'
```

```
## The following object is masked from 'package:MASS':
##
##     select
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(caret) # For confusion matrix
```

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

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

```r
# Estimating class-specific means (mu_k)
mu_k <- train_selected %>%
  group_by(Digit) %>%
  summarise(across(starts_with("Pixel"), mean, .names = "mean_{.col}"))

# Calculating prior probabilities (pi_k)
n_k <- table(train_selected$Digit)
total_n <- sum(n_k)
pi_k <- n_k / total_n

# Estimating the pooled covariance matrix (Sigma)
sigma_list <- lapply(split(train_selected[, -1], train_selected$Digit), cov)
Sigma <- Reduce(`+`, Map(function(x, y) x * (y-1), sigma_list, n_k)) / (sum(n_k) - length(n_k))

# Inverting the pooled covariance matrix
Sigma_inv <- solve(Sigma)

# Defining the decision score function delta_k(x)
delta <- function(x, mu, Sigma_inv, pi) {
  t(mu) %*% Sigma_inv %*% x - 0.5 * t(mu) %*% Sigma_inv %*% mu + log(pi)
}

# Apply the delta function to each observation in the test set
test_scores <- apply(test_selected[, -1], 1, function(x) {
  sapply(1:nrow(mu_k), function(i) delta(x, as.numeric(mu_k[i, -1]), Sigma_inv, pi_k[names(pi_k) == as.
})

# Predicting the class with the highest score
predictions <- apply(test_scores, 2, which.max)
predicted_labels <- names(pi_k)[predictions]

# Generating the actual labels
actual_labels <- test_selected$Digit

# Calculating the accuracy
accuracy <- sum(predicted_labels == actual_labels) / length(actual_labels)
cat("Test Accuracy:", accuracy, "%\n")
```

```
## Test Accuracy: 0.8903743 %
```

```r
# Generating the confusion matrix
confusionMatrix <- confusionMatrix(as.factor(predicted_labels), as.factor(actual_labels), mode = "every
print(confusionMatrix$table)
```

```
##           Reference
## Prediction   1   7   9
##          1 125   3   2
##          7   2 111  21
##          9   1  12  97
```

We manually implemented the Linear Discriminant Analysis (LDA) model by estimating the necessary parameters: the class-specific means (μ_k), the pooled covariance matrix (Σ), and the prior probabilities (π_k)

for each digit class (1, 7, and 9). First, we computed the mean vector for each class using the training data, followed by calculating the pooled covariance matrix, which is the weighted sum of class-specific covariances. We then inverted this covariance matrix to use in the LDA decision rule. The decision function $\delta_k(x)$ for each class was derived and applied to the test data to compute decision scores. These scores were used to predict the class label by selecting the class with the highest score for each test observation. After generating the predicted class labels, we computed the accuracy, which turned out to be **89.04%**, identical to the performance obtained using the `lda()` function in part A. We also generated a confusion matrix to analyze how well the model predicted each class. The confusion matrix showed that the model performed well overall, though it struggled somewhat with distinguishing between digits 7 and 9, as evidenced by some misclassifications. This exercise illustrates how the underlying mathematics of LDA can be manually implemented to achieve results similar to built-in functions.

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.

```
library(MASS)  # For LDA function
library(caret) # For confusion matrix

# Step 1: Fit the LDA model using the training set
lda_model <- lda(Digit ~ ., data = train_selected)

# Step 2: Make predictions on the test set using the fitted LDA model
lda_predictions <- predict(lda_model, test_selected)

# Step 3: Output the predicted class labels from the test set
predicted_labels <- lda_predictions$class

# Generating the actual labels
actual_labels <- test_selected$Digit

# Step 4: Calculate the accuracy of the predictions
accuracy <- sum(predicted_labels == actual_labels) / length(actual_labels)
cat("Test Accuracy:", accuracy, "%\n")
```

