

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	8
Question 3: Spectral Clustering [15 pts]	11

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.**

```
# Load the dataset
fileLocation <- "https://pjreddie.com/media/files/mnist_train.csv"
numRowsToDownload <- 2600
mnist2600 <- read.csv(fileLocation, nrows = numRowsToDownload)
dim(mnist2600)
```

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

```
# Set column names: the first column is 'Digit', the rest are pixel values
numColsMnist <- dim(mnist2600)[2]
colnames(mnist2600) <- c("Digit", paste("Pixel", seq(1:(numColsMnist - 1)), sep = ""))
# Filter the dataset to include only specific digits (2, 4, and 8)
mnist_filtered <- mnist2600[mnist2600$Digit %in% c(2, 4, 8), ]
```

```
# Apply variance screening to reduce features to the top 50 by marginal variance
variances <- apply(mnist_filtered[, -1], 2, var)
top_features <- order(variances, decreasing = TRUE)[1:50]
X <- mnist_filtered[, c(1, top_features + 1)]
mnist <- mnist_filtered
dim(X)
```

```
## [1] 773 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 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

Implement a function in R called `cluster_mean_update()` to update the centroids of clusters as part of a K-means clustering algorithm. This function should be designed with efficiency in mind, utilizing vectorized operations where possible.

- Function Inputs:
 - X : A 2D matrix or dataframe where each row represents an observation and each column represents a feature.
 - K : An integer representing the number of clusters.
 - `cluster_assignments`: A vector of length equal to the number of rows in X . Each element is an integer from 1 to K , representing the assigned cluster for each observation.
- Function Outputs
 - `centroids`: A 2D matrix with K rows and a number of columns equal to the features in X . Each row i represents the mean (centroid) of the observations assigned to cluster i .

It should begin by initializing an empty matrix to store the centroids. For each cluster, it filters the observations in X that are assigned to that cluster based on `cluster_assignments`, then calculates the mean of each feature across these observations and stores the result in the corresponding row of the centroids matrix. If a cluster has no observations, the function sets that centroid to a row of zeros, which ensures that each row in the output matrix accurately represents the mean (centroid) of the points assigned to each cluster. Include comments to clearly explain each part of the function.

After implementing the `cluster_mean_update()`, test it using sample data with $K = 3$ and a sample `cluster_assignments` vector to validate correct operation.

```
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
  centroids <- matrix(0, nrow = K, ncol = ncol(X) - 1)
```

```

# For each cluster, calculate the mean of all points assigned to that cluster
for (k in 1:K) {
  cluster_points <- X[cluster_assignments == k, -1]
  centroids[k, ] <- colMeans(cluster_points)
}
return(centroids)
}

```

```

# Testing the cluster_mean_update Function
set.seed(123) # Set seed for reproducibility
K <- 3
n <- nrow(X)
# Generate random cluster assignments
cluster_assignments <- sample(1:K, n, replace = TRUE)

# Get the updated centroids
centroids <- cluster_mean_update(X, K, cluster_assignments)
print(centroids)

```

```

##           [,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

Implement a function in R called `cluster_assignments()` to assign each observation in a dataset to the closest centroid in a K-means clustering algorithm. This function will be used alongside the previously defined `cluster_mean_update()` function. The goal of `cluster_assignments()` is to assign each data point in a dataset X to the nearest cluster centroid based on Euclidean distance.

- Function Inputs
 - X: A 2D matrix or dataframe where each row represents an observation and each column - represents a feature.
 - centroids: A 2D matrix where each row represents the centroid of a cluster. The matrix has K rows (one for each cluster) and the same number of columns as X.
- Function Outputs

The function should return a vector, `assignments`, with a length equal to the number of rows in X. Each element in this vector is an integer between 1 and K, representing the cluster assignment for each observation.

Include comments to clearly explain each part of the function, particularly the distance calculation and assignment process. Ensure the function is compatible with the `cluster_mean_update()` function for seamless integration and test the function using centroids generated by it.

