Module 5 Activity

Assigned at the start of Module 5

Due at the end of Module 5

Weekly Discussion Forum Participation

Each week, you are required to participate in the module's discussion forum. The discussion forum consists of the week's Module Activity, which is released at the beginning of the module. You must complete/attempt the activity before you can post about the activity and anything that relates to the topic.

Grading of the Discussion

1. Initial Post:

Create your thread by **Day 5 (Saturday night at midnight, PST).**

2. Responses:

Respond to at least two other posts by **Day 7 (Monday night at midnight, PST).**

Grading Criteria:

Your participation will be graded as follows:

Full Credit (100 points):

- Submit your initial post by **Day 5.**
- Respond to at least two other posts by **Day 7.**

Half Credit (50 points):

- If your initial post is late but you respond to two other posts.
- If your initial post is on time but you fail to respond to at least two other posts.

No Credit (0 points):

- If both your initial post and responses are late.
- If you fail to submit an initial post and do not respond to any others.

Additional Notes:

- **Late Initial Posts:** Late posts will automatically receive half credit if two responses are completed on time.
- **Substance Matters:** Responses must be thoughtful and constructive. Comments like "Great post!" or "I agree!" without further explanation will not earn credit.
- **Balance Participation:** Aim to engage with threads that have fewer or no responses to ensure a balanced discussion.

Avoid:

- A number of posts within a very short time-frame, especially immediately prior to the posting deadline.
- Posts that complement another post, and then consist of a summary of that.

Activity: Extending and Analyzing the Expectation-Maximization Algorithm

This activity is designed to deepen your understanding of the Expectation-Maximization (EM) algorithm and its computational efficiency. Below, you will find a code implementation of the EM algorithm for a Gaussian Mixture Model. Your task is twofold:

Task 1: Extend the Algorithm

Modify the code to run over multiple iterations. Implement:

- **A maximum iteration count** to limit the number of iterations.
- **A stopping criterion** based on convergence (e.g., a small change in parameter values between iterations).

Task 2: Analyze Runtime

Add line-by-line runtime comments to the code to measure the execution time of each section. Using these measurements, derive the **runtime complexity** of the algorithm in terms of the number of data points ((n)) and components ((k)).

Instructions

Part 1: Extending the Algorithm

1. Add a `for` loop or `while` loop to iterate over the Expectation-Maximization steps until:
   - A **maximum number of iterations** is reached (e.g., 100).

- The parameter values (`mu`, `sigma`, `pk`) converge, i.e., the change in values is below a predefined threshold (e.g., (10^{-4})).

2. Modify the algorithm to check for convergence at the end of each iteration:
   - Use a metric like the **norm of the difference in means** (`mu`) or the **log-likelihood** of the data.
   - Print the number of iterations the algorithm required before convergence.

