

# 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	7
Question 3: Spectral Clustering [15 pts]	9

## 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 write your own code or modify the code generated by the AI tool in any of the function definitions.**

```
# inputs to download file
fileLocation <- "https://pjreddie.com/media/files/mnist_train.csv"
numRowsToDownload <- 2600
localFileName <- paste0("mnist_first", numRowsToDownload, ".RData")

# download the data and add column names
mnist2600 <- read.csv(fileLocation, nrow = numRowsToDownload)
numColsMnist <- dim(mnist2600)[2]
colnames(mnist2600) <- 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(mnist2600, file = localFileName)
```

```
# you can load the data with the following code
#load(file = localFileName)
dim(mnist2600)
```

```
## [1] 2600 785
```

```
# Subset data to include only digits 2, 4, and 8
mnist <- mnist2600[mnist2600$Digit %in% c(2, 4, 8), ]
# Calculate variance for each pixel column
pixel_vars <- apply(mnist[, -1], 2, var)
# Select top 250 pixel columns with the highest variance
top_pixels <- names(sort(pixel_vars, decreasing = TRUE))[1:50]
# Subset training and test sets to only include these top 250 pixels
num_top_50 <- mnist[, c("Digit", top_pixels)]
```

- 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 be 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. ##### Prompt Write a function called `cluster_mean_update()` that takes in three arguments, the data  $X$ , the number of clusters  $K$ , and the cluster assignments. Outputs the updated centroids.

```
cluster_mean_update <- function(X, K, cluster_assignments) {
  # Initialize a matrix to store the updated centroids
  centroids <- matrix(0, nrow = K, ncol = ncol(X))

  # Loop through each cluster to calculate the mean for each one
  for (k in 1:K) {
    # Subset the data for the current cluster
    cluster_data <- X[cluster_assignments == k, ]

    # Calculate the mean of each column (i.e., each pixel) for the current cluster
    centroids[k, ] <- colMeans(cluster_data)
  }

  return(centroids)
}
```

```
set.seed(123) # Set seed for reproducibility
K <- 3
# Generate random cluster assignments
random_cluster_assignments <- sample(1:K, nrow(num_top_50), replace = TRUE)

# Get the updated centroids
centroids <- cluster_mean_update(num_top_50[, -1], K, random_cluster_assignments)
print(centroids)
```

## Testing the cluster\_mean\_update Function

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]
## [1,] 114.0315 119.0906 108.41732 106.5945 129.6260 113.0669 117.6535 106.92126
## [2,] 112.7598 118.4921  93.87008 102.9961 112.7559 103.4606 108.1181  93.43701
## [3,] 115.9660 125.5094 108.32075 106.6377 107.4377 107.9774 114.6755 103.85660
##           [,9]      [,10]     [,11]     [,12]     [,13]     [,14]     [,15]     [,16]
## [1,] 110.2165 123.1772 127.9213 107.3307 89.70079 126.6299 118.1890 135.1732
## [2,] 122.9724 108.2835 114.1535 117.3346 96.20079 115.2323 120.2717 128.7913
## [3,] 119.1358 102.3245 116.4264 115.2717 95.82642 123.5057 127.6226 134.6981
##           [,17]     [,18]     [,19]     [,20]     [,21]     [,22]     [,23]     [,24]
## [1,] 122.6417 113.7756 132.5394  98.58661 148.5197 101.11024 104.08661 133.5512
## [2,] 118.3425 115.0669 128.9173  97.07480 139.5669  94.74016 103.37795 125.2520
## [3,] 111.0189 125.0264 133.6830 101.81509 141.3925  93.99623  96.86038 134.3509
##           [,25]     [,26]     [,27]     [,28]     [,29]     [,30]     [,31]
## [1,] 109.3819 102.71260 110.1496 121.2480 109.00000  93.20866 107.16929
## [2,] 106.6417  98.40945 107.7244 128.2008  96.87795 113.20472  96.73622
## [3,] 108.3509  95.27547 107.6792 127.2566  94.23396 101.51698  98.89811
##           [,32]     [,33]     [,34]     [,35]     [,36]     [,37]     [,38]
## [1,]  97.31890 110.79921 120.42126 130.2244 109.5276 114.37795 112.88583
## [2,]  91.92520  99.56299 101.03150 145.1654 110.4449 103.33071  98.14173
## [3,]  96.70566 104.27170  93.30189 134.6830 114.8226  95.91698  91.52830
##           [,39]     [,40]     [,41]     [,42]     [,43]     [,44]     [,45]     [,46]
## [1,] 123.5433 106.13780 110.6181 147.0591 103.59449 130.7874 107.35039 145.7717
## [2,] 115.2008  97.06693 130.1575 148.7244  98.70866 141.5787 119.51969 144.8346
## [3,] 114.9585  94.45283 121.8340 152.4453 106.43019 126.8151  91.91698 144.6642
##           [,47]     [,48]     [,49]     [,50]
## [1,] 103.4724 95.87008 105.4646 114.1063
## [2,] 117.9331 85.37795 125.3346 110.2756
## [3,] 112.4981 84.90566 111.4679 113.4491
```

- 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. `####` Prompt Write a function called `cluster_assignments()` that takes in two arguments, the data  $X$  and the centroids. Outputs the cluster assignments.