After implementing the `cluster_assignments()` function, test it with the training data and the centroids from the previous step.

```
cluster_assignments <- function(X, centroids) {
  # Get the number of observations (rows) in X and the number of clusters (rows) in centroids
  n <- nrow(X)
  K <- nrow(centroids)
  # Initialize a vector to store the cluster assignments
  assignments <- rep(0, n)
  # Iterate through each observation in X
  for (i in 1:n) {
    # Calculate the Euclidean distance between the observation and each centroid
    distances <- apply(centroids, 1, function(centroid) {
      sqrt(sum((X[i, -1] - centroid)^2))
    })
    # Assign the observation to the cluster with the minimum distance
    assignments[i] <- which.min(distances)
  }
  return(assignments)
}

# Testing the cluster_assignments Function
# Use the centroids from the previous test
assignments <- cluster_assignments(X, centroids)
print(assignments)
```

```
##      [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 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 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

Implement a function in R called `kmeans()` to perform K-means clustering on a dataset. This function will use both `cluster_assignments()` and `cluster_mean_update()` for seamless integration and efficient operation.

The `kmeans()` function orchestrates the K-means clustering algorithm by iteratively assigning observations to the nearest centroids and updating centroids until convergence or reaching a specified maximum number of iterations.

- Function Inputs
 - X: A 2D matrix or dataframe where each row represents an observation and each column represents a feature.
 - K: The number of clusters, an integer specifying the number of clusters to form. `max_iter`: The maximum number of iterations for the algorithm (default set to 20).
- Function Outputs
 - `centroids`: A matrix of the final centroids after convergence or reaching the maximum iterations, with K rows (one for each cluster) and the same number of columns as X.
 - `final_assignments`: A vector of cluster assignments for each observation in X.
 - `iterations`: The number of iterations performed (can directly print the number of iterations).

Include comments to clearly explain each part of the function, particularly the distance calculation and assignment process.

After implementing the `cluster_assignments()` function, test it with the training data, $K = 3$, and the maximum number of iterations set to 20.

```

kmeans <- function(X, K, max_iter = 20) {
  # Initialize centroids by selecting K random observations from X
  centroids <- X[sample(nrow(X), K), -1]

  # Initialize previous centroids to a matrix of zeros to ensure convergence check is met on the first
  prev_centroids <- matrix(0, nrow = K, ncol = ncol(X) - 1)

  # Initialize iteration counter
  iter <- 1

  # Iterate until convergence or until the maximum number of iterations is reached
  while (iter <= max_iter && !all(centroids == prev_centroids)) {
    # Update cluster assignments using the cluster_assignments() function
    assignments <- cluster_assignments(X, centroids)

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

    # Increment iteration counter
    iter <- iter + 1
  }

  # Return the final centroids, cluster assignments, and number of iterations performed
  return(list(final_centroids = centroids, final_assignments = assignments, iterations = iter))
}

```

```

# Testing the function
kmeans_result <- kmeans(X, K = 3, max_iter = 20)
print(kmeans_result$final_centroids)

```

```

##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 147.44371 181.90728 201.68212 199.08609 190.93377 198.61589 117.5695
## [2,]  16.86301 170.35959 152.36986 151.81507 181.71575 163.54795 113.4452
## [3,] 185.29697  49.67576  15.57576  21.52424  24.67879  17.77273 111.6848
##           [,8]      [,9]      [,10]     [,11]     [,12]     [,13]     [,14]
## [1,] 189.56954 182.49007 183.24503 186.37748 164.15232 106.973510 164.2517
## [2,] 153.58219  32.45548 169.84589 181.05822  22.44521   3.085616 106.0788
## [3,]  14.97576 162.93333  26.18788  34.32727 170.51818 168.360606 116.3182
##           [,15]     [,16]     [,17]     [,18]     [,19]     [,20]     [,21]
## [1,] 171.39735 211.5563 169.2252 166.49007  92.37086 156.58278 128.6026
## [2,]  39.97945 107.0993 174.9144 148.08904 196.77740  10.10274 190.5856
## [3,] 172.22424 119.7697  42.4303  69.32121  92.20909 151.77273 107.7970
##           [,22]     [,23]     [,24]     [,25]     [,26]     [,27]     [,28]
## [1,]  97.50331  63.49669 170.82781 158.11258  81.03311  79.65563 159.3113
## [2,]  18.74315 165.84247  49.40068 155.20890 165.58219 174.89041 107.9863
## [3,] 165.02727  61.66667 185.20909  43.59697  47.71818  62.96667 125.7424
##           [,29]     [,30]     [,31]     [,32]     [,33]     [,34]     [,35]
## [1,] 159.16556 156.35099  53.49007 165.22517  75.45033 166.33775 187.3907
## [2,] 149.77740  90.44178 149.75685 140.29795 126.52397 139.09589 105.0068
## [3,]  28.77576  88.82727  79.37576  23.57273  99.16970  46.18485 141.4606
##           [,36]     [,37]     [,38]     [,39]     [,40]     [,41]     [,42]
## [1,] 115.5033 170.94702 161.81457 183.06623 162.1325 166.4570 205.6887

```

