

HW10_zilinw3

Zilin Wang (zilinw3)

2024-11-04

Contents

Question 1: K-means Clustering [65 pts]	1
Question 2: Hierarchical Clustering	10
Question 3: Spectral Clustering [15 pts]	14

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.**

- [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.

Answer:

Prompt:

We will again use the MNIST dataset. We will use the first 2600 observations of it:

```
# 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)
```

Please

1. Preparing the Data

Filtering the dataset to include only specific digits (2, 4, and 8).

Applying variance screening to reduce features to the top 50 by marginal variance.

2. Creating the cluster_mean_update() Function

Write a function called cluster_mean_update() in R that takes three arguments:

-X: A data matrix where each row represents an observation (a digit image) and each column represents a feature (one of the 50 selected pixel values).

-K: The integer number of clusters (e.g., 3 for this exercise).

-cluster_assignments: A vector of length equal to the number of rows in X. Each entry is an integer between 1 and K, indicating the cluster assignment of each observation.

The function should return an updated centroid matrix. This matrix should have K rows and the same number of columns as X (i.e., 50 columns). Each row in the output matrix represents the mean of the points assigned to that cluster.

Requirements:

-Initialize an empty matrix to store the centroids.

-For each cluster, calculate the mean of all points in X assigned to that cluster and store this mean in the centroid matrix.

-Ensure that the function includes comments explaining each step.

-Test the function with sample data: Use K = 3 and a randomly generated cluster_assignments vector.

Here is the code:

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

# 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)]

cluster_mean_update <- function(X, K, cluster_assignments) {
  # Initialize an empty matrix to store the 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
set.seed(42)
K <- 3
n <- nrow(X)
cluster_assignments <- sample(1:K, n, replace = TRUE)

centroids <- cluster_mean_update(X, K, cluster_assignments)
print(centroids)

##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]
## [1,] 112.6875 113.5781 100.2422 106.82422 114.1055 103.9766 108.3555 93.62891
## [2,] 112.8809 128.0217 101.3321 97.92419 120.1661 102.0181 127.6065 100.59928
## [3,] 117.5833 121.1167 109.8125 112.59583 114.7458 119.7292 102.7042 110.74167
##           [,9]      [,10]      [,11]      [,12]      [,13]      [,14]      [,15]      [,16]
## [1,] 112.8398 107.2031 114.0508 105.0820 91.23047 120.8008 116.6211 133.1211
## [2,] 120.3357 108.0361 126.2022 112.9892 93.51986 128.3935 116.7978 131.1191
## [3,] 119.0875 118.9042 117.4375 122.5542 97.30417 115.3000 134.0875 134.7625
##           [,17]      [,18]      [,19]      [,20]      [,21]      [,22]      [,23]      [,24]
## [1,] 111.0781 110.6562 128.6211 94.69141 129.1680 92.41406 98.98047 125.6328
## [2,] 114.9711 123.1047 129.2852 95.40433 151.1300 97.77978 109.82671 128.9783
## [3,] 126.4458 120.1250 137.9042 108.37917 148.8042 99.63333 94.17917 139.3750
##           [,25]      [,26]      [,27]      [,28]      [,29]      [,30]      [,31]
## [1,] 101.5117 96.35547 101.9766 121.1719 89.42578 106.74219 99.25781
## [2,] 119.9567 101.42238 110.4621 137.5307 113.51986 101.59567 107.86282
## [3,] 101.5333 98.21667 113.2125 116.5292 95.52917 99.42917 94.63333
##           [,32]      [,33]      [,34]      [,35]      [,36]      [,37]      [,38]
## [1,] 93.45703 98.82422 96.36719 136.3789 109.2852 106.10156 89.41016
## [2,] 96.89531 120.10830 107.40794 138.4765 121.2708 94.63177 110.45126
## [3,] 95.54167 93.72917 110.63333 134.8708 103.0500 113.92083 101.55000
##           [,39]      [,40]      [,41]      [,42]      [,43]      [,44]      [,45]      [,46]
## [1,] 112.7578 97.01953 123.8750 144.8906 100.11719 130.7227 101.8555 136.5078
## [2,] 115.2563 97.28159 122.8159 148.7184 98.31047 135.7726 107.1986 145.5126
## [3,] 126.3042 103.58333 115.4625 155.1667 111.36250 132.1375 109.2250 153.7375
##           [,47]      [,48]      [,49]      [,50]
## [1,] 101.4141 92.39844 114.3359 106.3555
## [2,] 111.8014 84.34296 114.2274 111.6751

```

