Assignment13: Illustration of the Expectation-Maximization Algorithm using MNIST Dataset

yifan

2025-05-14

Introduction

In class, we learned the general expectation-maximization (E-M) algorithm:[1]

Given a joint distribution $p(X, Z|\theta)$ over observed variables X and latent variables Z, governed by parameters θ, the goal is to maximize the likelihood function $p(X|\theta)$ with respect to θ.

1. Choose an initial setting for the parameters θ^{old}.
2. **E step**: Evaluate $p(Z|X, \theta^{\text{old}})$.
3. **M step**: Evaluate θ^{new} given by

$$\theta^{\text{new}} = \arg\max_{\theta} Q(\theta, \theta^{\text{old}})$$

where

$$Q(\theta, \theta^{\text{old}}) = \sum_{Z} p(Z|X, \theta^{\text{old}}) \ln p(X, Z|\theta).$$

4. Check for convergence of either the log likelihood or the parameter values. If the convergence criterion is not satisfied, then let

$$\theta^{\text{old}} \leftarrow \theta^{\text{new}}$$

and return to step 2.

However, to illustrate this algorithm, we would probably need a more specific example of how this algorithm can be applied. Let's consider a finite mixture of Bernoulli distributions, as mentioned in the textbook:[2]

$$p(\mathbf{x}|\mu,\pi) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|\mu_k) \tag{1}$$

Here, $\mathbf{x} = (x_1, \cdots, x_D)^T \in R^D, \mu = (\mu_1, \cdots, \mu_k) \in R^{D\times k}, \pi = (\pi_1, \cdots, \pi_k) \in R^k$ and

$$p(\mathbf{x}|\mu_k) = \prod_{i=1}^{D} \mu_{ki}^{x_i}(1-\mu_{ki})^{(1-x_i)} \tag{2}$$

Scenario Description

How could we understand the above model more intuitively? In textbook, a scenario in machine learning is considered. We consider an MNIST dataset where each pixel's grayscale value has been binarized to 0 or 1. In this case, since the images in the MNIST dataset have a resolution of 28×28 pixels, here $D = 784$. N represents the number of images, and K represents the number of image class (without selection, $K = 10$). Our goal is to estimate the unknown parameters using maximum-likelihood.

R Implementation

First, since MNIST data is in grey scale $[0, 255]$, we have to convert it to a binary scale using a threshold of 128:

```
# Load MNIST data
mnist <- getOMLDataSet(data.id = 554)
mnist_data <- mnist$data
mnist_data <- mnist_data[, -ncol(mnist_data)] # Remove target column

# Convert to binary scale [0,1] with threshold 128
mnist3 <- as.data.frame(ifelse(mnist_data/128 >= 1, 1, 0))
```

Implementation of the function to plot the image:

```
# Function to plot MNIST data
show <- function(image) {
```

```
  image_matrix <- matrix(as.numeric(image), nrow = 28, byrow = TRUE)
  image_matrix <- t(apply(image_matrix, 2, rev)) # Rotate for proper
      orientation

  ggplot(data = reshape2::melt(image_matrix), aes(x = Var1, y = Var2, fill
      = value)) +
    geom_tile() +
    scale_fill_gradient(low = "white", high = "black") +
    theme_void() +
    theme(legend.position = "none") +
    coord_equal()
}
```

Implementation of the function to calculate the Bernoulli probability (2):

```
# Bernoulli probability calculation
bernoulli <- function(data, means) {
  N <- nrow(data)
  K <- nrow(means)
  prob <- matrix(0, nrow = N, ncol = K)

  for (i in 1:N) {
    for (k in 1:K) {
      prob[i, k] <- prod((means[k, ] ^ data[i, ]) * ((1 - means[k, ]) ^ (1
          - data[i, ])))
    }
  }

  return(prob)
}
```

Implementation of the function to calculate the responsibilities (E step)

$$\gamma(z_{nk}) = \frac{\pi_k p(\mathbf{x_n}|\mu_k)}{\sum_{j=1}^{k} \pi_j p(\mathbf{x_n}|\mu_j)} \tag{3}$$

```
# Responsibility calculation
respBernoulli <- function(data, weights, means) {
  # Step 1: Calculate p(x|means)
  prob <- bernoulli(data, means)

