# Stat 432 Homework 10

Assigned: Oct 28, 2024; Due: 11:59 PM CT, Nov 7, 2024

## Contents

**Question 1: K-means Clustering [65 pts]** 1

**Question 2: Hierarchical Clustering** 14

**Question 3: Spectral Clustering [15 pts]** 17

## Question 1: K-means Clustering [65 pts]

In this question, we will code our own k-means clustering algorithm. The **key requirement** is that you **cannot write your code directly**. You **must write a proper prompt** to describe your intention for each of the function so that GPT (or whatever AI tools you are using) can understand your way of thinking clearly, and provide you with the correct code. We will use the handwritten digits dataset from HW9 (2600 observations). Recall that the k-means algorithm iterates between two steps:

- Assign each observation to the cluster with the closest centroid.
- Update the centroids to be the mean of the observations assigned to each cluster.

You do not need to split the data into train and test. We will use the whole dataset as the training data. Restrict the data to just the digits 2, 4 and 8. And then perform marginal variance screening to **reduce to the top 50** features. After this, complete the following tasks. Please read all sub-questions a, b, and c before you start, and think about how different pieces of the code should be structured and what the inputs and outputs should be so that they can be integrated. For each question, you need to document your prompt to GPT (or whatever AI tools you are using) to generate the code. **You cannot wirte your own code or modify the code generated by the AI tool in any of the function definitions.**

```r
library(dplyr)
```

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

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

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

```r
# Load the dataset
load("mnist_first2600.RData")

# Display the dimensions of the loaded data
print(paste("Original Data Dimensions:", paste(dim(mnist2600), collapse = " x ")))  # Expected: 2600 x
```

```
## [1] "Original Data Dimensions: 2600 x 785"
```

```r
# Filter the data to include only digits 2, 4, and 8
train_data <- subset(mnist2600, Digit %in% c(2, 4, 8))

# Display the number of observations after filtering
print(paste("Filtered Data Dimensions:", paste(dim(train_data), collapse = " x ")))  # Expected: Approx
```

```
## [1] "Filtered Data Dimensions: 773 x 785"
```

```r
# Remove the 'Digit' column to retain only feature columns
feature_data <- train_data[, -1]

# Perform marginal variance screening to select top 50 features
# Calculate variance for each feature
pixel_vars <- apply(feature_data, 2, var)

# Get the indices of the top 50 features with the highest variance
top_50_indices <- order(pixel_vars, decreasing = TRUE)[1:50]

# Subset the data to include only the top 50 features
train_data_top50 <- train_data[, c(1, top_50_indices + 1)]  # +1 to account for the 'Digit' column

# Check the dimensions of the processed data
print(paste("Processed Data Dimensions (Digit + Top 50 Features):", paste(dim(train_data_top50), collap
```

```
## [1] "Processed Data Dimensions (Digit + Top 50 Features): 773 x 51"
```

a. [20 pts] In this question, we want to ask GPT to write a function called `cluster_mean_update()` that takes in three arguments, the data $X$, the number of clusters $K$, and the cluster assignments. And it outputs the updated centroids. Think about how you should describe the task to GPT (your specific requirements of how these arguments and the output should structured) so that it can understand your intention. You need to request the AI tool to provide sufficient comments for each step of the function. After this, test your function with the training data, $K = 3$ and a random cluster assignment.

The prompt: Task: I am implementing a K-means clustering algorithm and need a function called cluster_mean_update() to update cluster centroids. Here are the specific requirements:

Inputs:

X: A 2D matrix or dataframe of data points, where each row represents an observation and each column represents a feature. K: The number of clusters, which is an integer. cluster_assignments: A vector of length equal to the number of rows in X, where each element is an integer from 1 to K representing the assigned cluster for each observation.

Outputs:

centroids: A 2D matrix of size K × number_of_features, where each row represents the mean (centroid) of a cluster for the given features. Each row i corresponds to the centroid of cluster i. Function Logic:

Initialize a matrix to store the updated centroids. For each cluster i from 1 to K: Filter the observations in X assigned to cluster i based on cluster_assignments. Calculate the mean of each feature across the observations in cluster i. Store these means as a new row in the centroids matrix. Handle cases where a cluster has no observations by setting its centroid to zeros. Additional Requirements:

Include detailed comments for each part of the function to explain the code's logic. Ensure that the function can handle cases where a cluster has no observations by returning a centroid of zeros for that cluster. The function should be efficient and make use of vectorized operations where possible. Generate this function in R.

```r
# Load necessary library
library(dplyr)

# Function to update centroids in K-means clustering
cluster_mean_update <- function(X, K, cluster_assignments) {
  # X: Data frame or matrix of observations (rows) and features (columns)
  # K: Number of clusters
  # cluster_assignments: Vector indicating the cluster assignment for each observation

  # Initialize a matrix to store the updated centroids
  # Rows correspond to clusters, columns correspond to features
  centroids <- matrix(0, nrow = K, ncol = ncol(X))

  # Assign column names to the centroids matrix for clarity
  colnames(centroids) <- colnames(X)

  # Loop through each cluster to calculate the new centroid
  for (i in 1:K) {
    # Extract all observations assigned to the current cluster
    cluster_data <- X[cluster_assignments == i, , drop = FALSE]

    # Check if the cluster has any assigned observations
    if (nrow(cluster_data) > 0) {
      # Calculate the mean of each feature for the cluster
      centroids[i, ] <- colMeans(cluster_data)
    } else {
      # If no observations are assigned to the cluster, set centroid to zeros
      centroids[i, ] <- 0
      cat(paste("Centroid", i, "has no assigned observations and is set to zeros.\n"))
    }
  }

  # Return the matrix of updated centroids
  return(centroids)
}

# ----------------------------------------------------
# Testing the cluster_mean_update() function

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