Part 2: Analyzing Runtime

1. Import Python's `time` module and measure the runtime of each section of the code:
   - **Initialization**: Measure the time to compute the initial `mu`, `sigma`, and `pk`.
   - **E-Step**: Measure the time to compute probabilities and numerators.
   - **M-Step**: Measure the time to update the parameters.
2. Add comments next to each runtime measurement to document how long the operation took.
3. Using the runtimes, analyze the algorithm's complexity:
   - Consider (n), the number of data points.
   - Consider (k), the number of Gaussian components.
   - Derive the overall runtime complexity as a function of (n) and (k).

```
import numpy as np
import math

# Define the initial data columns
column_1 = np.array([1, 4, 1, 4])
column_2 = np.array([2, 2, 3, 3])

# Create a 2-column dataset
x = np.column_stack((column_1, column_2))
print(f"x {x}")

# Compute the means and sample standard deviations of each column
column_means = np.mean(x, axis=0)
print(f"column_means {column_means}")

column_stddevs = np.std(x, axis=0, ddof=1)  # Sample standard
deviation (ddof=1)
print(f"column_stddevs {column_stddevs}")

# Compute the average of the standard deviations across columns
average_of_values = np.mean(column_stddevs)

# Create an array of uniform standard deviations using the average
value
std_deviations = np.full_like(column_stddevs, average_of_values)
print(f"std_deviations {std_deviations}")
```

```python
# Initialize means (mu) based on the data
# cluster_mean = overall_mean + (k * overall_std_dev)
# k = np.array([-0.1867, 0.7257])
mu = ((column_means.reshape(1, -1) * np.array([1, 1]).reshape(-1, 1))
+
      (column_stddevs.reshape(1, -1) * np.array([-0.1867,
0.7257]).reshape(-1, 1)))
print(f"mu {mu}")

# Initialize standard deviations (sigma) for cluster
sigma = std_deviations.reshape(1, -1) * np.array([1, 1]).reshape(1, -
1)
print(f"sigma {sigma}")

# Initialize the prior probabilities (pk) i.e. Mixing Weights
pk = np.array([1, 1]).reshape(1, -1) / 2
print(f"pk {pk}")

# Define the Gaussian probability density function
def g(x, mu, sigma):
    temp = 1 / ((((2 * math.pi) ** 0.5) * sigma) ** 2)  # Gaussian
normalization constant
    temp2 = (np.linalg.norm(x - mu) / sigma) ** 2       # Squared
Mahalanobis distance
    temp3 = np.exp(-0.5 * temp2)                        # Exponential
factor
    return temp * temp3

# Helper function: Sum every k-th value in a list
def sum_every_kth_value_list(arr, k):
    result = []
    for i in range(k):
        sum_val = sum(arr[i::k])
        result.append(sum_val)
    return result

# Helper function: Sum values in chunks of size k
def sum_every_k_values(arr, k):
    if k <= 0 or len(arr) % k != 0:
        return "Invalid input"
    return [sum(arr[i:i + k]) for i in range(0, len(arr), k)]

# Helper function: Get the n-th set of k values from a list
def get_nth_set_of_k_values(arr, k, n):
    if k <= 0 or n <= 0:
        return "Invalid input"
    start_index = (n - 1) * k
    end_index = start_index + k
    return arr[start_index:end_index]
```

```
# E-step: Calculate numerators for updating probabilities
numerators = []
for j in range(0, mu.shape[0]):  # Iterate over each Gaussian
component
    for i in range(0, x.shape[0]):  # Iterate over each data point
        value = g(x[i], mu[j], sigma[0][j]) * pk[0][j]
        numerators.append(value)

# Compute denominators for normalizing probabilities
denominators = sum_every_kth_value_list(numerators,
int(len(numerators) / 2))

# Update probabilities for each data point and each component
new_p = []
for i in range(0, len(numerators)):
    new_p.append(numerators[i] / denominators[i % int(len(numerators)
/ 2)])

# Compute p_k_n (updated prior probabilities)
p_k_n = sum_every_k_values(new_p, int(len(numerators) / 2))

# M-step: Update the means (mu) and standard deviations (sigma)
new_mu = []
new_std = []
for i in range(1, 3):  # Iterate over components
    temp = 0
    temp2 = 0
    for j in range(0, x.shape[0]):  # Update mean
        temp += get_nth_set_of_k_values(new_p, 4, i)[j] * x[j]
    temp = temp / p_k_n[i - 1]
    new_mu.append(temp)

    for k in range(0, x.shape[0]):  # Update standard deviation
        temp2 += get_nth_set_of_k_values(new_p, 4, i)[k] *
(np.linalg.norm(x[k] - new_mu[i - 1]) ** 2)
    temp2 = (temp2 / (2 * p_k_n[i - 1])) ** 0.5
    new_std.append(temp2)

# Update prior probabilities
updated_p = []
for i in range(0, 2):
    updated_p.append(p_k_n[i] / int(len(numerators) / 2))

# Final updated parameters: new means, new standard deviations, and
updated priors
new_mu, new_std, updated_p

x [[1 2]
 [4 2]
```

```
 [1 3]
 [4 3]]
column_means [2.5 2.5]
column_stddevs [1.73205081 0.57735027]
std_deviations [1.15470054 1.15470054]
mu [[2.17662611 2.3922087 ]
 [3.75694927 2.91898309]]
sigma [[1.15470054 1.15470054]]
pk [[0.5 0.5]]

([array([1.6232562 , 2.47791081]), array([3.69828951, 2.5301904 ])],
 [np.float64(0.9302774776426431), np.float64(0.7291058883137361)],
 [np.float64(0.5774796490102341), np.float64(0.4225203509897659)])
```

Understand the given code

1.0 Analyze the data and initializing the paramters

Number of Clusters (or Mixtures): We have 2 clusters. Let's call them Cluster A and Cluster B.