```
## [3,] 121.3250 89.66667 113.5458 120.4000
```

- 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.

Answer:

Prompt:

Write a function called `cluster_assignments()` in R that takes two arguments:

- X : A data matrix where each row represents an observation (an image of a digit) and each column represents a feature (one of the 50 selected pixel values).

-centroids: A matrix where each row represents the centroid of a cluster. The matrix has K rows (one for each cluster) and 50 columns (one for each feature).

The function should return a vector of cluster assignments. This vector should have a length equal to the number of rows in X , with each element being an integer between 1 and K . Each observation is assigned to the cluster whose centroid is closest, based on Euclidean distance.

Requirements:

-For each observation in X , compute the Euclidean distance to each centroid.

-Assign the observation to the cluster with the minimum distance.

-Include comments explaining the logic of each step, especially the distance calculation and assignment process.

-Test the function using the centroids generated by the `cluster_mean_update()` function from part (a).

Here is the code:

```
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)  
}  
  
# Assuming you have already run the cluster_mean_update() function
```

```
assignments <- cluster_assignments(X, centroids)
print(assignments)
```

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

- 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.

Answer:

Prompt:

Write a function called `kmeans()` in R that performs k-means clustering on a dataset. This function should take the following arguments:

- X: A data matrix where each row represents an observation (a digit image) and each column represents a feature (one of the 50 selected pixel values).
- K: The integer number of clusters.
- max_iter: The maximum number of iterations for the algorithm (set to 20 by default).

The function should return:

- final_centroids: The final centroids matrix after convergence or reaching the maximum number of iterations.
- final_assignments: A vector of cluster assignments for each observation in X.

Requirements:

- Initialize the centroids by selecting K random observations from X.
- In each iteration, use the `cluster_assignments()` function to update the cluster assignments for each observation.
- Use the `cluster_mean_update()` function to update the centroids based on the new assignments.
- Stop if the centroids do not change from one iteration to the next, indicating convergence.
- Include detailed comments explaining each step, including initialization, iteration, and convergence check.
- Test the function on X with $K = 3$ and `max_iter = 20`.

Here is the code:

```
kmeans <- function(X, K, max_iter = 20) {  
  # Initialize the centroids by selecting K random observations from X  
  centroids <- X[sample(nrow(X), K), -1]  
  
  # Initialize the previous centroids to a matrix of zeros, so that the convergence condition is met on  
  prev_centroids <- matrix(0, nrow = K, ncol = ncol(X) - 1)  
  
  # Iterate until convergence or maximum iterations reached  
  iter <- 1  
  while (iter <= max_iter && !all(centroids == prev_centroids)) {  
    # Update the cluster assignments using the cluster_assignments() function  
    assignments <- cluster_assignments(X, centroids)  
  
    # Update the centroids using the cluster_mean_update() function  
    prev_centroids <- centroids  
    centroids <- cluster_mean_update(X, K, assignments)  
  
    iter <- iter + 1  
  }  
  
  # Return the final centroids and cluster assignments  
  return(list(final_centroids = centroids, final_assignments = assignments))  
}  
  