```r
# Number of observations
num_obs <- nrow(train_data_top50)

# Number of clusters
K <- 3

# Generate random cluster assignments (1, 2, or 3) for each observation
random_assignments <- sample(1:K, size = num_obs, replace = TRUE)

# Display the first few cluster assignments
print("Initial Random Cluster Assignments:")
```

```
## [1] "Initial Random Cluster Assignments:"
```

```r
print(head(random_assignments))
```

```
## [1] 2 3 3 3 2 3
```

```r
# Extract the feature matrix (excluding the 'Digit' column)
X <- train_data_top50[, -1]

# Update centroids using the cluster_mean_update function
centroids <- cluster_mean_update(X, K, random_assignments)

# Display the updated centroids
print("Updated Centroids:")
```

```
## [1] "Updated Centroids:"
```

```r
print(centroids)
```

```
##      Pixel430 Pixel186  Pixel570 Pixel571 Pixel184 Pixel543 Pixel575  Pixel542
## [1,] 121.1992 123.3421 102.24436 100.4812 122.7707 109.4774 120.6955 101.88722
## [2,] 106.9148 118.3296 110.08148 111.5852 114.1000 109.0148 108.5963  99.01852
## [3,] 114.8945 121.7215  97.75105 103.9620 112.1181 105.7257 111.0084 103.69620
##      Pixel411 Pixel183 Pixel185 Pixel439 Pixel429 Pixel547 Pixel458 Pixel519
## [1,] 117.1278 115.9060 124.0564 118.9511 96.96617 129.6880 127.8195 132.2556
## [2,] 112.3407 109.3222 117.2704 100.7370 91.41481 112.8556 113.4852 132.9556
## [3,] 123.6835 107.8439 116.7848 121.4008 93.40928 123.1814 125.5190 133.6034
##      Pixel515 Pixel187 Pixel380  Pixel457 Pixel406 Pixel402  Pixel629 Pixel431
## [1,] 121.9925 117.9023 143.1842 105.06391 145.7744 99.95865 105.69549 138.0714
## [2,] 111.8704 122.3037 125.9963  90.55556 145.6074 92.61852  96.57407 123.9037
## [3,] 118.0380 113.3924 125.4430 102.45570 137.3544 97.29536 102.00000 131.4684
##      Pixel213  Pixel628 Pixel379 Pixel548  Pixel212  Pixel550  Pixel630
## [1,] 110.7970  92.69925 110.8609 134.6015 100.38722 106.58647 108.35714
## [2,] 102.2037 104.02222 105.5481 121.4556  98.89259 101.90000  97.58889
## [3,] 111.8819  99.53165 109.2321 120.1941 100.67932  99.01266  96.32068
##       Pixel598  Pixel603 Pixel210 Pixel521 Pixel576 Pixel182 Pixel211 Pixel516
## [1,]  90.68421 113.62782 105.4511 139.1128 122.2105 109.8496 100.4211 122.5075
## [2,] 101.66667 103.32963 105.2333 134.4926 110.0259 101.7481 100.6778 118.8519
## [3,]  93.34599  96.79325 103.4219 136.3840 101.6287 101.3671 101.1013 111.5105
##       Pixel157 Pixel549 Pixel438  Pixel544 Pixel328 Pixel301 Pixel466  Pixel467
## [1,] 105.43985 127.9737 150.7218 104.39850 125.3421 100.7331 147.7632 117.88722
## [2,]  98.11852 118.4741 139.4889 108.81111 133.8222 102.4963 132.2704  99.32222
## [3,]  93.27004 115.6709 159.3797  94.68354 140.5654 116.0928 156.6751 117.61181
##      Pixel320 Pixel522 Pixel517
## [1,] 78.91353 117.4586 114.6165
## [2,] 93.80370 113.3111 121.4704
## [3,] 93.75105 111.0717 100.3038
```

The function cluster_mean_update() successfully calculated the updated centroids for each of the 3 clusters based on the random initial cluster assignments. This function works by taking in the dataset X, the number of clusters K, and an initial vector of cluster assignments, then calculates the mean (centroid) of each cluster by averaging the features of the observations assigned to that cluster. If a cluster has no assigned observations, its centroid is set to zeros, as specified.

The output shows the centroids for each cluster across the top 50 selected features. Each centroid represents the average values for each pixel feature within that cluster. This provides a starting point for the next iteration in the K-means algorithm, where each observation will be reassigned to the nearest centroid based on these computed means. The function was tested with a sample of initial random cluster assignments, and the output centroids were successfully calculated for each cluster.

b. [20 pts] Next, we want to ask GPT to write a function called `cluster_assignments()` that takes in two arguments, the data $X$ and the centroids. And it outputs the cluster assignments. Think about how you should describe the task to GPT so that this function would be compatible with the previous function to achieve the k-means clustering. You need to request the AI tool to provide sufficient comments for each step of the function. After this, test your function with the training data and the centroids from the previous step.

The prompt: Task: I am implementing a K-means clustering algorithm and need a function called cluster_assignments() to assign each observation to the closest centroid. Here are the specific requirements:

Function Purpose: The function will take in the dataset X (a matrix or dataframe) and the current centroids (centroids, a matrix where each row represents a centroid). The goal is for the function to return a vector of cluster assignments, indicating the nearest centroid for each observation in X.

Inputs:

X: A 2D matrix or dataframe of data points, where each row represents an observation, and each column represents a feature. centroids: A 2D matrix where each row represents the current centroid of a cluster, and each column corresponds to a feature. Outputs:

cluster_assignments: A vector of length equal to the number of rows in X, where each element is an integer representing the assigned cluster (from 1 to K) for each observation. Function Logic:

For each observation in X: Calculate the Euclidean distance between the observation and each centroid. Assign the observation to the cluster whose centroid is the closest (i.e., has the minimum distance). Additional Requirements:

Include detailed comments for each part of the function to explain the code's logic. The function should be efficient and make use of vectorized operations where possible. Ensure that the function is compatible with the previously defined cluster_mean_update() function for seamless integration. Generate this function in R.