Number of Features (or Dimensions) in our Data: Our data x has 2 columns (features). Each data point is a 2D coordinate, like [1, 2].

```
[[1 2]
 [4 2]
 [1 3]
 [4 3]]
```

1.1 Why mu is (2, 2)?

```
[[2.17662611 2.3922087 ]
 [3.75694927 2.91898309]]
```

The mean (mu) describes the center of a cluster. Since our data points live in a 2-dimensional space, the center of each cluster must also be a 2-dimensional coordinate.

1.2 pk (Mixing Weights) - Shape (1, 2)

```
[[0.5 0.5]]
```

pk represents the overall probability or proportion of the entire dataset belonging to a cluster. Therefore it is a single number per cluster.

- pk[0] = Probability of being in Cluster A (e.g., 0.5)

- pk[1] = Probability of being in Cluster B (e.g., 0.5)

1.3 sigma (Standard Deviations) - Shape (1, 2)

```
[[1.15470054 1.15470054]]
```

Here, we assumes a simple type of Gaussian distribution called a Spherical or Isotropic Gaussian.

This means we assume that the spread (standard deviation) of a cluster is the same in all directions (for all features).

So, for sigma, the code is designed to hold:

- A single standard deviation value that applies to all features of Cluster A.
- A single standard deviation value that applies to all features of Cluster B.

1.4 What would sigma look like if we didn't make this assumption?

If we allowed each feature in each cluster to have its own standard deviation (a "Diagonal" GMM), then sigma would have the exact same shape as mu: (2, 2).

```
# For a more complex "Diagonal" GMM, sigma would be (2, 2)
sigma = [[std_dev_for_cluster_A_feature1,
std_dev_for_cluster_A_feature2],
         [std_dev_for_cluster_B_feature1,
std_dev_for_cluster_B_feature2]]
```

2. Understand Helper functions

2.1 Gaussian probability density function

```
def g(x, mu, sigma):
    temp = 1 / ((((2 * math.pi) ** 0.5) * sigma) ** 2)  # Gaussian
normalization constant
    temp2 = (np.linalg.norm(x - mu) / sigma) ** 2       # Squared
Mahalanobis distance
    temp3 = np.exp(-0.5 * temp2)                        # Exponential
factor
    return temp * temp3
```

2.1.1 High-Level Concept: The "Likelihood" Calculator

\begin{align} G(x,\mu,\sigma,N) = \frac{1}{(\sigma\sqrt{2\pi})^N} \exp{\frac{-{(x-\mu)}^2}{2{\sigma^2}}} \end{align}

[Image Source: https://miro.medium.com/v2/resize:fit:683/1*9rdkuNxjx5bCACeo1CVjpA.png]

Imagine we have a single 2D "bell curve" (a Gaussian distribution) floating over a graph.

- mu: Is the (x, y) coordinate of the peak of the bell curve.
- sigma: Is a single number that describes how "wide" or "spread out" the bell is. A smaller sigma means a sharper, narrower peak. A larger sigma means a flatter, wider peak.
- x: Is a specific point on the graph, like [1, 2].

The function `g(x, mu, sigma)` answers the question: "What is the height of the bell curve at point x?"

This "height" isn't a probability (which is always between 0 and 1), but a probability density. A higher value means the point x is more "likely" or "plausible" to have come from this specific bell curve.