```
cluster_assignments <- function(X, centroids) {
  # Initialize a vector to store the cluster assignments
  assignments <- integer(nrow(X))

  # Loop through each data point
  for (i in 1:nrow(X)) {
    # Calculate the Euclidean distance between the data point and each centroid
    distances <- apply(centroids, 1, function(centroid) sum((X[i, ] - centroid)^2))

    # Assign the data point to the closest centroid
    assignments[i] <- which.min(distances)
  }
}
```

```

    return(assignments)
}

```

```

# Use the centroids from the previous test
assignments <- cluster_assignments(num_top_50[, -1], centroids)
print(assignments)

```

### Testing the cluster\_assignments Function

```

## [1] 2 1 2 2 1 2 1 3 1 1 1 1 3 1 2 2 3 2 1 1 1 2 3 1 1 2 2 1 2 3 1 2 2 1 3 3 3
## [38] 1 1 1 1 2 1 3 2 3 1 2 1 1 2 3 1 2 1 2 1 3 2 1 2 1 2 2 1 1 1 2 2 3 1 3 3 2
## [75] 2 2 3 2 2 1 2 3 3 1 2 2 1 2 2 1 3 3 2 3 1 2 1 3 3 1 3 1 1 3 3 3 1 3 3 2 1
## [112] 1 1 3 3 1 1 2 3 3 3 3 3 1 3 2 2 3 3 3 1 3 1 2 2 1 3 3 3 2 1 1 1 1 1 1 1 1
## [149] 2 1 1 3 1 2 3 1 2 1 1 2 3 2 1 1 1 2 3 2 1 3 1 1 1 3 3 1 2 3 3 1 1 1 3 3 2
## [186] 1 3 3 3 3 2 2 2 2 3 1 1 2 1 2 3 2 2 1 3 1 1 1 3 2 2 1 1 1 2 1 1 1 1 2 1 1
## [223] 1 1 1 1 1 1 2 1 2 1 1 2 2 3 2 1 1 1 1 1 3 1 1 3 3 2 2 2 1 2 2 2 3 1 3 1 1
## [260] 2 1 2 2 1 1 3 1 2 2 1 1 2 1 1 1 3 3 3 2 1 3 1 1 2 1 1 2 2 1 2 2 2 2 3 3 2
## [297] 3 2 1 1 1 1 3 2 3 1 1 3 2 2 1 1 1 1 2 2 1 3 1 2 3 3 2 1 1 2 3 2 1 3 1 3 1
## [334] 1 2 3 2 2 3 3 2 3 1 2 3 1 2 1 1 3 2 1 3 1 2 1 1 2 1 2 1 3 2 3 1 1 3 1 2 1
## [371] 3 1 1 2 3 1 3 1 1 2 2 2 1 3 1 1 1 1 1 3 1 1 3 1 1 1 3 1 1 3 2 2 3 1 1 1 3
## [408] 1 1 1 1 1 2 3 1 1 2 1 1 1 2 2 2 2 3 1 1 2 3 1 3 1 3 3 1 2 1 2 1 2 3 3 1 3
## [445] 1 3 3 3 1 1 3 2 3 1 1 1 3 1 2 3 1 2 2 2 2 1 1 1 1 2 1 3 3 1 1 2 2 3 2 3 1
## [482] 2 1 3 2 1 1 2 3 2 3 3 3 2 2 3 3 3 1 1 1 3 3 1 1 2 2 1 1 1 3 1 3 1 1 3 3 2
## [519] 3 2 3 1 2 3 1 1 1 1 1 1 2 1 3 2 3 1 1 2 2 1 2 2 3 1 3 2 3 2 1 1 1 3 2 3 1
## [556] 3 3 1 1 3 1 1 2 3 3 2 2 1 1 1 2 1 1 1 3 3 2 1 2 2 3 3 3 1 1 1 2 1 1 1 1 1
## [593] 1 1 1 2 3 3 3 3 1 2 2 2 2 1 1 1 1 1 3 1 2 2 1 1 1 1 1 2 1 1 2 2 2 1 1 1 2
## [630] 1 1 1 1 2 2 1 1 2 1 2 2 1 2 2 2 2 1 2 3 3 1 1 1 3 2 3 3 1 3 3 2 1 2 1 2 3
## [667] 2 1 3 1 1 3 3 1 1 3 1 1 2 3 3 3 1 3 3 1 3 1 2 1 3 1 2 2 1 3 3 2 1 1 3 2 3
## [704] 1 1 2 3 1 1 3 3 3 1 2 1 2 3 2 2 1 2 3 2 3 3 3 3 1 1 1 1 1 1 1 2 2 1 3 1 2
## [741] 3 1 1 1 1 1 2 2 2 1 1 1 1 1 1 3 3 1 3 3 2 1 3 2 1 3 1 1 1 3 2 3 1

```

- 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. *#### Prompt Write a function called `kmeans()` that takes in three arguments, the data  $X$ , the number of clusters  $K$ , and the maximum number of iterations. Outputs the cluster assignments and the final centroids.*

