Stat 432 Homework 10 Jerry Liang

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

Contents

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.**

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.

I need a function named cluster_mean_update that takes in a 2D NumPy array X (observations as rows, features as columns), an integer K representing the number of clusters, and a 1D array cluster_assignments indicating cluster membership for each observation. The function should: Initialize an array centroids of shape (K, number of features) to store updated cluster centroids.For each cluster (from 0 to K-1):Collect all observations assigned to the cluster and compute the mean of these observations along the feature axis and set this as the new centroid for the cluster. It shall Handle cases where no observations are assigned to a cluster by keeping the centroid unchanged or reinitializing as needed. Lastly Return the centroids array.

```r
cluster_mean_update <- function(X, K, cluster_assignments) {
  # Get the number of features
  num_features <- ncol(X)

  # Initialize centroids matrix
  centroids <- matrix(0, nrow = K, ncol = num_features)

  for (k in 1:K) {
    # Get indices of observations belonging to cluster k
    cluster_points <- X[cluster_assignments == (k - 1), , drop = FALSE]

    if (nrow(cluster_points) > 0) {
      # Update the centroid with the mean of the cluster points
      centroids[k, ] <- colMeans(cluster_points)
    } else {
      # Handle the case where no observations are assigned to the cluster
      # Reinitialize the centroid randomly from existing data points
      centroids[k, ] <- X[sample(1:nrow(X), 1), ]
    }
  }

  return(centroids)
}
```

```r
  # 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, nrows = 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
```

```r
  library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.1     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
```

```
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
numColsMnist <- dim(mnist2600)[2]
colnames(mnist2600) <- c("Digit", paste("Pixel", seq(1:(numColsMnist - 1)), sep = ""))

# Split data into training and test sets
train_data <- mnist2600[1:1300, ]
test_data <- mnist2600[1301:2600, ]

# Filter to keep only digits 2, 4, and 8
train_data <- train_data %>% filter(Digit %in% c(2, 4, 8))
test_data <- test_data %>% filter(Digit %in% c(2, 4, 8))

# Calculate the variance for each pixel column
pixel_variances <- sapply(train_data[ , -1], var)

# Select the top 250 pixel columns with the highest variance
top_50_pixels <- names(sort(pixel_variances, decreasing = TRUE))[1:50]
train_data1 <- train_data
test_data1 <- test_data
# Keep only the top 250 pixels and the Digit column in the training and test sets
train_data <- train_data %>% select(Digit, all_of(top_50_pixels))
test_data <- test_data %>% select(Digit, all_of(top_50_pixels))
```

```
set.seed(123)
test_data <- matrix(runif(50), nrow = 10, ncol = 5) # 10 observations, 5 features

# Sample cluster assignments (assigning each observation to a cluster)
cluster_assignments <- sample(0:2, 10, replace = TRUE) # Assign to clusters 0, 1, or 2

# Number of clusters
K <- 3

# Run the function
centroids <- cluster_mean_update(test_data, K, cluster_assignments)

# Print the centroids
print("Updated Centroids:")
```

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

```
print(centroids)
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.4371930 0.5808418 0.6243757 0.5332229 0.2854531
## [2,] 0.7342864 0.1745062 0.5998859 0.3915366 0.1927394
## [3,] 0.7093124 0.6601571 0.6113956 0.6431307 0.5470732
```

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.

I need to create a function named cluster_assignments() that assigns each data point in the dataset X to the nearest cluster based on the given centroids. This function should be compatible with the previously created cluster_mean_update() function so that they can work together to perform k-means clustering.

```
cluster_assignments <- function(X, centroids) {
  # Get the number of data points and clusters
  num_points <- nrow(X)
  K <- nrow(centroids)

  # Initialize vector to store the cluster assignments for each point
  assignments <- integer(num_points)

  for (i in 1:num_points) {
    # Calculate the distance from each data point to each centroid
    distances <- apply(centroids, 1, function(centroid) sum((X[i, ] - centroid)^2))

    # Assign the data point to the nearest centroid (minimum distance)
    assignments[i] <- which.min(distances) - 1  # Adjust for zero-based indexing if needed
  }

  return(assignments)
}
```

```
new_assignments <- cluster_assignments(test_data, centroids)

# Print the results
print("Cluster Assignments for each data point:")
```

```
## [1] "Cluster Assignments for each data point:"
```

```
print(new_assignments)
```

```
##  [1] 0 2 0 2 1 0 0 1 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.

I need a single R function kmeans() that implements the k-means clustering algorithm. Based on the previous functions I have.