2.1.2 Gaussian normalization constant

```
    temp = 1 / ((((2 * math.pi) ** 0.5) * sigma) ** 2)  # Gaussian
normalization constant
```

- **Purpose:** This calculates the Normalization Term. Its job is to control the height of the bell curve's peak to ensure that the total volume under the 2D surface equals 1.
- **Math:** The code calculates 1 / (σ√(2π)), which simplifies to `1 / √(2π * σ^2)`.
- **Why this formula?** This is the correct normalization constant for a 2-dimensional isotropic Gaussian. For a 1D Gaussian, it would just be 1 / (σ√(2π)). The squaring (** 2) is what adapts it for 2D.

2.1.3 Squared Mahalanobis distance

\begin{align} M = \sqrt{\frac{(x-\mu)^2}{\sigma^2}} \end{align}

```
    temp2 = (np.linalg.norm(x - mu) / sigma) ** 2       # Squared
Mahalanobis distance
```

- **Purpose**: This calculates the Squared Distance Term, which measures how far away the point x is from the center mu, scaled by the standard deviation.
- **How it works:**
  - `x - mu`: Calculates the vector difference between the point and the mean. e.g., `[1, 2] - [2, 3] = [-1, -1]`.
  - `np.linalg.norm(...)`: Calculates the standard Euclidean (straight-line) distance of that vector. e.g., `norm([-1, -1])` is `sqrt((-1)^2 + (-1)^2) = sqrt(2)`.
  - `... / sigma:` Scales this distance. It answers "how many standard deviations away is the point?"
  - `(...) ** 2`: The result is squared.
- **Formal Name:** This is a simplified version of the Squared Mahalanobis Distance for an isotropic distribution.

2.1.4 Exponential factor

\begin{align} G(x,\mu,\sigma,N) = \frac{1}{(\sigma\sqrt{2\pi})^N} \exp{\frac{-M^2}{2}} \ end{align}

```
    temp3 = np.exp(-0.5 * temp2)                          # Exponential factor
```

- **Purpose:** This creates the actual "bell curve" shape.
- **How it works:**
  - It takes the distance we just calculated (temp2) and plugs it into the exponential function e^(-0.5 * distance).
  - If x is very close to mu, then temp2 is small, `-0.5 * temp2` is close to 0, and `exp(...)` is close to 1.
  - If x is far from mu, then temp2 is large, `-0.5 * temp2` is a large negative number, and `exp(...)` is close to 0.

2.1.5 Probabolity Density

```
    return temp * temp3
```

- **Purpose:** This combines the pieces.
- **How it works:** It multiplies the peak height of the curve (temp) by the exponential decay factor (temp3). The result is the final probability density (the "height" of the bell curve) at point x.

2.1.6 Usage

- This function is a self-contained calculator for the likelihood of a single point x belonging to a single, simple Gaussian cluster defined by mu and sigma. - In the EM algorithm, we would call this function repeatedly: once for each data point and for each cluster, to see how well each point "fits" into each of the possible clusters.

2.2 Calculating "Effective" Cluster responsibilities i.e. sizes

```
# Helper function: Sum every k-th value in a list
def sum_every_kth_value_list(arr, k):
    result = []
    for i in range(k):
        sum_val = sum(arr[i::k])
        result.append(sum_val)
    return result
```

This function, sum_every_kth_value_list, is a utility designed to be used within the M-Step (Maximization Step) of the EM algorithm.

2.2.1 High-Level Goal: Calculating "Effective" Cluster Sizes

In the M-Step, we need to update our model parameters (mu, sigma, pk). To do this, we first need to calculate a crucial value for each cluster, often called N_k:

- N_k = The "effective number of points" assigned to cluster k.
- This isn't a simple count. It's the sum of all the responsibilities that cluster k has across all data points.
- For example, if the responsibilities (the gamma matrix from the E-Step) for Cluster A are [0.8, 0.1, 0.7], then N_A = 0.8 + 0.1 + 0.7 = 1.6. It's as if Cluster A is "responsible for" 1.6 data points.
- This N_k value is the denominator used when updating the mean (mu) and standard deviation (sigma), and the numerator when updating the mixing weight (pk).