```

kmeans <- function(X, K, max_iters = 100) {
  # Step 1: Randomly initialize K centroids by selecting K random rows from X
  set.seed(123) # Set seed for reproducibility
  initial_centroids <- X[sample(1:nrow(X), K), ]

  # Initialize cluster assignments
  cluster_assignments <- rep(0, nrow(X))

  for (iter in 1:max_iters) {

```

```

# Step 2: Assign points to the nearest centroid
cluster_assignments <- cluster_assignments(X, initial_centroids)

# Step 3: Update centroids based on current assignments
new_centroids <- cluster_mean_update(X, K, cluster_assignments)

# Check for convergence (if centroids do not change)
if (all(initial_centroids == new_centroids)) {
  cat("Converged in", iter, "iterations.\n")
  break
}

# Update centroids for the next iteration
initial_centroids <- new_centroids
}

# Return the final cluster assignments and centroids
list(cluster_assignments = cluster_assignments, centroids = initial_centroids)
}

```

```

# Run k-means clustering on num_top_50 with K = 3 and max_iters = 20
kmeans_result <- kmeans(num_top_50[, -1], K = 3, max_iters = 20)

```

## Testing the kmeans Function

```
## Converged in 13 iterations.
```

```

# Print the final cluster assignments and centroids
print(kmeans_result$cluster_assignments)

```

```

## [1] 2 1 2 2 3 2 1 3 1 1 1 3 3 1 2 2 2 2 3 1 1 2 2 1 1 3 1 2 2 1 2 2 3
## [38] 3 1 2 1 3 1 2 2 2 1 2 3 1 1 1 1 2 1 3 1 1 2 1 2 1 1 2 3 1 1 2 2 3 1 3 2 2
## [75] 2 2 2 2 2 1 2 3 1 1 2 2 1 1 2 1 2 2 2 2 1 2 1 3 3 1 2 1 1 2 2 2 1 2 2 2 1
## [112] 1 3 3 2 1 1 2 3 1 3 2 1 1 3 2 2 3 2 3 1 3 3 2 2 1 3 3 2 2 1 1 1 3 1 1 1 1
## [149] 2 3 1 2 1 2 2 1 3 1 1 2 1 2 1 1 3 3 2 2 3 2 1 3 3 3 2 1 2 3 3 3 1 3 3 3 2
## [186] 1 3 2 3 1 3 2 2 2 1 1 3 2 1 3 3 2 2 3 2 3 1 1 1 2 2 3 3 3 2 1 1 3 1 2 2 1
## [223] 3 1 1 1 1 1 2 1 3 1 3 3 2 2 2 3 1 1 1 1 3 1 1 2 2 2 2 2 3 2 2 2 3 1 2 1 1
## [260] 2 3 2 2 1 1 2 2 2 2 1 3 2 1 1 1 2 2 3 2 3 2 1 1 2 1 1 2 2 1 2 2 3 2 2 3 1
## [297] 2 3 3 3 1 1 1 2 2 3 1 1 2 3 3 3 3 3 3 2 3 3 3 2 1 2 2 1 1 2 3 2 1 2 1 3 3
## [334] 3 2 2 2 2 3 3 2 2 3 2 2 1 2 3 1 2 2 1 2 1 2 2 1 2 1 2 1 2 2 3 1 1 3 1 2 1
## [371] 2 3 1 2 2 1 1 1 3 2 2 2 1 2 1 3 1 1 1 2 1 1 3 1 1 3 2 2 3 2 3 2 2 3 1 1 3
## [408] 1 1 1 1 1 2 2 1 1 2 3 3 1 2 3 2 2 2 1 1 2 2 1 3 1 3 2 1 2 3 2 1 2 3 2 3 2
## [445] 3 2 2 2 1 1 2 2 2 1 3 1 2 1 2 2 3 2 2 2 2 3 3 1 1 2 1 2 3 1 1 3 2 1 2 2 1
## [482] 2 1 3 2 1 3 2 2 3 2 2 2 2 2 2 2 2 1 1 3 3 1 1 1 2 3 1 1 1 2 3 3 1 1 2 2 2
## [519] 2 1 2 1 2 3 1 1 1 3 1 1 2 3 2 2 2 3 1 3 2 3 2 2 3 1 2 1 1 2 1 1 1 2 2 2 1
## [556] 3 3 3 1 2 3 1 2 2 3 2 2 1 1 2 2 3 1 3 2 1 2 3 2 2 3 3 3 1 1 1 2 1 1 1 3 1
## [593] 1 3 1 3 2 2 1 2 1 2 3 2 1 1 1 1 1 3 2 1 2 2 1 1 1 3 1 2 1 1 2 2 2 1 3 1 2
## [630] 1 3 3 1 2 2 2 3 1 1 2 2 3 2 1 2 2 1 2 2 2 3 3 1 2 3 2 2 1 2 2 3 1 2 3 2 2
## [667] 2 1 2 1 1 2 2 2 1 1 3 1 2 2 2 3 3 2 2 1 2 3 2 1 2 1 2 2 3 3 3 1 1 1 2 2 3
## [704] 1 1 3 3 3 1 2 2 3 3 1 1 3 2 2 2 1 2 1 2 3 2 3 2 2 3 2 3 3 1 2 2 2 1 2 1 2
## [741] 2 3 1 1 3 1 2 2 1 1 1 1 1 1 2 2 1 2 3 2 3 3 2 1 2 1 2 1 2 1 2 1 2 1

```