# Testing  
result <- kmeans(X, K = 3, max_iter = 20)  
print(result$final_centroids)
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]  
## [1,] 16.86301 170.35959 152.36986 151.81507 181.71575 163.54795 113.4452  
## [2,] 147.44371 181.90728 201.68212 199.08609 190.93377 198.61589 117.5695  
## [3,] 185.29697 49.67576 15.57576 21.52424 24.67879 17.77273 111.6848  
##           [,8]      [,9]      [,10]     [,11]     [,12]     [,13]     [,14]  
## [1,] 153.58219 32.45548 169.84589 181.05822 22.44521 3.085616 106.0788  
## [2,] 189.56954 182.49007 183.24503 186.37748 164.15232 106.973510 164.2517  
## [3,] 14.97576 162.93333 26.18788 34.32727 170.51818 168.360606 116.3182  
##           [,15]     [,16]     [,17]     [,18]     [,19]     [,20]     [,21]  
## [1,] 39.97945 107.0993 174.9144 148.08904 196.77740 10.10274 190.5856  
## [2,] 171.39735 211.5563 169.2252 166.49007 92.37086 156.58278 128.6026  
## [3,] 172.22424 119.7697 42.4303 69.32121 92.20909 151.77273 107.7970  
##           [,22]     [,23]     [,24]     [,25]     [,26]     [,27]     [,28]  
## [1,] 18.74315 165.84247 49.40068 155.20890 165.58219 174.89041 107.9863  
## [2,] 97.50331 63.49669 170.82781 158.11258 81.03311 79.65563 159.3113  
## [3,] 165.02727 61.66667 185.20909 43.59697 47.71818 62.96667 125.7424  
##           [,29]     [,30]     [,31]     [,32]     [,33]     [,34]     [,35]  
## [1,] 149.77740 90.44178 149.75685 140.29795 126.52397 139.09589 105.0068  
## [2,] 159.16556 156.35099 53.49007 165.22517 75.45033 166.33775 187.3907  
## [3,] 28.77576 88.82727 79.37576 23.57273 99.16970 46.18485 141.4606  
##           [,36]     [,37]     [,38]     [,39]     [,40]     [,41]     [,42]  
## [1,] 113.1678 146.08219 144.07192 164.54795 146.8801 104.0240 57.9726  
## [2,] 115.5033 170.94702 161.81457 183.06623 162.1325 166.4570 205.6887  
## [3,] 108.5303 37.11212 34.40303 46.70909 28.1000 114.9485 204.6667
```

```
##           [,43]      [,44]      [,45]      [,46]      [,47]      [,48]      [,49]
## [1,] 140.81164 112.6952 102.91781  59.68151  35.80137 106.55137  81.56507
## [2,] 197.13907 176.4702 146.91391 197.88742 149.11258  33.53642 169.05960
## [3,]  26.37576 131.0091  90.14242 196.49091 160.84545  98.06061 117.62727
##           [,50]
## [1,] 136.54110
## [2,] 196.39073
## [3,]  53.12727
```

```
print(result$final_assignments)
```

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

- 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).

Answer:

```
# Extract true labels and predicted assignments
true_labels <- X$Digit
predicted_labels <- result$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)

##
## 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

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  87 124  45
##           4   7   8 274
##           8 198  19  11

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

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

```

My code is working as expected because we got the 77.1% accuracy. The accuracy of the clustering is 77.1%, meaning that approximately 77% of the samples were assigned to the cluster that corresponds to their true label after applying the predominant label mapping. This is a reasonable result for k-means clustering on a dataset like MNIST, especially with only 50 features after variance-based feature selection..

Maybe the code generated by GPT can be improved by avoiding loops where possible, as loops in R can be relatively slow and inefficient, especially for larger datasets. In R, vectorized operations and matrix manipulations are typically much faster than iterative loops. We can vectorizing the distance calculation in `cluster_assignments()`. In the original `cluster_assignments()` function, we calculate the Euclidean distance from each observation to each centroid using a for loop. This can be inefficient, especially with larger datasets. Instead of looping through each centroid, we can leverage matrix operations to compute the squared Euclidean distances between each observation and each centroid simultaneously. Instead of iterating over each centroid and computing the distance individually, we can use matrix operations with broadcasting to achieve this more efficiently.

Also, K-means is sensitive to initial centroid placements, which can lead to suboptimal clustering. By running k-means with multiple random initializations and choosing the result with the lowest within-cluster variance (inertia), we can often achieve a better clustering solution. This can be done by modifying the `kmeans()` function to repeat the clustering process a fixed number of times (say, 10), storing the results of each run, and selecting the one with the best fit.

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?

Answer:

```
library(dplyr)

# Calculate the distance matrix using Euclidean distance
# Assuming 'X' is the data matrix with only the pixel values (50 columns)
distance_matrix <- dist(X, method = "euclidean")

# Perform hierarchical clustering with different linkage methods
# Single Linkage
hc_single <- hclust(distance_matrix, method = "single")