```

```
  # Step 2: Calculate numerator
  prob <- prob * matrix(weights, nrow = nrow(data), ncol =
      length(weights), byrow = TRUE)

  # Step 3: Calculate denominator
  row_sums <- rowSums(prob)

  # Step 4: Calculate responsibilities
  if (any(row_sums == 0)) {
    warning("Division by zero occurred in responsibility calculations!")
    return(prob)
  }

  prob <- prob / row_sums
  return(prob)
}
```

Implementation of the function used to re-estimate the parameters using the current responsibilities (M step)

$$\begin{cases} \mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(Z_{nk}) \mathbf{x_n} \\ \pi_k = \frac{N_k}{N} \end{cases} \tag{4}$$

where $N_k = \sum_{n=1}^{N} \gamma(Z_{nk})$:

```
# M-step for Bernoulli mixture
bernoulliMStep <- function(data, resp) {
  N <- nrow(data)
  D <- ncol(data)
  K <- ncol(resp)

  Nk <- colSums(resp)
  mus <- matrix(0, nrow = K, ncol = D)

  for (k in 1:K) {
    mus[k, ] <- colSums(resp[, k] * data)
    if (Nk[k] == 0) {
      warning("Division by zero occurred in Mixture of Bernoulli Dist
          M-Step!")
      next
    }
```

```
    mus[k, ] <- mus[k, ] / Nk[k]
  }

  return(list(weights = Nk / N, means = mus))
}
```

Implementation of log-likelihood calculation for determination of convergence:

$$E_Z[\ln p(\mathbf{X}, \mathbf{Z}|\mu, \pi)] = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk}) \left\{ \ln \pi_k + \sum_{i=1}^{D} [x_{ni} \ln \mu_{ki} + (1 - x_{ni}) \ln(1 - \mu_{ki})] \right\} \tag{5}$$

```
# Log-likelihood calculation
llBernoulli <- function(data, weights, means) {
  N <- nrow(data)
  K <- nrow(means)

  resp <- respBernoulli(data, weights, means)

  ll <- 0
  for (i in 1:N) {
    sumK <- 0
    for (k in 1:K) {
      temp1 <- (means[k, ] ^ data[i, ]) * ((1 - means[k, ]) ^ (1 - data[i,
          ]))
      temp1 <- log(pmax(temp1, 1e-50)) # Avoid log(0)
      sumK <- sumK + resp[i, k] * (log(weights[k]) + sum(temp1))
    }
    ll <- ll + sumK
  }

  return(list(ll = ll, resp = resp))
}
```

R function to run the EM algorithm:

```
# EM algorithm for mixture of Bernoulli distributions
mixOfBernoulliEM <- function(data, init_weights, init_means, maxiters =
    1000, relgap = 1e-4, verbose = FALSE) {
  N <- nrow(data)
  D <- ncol(data)
```

```
  K <- nrow(init_means)

  # Initialize
  weights <- init_weights
  means <- init_means
  ll_result <- llBernoulli(data, weights, means)
  ll_old <- ll_result$ll
  resp <- ll_result$resp

  for (i in 1:maxiters) {
    if (verbose && (i %% 5 == 0)) {
      cat(sprintf("iteration %d:\n", i))
      cat("  ", weights, "\n")
      cat(sprintf(" %.6f\n", ll_old))
    }

    # M Step
    mstep_result <- bernoulliMStep(data, resp)
    weights <- mstep_result$weights
    means <- mstep_result$means

    # Convergence check
    ll_result <- llBernoulli(data, weights, means)
    ll <- ll_result$ll
    resp <- ll_result$resp

    if (abs(ll - ll_old) < relgap) {
      cat(sprintf("Relative gap: %.8f at iterations %d\n", ll - ll_old, i))
      break
    } else {
      ll_old <- ll
    }
  }

  return(list(weights = weights, means = means))
}
```

R code to pick specific digits from the MNIST dataset:

```
# Function to pick specific digits
pickData <- function(digits, N) {
```

```
  set.seed(30)
  shuffled_idx <- sample(nrow(mnist3))
  sData <- mnist3[shuffled_idx, ]
  sTarget <- mnist$data[shuffled_idx, ncol(mnist$data)]

  selected <- sData[as.integer(sTarget) %in% digits, ]
  selected <- selected[sample(nrow(selected), N), ]
  return(selected)
}
```