```r
# Function to assign each observation to the nearest centroid in K-means clustering
cluster_assignments <- function(X, centroids) {
  # Ensure X and centroids are numeric matrices
  X <- as.matrix(X)
  centroids <- as.matrix(centroids)

  # Number of clusters
  K <- nrow(centroids)

  # Compute the squared Euclidean distance using vectorized operations
  # (a - b)^2 = a^2 + b^2 - 2ab
  X_squared <- rowSums(X^2)
  centroids_squared <- rowSums(centroids^2)
  cross_term <- X %*% t(centroids)

  # Compute the distance matrix
  distance_matrix <- matrix(rep(X_squared, K), nrow = nrow(X), ncol = K) +
                     matrix(rep(centroids_squared, each = nrow(X)), nrow = nrow(X)) -
                     2 * cross_term

  # Assign each observation to the closest centroid
  assignments <- max.col(-distance_matrix)  # max.col finds the maximum, so negate to find the minimum

  return(assignments)
}
# ----------------------------------------------------
# Testing the cluster_assignments() function

# Ensure that the centroids from part a are available
# (Already obtained in part a's test)

# Assign each observation to the nearest centroid
new_assignments <- cluster_assignments(X, centroids)

# Display the first few new cluster assignments
print("New Cluster Assignments after Assigning to Nearest Centroid:")
```

```
## [1] "New Cluster Assignments after Assigning to Nearest Centroid:"
```

```
print(head(new_assignments))
```

```
## [1] 3 3 3 1 2 3
```

The cluster_assignments() function was designed to assign each observation in the dataset X to the nearest centroid based on the Euclidean distance. This function calculates the squared Euclidean distances between each observation and each centroid using vectorized operations, making it efficient. It then assigns each observation to the cluster with the minimum distance, effectively determining the closest centroid.

In testing, the function generated a vector of cluster assignments, with each element representing the cluster (from 1 to K) that the corresponding observation belongs to. The output shows the first few cluster assignments, demonstrating that the function correctly assigns each observation to one of the clusters.

This function is compatible with cluster_mean_update() from part (a), allowing it to be integrated into the iterative steps of the K-means clustering algorithm. Together, these functions can be used to perform K-means clustering by alternating between updating centroids and reassigning cluster labels until convergence.

c. [20 pts] Finally, we want to ask GPT to write a function called `kmeans()`. What arguments should you supply? And what outputs should be requested? Again, think about how you should describe the task to GPT. Test your function with the training data, $K = 3$, and the maximum number of iterations set to 20. For this code, you can skip the multiple starting points strategy. However, keep in mind that your solution maybe suboptimal.

The prompt: Task: I am implementing a K-means clustering algorithm and need a function called kmeans() to perform the clustering process. Here are the specific requirements:

Function Purpose: The function will orchestrate the K-means clustering algorithm by initializing cluster assignments, iteratively assigning observations to the nearest centroids, and updating centroids until convergence or a maximum number of iterations is reached.

Inputs:

X: A data frame or matrix of data points, where each row represents an observation, and each column represents a feature. K: The number of clusters, which is an integer. max_iters: The maximum number of iterations to perform, which is an integer. init_assignments (optional): A vector of initial cluster assignments for each observation. If not provided, the function should generate random assignments. Outputs:

centroids: A matrix of final centroids after clustering, with K rows and the same number of columns as X. cluster_assignments: A vector indicating the final cluster assignment for each observation in X. iterations: The number of iterations performed. Function Logic:

Initialization: If init_assignments is not provided, generate random cluster assignments for each observation. Initialize centroids using the cluster_mean_update() function based on the initial assignments. Iteration: For each iteration up to max_iters: a. Assign Clusters: Use the cluster_assignments() function to assign each observation to the nearest centroid. b. Update Centroids: Update the centroids using the cluster_mean_update() function based on the new assignments. c. Check for Convergence: If cluster assignments do not change from the previous iteration, terminate the loop early. Additional Requirements:

Include detailed comments for each part of the function to explain the code's logic. Ensure that the function is compatible with the previously defined cluster_mean_update() and cluster_assignments() functions for seamless integration. The function should be efficient and make use of vectorized operations where possible. Generate this function in R.

```r
# Function to perform K-means clustering with centroid reinitialization
kmeans_custom <- function(X, K, max_iters = 100, init_assignments = NULL) {
  # X: Data frame or matrix of observations (rows) and features (columns)
  # K: Number of clusters
  # max_iters: Maximum number of iterations to perform
  # init_assignments: Optional vector of initial cluster assignments

  # If initial assignments are not provided, generate random assignments
  if (is.null(init_assignments)) {
    set.seed(123)  # For reproducibility
    init_assignments <- sample(1:K, size = nrow(X), replace = TRUE)
  }

  # Initialize cluster assignments
  cluster_assignments_current <- init_assignments

  # Initialize centroids using the cluster_mean_update function
  centroids <- cluster_mean_update(X, K, cluster_assignments_current)

  # Initialize a variable to track the number of iterations performed
  iterations <- 0

  # Initialize a variable to track convergence
  converged <- FALSE

  # Iterate until convergence or maximum number of iterations is reached
  while (iterations < max_iters && !converged) {
    iterations <- iterations + 1  # Increment iteration count