```
set.seed(123)
kmeans <- function(X, K, max_iterations = 20) {
  # Randomly initialize K centroids from the data points
```

```r
  centroids <- X[sample(1:nrow(X), K), ]

  for (iteration in 1:max_iterations) {
    # Step 1: Assign clusters to data points based on current centroids
    assignments <- cluster_assignments(X, centroids)

    # Step 2: Update centroids based on current cluster assignments
    new_centroids <- cluster_mean_update(X, K, assignments)

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

  # Return final centroids and cluster assignments
  return(list(centroids = centroids, assignments = assignments))
}
```

```r
final_centroids <- kmeans(test_data, K = 3, max_iterations = 20)

# Print the final centroids
print("Final Centroids:")
```

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

```r
print(final_centroids)
```

```
## $centroids
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.2473703 0.8447430 0.7461922 0.7105085 0.2317768
## [2,] 0.4566147 0.9545036 0.1471136 0.2316258 0.8578277
## [3,] 0.7639582 0.2908267 0.6283578 0.5025714 0.3168010
##
## $assignments
##  [1] 0 2 0 2 2 0 2 2 2 1
```

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).

```r
map_clusters_to_labels <- function(assignments, true_labels) {
  unique_clusters <- unique(assignments)
  cluster_label_map <- integer(length(unique_clusters))

  for (cluster in unique_clusters) {
    # Find the most common true label in each cluster
    cluster_indices <- which(assignments == cluster)
    common_label <- as.numeric(names(sort(table(true_labels[cluster_indices]), decreasing = TRUE)[1]))
    cluster_label_map[cluster + 1] <- common_label  # Adjust for zero-based indexing
  }
```

```
  return(cluster_label_map)
}

# Apply k-means clustering to the training data
X <- as.matrix(train_data[, -1])  # Remove 'Digit' column for clustering
true_labels <- train_data$Digit

result <- kmeans(X, K = 3, max_iterations = 20)
assignments <- result$assignments

# Map cluster assignments to labels
label_map <- map_clusters_to_labels(assignments, true_labels)

# Apply the mapping to the assignments
mapped_assignments <- label_map[assignments + 1]  # Adjust for zero-based indexing