2.2.2 Why This Specific Function is Needed

We might think, "Can't we just sum the columns of the gamma matrix from the E-Step?" Yes, and np.sum(gamma, axis=0) would do that perfectly if gamma were a 2D matrix.

This helper function exists because the code it supports is likely passing the gamma matrix as a flattened, 1D array.

Let's see how. Imagine our gamma matrix from the E-Step looks like this for 3 data points and 2 clusters:

```
gamma = [[0.8, 0.2],   # Responsibilities for Point 0
         [0.1, 0.9],   # Responsibilities for Point 1
         [0.7, 0.3]]   # Responsibilities for Point 2
```

If you flatten this matrix into a 1D list (arr), it becomes:

```
arr = [0.8, 0.2, 0.1, 0.9, 0.7, 0.3]
```

Now, how do you sum the first column `(0.8 + 0.1 + 0.7)` and the second column `(0.2 + 0.9 + 0.3)` from this 1D list? That's exactly what sum_every_kth_value_list does!

2.3 Verification method to check the integrity of the output from the E-Step.

```
# Helper function: Sum values in chunks of size k
def sum_every_k_values(arr, k):
    if k <= 0 or len(arr) % k != 0:
        return "Invalid input"
    return [sum(arr[i:i + k]) for i in range(0, len(arr), k)]
```

The EM algorithm can run perfectly without it. It's a diagnostic tool for the programmer.

2.3.1 What the E-Step Guarantees?

The e_step function calculates the responsibility matrix (gamma). A fundamental property of this gamma matrix is that for any given data point, its responsibilities across all clusters must sum to 1.

Let's look at our gamma matrix:

```
gamma = [[0.8, 0.2],   # <-- Sum is 1.0. Point 0 is 80% A, 20% B.
         [0.1, 0.9],   # <-- Sum is 1.0. Point 1 is 10% A, 90% B.
         [0.7, 0.3]]   # <-- Sum is 1.0. Point 2 is 70% A, 30% B.
```

The E-Step's normalization guarantees that every row sums to 1.

2.3.2 How This Function Checks That Guarantee?

Now, imagine we flatten this gamma matrix into a 1D arr:

```
arr = [0.8, 0.2, 0.1, 0.9, 0.7, 0.3]
```

The function `sum_every_k_values` is designed to sum the values that correspond to each original row. It groups the flattened array into chunks and sums each chunk.

2.4 Inspect the results for one specific data point at a time

```
# Helper function: Get the n-th set of k values from a list
def get_nth_set_of_k_values(arr, k, n):
    if k <= 0 or n <= 0:
        return "Invalid input"
    start_index = (n - 1) * k
    end_index = start_index + k
    return arr[start_index:end_index]
```

This helper function, get_nth_set_of_k_values, is a utility designed to work with the flattened output of the E-Step. Its primary purpose is to inspect or process the results for one specific data point at a time.

2.4.1 High-Level Goal: "Zooming In" on a Single Data Point

- Imagine the E-Step has just finished. It produced a gamma matrix of responsibilities, which we then flattened into a single long list (arr).
- The purpose of this function is to answer the question: "What were the calculated responsibilities for the n-th data point?"
- It allows you to "un-flatten" the array for just one data point of interest.

2.4.2 Why It's Needed?

- This function is necessary because the data is stored in a non-intuitive, flattened format.
- If gamma were a 2D matrix, getting the responsibilities for the 3rd data point (index 2) would be simple: gamma[2, :].
- But since gamma is flattened into arr, we can't use simple indexing. We need a function to calculate the correct starting and ending position of the 3rd data point's information within that long list.

2.5 E Step

```
# E-step: Calculate numerators for updating probabilities
numerators = []
for j in range(0, mu.shape[0]):  # Iterate over each Gaussian
component
    for i in range(0, x.shape[0]):  # Iterate over each data point
        value = g(x[i], mu[j], sigma[0][j]) * pk[0][j]
        numerators.append(value)

# Compute denominators for normalizing probabilities
denominators = sum_every_kth_value_list(numerators,
int(len(numerators) / 2))

# Update probabilities for each data point and each component
new_p = []
for i in range(0, len(numerators)):
    new_p.append(numerators[i] / denominators[i %
int(len(numerators) / 2)])

# Compute p_k_n (updated prior probabilities)
p_k_n = sum_every_k_values(new_p, int(len(numerators) / 2))
```

This block of code is a fascinating and highly procedural implementation of the entire E-Step and the very first calculation of the M-Step.