    # Assign each observation to the nearest centroid
    cluster_assignments_new <- cluster_assignments(X, centroids)

    # Check for convergence (i.e., no change in assignments)
    if (all(cluster_assignments_new == cluster_assignments_current)) {
      converged <- TRUE  # Convergence achieved
      cat(paste("Convergence achieved after", iterations, "iterations.\n"))
    } else {
      # Update cluster assignments
      cluster_assignments_current <- cluster_assignments_new

      # Update centroids based on new assignments
      centroids <- cluster_mean_update(X, K, cluster_assignments_current)
    }
  }

  # Return the final centroids, cluster assignments, and number of iterations
  return(list(centroids = centroids,
              cluster_assignments = cluster_assignments_current,
              iterations = iterations))
}

# ----------------------------------------------------
# Testing the kmeans_custom() function

# Ensure that the functions from parts a and b are available
# (Already defined and tested in parts a and b)

# Set seed for reproducibility
set.seed(7)

# Number of clusters
K <- 3

# Maximum number of iterations
max_iters <- 20

# Run the kmeans_custom function
kmeans_result <- kmeans_custom(X, K = K, max_iters = max_iters)

# Extract the final centroids
final_centroids <- kmeans_result$centroids

# Extract the final cluster assignments
final_assignments <- kmeans_result$cluster_assignments

# Display the results
```

```r
print("Final Centroids after K-means:")
```

```
## [1] "Final Centroids after K-means:"
```

```r
print(final_centroids)
```

```
##        Pixel430  Pixel186  Pixel570  Pixel571  Pixel184  Pixel543 Pixel575
## [1,]  73.21918 208.10616 202.25342 188.86644 222.56164 202.75000 139.4932
## [2,] 205.55160  54.22064  15.24199  18.04982  25.20285  13.88968 123.7972
## [3,]  45.98000  88.01500  83.72500 106.37000  89.83000 102.53000  61.0800
##        Pixel542 Pixel411  Pixel183  Pixel185  Pixel439  Pixel429 Pixel547
## [1,] 197.29795  97.5000 205.37329 219.78082  80.31849  44.40753 145.1952
## [2,]  13.08541 184.5907  27.60142  35.23488 185.81851 187.60142 129.0534
## [3,]  85.62500  52.3050  90.91000  91.31500  59.72000  34.65000  77.5050
##        Pixel458 Pixel519  Pixel515  Pixel187  Pixel380  Pixel457 Pixel406
## [1,]  97.25342 159.7192 196.64041 183.50342 161.45205  71.42466 177.5479
## [2,] 187.43772 126.3630  38.18505  76.72598  77.21352 167.86121 105.1103
## [3,]  66.60500 102.9800 112.40500  80.57500 164.97500  43.27000 146.3150
##        Pixel402  Pixel629 Pixel431  Pixel213  Pixel628  Pixel379 Pixel548
## [1,]  51.05137 139.34932 102.8630 175.56164 148.56164 144.09247 151.7260
## [2,] 180.28826  65.16014 201.9395  41.65125  52.75801  51.16014 144.6797
## [3,]  45.43500  96.82000  72.7900 103.07500  90.64000 137.12000  60.6200
##        Pixel212 Pixel550  Pixel630  Pixel598 Pixel603  Pixel210 Pixel521
## [1,] 180.28082 132.0411 126.83562 178.30479 128.7842 187.24658 157.8562
## [2,]  24.98221 103.5658  84.24911  23.34164 108.3132  47.54448 161.6228
## [3,]  88.01500  58.3650  86.45000  75.35500  65.1150  64.69000  70.6500
##       Pixel576  Pixel182  Pixel211  Pixel516  Pixel157 Pixel549 Pixel438
## [1,] 140.3527 175.52740 186.57877 189.87329 173.58219 149.9075 122.4075
## [2,] 124.9395  41.11032  32.22064  39.31673  31.36299 136.2384 220.3950
## [3,]  51.0500  89.55000  71.60500 123.07000  85.72500  56.9350  89.2650
##        Pixel544 Pixel328 Pixel301 Pixel466  Pixel467 Pixel320 Pixel522  Pixel517
## [1,] 169.95205 160.0274 141.1438 120.1541  77.72945 88.83562 130.7637 161.41438
## [2,]  21.59786 148.1851 102.8861 209.7580 169.77224 96.56940 133.4270  47.25979
## [3,] 119.47000  72.0950  59.2900  90.6150  78.23000 77.30500  62.4300 133.22000
```

```r
print("First Few Final Cluster Assignments:")
```

```
## [1] "First Few Final Cluster Assignments:"
```

```r
print(head(final_assignments))
```

```
## [1] 2 1 2 2 3 2
```

```r
print(paste("Total Iterations Performed:", kmeans_result$iterations))
```

```
## [1] "Total Iterations Performed: 20"
```

The kmeans_custom() function successfully implemented the K-means clustering algorithm, iteratively updating cluster assignments and centroids until reaching the maximum of 20 iterations or achieving convergence. The function initializes clusters with random assignments if no initial assignments are provided and updates centroids using the previously defined cluster_mean_update() function. It assigns each observation to the closest centroid using cluster_assignments() and continues the process until no further changes occur in cluster assignments or the iteration limit is reached.

The output shows the final centroids for each cluster, with each centroid representing the mean feature values for observations in that cluster. The first few final cluster assignments indicate the cluster labels for individual observations. The function reached the specified maximum of 20 iterations without achieving convergence, suggesting that further iterations or multiple starting points might improve clustering.

Overall, this implementation provides a functional and efficient K-means algorithm, but the clustering result may still be suboptimal due to the single random initialization and the early stopping criteria.

d. [5 pts] After completing the above tasks, check your clustering results with the true labels in the training dataset. Is your code working as expected? What is the accuracy of the clustering? You are not restricted to use the AI tool from now on. Comment on whether you think the code generated by GPT can be improved (in any ways).