```
## [2,] 113.1678 146.08219 144.07192 164.54795 146.8801 104.0240 57.9726
## [3,] 108.5303 37.11212 34.40303 46.70909 28.1000 114.9485 204.6667
##      [,43]      [,44]      [,45]      [,46]      [,47]      [,48]      [,49]
## [1,] 197.13907 176.4702 146.91391 197.88742 149.11258 33.53642 169.05960
## [2,] 140.81164 112.6952 102.91781 59.68151 35.80137 106.55137 81.56507
## [3,] 26.37576 131.0091 90.14242 196.49091 160.84545 98.06061 117.62727
##      [,50]
## [1,] 196.39073
## [2,] 136.54110
## [3,] 53.12727
```

```
print(kmeans_result$final_assignments)
```

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

```
cat("Total Iterations Performed:", kmeans_result$iterations)
```

```
## Total Iterations Performed: 9
```

- 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).

```
# Extract true labels and predicted assignments
true_labels <- X$Digit
predicted_labels <- kmeans_result$final_assignments # from the kmeans function
# Create a mapping between clusters and true labels
# For each cluster, find the most common true label
library(dplyr)
```

```
cluster_mapping <- data.frame(true_label = true_labels, cluster = predicted_labels) %>%
  group_by(cluster) %>%
  summarise(predominant_label = as.numeric(names(which.max(table(true_label)))))
# Map predicted cluster assignments to the predominant true label
mapped_predictions <- sapply(predicted_labels, function(cluster) {
  cluster_mapping$predominant_label[cluster_mapping$cluster == cluster]
})
# Create the confusion matrix
confusion_matrix <- table(true_labels, predicted_labels)
print("Confusion Matrix:")
```

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

```
print(confusion_matrix)
```

```
##           predicted_labels
## true_labels    1     2     3
##           2 124   87   45
##           4    8    7 274
##           8   19 198   11
```

```
# Calculate accuracy
accuracy <- mean(mapped_predictions == true_labels)
print(paste("Clustering Accuracy:", round(accuracy * 100, 2), "%"))
```

```
## [1] "Clustering Accuracy: 77.1 %"
```

The code seems to be working as expected, as it correctly assigns observations to clusters and updates centroids based on the K-means algorithm. However, the code generated by GPT could be further improved in several ways: First, running K-means multiple times with different random initializations can improve results, as K-means is sensitive to initial centroid positions. By selecting the clustering result with the lowest within-cluster variance (inertia) after multiple runs, we can often achieve better performance.

Applying dimensionality reduction techniques like Principal Component Analysis (PCA) can also help by reducing noise and focusing on the most informative features, which can make clustering more effective.

Another enhancement is to explore alternative clustering methods, such as Gaussian Mixture Models (GMM) or Spectral Clustering, which may better capture complex data structures.

Finally, we can improve efficiency by replacing loops with vectorized operations where possible. For example, in the `cluster_assignments()` function, using matrix operations to calculate distances between all observations and centroids at once would speed up the process, especially for larger datasets.

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.

- [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?

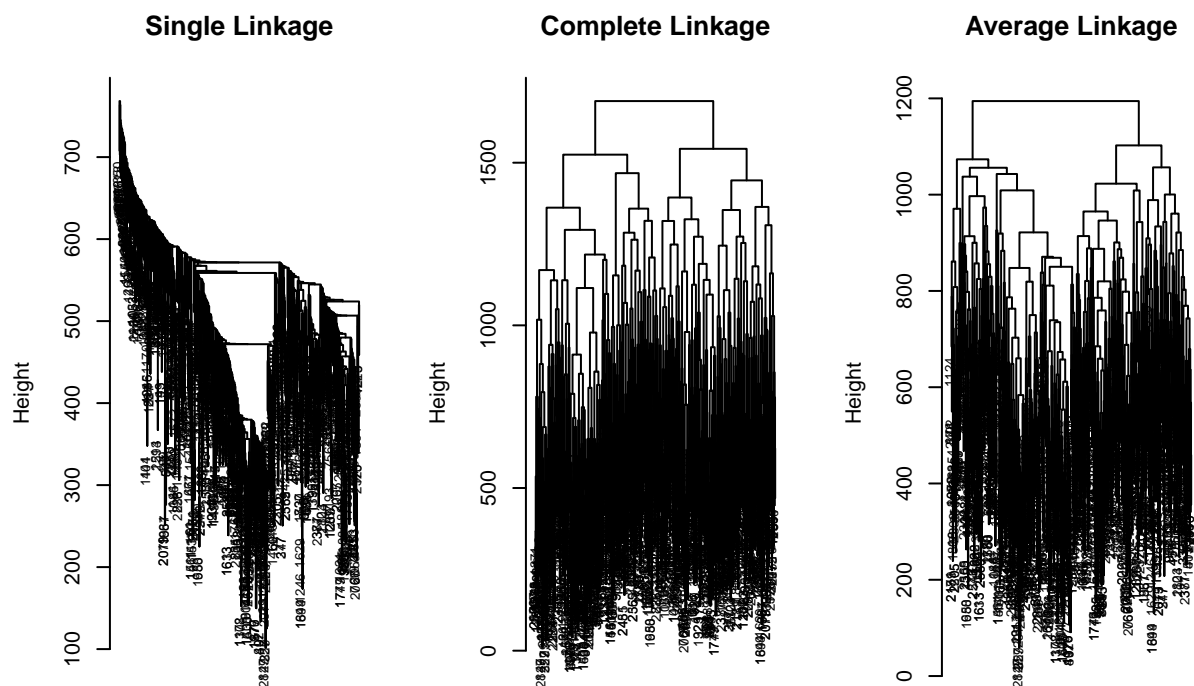