2.5.1 Conceptual Overview

Let's map the code variables to their mathematical meaning in the EM algorithm. This is the key to understanding the code.

Code Variable	Mathematical Meaning	Role in E-M Algorithm
`numerators`	The joint probability `P(x_i, C_j)` for every point `i` and cluster `j`. This is `P(x_i | C_j) * P(C_j)`.	Intermediate step in E-Step.
`denominators`	The marginal probability `P(x_i)` for every point `i`. This is `Sum over all k of P(x_i, C_k)`.	Intermediate step in E-Step.
`new_p`	The responsibility `P(C_j | x_i)`. This is `P(x_i, C_j) / P(x_i)`.	**Final output of the E-Step.** This is the `gamma` matrix, but flattened.
`p_k_n`	The "effective number of points" `N_k` for each cluster `k`. This is `Sum over all i of P(C_k | x_i)`.	**First calculation of the M-Step.**

2.5.2 Calculating the Numerators (Joint Probabilities)

```
# E-step: Calculate numerators for updating probabilities
numerators = []
for j in range(0, mu.shape[0]):  # Iterate over each Gaussian
component (cluster)
    for i in range(0, x.shape[0]):  # Iterate over each data point
        value = g(x[i], mu[j], sigma[0][j]) * pk[0][j]
        numerators.append(value)
```

- **Goal**: Calculate P(x_i, C_j), the joint probability of seeing data point i AND it belonging to cluster j.
- **How it Works**: The code loops through each cluster first, and then through each data point. For each (point, cluster) pair, it calculates g(...) (the likelihood P(x_i|C_j)) and multiplies it by the cluster's prior probability pk (P(C_j)).
- **The Critical Detail (Data Layout):** Because the outer loop is over clusters, the numerators list is ordered in a "cluster-major" format. For 2 clusters (C0, C1) and 3 data points (x0, x1, x2), the list will look like: [P(x0,C0), P(x1,C0), P(x2,C0), P(x0,C1), P(x1,C1), P(x2,C1)]

2.5.3 Calculating the Denominators (Marginal Probabilities)

```
# Compute denominators for normalizing probabilities
denominators = sum_every_kth_value_list(numerators,
int(len(numerators) / 2))
```

- **Goal:** Calculate P(x_i), the total probability of observing data point i, regardless of which cluster it came from. This is the normalization factor needed for the E-Step. Mathematically, P(x_i) = P(x_i, C0) + P(x_i, C1).
- **How it Works:** This is where the cleverness comes in. It uses sum_every_kth_value_list on the strangely ordered numerators list. With k = N (number of data points), it correctly groups the values for each point.
  - The first element of denominators will be numerators[0] + numerators[N], which is P(x0,C0) + P(x0,C1) = P(x0).

  - The second element will be numerators[1] + numerators[N+1], which is P(x1,C0) + P(x1,C1) = P(x1). And so on.
- **Result:** denominators is a list [P(x0), P(x1), P(x2), ...].

2.5.4 Calculating new_p (The Responsibilities) - The E-Step Conclusion

```
# Update probabilities for each data point and each component
new_p = []
for i in range(0, len(numerators)):
    new_p.append(numerators[i] / denominators[i %
int(len(numerators) / 2)])
```

- **Goal**: This is the final step of the E-Step. It calculates the responsibility P(C_j | x_i), which is the probability of cluster j given data point i.
- **How it Works**: It divides each element in numerators by the corresponding element in denominators. The i % N logic correctly maps the i-th element of the "cluster-major" numerators list to the correct denominator for its data point.
  - e.g., numerators[N] (which is P(x0,C1)) is divided by denominators[0] (which is P(x0)) to get P(C1|x0).
- **Result:** new_p is the final, flattened responsibility matrix (gamma). This concludes the E-Step.