# Calculate the accuracy
accuracy <- sum(mapped_assignments == true_labels) / length(true_labels)

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

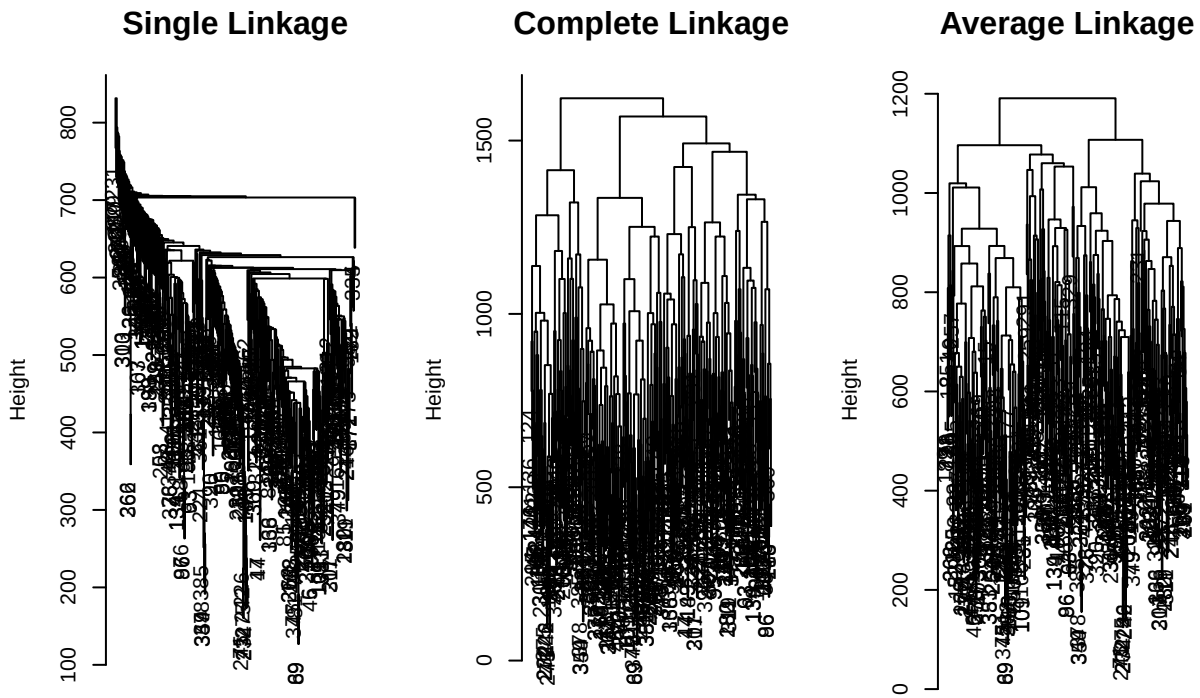
```
## [1] "Clustering Accuracy: 77.06 %"
```

We can simplify the indexing in function, and can make iteration or Ks variable so we can change as we like.
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?

```
X <- as.matrix(train_data)

distance_matrix <- dist(X, method = "euclidean")

# Perform hierarchical clustering with different linkage methods
hclust_single <- hclust(distance_matrix, method = "single")
hclust_complete <- hclust(distance_matrix, method = "complete")
hclust_average <- hclust(distance_matrix, method = "average")

# Plot the dendrograms for each linkage method
par(mfrow = c(1, 3))  # Set up the plotting area to show three plots side-by-side

# Single linkage dendrogram
plot(hclust_single, main = "Single Linkage", sub = "", xlab = "", cex.main = 1.5)

# Complete linkage dendrogram
plot(hclust_complete, main = "Complete Linkage", sub = "", xlab = "", cex.main = 1.5)

# Average linkage dendrogram
plot(hclust_average, main = "Average Linkage", sub = "", xlab = "", cex.main = 1.5)
```