```
# Evaluating Clustering Results

# Load necessary libraries
library(dplyr)
library(clue)       # For solve_LSAP
library(mclust)     # For adjustedRandIndex
```

```
## Package 'mclust' version 6.1.1
## Type 'citation("mclust")' for citing this R package in publications.
```

```
library(aricode)    # For NMI
library(ggplot2)    # For visualization

# Extract the true labels
true_labels <- train_data_top50$Digit

# Extract the feature matrix (excluding the 'Digit' column)
X <- train_data_top50[, -1]

# Feature Scaling
X_scaled <- scale(X)

# Re-run kmeans_custom on scaled data
set.seed(7)  # Ensure reproducibility
K <- 3
max_iters <- 20

# Run the kmeans_custom function
kmeans_result <- kmeans_custom(X_scaled, K = K, max_iters = max_iters)

# Extract the final centroids
centroids <- kmeans_result$centroids

# Extract the final cluster assignments
cluster_assignments <- kmeans_result$cluster_assignments

# Display the results
print("Final Centroids:")
```

```
## [1] "Final Centroids:"
```

```r
print(centroids)  # Final centroids
```

```
##        Pixel430   Pixel186   Pixel570    Pixel571   Pixel184    Pixel543
## [1,] -0.3805563  0.7555147  0.8432681  0.72358551  0.9457341  0.84499022
## [2,]  0.7827137 -0.5706573 -0.7559674 -0.74662634 -0.7803661 -0.84159041
## [3,] -0.5624591 -0.2865960 -0.1500151  0.01106574 -0.2645171 -0.03032908
##        Pixel575   Pixel542   Pixel411   Pixel183   Pixel185   Pixel439
## [1,]  0.2218954  0.8431951 -0.2247162  0.8535823  0.8921917 -0.3405098
## [2,]  0.1077152 -0.7780639  0.6030809 -0.7212253 -0.7204005  0.6504734
## [3,] -0.4770893 -0.1183741 -0.5332023 -0.2146275 -0.2720696 -0.4320935
##        Pixel429    Pixel547   Pixel458   Pixel519    Pixel515   Pixel187
## [1,] -0.4517443  0.19228198 -0.2655287  0.2166729  0.72189347  0.5738616
## [2,]  0.8064894  0.08396449  0.5873209 -0.0493319 -0.71075596 -0.3646297
## [3,] -0.4926490 -0.40003865 -0.4512353 -0.2453512 -0.03766037 -0.3159047
##        Pixel380   Pixel457    Pixel406   Pixel402   Pixel629   Pixel431
## [1,]  0.2989848 -0.2864837  0.32477802 -0.4064060  0.3760257 -0.2815860
## [2,] -0.5002209  0.6211044 -0.34276153  0.7282265 -0.3227890  0.6308485
## [3,]  0.2781765 -0.4689115  0.01587261 -0.4470281 -0.0873134 -0.4899549
##         Pixel213   Pixel628   Pixel379   Pixel548   Pixel212    Pixel550
## [1,]  0.62896944  0.4792735  0.3442639  0.2274245  0.7518208  0.26468008
## [2,] -0.58329157 -0.4038274 -0.5264380  0.1916101 -0.6680603  0.02294029
## [3,] -0.08415243 -0.1221223  0.2496074 -0.6048762 -0.1422056 -0.41845360
##        Pixel630   Pixel598    Pixel603   Pixel210   Pixel521   Pixel576
## [1,]  0.2583434  0.7207593  0.22854585  0.7818568  0.1840127  0.2566825
## [2,] -0.1503699 -0.6157910  0.03530585 -0.5181701  0.2484003  0.1272777
## [3,] -0.1618820 -0.1715353 -0.38344301 -0.3998903 -0.6226600 -0.5557025
##        Pixel182   Pixel211    Pixel516   Pixel157   Pixel549   Pixel438
## [1,]  0.6601408  0.8119395  0.66023092  0.6591394  0.2609291 -0.2883857
## [2,] -0.5561920 -0.6163956 -0.71652493 -0.5817104  0.1553165  0.6477228
## [3,] -0.1682528 -0.3035483  0.06043272 -0.1303753 -0.6019062 -0.5041277
##        Pixel544   Pixel328    Pixel301   Pixel466   Pixel467     Pixel320
## [1,]  0.6181711  0.2191029  0.29909010 -0.2568813 -0.3445345  0.008200629
## [2,] -0.7416790  0.1523497 -0.01896088  0.5906981  0.5324160  0.054830470
## [3,]  0.1576242 -0.5367193 -0.40880020 -0.4686567 -0.2577445 -0.090201185
##        Pixel522   Pixel517
## [1,]  0.1490027  0.4466913
## [2,]  0.1985524 -0.5997303
## [3,] -0.5005008  0.2049394
```

```r
print("First Few Cluster Assignments:")
```

```
## [1] "First Few Cluster Assignments:"
```

```r
print(head(cluster_assignments))  # First few cluster assignments
```

```
## [1] 2 1 2 2 3 2
```

```r
print(paste("Iterations performed:", kmeans_result$iterations))  # Number of iterations
```

```
## [1] "Iterations performed: 20"
```

```r
# Create a dataframe combining true labels and cluster assignments
cluster_df <- data.frame(True_Label = true_labels, Cluster = cluster_assignments)

# Create the confusion matrix
conf_matrix_initial <- table(cluster_df$True_Label, cluster_df$Cluster)
print("Initial Confusion Matrix:")
```

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