2.5.5 Calculating p_k_n (Effective Cluster Sizes) - The M-Step Begins

```
# Compute p_k_n (updated prior probabilities)
p_k_n = sum_every_k_values(new_p, int(len(numerators) / 2))
```

- **Goal:** This is the first calculation of the M-Step. It computes N_k, the "effective number of points" in each cluster. This is done by summing up the responsibilities for each cluster over all data points.
- **How it Works:** It uses sum_every_k_values to sum the new_p list in chunks.
  - The first chunk is the first N elements of new_p, which are all the responsibilities for Cluster 0: [P(C0|x0), P(C0|x1), P(C0|x2), ...]. Summing them gives N_0.
  - The second chunk gives N_1, and so on.
- **Result:** p_k_n (a slightly confusing name) is a list [N_0, N_1, ...]. This value is the crucial denominator for updating the means and standard deviations in the rest of the M-Step. It is also used to update the mixing weights pk.

2.6 M-Step (Maximization Step)

```
# M-step: Update the means (mu) and standard deviations (sigma)
new_mu = []
new_std = []
for i in range(1, 3):  # Iterate over components
    temp = 0
    temp2 = 0
    for j in range(0, x.shape[0]):  # Update mean
        temp += get_nth_set_of_k_values(new_p, 4, i)[j] * x[j]
```

```
    temp = temp / p_k_n[i - 1]
    new_mu.append(temp)

    for k in range(0, x.shape[0]):  # Update standard deviation
        temp2 += get_nth_set_of_k_values(new_p, 4, i)[k] *
(np.linalg.norm(x[k] - new_mu[i - 1]) ** 2)
    temp2 = (temp2 / (2 * p_k_n[i - 1])) ** 0.5
    new_std.append(temp2)

# Update prior probabilities
updated_p = []
for i in range(0, 2):
    updated_p.append(p_k_n[i] / int(len(numerators) / 2))

# Final updated parameters: new means, new standard deviations, and
updated priors
new_mu, new_std, updated_p
```

This block of code is the implementation of the M-Step (Maximization Step), the second major phase of the E-M algorithm cycle.

2.6.1 High-Level Goal: The "Updating" Step

- The purpose of the M-Step is to take the "soft assignments" or responsibilities calculated in the E-Step (new_p) and use them to compute a new, better set of model parameters. It "maximizes" the likelihood of the data given these responsibilities.

1. This specific code updates two of the three parameters:

- mu (the means of the clusters)
- sigma (the standard deviations of the clusters)

1. It does this by calculating a weighted average for each parameter.

2.6.2 Conceptual Formulas

The code is implementing the following update rules:

1. New Mean (μ_k): The new center of a cluster k is the weighted average of all data points, where the weight for each point is its responsibility to that cluster.

- new_μ_k = (Σ [P(C_k|x_i) * x_i]) / (Σ [P(C_k|x_i)])
- The denominator is simply N_k, which was already calculated as p_k_n.

1. New Variance (σ_k^2): The new variance of a cluster k is the weighted average of the squared distances of all points from the cluster's new mean.

- new_σ_k^2 = (Σ [P(C_k|x_i) * $||x_i - new\_\mu_k||^2$]) / (D * Σ [P(C_k|x_i)]) Where D is the number of dimensions of the data (in this case, D=2). The code then takes the square root to get the standard deviation σ_k.

2.6.3 Updating the Mixing Weights (pk)

```
updated_p = []
for i in range(0, 2):
    updated_p.append(p_k_n[i] / int(len(numerators) / 2))
```

- The mixing weight (p_k or π_k) represents our belief about the proportion of the entire dataset that belongs to a specific cluster k.
- For example, if we have two clusters, A and B, their mixing weights might be 0.4 and 0.6, meaning we believe 40% of the data comes from cluster A and 60% from cluster B.
- The goal of this code is to update these proportions based on the results of the E-Step.