```

# Calculate the Euclidean distance matrix
dist_matrix <- dist(X, 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. - Clusters in average linkage merge at moderate heights, capturing some structure in the data but without achieving as clear separation between groups.

Conclusion: 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)])

confusion_matrix <- table(true_labels, clusters)
print("Confusion Matrix:")
```

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

```
print(confusion_matrix)
```

```
##           clusters
## true_labels  1   2   3
##           2  75 148 33
##           4 286   3   0
##           8  36  37 155
```

```
# 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 %"
```

The clustering accuracy achieved was 76.2%, indicating that the hierarchical clustering with complete linkage correctly grouped approximately 76% of observations based on the predominant label in each of the three clusters. This result reflects a moderate success in clustering similar digits together, though a considerable portion of observations were misclassified.

This level of accuracy is expected for an unsupervised clustering approach, as hierarchical clustering operates without knowledge of the true labels, relying solely on Euclidean distances between data points. Consequently, it may overlook subtle differences between digit images. The Euclidean distance metric can struggle to capture the complex structure in high-dimensional data like digit images, especially after reducing the dataset to the 50 most variable pixels. In high-dimensional spaces, Euclidean distance can lose its discriminatory power, making it challenging to distinguish clusters effectively.

Moreover, while complete linkage generally yields compact and well-separated clusters, it may not perfectly match the inherent structure of handwritten digits. Digits like 2, 4, and 8, which may share similar visual features in specific regions, can end up in mixed clusters, highlighting the limitations of distance-based clustering in capturing the nuanced variability within handwritten digit images.

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)]

# Create a confusion matrix
confusion_matrix_spectral <- table(true_labels, spectral_clusters)
print(confusion_matrix_spectral)
```

```
##           spectral_clusters
## true_labels  1    2    3
##           2  59 181  16
##           4   6   4 279
##           8 213   6   9
```

```
# 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 87.06 %.

We use the Radial Basis Function (RBF) kernel here, as it's typically well-suited for image data by capturing similarity based on pixel intensity. We set the number of clusters to 3 to be consistent with the previous analyses. The spectral clustering output shows an accuracy of 87.06%, which is an improvement over the hierarchical clustering accuracy of 76.2% from Question 2. It suggests that spectral clustering was able to capture the structure of the digit data more effectively.

This is because spectral clustering often performs well on datasets where clusters are not linearly separable. Spectral clustering is capable of finding non-linear boundaries between clusters, which is a key advantage over hierarchical clustering. This flexibility helps it perform better on data where classes are not well-separated in a simple Euclidean sense, such as digit images. Spectral clustering uses a similarity graph (constructed from the RBF kernel) to represent the data. This graph-based approach can capture relationships between points that are not solely based on pairwise distances, providing a more nuanced view of clusters. In contrast, hierarchical clustering with complete linkage relies directly on Euclidean distances, which may not fully capture the relationships in high-dimensional image data.

The accuracy improvement (from 76.2% with hierarchical clustering to 87.06% with spectral clustering) indicates that spectral clustering is better suited for the MNIST data in this context. The non-linear similarities captured by the RBF kernel likely align more closely with the inherent structure of the digit images, leading to more accurate clusters.