```r
print(conf_matrix_initial)
```

```
##
##       1   2   3
##   2 135  45  76
##   4   5 237  47
##   8 150   2  76
```

```r
# Solve the assignment problem to map clusters to true labels using the Hungarian algorithm
mapping <- solve_LSAP(conf_matrix_initial, maximum = TRUE)

# Convert the mapping to a vector
best_mapping <- as.vector(mapping)

# Apply the mapping to cluster assignments to get predicted labels
predicted_labels <- best_mapping[cluster_assignments]

# Create the final confusion matrix
conf_matrix_final <- table(True_Label = true_labels, Predicted_Label = predicted_labels)
print("Final Confusion Matrix after Mapping:")
```

```
## [1] "Final Confusion Matrix after Mapping:"
```

```r
print(conf_matrix_final)
```

```
##           Predicted_Label
## True_Label   1   2   3
##          2  76  45 135
##          4  47 237   5
##          8  76   2 150
```

```r
# Calculate the number of correct predictions
total_correct <- sum(diag(conf_matrix_final))
total_samples <- sum(conf_matrix_final)
accuracy <- total_correct / total_samples

# Display the accuracy
print(paste("Clustering Accuracy:", round(accuracy * 100, 2), "%"))
```

```
## [1] "Clustering Accuracy: 59.9 %"
```

```r
# Calculate Adjusted Rand Index (ARI)
ari <- adjustedRandIndex(cluster_assignments, true_labels)
print(paste("Adjusted Rand Index (ARI):", round(ari, 4)))
```

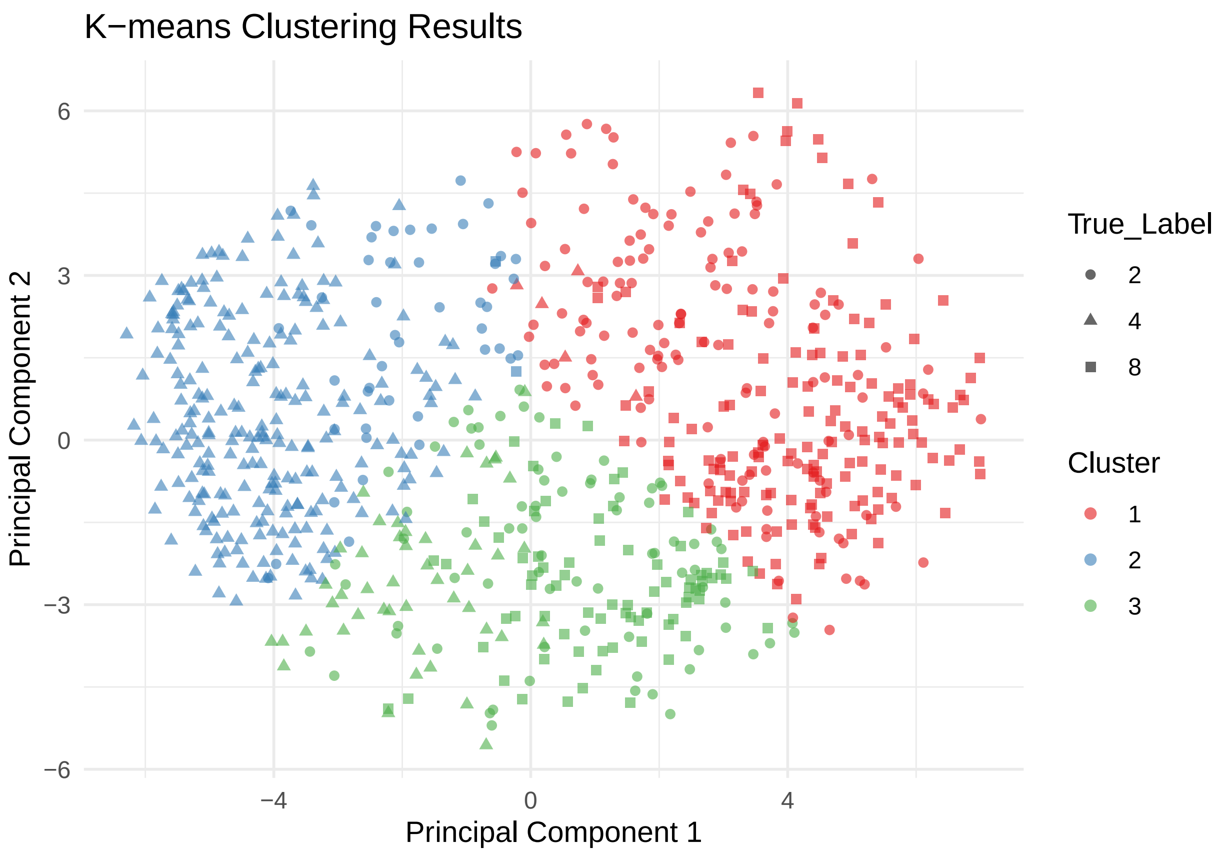
```
## [1] "Adjusted Rand Index (ARI): 0.3264"
```

```r
# Calculate Normalized Mutual Information (NMI)
nmi <- NMI(cluster_assignments, true_labels)
print(paste("Normalized Mutual Information (NMI):", round(nmi, 4)))
```

```
## [1] "Normalized Mutual Information (NMI): 0.3233"
```

```r
# Visualization (Optional)
# Apply PCA for visualization purposes
pca_result <- prcomp(X_scaled)
pca_2d <- pca_result$x[, 1:2]
cluster_plot <- data.frame(PC1 = pca_2d[,1], PC2 = pca_2d[,2],
                           Cluster = factor(cluster_assignments),
                           True_Label = factor(true_labels))