```
## Test Accuracy: 0.8903743 %
```

```
# Step 5: Generate the confusion matrix
confusionMatrix <- confusionMatrix(as.factor(predicted_labels), as.factor(actual_labels), mode = "every
print(confusionMatrix$table)
```

```
##           Reference
## Prediction   1   7   9
##          1 125   3   2
##          7   2 111  21
##          9   1  12  97
```

```
# Step 6: Report accuracy and confusion matrix
print(paste("Accuracy of the LDA model:", accuracy))
```

```
## [1] "Accuracy of the LDA model: 0.890374331550802"
```

```r
print("Confusion Matrix:")
```

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

```r
print(confusionMatrix$table)
```

```
##           Reference
## Prediction   1   7   9
##          1 125   3   2
##          7   2 111  21
##          9   1  12  97
```

We used the `lda()` function from the `MASS` package to fit the LDA model on the same dataset, with the training data consisting of the top 300 pixels for digits 1, 7, and 9. After fitting the model, we used it to predict the class labels for the test set. The accuracy of the predictions was **89.04%**, which is identical to the accuracy obtained from our manual implementation in part b. We also generated a confusion matrix to compare the predicted labels with the actual ones.

The confusion matrix 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.

```r
library(MASS)  # For QDA function

# Attempt to fit the QDA model using the training set
qda_model <- try(qda(Digit ~ ., data = train_selected), silent = TRUE)

# Check if the model fitting was successful
if (inherits(qda_model, "try-error")) {
  cat("QDA model fitting failed with error:", qda_model)
} else {
  # Make predictions on the test set using the fitted QDA model
  qda_predictions <- predict(qda_model, test_selected)

  # Output the predicted class labels from the test set
  predicted_labels <- qda_predictions$class

  # Generating the actual labels
  actual_labels <- test_selected$Digit

  # Calculate the accuracy of the predictions
  accuracy <- sum(predicted_labels == actual_labels) / length(actual_labels)
  cat("Test Accuracy:", accuracy, "%\n")
}
```