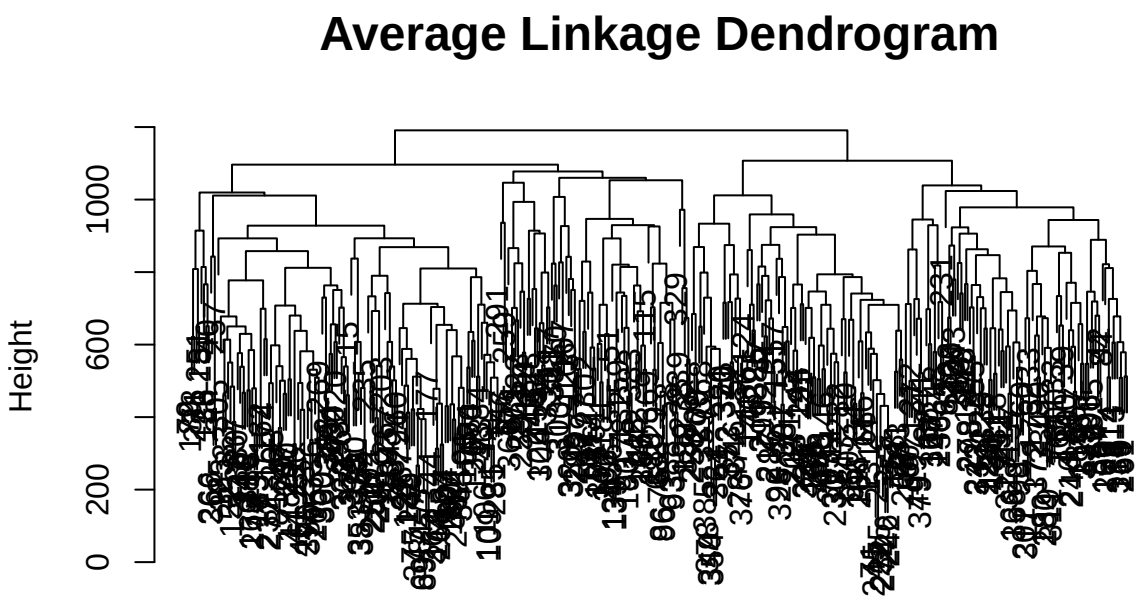



```
par(mfrow = c(1, 1))
```

Average linkage dendrogram seems to be the best since it has well-separated branches and the clustering method is finding relatively consistent cluster. Also, In average linkage clusters merge at a higher height, closer to the top.

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.

```
true_labels <- train_data$Digit
distance_matrix <- dist(X, method = "euclidean")

# Perform hierarchical clustering using average linkage
hclust_avg <- hclust(distance_matrix, method = "average")

# Plot the dendrogram (optional for visualization)
plot(hclust_avg, main = "Average Linkage Dendrogram", sub = "", xlab = "", cex.main = 1.5)
```




```r
# Cut the dendrogram to obtain 3 clusters
cluster_assignments <- cutree(hclust_avg, k = 3)

# Function to map clusters to true labels for accuracy calculation
map_clusters_to_labels <- function(assignments, true_labels) {
  unique_clusters <- unique(assignments)
  cluster_label_map <- integer(length(unique_clusters))

  for (cluster in unique_clusters) {
    cluster_indices <- which(assignments == cluster)
    common_label <- as.numeric(names(sort(table(true_labels[cluster_indices]), decreasing = TRUE)[1]))
    cluster_label_map[cluster] <- common_label
  }

  return(cluster_label_map)
}

# Map cluster assignments to the most common true labels
label_map <- map_clusters_to_labels(cluster_assignments, true_labels)
mapped_assignments <- label_map[cluster_assignments]

# Calculate the accuracy
accuracy <- sum(mapped_assignments == true_labels) / length(true_labels)

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

```
## [1] "Clustering Accuracy: 63.14 %"
```

An accuracy of 63.14% is a moderate value, there is still considerable discrepancy between the clustering output and the true labels. # 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.

```
library(kernlab)
```

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

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

```
## The following object is masked from 'package:purrr':
##
##     cross
```

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

```
# Prepare training and test data
# Assume train_data1 and test_data1 are already loaded and preprocessed
X_train <- as.matrix(train_data1)
true_labels_train <- train_data1$Digit   # Extract the true labels for training data

# Specify the kernel type and the number of clusters
# Choose from: "rbfdot" (Radial Basis Function), "polydot" (Polynomial), etc.
kernel_type <- "rbfdot"  # Radial Basis Function kernel
num_clusters <- 3        # Based on the dataset's structure

# Apply spectral clustering using specc()
spectral_clustering_result <- specc(X_train, centers = num_clusters, kernel = kernel_type)

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

# Function to map clusters to the most common true labels for accuracy calculation
map_clusters_to_labels <- function(assignments, true_labels) {
  unique_clusters <- unique(assignments)
  cluster_label_map <- integer(length(unique_clusters))

  for (cluster in unique_clusters) {
    cluster_indices <- which(assignments == cluster)
    common_label <- as.numeric(names(sort(table(true_labels[cluster_indices]), decreasing = TRUE)[1]))
    cluster_label_map[cluster] <- common_label
  }

  return(cluster_label_map)
```

```r
}

# Map cluster assignments to the most common true labels
label_map <- map_clusters_to_labels(spectral_assignments, true_labels_train)
mapped_assignments <- label_map[spectral_assignments]

# Calculate the accuracy
accuracy <- sum(mapped_assignments == true_labels_train) / length(true_labels_train)

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

```
## [1] "Spectral Clustering Accuracy: 85.05 %"
```

This is the method with highest accuracy compared to the previous ones.