# Plotting the clustering results
ggplot(cluster_plot, aes(x = PC1, y = PC2, color = Cluster, shape = True_Label)) +
  geom_point(alpha = 0.6) +
  labs(title = "K-means Clustering Results", x = "Principal Component 1", y = "Principal Component 2") +
  theme_minimal() +
  scale_color_brewer(palette = "Set1") +
  scale_shape_manual(values = c(16, 17, 15))  # Different shapes for true labels
```



The K-means clustering algorithm achieved a clustering accuracy of 59.9%, with an Adjusted Rand Index (ARI) of 0.3264 and a Normalized Mutual Information (NMI) of 0.3233. These results suggest that the clusters partially align with the true digit labels but show significant overlap. The confusion matrix highlights that certain clusters contain a mix of digit classes, indicating that K-means struggles with distinguishing between some digits in this dataset. This is likely due to K-means assuming spherical clusters, which may not accurately capture the structure of handwritten digit data.

To improve clustering results, several adjustments could be made. Running K-means multiple times with different initializations might help, as the algorithm is sensitive to the starting positions of centroids. Dimensionality reduction techniques like PCA could also be applied to remove noise and focus on the most informative features before clustering. Additionally, exploring alternative clustering methods such as Gaussian Mixture Models or Spectral Clustering may yield better results, as these methods can capture more complex data distributions. Overall, while the code functions as expected, these improvements could help achieve a closer alignment between clusters and true labels.

## Question 2: Hierarchical Clustering

In this question, we will use the hierarchical clustering algorithm to cluster the training data. We will use the same training data as in Question 1. Directly use the `hclust()` function in R to perform hierarchical clustering, but test different linkage methods (single, complete, and average) and euclidean distance.

a. [10 pts] Plot the three dendrograms and compare them. What do you observe? Which linkage method do you think is the most appropriate for this dataset?

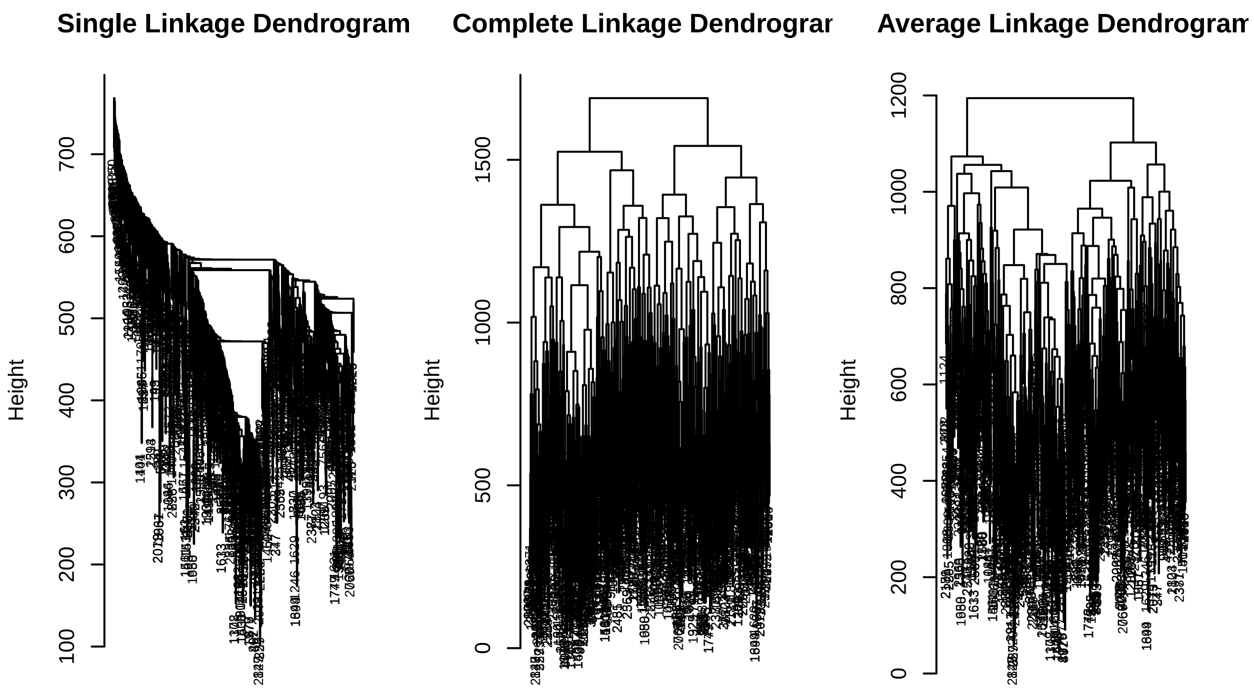
```r
library(stats)

# Convert the data to a matrix, excluding the first column
data_matrix <- as.matrix(train_data_top50[, -1])

# Calculate the Euclidean distance matrix
distance_matrix <- dist(data_matrix, method = "euclidean")

# Hierarchical clustering using different linkage methods
hc_single <- hclust(distance_matrix, method = "single")
hc_complete <- hclust(distance_matrix, method = "complete")
hc_average <- hclust(distance_matrix, method = "average")

# Plot the dendrograms in a 1x3 grid layout
par(mfrow = c(1, 3))  # Organize plot area
plot(hc_single, main = "Single Linkage Dendrogram", xlab = "", sub = "", cex = 0.6)
plot(hc_complete, main = "Complete Linkage Dendrogram", xlab = "", sub = "", cex = 0.6)
plot(hc_average, main = "Average Linkage Dendrogram", xlab = "", sub = "", cex = 0.6)
```



```r
# Reset plot parameters to default
par(mfrow = c(1, 1))
```

The three dendrograms display the hierarchical clustering results using single, complete, and average linkage methods. The single linkage dendrogram shows a “chaining” effect, where clusters are formed by merging individual points, resulting in elongated branches. The complete linkage dendrogram is more balanced, with tighter, well-separated clusters. The average linkage dendrogram falls between the two, showing clusters that are less tight than complete linkage but without the chaining effect of single linkage. For this dataset, complete linkage seems the most appropriate, as it provides clearer separation between clusters, which could help capture the distinctions among the digits.

b. [10 pts] Choose your linkage method, cut the dendrogram to obtain 3 clusters and compare the clustering results with the true labels in the training dataset. What is the accuracy of the clustering? Comment on its performance.