```
## QDA model fitting failed with error: Error in qda.default(x, grouping, ...) :
##   some group is too small for 'qda'
```

We attempted to fit a Quadratic Discriminant Analysis (QDA) model using the `qda()` function from the `MASS` package. However, the model fitting failed with the error: "some group is too small for 'qda'." This error occurs because QDA requires the estimation of a separate covariance matrix for each class, and if the number of observations for any class is too small relative to the number of features, the covariance matrix becomes singular (non-invertible). This is a common issue when dealing with high-dimensional data, as we are using 300 features (pixels) in this case, but the number of observations in each class may not be sufficient to estimate a reliable covariance matrix.

QDA is more sensitive to the size of the training data for each class compared to LDA because it estimates a different covariance matrix for each class. When a class has too few observations relative to the number of features, the covariance matrix is singular, and QDA cannot proceed. This is likely why the model fitting failed.

To address this issue, we can take the following steps:

Solution 1: Regularization A common approach to handling singular covariance matrices in QDA is to add regularization by modifying the covariance matrix estimation. One method is to apply ridge regularization (shrinkage) by adding a small constant to the diagonal of the covariance matrix. Mathematically, we adjust the covariance matrix for each class Σ_k as follows:

$$\Sigma_k' = \Sigma_k + \lambda I$$

where λ is a small constant and I is the identity matrix. This adjustment ensures that the covariance matrix is invertible by increasing the diagonal elements, thus stabilizing the estimation. This is a form of regularized QDA, which is often more robust in high-dimensional settings.

Solution 2: Dimensionality Reduction Another way to address this issue is to reduce the dimensionality of the feature space before applying QDA. This can be achieved using techniques such as Principal Component Analysis (PCA), which projects the data onto a lower-dimensional space while retaining the most important variance. By reducing the number of features, we decrease the complexity of the covariance matrices that need to be estimated, making it more likely that QDA will work. Mathematically, if we reduce the dimensionality from 300 features to, say, 50 principal components, we are now estimating covariance matrices in a 50-dimensional space instead of 300, which reduces the risk of singularity.

Mathematical Justification for Regularization:

In QDA, the decision function $\delta_k(x)$ for classifying an observation x into class k is given by:

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

When Σ_k is singular or near-singular, it cannot be inverted, which leads to the failure of the QDA model. By applying regularization, we modify the covariance matrix as:

$$\Sigma_k' = \Sigma_k + \lambda I$$

This ensures that Σ_k' is invertible, preventing numerical issues during matrix inversion. Regularization controls the condition number of the covariance matrix and stabilizes the decision boundaries in high-dimensional space.

These solutions would help overcome the problem of small group sizes in QDA and allow the model to fit successfully.

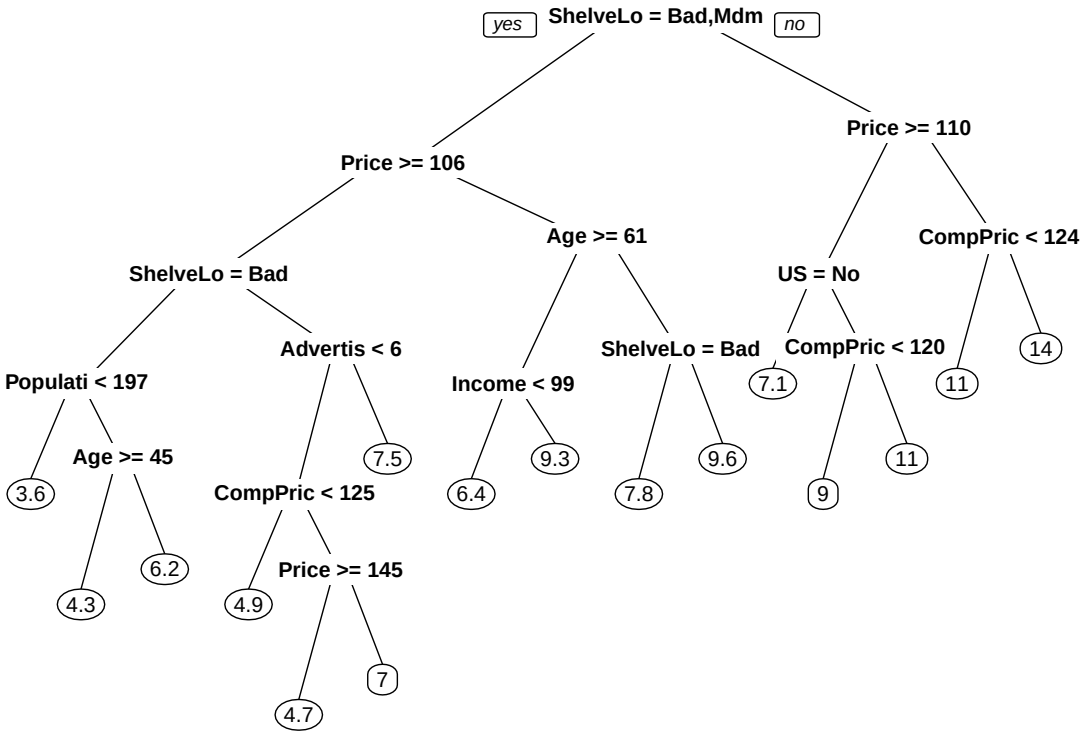
Question 2: Regression Trees (40 points)

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

```
# 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.

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

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

# Plot the tree
prp(tree_model)
```




```
# Predicting on the test set
pred_test <- predict(tree_model, newdata = test_Carseats)