Implementation of the experiments function:

```
# Experiment function
experiments <- function(digits, K, N, iters = 100, verbose = FALSE) {
  expData <- pickData(digits, N)
  D <- ncol(expData)

  initWts <- runif(K, 0.25, 0.75)
  initWts <- initWts / sum(initWts)

  initMeans <- as.matrix(expData[sample(nrow(expData), K, replace =
      FALSE), ])
  initMeans <- initMeans * (0.6 + 0.8 * matrix(runif(K * D), nrow = K,
      ncol = D))
  initMeans <- pmin(pmax(initMeans, 0.05), 0.95)

  return(mixOfBernoulliEM(expData, initWts, initMeans, maxiters = iters,
      relgap = 1e-15, verbose = verbose))
}
```

Note that in contrast to the python code which initialized the mean values using uniform distribution, I added randomness to the initialization of the mean values. In this way, the algorithm can be more stable, especially when the number of samples is small.

Clustering Performance Evaluation for Different Choice of K

Of course, we could easily run the code to produce the result illustrated in Figure 9.10 in the textbook. However, it is hard to judge the performance of our algorithm based on that figure.

Therefore, we could take a step further, and sort our data into training set and testing set. However, our task is somewhat different from traditional machine learning: our output can be seen as a result of **clustering**, rather than **classification**. In other words, it would be difficult to calculate the accuracy of our model on the testing set, since we only cluster the images in the testing set, without labeling them.

However, there are other ways to evaluate our clustering performance. Say that K types of digits are included. Since in this section, we assume we know the ground truth K, the mixture model would output K clusters.

We evaluate the performance of our mixture model by the metric **adjusted rand index**(ARI).[3] It is a widely used metric for the evaluation of clustering performance:

$$ARI = \frac{RI - E(RI)}{max(RI) - E(RI)} \tag{6}$$

which can be understood intuitively as normalizing RI into the range of $[-1, 1]$. Here,

$$RI = \frac{TP + TN}{TP + TN + FP + FN} \tag{7}$$

Where we consider the pairs in the testing set and categorize them into:

- True Positives (TP): These are pairs of elements that are correctly grouped together in both the predicted and actual clustering.
- True Negatives (TN): These are pairs of elements that are correctly separated in both the predicted and actual clustering.
- False Positives (FP): These are pairs of elements that are grouped together in the predicted clustering but were not grouped together in the actual clustering.
- False Negatives (FN): These are pairs of elements that are grouped together in the actual clustering but were not grouped together in the predicted clustering.

A positive ARI value indicates that the performance of the model is better than random grouping.

Now, we implement this in R.

First, we need to revise the pickData function to return the true label of the testing set:

```
pickData <- function(digits, N) {
  # Shuffle data and labels while maintaining correspondence
  set.seed(30)
  shuffled_indices <- sample(nrow(mnist3))
  sData <- mnist3[shuffled_indices, ]
  sTarget <- mnist.target[shuffled_indices]

  # Filter data for specified digits and collect corresponding labels
  filtered_data <- list()
  filtered_labels <- list()
  for (i in 1:nrow(sData)) {
    if (as.integer(sTarget[i]) %in% digits) {
      filtered_data <- append(filtered_data, list(as.numeric(sData[i, ])))
      filtered_labels <- append(filtered_labels, as.integer(sTarget[i]))
    }
  }

  # Convert to matrices
  filtered_data <- do.call(rbind, filtered_data)
  filtered_labels <- unlist(filtered_labels)

  # Randomly select N samples while maintaining data-label correspondence
  set.seed(30)
  selected_indices <- sample(nrow(filtered_data), N)

  selected_data <- filtered_data[selected_indices, ]
  selected_labels <- filtered_labels[selected_indices]

  return(list(data = selected_data, labels = selected_labels))
}
```

Then we implement a function to divide data into training set and testing set:

```
dataPartition <- function(digits, N1, N2) {
  # Get combined data and labels
  result <- pickData(digits, N1 + N2)
  data <- result$data
```

```
  labels <- result$labels

  # Split into training and testing sets
  training_set <- data[1:N1, ]
  testing_set <- data[(N1+1):(N1+N2), ]
  testing_set_labels <- labels[(N1+1):(N1+N2)]

  return(list(
    training_set = training_set,
    testing_set = testing_set,
    testing_set_labels = testing_set_labels
  ))
}
```