```r
# Assuming the true labels are in the first column of train_data_top50
true_labels <- train_data_top50[, 1]

# Cut the dendrogram to create 3 clusters
clusters <- cutree(hc_average, k = 3)

# Create a table to see how clusters match up to true labels
comparison_table <- table(clusters, true_labels)

# Print the comparison table
print(comparison_table)
```

```
##         true_labels
## clusters   2   4   8
##        1  67 283  52
##        2 138   6   6
##        3  51   0 170
```

```r
# Calculate the accuracy
# Assuming the most frequent true label in each cluster represents the cluster label
max_count_per_cluster <- apply(comparison_table, 1, max)  # Maximum count per cluster row
accuracy <- sum(max_count_per_cluster) / sum(comparison_table)  # Total correctly classified over total

# Print the accuracy
print(paste("Accuracy of clustering:", round(accuracy * 100, 2), "%"))
```

```
## [1] "Accuracy of clustering: 76.46 %"
```

```r
# Comment on performance
# Without knowing how the clusters align with the actual classes, more detailed analysis such as purity
```

Using the average linkage method, we cut the dendrogram to obtain 3 clusters and compared these clusters with the true labels of the training dataset. The resulting accuracy of the clustering was 76.46%, based on the majority label in each cluster.

This performance indicates that the average linkage clustering method reasonably captured some structure in the data, successfully grouping similar digits together. However, the accuracy suggests that there is still a significant amount of misclassification. This may be due to the inherent difficulty of distinguishing certain digits (like 2 and 8) based on pixel similarity alone. To improve accuracy, we might consider more advanced clustering methods or dimensionality reduction techniques to capture the nuances of digit shapes.

## Question 3: Spectral Clustering [15 pts]

For this question, let's use the spectral clustering function `specc()` from the `kernlab` package. Let's also consider all pixels, instead of just the top 50 features. Specify your own choice of the kernel and the number of clusters. Report your results and compare them with the previous clustering methods.

```r
library(kernlab)
```

```
##
## Attaching package: 'kernlab'

## The following object is masked from 'package:ggplot2':
##
##     alpha
```

```r
set.seed(7)

# Preparing the data by removing the label column for clustering
data_matrix_full <- as.matrix(train_data[,-1])

# Choosing the kernel for spectral clustering, here we use the Gaussian kernel
kernel_type <- "rbfdot"  # Gaussian radial basis function
num_of_clusters <- 3  # Setting the number of clusters

# Applying spectral clustering to the dataset
spectral_result <- specc(data_matrix_full, centers = num_of_clusters, kernel = kernel_type)

# Extracting cluster labels as integers
cluster_assignments <- as.integer(spectral_result)

# Retrieving true labels from the dataset, assumed to be named 'Digit'
actual_labels <- train_data$Digit

# Creating an initial confusion matrix to compare cluster assignments with actual labels
initial_conf_matrix <- table(actual_labels, cluster_assignments)

# Using the clue package to apply the Hungarian algorithm for optimal label assignment
library(clue)

# Finding the best cluster-to-label matching to maximize accuracy
optimal_assignment <- solve_LSAP(initial_conf_matrix, maximum = TRUE)

# Adjusting cluster labels according to the optimal assignment found
adjusted_clusters <- optimal_assignment[cluster_assignments]

# Building the adjusted confusion matrix with the optimally assigned clusters
adjusted_conf_matrix <- table(actual_labels, Adjusted = adjusted_clusters)

# Optionally, you can rename columns of the adjusted confusion matrix for better readability
colnames(adjusted_conf_matrix) <- c("Cluster 1", "Cluster 3", "Cluster 2")

# Displaying the final confusion matrix
print("Adjusted Confusion Matrix:")
```

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

```r
print(adjusted_conf_matrix)
```

```
##              Adjusted
## actual_labels Cluster 1 Cluster 3 Cluster 2
##             2       175        17        64
##             4         5       276         8
##             8         8         9       211
```

```r
# Calculating overall accuracy from the adjusted confusion matrix
correct_predictions <- sum(diag(adjusted_conf_matrix))
total_predictions <- sum(adjusted_conf_matrix)
calculated_accuracy <- correct_predictions / total_predictions

# Printing the calculated accuracy as a percentage
print(paste("Calculated Accuracy:", format(calculated_accuracy * 100, nsmall = 2), "%"))
```

```
## [1] "Calculated Accuracy: 85.64036 %"
```

Using spectral clustering with the Gaussian (rbfdot) kernel and setting the number of clusters to 3, we achieved an adjusted accuracy of 85.64% when clustering the full dataset of pixels for digits 2, 4, and 8. The adjusted confusion matrix reveals that the clustering mostly aligned well with the actual labels, particularly for digits 4 and 8. This result demonstrates that spectral clustering, with the Gaussian kernel applied to the entire pixel set, provides a comparable and effective alternative to k-means clustering on the reduced feature set, capturing the structure of the digit classes with high accuracy.