# Calculate MSE
mse <- mean((pred_test - test_Carseats$Sales)^2)
mse
```

```
## [1] 4.51347
```

Based on the regression tree model, the observations with the highest predicted sales (14 units) occur when the shelf location is either not bad or medium (ShelveLo Bad, Mdm), the product price is high (greater than or equal to 110), and the competitor's price (CompPrice) is low (less than 124). This suggests that sales are maximized when the product is positioned as premium with a high price, and competitors are offering similar products at a relatively lower price. On the other hand, the observations with the lowest predicted sales (3.0 units) occur when the shelving location (ShelveLo) is poor, the population is small (less than 197), and the average customer age is 45 or older. These factors likely lead to decreased visibility in stores, a limited customer base, and an older demographic that may not be inclined to purchase the product. When applying the tree model to the test set, the Mean Squared Error (MSE) was calculated to be 4.51347. This value indicates that the model has reasonable predictive accuracy but could benefit from further tuning or alternative models.

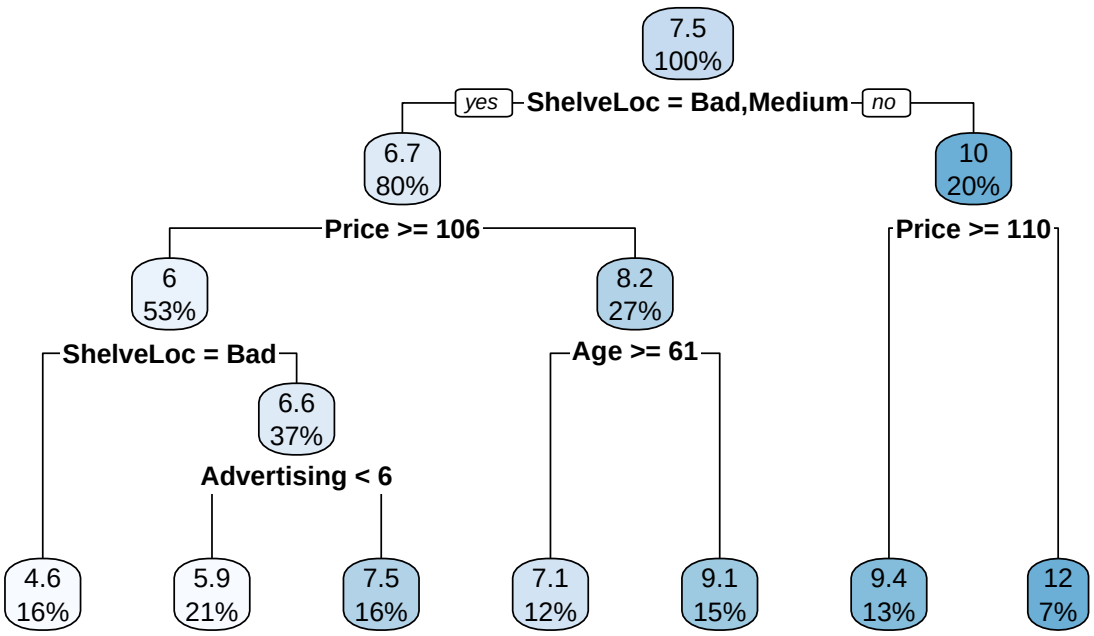
b. [20 pts] Set the seed to 7 at the beginning of the chunk and do this question in a single chunk so the seed doesn't get switched. Find the largest complexity parameter value of the tree you grew in part a) that will ensure that the cross-validation error < min(cross-validation error) + cross-validation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

```r
# Set 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
```

```r
# 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"])
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 rpart.plot
rpart.plot(pruned_tree, main = "Pruned Regression Tree for Carseats Sales")
```




```
# 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(mse_pruned)
```

```
## [1] 4.36973
```

```
# Print the optimal complexity parameter value
print(optimal_cp)
```

```
## [1] 0.0295811
```

After setting the seed for reproducibility, we fitted a regression tree and used cross-validation to determine the largest complexity parameter (CP) that minimized the cross-validation error while allowing for a standard deviation buffer. 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 used to predict sales on the test set, and the resulting Mean Squared Error (MSE)

was 4.36973, an improvement over the unpruned tree's MSE of 4.51347. This reduction in MSE suggests that the pruned tree performs better on unseen data by reducing overfitting and improving generalization. The pruning process successfully simplified the model while maintaining strong predictive accuracy.