Implementation for generated the predicted cluster for each data sample in the testing set (simply picking out the largest value in the posterior distribution) and the implementation of ARI:

```
library(gtools) # For comb() function

convert_cluster <- function(posterior) {
  # Convert posterior probabilities to cluster assignments
  # Equivalent to np.argmax(posterior, axis=1)
  apply(posterior, 1, which.max) - 1 # Subtract 1 for 0-based indexing
}

adjusted_rand_index <- function(true_labels, pred_labels) {
  # Calculate the adjusted Rand index for clustering evaluation

  # Create contingency table
  contingency <- table(factor(pred_labels, levels = 0:max(pred_labels)),
                       factor(true_labels, levels = 0:max(true_labels)))

  # Convert to matrix
  contingency <- as.matrix(contingency)

  # Calculate sums
  sum_comb_c <- sum(comb(contingency, 2))
  sum_comb_a <- sum(comb(rowSums(contingency), 2))
  sum_comb_b <- sum(comb(colSums(contingency), 2))

```

```
  n <- length(true_labels)
  expected_ri <- sum_comb_a * sum_comb_b / comb(n, 2)
  max_ri <- 0.5 * (sum_comb_a + sum_comb_b)

  ari <- (sum_comb_c - expected_ri) / (max_ri - expected_ri)
  return(ari)
}
```

Finally, we note that the EM algorithm is in fact not very stable, since we initialize the initial value of π_k and $\mu_k i$ randomly. Sometimes, the algorithm would fall into locally optimal solution. Therefore, we choose to run the algorithm B times and calculate the mean and standard error of $ARI_i, i = 1, \cdots, B$.

```
# Run EM algorithm B times and calculate Adjusted Rand Index for each run
run_EM <- function(digits, K, N1, N2, B) {
  results <- numeric(B) # Initialize vector to store results

  for (i in 1:B) {
    # Partition data into training and testing sets
    partitioned_data <- dataPartition(digits, N1, N2)
    training_set <- partitioned_data$training_set
    testing_set <- partitioned_data$testing_set
    testing_set_labels <- partitioned_data$testing_set_labels

    # Run EM experiments and get final means
    finMeans <- experiments(K, training_set = training_set, verbose = TRUE)

    # Calculate cluster assignments for test data
    cluster_label <- convert_cluster(respBernoulli(testing_set,
        finMeans[[1]], finMeans[[2]]))

    # Compute Adjusted Rand Index and store result
    results[i] <- adjusted_rand_index(testing_set_labels, cluster_label)
  }

  return(results)
}

# Calculate mean and standard deviation while handling missing/zero values
mean_and_std <- function(vector) {
  # Convert input to numeric vector and handle missing values
```

```
  vec <- as.numeric(vector)

  # Filter out NA/NaN and zero values
  filtered_vec <- vec[!is.na(vec) & vec != 0]

  # Return NA if no valid values remain
  if (length(filtered_vec) == 0) {
    return(list(mean = NA, std = NA))
  }

  # Calculate mean and sample standard deviation (n-1 denominator)
  mean_val <- mean(filtered_vec)
  std_val <- sd(filtered_vec)

  return(list(mean = mean_val, std = std_val))
}
```

In practice, to test the performance of our mixture model on the task of k-clustering, we simply choose the digits $[0, \cdots, k-1]$. We make sure that for each digit category, there are approximately 1000 training images and 1000 testing images, i.e. $N_1 = N_2 = 1000k$. The number of experiments conducted for each clustering task is set to $B = 10$, and the experiment is conducted on a L4 GPU on colab.

The results are shown in Figure 1:

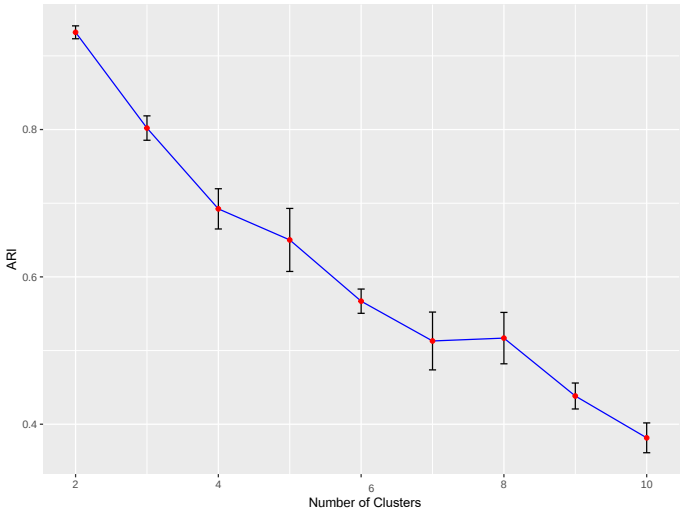

Figure 1: Model Performance

Note that the data is represented as mean and 95% confidence intervals. Here,

$$\begin{cases} ARI_{mean_i} = \sum_{j=1}^{B} ARI_{ij}/10 \\ ARI_{sd_i} = \sum_{j=1}^{B} (ARI_{ij} - ARI_{mean_i})^2/9 \\ ARI_{CI_i} = [ARI_{mean_i} - t_{\alpha/2,df} ARI_{sd_i}/\sqrt{10}, ARI_{mean_i} - t_{\alpha/2,df} ARI_{sd_i}/\sqrt{10}], \end{cases} \tag{8}$$

Where $\alpha = 0.05, df = 9, t_{\alpha/2,df} \approx 2.2622$.

From the figure above, it is obvious that the clustering performance decreases as the number of clusters increases, though the number of training images remains at approximately 1000. However, there isn't an obvious pattern for the width of 95% confidence intervals, subject to the value of K. This may be due to the small number of experiments we conducted on each choice of K ($B = 10$).

Criterion for the Selection of K

Now in this section, we assume that the ground truth K is unknown, and we try to come up with a criterion to estimate K.

First, since we could obtain the posterior likelihood:

$$lnp(\mathbf{X}|\mu, \pi) = \sum_{n=1}^{N} ln\{\sum_{k=1}^{K} \pi_k p(\mathbf{X_n}|\mu_k)\} \tag{9}$$

```
loglikelihood <- function(data, weights, means) {
  # Calculate probability of each sample under each component: p(x_n|_k)
  prob <- bernoulli(data, means) # Shape (N, K)

  # Weighted sum: sum_{k=1}^K _k p(x_n|_k)
  weighted_prob <- prob * matrix(weights, nrow = nrow(data), ncol =
      length(weights), byrow = TRUE)

  # Compute log-likelihood: sum_n ln{sum_k _k p(x_n|_k)}
  # Using log(sum) instead of logSumExp for simplicity (add protection if
      needed)
  log_likelihood <- sum(log(rowSums(weighted_prob)))

  return(log_likelihood)
}
```

One may want to choose K using the Akaike information criterion (AIC):[4]

$$\hat{k}_{AIC} = argmin_k\{2p_k - 2ln(\hat{L})\} \tag{10}$$

Or the Bayesian information criterion (BIC):[5]

$$\hat{k}_{BIC} = argmin_k\{p_k ln(n) - 2ln(\hat{L})\} \tag{11}$$

Here, $ln(\hat{L})$ represents the maximum value of the log-likelihood function, $p_k = kD+(k-1)$ represents the number of parameters included in the mixture model when $K = k$. Note that although there are k parameters included in $\pi = (\pi_1, \cdots, \pi_k)$, since $\sum_{i=1}^{k} \pi_i = 1$, the number of free valuable is only $k - 1$.

These two criteria both view $-2ln(\hat{L})$ as a cost function, with additional penalty terms $2p_k, ln(n)p_k$ to penalize the complexity of the model, respectively. However, after trying both criteria, I found that the cost function dominates overwhelmingly in both, and the penalty term hardly has any effect, leading them to consistently overestimate the value of K.

Therefore, we would probably need a different selection criterion. When the ground truth K is fixed, $ln(\hat{L})$ is a monotonically increasing function, and in practice, I have found that it is generally a concave function. Therefore, we could apply the **elbow criterion**. Actually, this criterion is widely applied to the selection of clustering number. The idea behind the elbow criterion is to identify the inflection point where the log-likelihood transitions from rapid growth to slow growth.