```
print(kmeans_result$centroids)
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,]  73.93141 210.82671 204.42238 189.33213 224.79783 204.74368 143.08664
## [2,] 198.35032  56.51592  21.61465  26.59873  28.01274  22.46815 113.84076
## [3,]  30.63187  95.93956  91.61538 113.72527 104.23626 109.02747  67.87912
##           [,8]      [,9]      [,10]     [,11]     [,12]     [,13]     [,14]
## [1,] 200.42960 100.08664 205.75451 223.06137  83.24549  45.24188 147.61372
## [2,]  19.78344 179.58599  29.06688  39.07962 185.40446 181.46815 122.55732
## [3,]  91.65934  36.74176 108.71429 100.44505  34.81319  17.03297  81.26374
##           [,15]     [,16]     [,17]     [,18]     [,19]     [,20]     [,21]
## [1,]  99.34657 160.7004 198.50903 184.01083 162.72563  73.33935 180.2960
## [2,] 185.40127 123.6019  43.78025  75.96815  83.19745 166.00955 107.8439
## [3,]  47.54945 106.6868 120.30769  90.29121 168.33516  23.28022 147.4615
##           [,22]     [,23]     [,24]     [,25]     [,26]     [,27]     [,28]
## [1,]  52.14801 139.14801 103.75090 181.59206 147.43321 146.83755 154.30686
## [2,] 174.54459  60.75796 196.42675  48.22611  49.83758  57.15605 133.91720
## [3,]  29.68681 113.96703  60.01099  99.66484 109.03846 138.75824  67.52747
##           [,29]     [,30]     [,31]     [,32]     [,33]     [,34]     [,35]
## [1,] 186.42599 131.51986 129.57401 179.59567 132.50181 190.96390 157.57040
## [2,]  31.74522  98.22611  77.43312  29.54777  98.39172  47.23567 151.05732
## [3,]  86.02747  66.24725  97.76923  80.59890  73.98901  72.77473  80.00549
##           [,36]     [,37]     [,38]     [,39]     [,40]     [,41]     [,42]
## [1,] 145.02527 174.93863 191.28881 191.85199 172.16606 151.15884 125.03610
## [2,] 112.52866  40.82484  35.53822  47.42675  35.14331 124.68471 217.24522
## [3,]  59.31319 106.80769  75.32967 126.75824  98.45604  68.24725  69.65385
##           [,43]     [,44]     [,45]     [,46]     [,47]     [,48]     [,49]
## [1,] 170.77617 161.28520 143.24910 122.77256  80.58123  91.85921 130.94585
## [2,]  30.44268 144.27389 101.56051 208.90127 171.22611  93.75478 127.21338
## [3,] 124.86264  70.37912  57.21429  68.93956  54.74176  75.01648  65.63187
##           [,50]
## [1,] 163.25632
## [2,]  55.88217
## [3,] 133.45055
```

- 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).