# Complete Linkage
hc_complete <- hclust(distance_matrix, method = "complete")

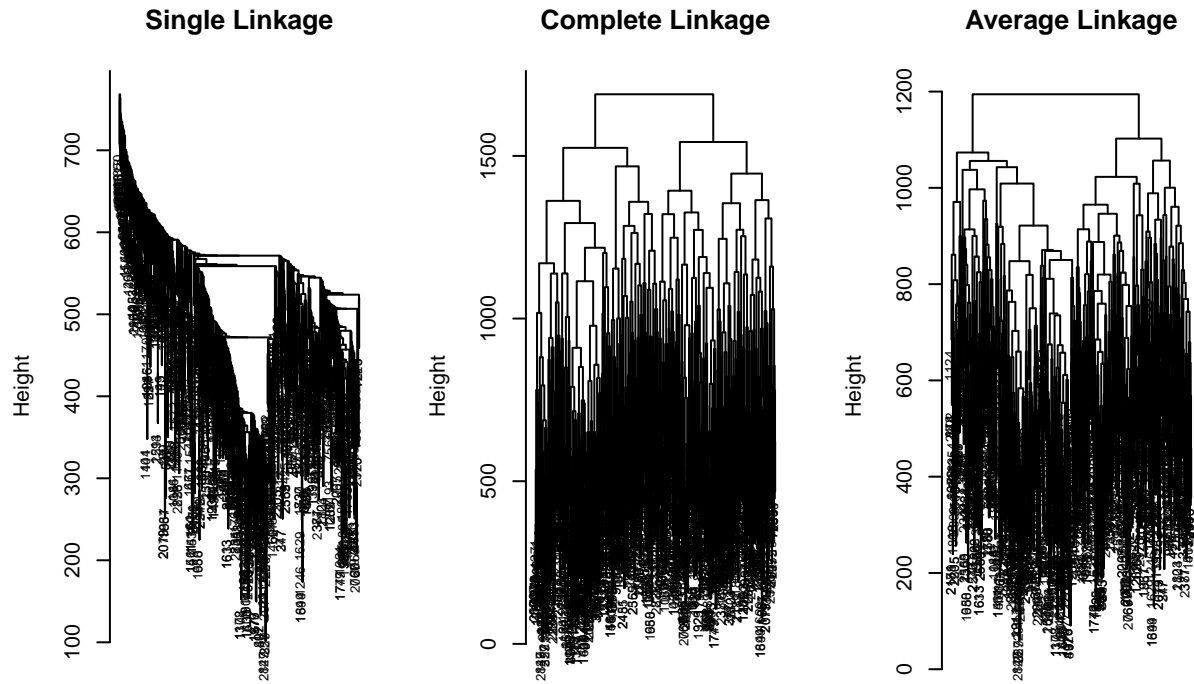
# Average Linkage
hc_average <- hclust(distance_matrix, method = "average")

# Plot the dendrograms
par(mfrow = c(1, 3))

# Single linkage dendrogram
plot(hc_single, main = "Single Linkage", xlab = "", sub = "", cex = 0.6)

# Complete linkage dendrogram
plot(hc_complete, main = "Complete Linkage", xlab = "", sub = "", cex = 0.6)

# Average linkage dendrogram
plot(hc_average, main = "Average Linkage", xlab = "", sub = "", cex = 0.6)
```



The single linkage method tends to produce a phenomenon called chaining, where clusters are formed by linking the closest points, even if they are stretched across larger distances. In the dendrogram, we can see that single linkage has created a few long, stretched-out clusters. These clusters have merged at relatively low height values, and the branches are not well separated. Single linkage may not be ideal for this dataset because it lacks compact clusters and tends to merge distant points, making it challenging to define clear, cohesive groups.

Complete linkage typically results in compact, well-separated clusters, as it uses the maximum distance between points in two clusters to determine their linkage. In this dendrogram, clusters appear more distinct and are separated at greater heights, suggesting stronger cohesion within clusters and clearer separations between clusters. The structure is more balanced, and clusters merge at higher height values than with single linkage, indicating well-separated groups that could more likely reflect true underlying patterns in the data. This visual structure suggests that complete linkage may be a suitable choice for clustering well-defined groups.

Average linkage provides a compromise between single and complete linkage by using the average distance between all points in two clusters to determine linkage. The resulting dendrogram is more balanced than single linkage, but not as compact and separated as complete linkage. Although clusters are somewhat cohesive, they merge at lower heights than in the complete linkage dendrogram. This suggests that average linkage might capture some structure but may not achieve the same level of separation between groups as complete linkage. Average linkage could be a reasonable option, but it may not produce clusters that are as distinct as those from complete linkage.

Therefore complete linkage appears to be the most appropriate method for this dataset because of the compact and well-separated clusters and the clear structure in dendrogram. Complete linkage produces clusters that are compact and well-separated, as observed by the larger height values at which clusters merge. This suggests that it captures distinct groups within the dataset, which is often desirable when we expect clear separations (e.g., different digit classes in MNIST). Also, the complete linkage dendrogram shows

a well-defined hierarchical structure, which can help identify distinct clusters when we cut the dendrogram. This is particularly useful when we aim to obtain a small number of clusters.

- 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.

Answer:

```
# Cut the dendrogram to form 3 clusters using complete linkage
cluster_assignments <- cutree(hc_complete, k = 3)

# Evaluate clustering accuracy by comparing with true labels
# True labels
true_labels <- X$Digit

# Create a data frame with true labels and predicted cluster assignments
clustering_results <- data.frame(true_label = true_labels, cluster = cluster_assignments)

# Map each cluster to the most common true label in that cluster
cluster_mapping <- clustering_results %>%
  group_by(cluster) %>%
  summarise(predominant_label = as.numeric(names(which.max(table(true_label)))))

# Reassign cluster labels based on predominant true label
mapped_predictions <- apply(cluster_assignments, function(cluster) {
  cluster_mapping$predominant_label[cluster_mapping$cluster == cluster]
})

# Create a confusion matrix
confusion_matrix <- table(true_labels, cluster_assignments)
print("Confusion Matrix:")
```

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

```
print(confusion_matrix)
```

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

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

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

The accuracy of the clustering is 76.2%. This indicates that the hierarchical clustering with complete linkage was able to correctly classify approximately 76% of the observations based on the predominant label in each of the three clusters. An accuracy of 76.2% suggests that the hierarchical clustering was moderately successful at grouping similar digits together but still misclassified a significant number of observations. This level of accuracy is reasonable for an unsupervised clustering method, given that hierarchical clustering doesn't use the true labels to guide the clustering. The algorithm relies solely on the Euclidean distances between data points, which may not capture all the nuanced differences between digit images.

Hierarchical clustering is purely based on distances between points, without any prior knowledge of class labels. This can lead to mixed clusters, especially if some digits (like 2, 4, and 8) have similar visual features in certain positions. Also, even after selecting the 50 most variable pixels, the data remains high-dimensional, which can make distance-based clustering methods like hierarchical clustering less effective. In high-dimensional spaces, Euclidean distance becomes less informative, often making it difficult to distinguish clusters effectively. What's more, although complete linkage generally provides compact and well-separated clusters, the method may not perfectly align with the structure of handwritten digits, which can have variations that aren't captured by simple distance-based clustering.

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.

Answer:

```
set.seed(42)
library(kernlab)

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

# Prepare the data by removing the Digit column
X_all <- as.matrix(mnist_filtered[, -1])

# Set number of clusters to 3
num_clusters <- 3

# Run spectral clustering with Gaussian (RBF) kernel and specified number of clusters
spectral_clustering_result <- specc(X_all, centers = num_clusters, kernel = "rbfdot")

# Extract the cluster assignments
cluster_assignments <- as.numeric(spectral_clustering_result)

# Evaluate clustering accuracy
# True labels from the 'Digit' column
true_labels <- mnist_filtered$Digit

# Create a data frame with true labels and predicted cluster assignments
clustering_results <- data.frame(true_label = true_labels, cluster = cluster_assignments)

# Map each cluster to the most common true label in that cluster
cluster_mapping <- clustering_results %>%
  group_by(cluster) %>%
  summarise(predominant_label = as.numeric(names(which.max(table(true_label)))))

# Reassign cluster labels based on predominant true label
mapped_predictions <- sapply(cluster_assignments, function(cluster) {
  cluster_mapping$predominant_label[cluster_mapping$cluster == cluster]
})

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

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

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

```
## [1] "Spectral Clustering Accuracy: 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. This 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.