However, this may require manually identifying the inflection point. Is there an algorithm to automatically select the value K? Our method here is to draw a straight line linking the first point and the last point of the curve. Then, we find out the point that has the maximum distance to that line (Figure 2):

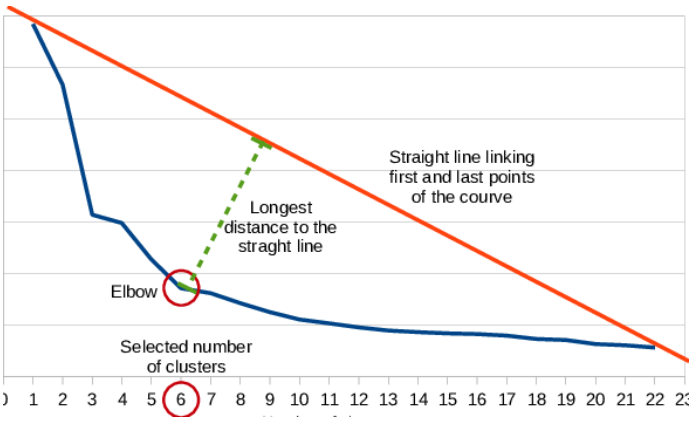

Figure 2: Illustration of the Elbow Criterion

Note that in contrast to Figure 2, our curve is monotonically increasing and concave, instead of monotonically decreasing and convex.

Implementation of the elbow criterion using R:

```
library(ggplot2)
library(kneed) # For KneeLocator equivalent

estimate_K_by_elbow <- function(digits, N, max_K = 10, iters = 50) {
  # Partition data into training and testing sets
  data_partition <- dataPartition(digits, N, N)
  training_set <- data_partition$training_set
  testing_set <- data_partition$testing_set
  testing_labels <- data_partition$testing_labels

  K_values <- 1:max_K
  log_likelihoods <- numeric(length(K_values))

  for (i in seq_along(K_values)) {
    K <- K_values[i]
    tryCatch({
      # Train model
      model <- experiments(K, training_set, iters = iters, verbose = FALSE)
      weights <- model$weights
      means <- model$means

      # Calculate log-likelihood (with stable implementation)
      if (K == 1) {
        # For K=1, use direct Bernoulli log-likelihood
        ll <- sum(training_set * log(means + 1e-100) +
              sum((1 - training_set) * log(1 - means + 1e-100))
      } else {
        # For K>1, use mixture log-likelihood
        ll <- loglikelihood(training_set, weights, means)
      }

      log_likelihoods[i] <- ll
      cat(sprintf("K=%d: log-likelihood=%.1f\n", K, ll))

    }, error = function(e) {
      cat(sprintf("K=%d failed: %s\n", K, conditionMessage(e)))
      log_likelihoods[i] <- -Inf # Mark failure
```

```
    })
  }

  # Automatic elbow detection
  kneedle <- KneeLocator(
    x = K_values,
    y = log_likelihoods,
    curve = "concave", # Log-likelihood curve is typically concave
    direction = "increasing"
  )
  best_K <- kneedle$elbow

  # Visualization
  elbow_plot <- ggplot(data.frame(K = K_values, LL = log_likelihoods),
      aes(x = K, y = LL)) +
    geom_line(color = "magenta", linewidth = 1) +
    geom_point(color = "magenta", size = 3) +
    labs(x = "Number of clusters (K)", y = "Log-Likelihood",
         title = "Elbow Method for K Selection") +
    theme_minimal() +
    theme(panel.grid.major = element_line(color = "gray", alpha = 0.3))

  if (!is.na(best_K)) {
    elbow_plot <- elbow_plot +
      geom_vline(xintercept = best_K, color = "red", linetype = "dashed",
          alpha = 0.5) +
      annotate("text", x = best_K + 0.1, y = median(log_likelihoods, na.rm
          = TRUE),
               label = paste("Elbow K =", best_K), color = "red")
  }

  print(elbow_plot)

  return(list(best_K = best_K, log_likelihoods = log_likelihoods))
}
```

Now, we test the performance of the elbow criterion. Set the ground truth of cluster numbers to $K = 4$, and run the R code:

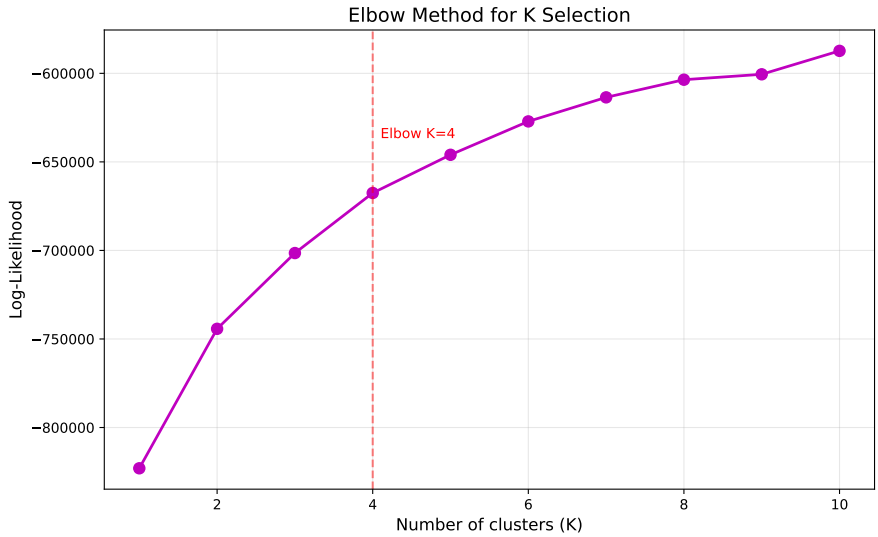

Figure 3: Elbow Criterion when $K = 4$

Set the ground truth of cluster numbers to $K = 5$, and run the R code:

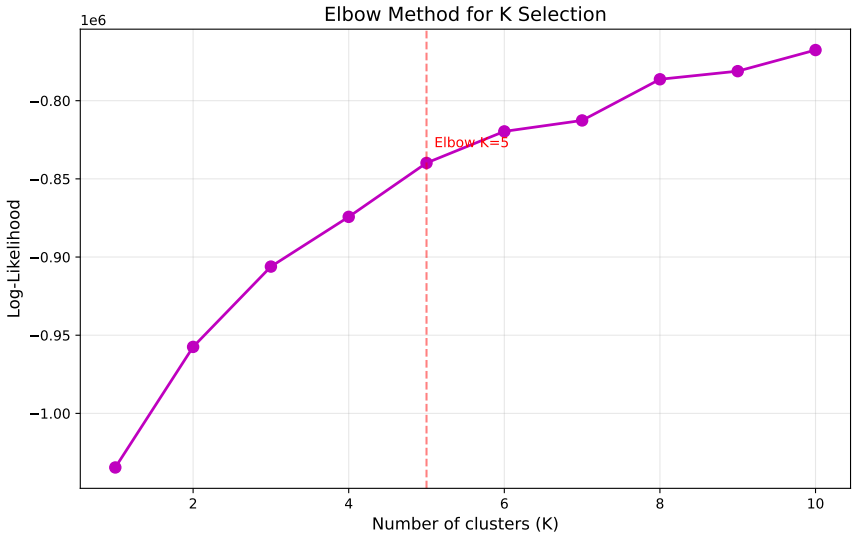

Figure 4: Elbow Criterion when $K = 5$

We could see that the elbow criterion successfully identifies the ground truth of K for both cases, demonstrating ideal performance.

Discussion

In this assignment, we applied the E-M algorithm to the MNIST dataset, and tested its clustering performance for different values of K. We could see that as K increases, the performance of our mixture model decreases rapidly. In fact, MNIST is a classical machine-learning dataset for classification, and current neural-networks achieves an accuracy of over 99% on testing sets.[6] It is obvious that neural-networks would outperform our traditional mixture model, which isn't surprising.

Moreover, we developed a criterion for estimation of the clustering number K using the elbow criterion. Due to time constraint, we only ran the algorithm two times, and both time we obtained the correct result. However, repetition is needed to further validify our method.

References

[1] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society: series B (methodological)*, 39(1):1–22, 1977.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, New York, 2006.

[3] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of classification*, 2:193–218, 1985.

[4] Hirotogu Akaike. Information theory and an extension of the maximum likelihood principle. In *Selected papers of hirotugu akaike*, pages 199–213. Springer, 1998.

[5] Gideon Schwarz. Estimating the dimension of a model. *The annals of statistics*, pages 461–464, 1978.

[6] Feiyang Chen, Nan Chen, Hanyang Mao, and Hanlin Hu. Assessing four neural networks on handwritten digit recognition dataset (mnist). *arXiv preprint arXiv:1811.08278*, 2018.