```
# True labels in the dataset
true_labels <- num_top_50$Digit

# Get the predicted cluster assignments from k-means results
predicted_clusters <- kmeans_result$cluster_assignments

# Create a mapping between clusters and true labels
label_mapping <- sapply(1:K, function(cluster) {
  # Find the true label that is most frequent in each cluster
  majority_label <- names(sort(table(true_labels[predicted_clusters == cluster]), decreasing = TRUE))[1]
  return(as.numeric(majority_label))
})
```

```

})

# Map each predicted cluster to the majority true label
predicted_labels <- sapply(predicted_clusters, function(cluster) label_mapping[cluster])

# Calculate accuracy
accuracy <- sum(predicted_labels == true_labels) / length(true_labels)
cat("Clustering Accuracy:", round(accuracy * 100, 2), "%\n")

## Clustering Accuracy: 62.35 %

```

## 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?

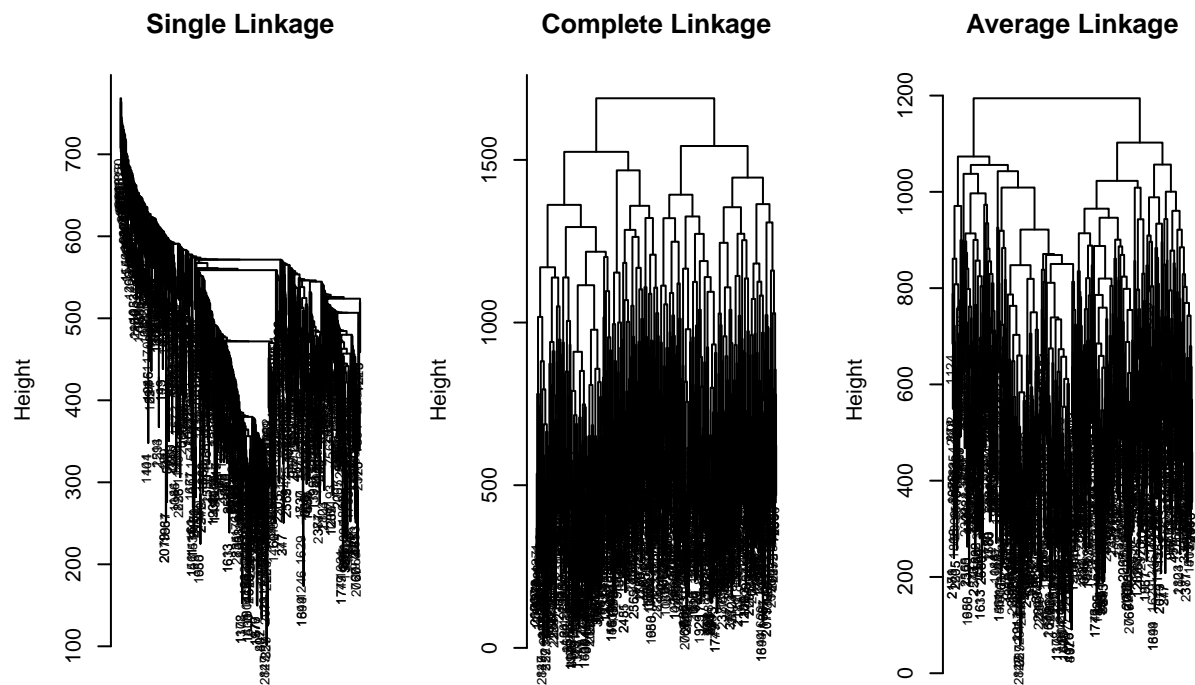
```

# Calculate the Euclidean distance matrix
dist_matrix <- dist(num_top_50, method = "euclidean")

# Perform hierarchical clustering with different linkage methods
hc_single <- hclust(dist_matrix, method = "single")
hc_complete <- hclust(dist_matrix, method = "complete")
hc_average <- hclust(dist_matrix, method = "average")

# Plot the dendrograms
par(mfrow = c(1, 3)) # Arrange plots side by side
plot(hc_single, main = "Single Linkage", xlab = "", sub = "", cex = 0.6)
plot(hc_complete, main = "Complete Linkage", xlab = "", sub = "", cex = 0.6)
plot(hc_average, main = "Average Linkage", xlab = "", sub = "", cex = 0.6)

```



```
par(mfrow = c(1, 1)) # Reset plot layout
```

Single Linkage: - The dendrogram is highly elongated with many “chained” clusters, where individual data points are linked one-by-one. - This chaining effect makes it harder to distinguish clear, well-separated clusters. - Single linkage is generally sensitive to noise and outliers, which can result in this kind of chaining pattern.

Complete Linkage: - The dendrogram is more balanced and compact, with clusters that appear well-separated. - This linkage method tends to produce more spherical clusters, which is often desirable if the data has distinct groups. - Complete linkage minimizes the maximum distance between clusters, making it less sensitive to outliers and better suited for identifying well-defined clusters.

Average Linkage: - The average linkage dendrogram is somewhat between single and complete linkage in terms of cluster compactness. - Clusters are generally balanced, though not as tight as with complete linkage.

Based on the structure observed in the dendrograms, complete linkage seems to be the most appropriate for this dataset. It provides well-separated, compact clusters that are less prone to the chaining effect seen in single linkage, making it easier to interpret the clustering structure.

- 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.

```
# Perform complete linkage clustering on the distance matrix
hc_complete <- hclust(dist_matrix, method = "complete")
```



```

# Cut the dendrogram to obtain 3 clusters
clusters <- cutree(hc_complete, k = 3)

# Create a contingency table between true labels and predicted cluster labels
table_result <- table(true_labels, clusters)

# Map each cluster label to the most frequent true label within that cluster
cluster_to_label <- sapply(1:ncol(table_result), function(col) {
  true_label <- names(which.max(table_result[, col]))
  return(true_label)
})

# Convert the mapping result to a named character vector
names(cluster_to_label) <- colnames(table_result)

# Replace cluster labels with the mapped labels
mapped_labels <- as.character(cluster_to_label[as.character(clusters)])

# Calculate accuracy by comparing mapped labels to true labels
accuracy <- sum(mapped_labels == true_labels) / length(true_labels)
print(paste("Clustering Accuracy:", round(accuracy * 100, 2), "%"))

## [1] "Clustering Accuracy: 76.2 %"

```

### 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.

```

#install.packages("kernlab")
library(kernlab)

## Warning: package 'kernlab' was built under R version 4.3.3

# Specify the number of clusters (for example, using the number of unique labels in the dataset)
num_clusters <- length(unique(mnist$Digit))

# Apply spectral clustering with an RBF kernel (Gaussian) and the specified number of clusters
spectral_clusters <- specc(as.matrix(mnist[, -1]), centers = num_clusters, kernel = "rbfdot")

# Extract the cluster assignments
cluster_labels <- as.factor(spectral_clusters@.Data)

# Compare with true labels
true_labels <- mnist$Digit
# Create a contingency table between true labels and cluster labels
table_result <- table(true_labels, cluster_labels)

```

```

# Map each cluster label to the true label with the highest count in each cluster
cluster_to_label <- apply(table_result, 2, function(x) names(which.max(x)))

# Replace cluster labels with mapped labels
mapped_labels <- cluster_to_label[as.character(cluster_labels)]

# Calculate accuracy
accuracy <- sum(mapped_labels == true_labels) / length(true_labels)
print(paste("Spectral Clustering Accuracy:", round(accuracy * 100, 2), "%"))

## [1] "Spectral Clustering Accuracy: 87.06 %"

```

The accuracy is higher than all of the previous methods.