2.6.4 Conceptual Formula

The update rule for the new mixing weight of a cluster k is very intuitive:

```
new_p_k = (Effective number of points in cluster k) / (Total number of points)
```

Mathematically, this is written as:

```
new_p_k = N_k / N
```

where:

- N_k = Σ [P(C_k|x_i)] (The sum of responsibilities for cluster k over all data points i).
- N = Total number of data points in the dataset.

2.6.5 final results of one full E-M iteration.

- new_mu: The list of updated means calculated in the previous code block.
- new_std: The list of updated standard deviations calculated in the previous code block.
- updated_p: The list of updated mixing weights calculated just now.

Task 1: Extend the Algorithm

Modify the code to run over multiple iterations. Implement:

- **A maximum iteration count** to limit the number of iterations.
- **A stopping criterion** based on convergence (e.g., a small change in parameter values between iterations).

```
# --- Iterative E-M Algorithm ---
# Define iteration controls
max_iterations = 100
convergence_threshold = 1e-5 # Stop when the change in means is very small
```

```
print("--- Initial Guesses for parametrs---")
print(f"Initial Means (mu):\n{mu}")
print(f"Initial Std Devs (sigma):\n{sigma}")
print(f"Initial Priors (pk):\n{pk}")

# Main loop for the E-M algorithm
for iteration in range(max_iterations):
    # Store old means to check for convergence later
    old_mu = mu.copy()

    # --- E-Step ---
    # (The logic here is identical to the original code)
    numerators = []
    # Note: mu.shape[0] correctly gets the number of clusters (k)
    for j in range(0, mu.shape[0]):
        for i in range(0, x.shape[0]):
            value = g(x[i], mu[j], sigma[0][j]) * pk[0][j]
            numerators.append(value)

    # Denominators are calculated based on the number of data points
    num_points = x.shape[0]
    denominators = sum_every_kth_value_list(numerators, num_points)

    # Update probabilities for each data point and each component
    new_p = []
    for i in range(0, len(numerators)):
        new_p.append(numerators[i] / denominators[i % num_points])

    # Compute p_k_n (updated prior probabilities)
    p_k_n = sum_every_k_values(new_p, num_points)

    # --- M-Step ---
    # (The logic here is identical to the original code, but we use
variables for clarity)
    new_mu = []
    new_std = []
    num_clusters = mu.shape[0]

    # Note: This loop was hardcoded for 2 clusters. A more general
solution
    # would be `for i in range(num_clusters):`
    for i in range(1, num_clusters + 1):
        temp = 0
        temp2 = 0
        for j in range(0, x.shape[0]):
            # Note: The '4' here was hardcoded for 4 data points.
            temp += get_nth_set_of_k_values(new_p, num_points, i)[j] *
x[j]
        temp = temp / p_k_n[i - 1]
        new_mu.append(temp)
```

```
        for k in range(0, x.shape[0]):
            # Note: The '4' here was hardcoded for 4 data points.
            temp2 += get_nth_set_of_k_values(new_p, num_points, i)[k]
* (np.linalg.norm(x[k] - new_mu[i - 1]) ** 2)
        # Note: The '2' here is hardcoded for 2 dimensions.
        temp2 = (temp2 / (2 * p_k_n[i - 1])) ** 0.5
        new_std.append(temp2)

    # Update prior probabilities
    updated_p = []
    for i in range(0, num_clusters):
        updated_p.append(p_k_n[i] / num_points)

    # --- Convergence Check & Parameter Update ---

    # Calculate the change in means. np.linalg.norm computes the
    #      Euclidean distance between the flattened old and new mean
matrices.
    change = np.linalg.norm(np.array(new_mu) - old_mu)

    print(f"Iteration {iteration + 1}: Change in means =
{change:.6f}")

    # Update the parameters for the next iteration
    # We must convert the lists back to numpy arrays with the correct
shapes
    mu = np.array(new_mu)
    sigma = np.array([new_std])
    pk = np.array([updated_p])

    # NEW: Check if the change is below the threshold
    if change < convergence_threshold:
        print(f"\nConvergence reached after {iteration + 1}
iterations.")
        break
    if iteration == max_iterations: # This belongs to the 'for' loop,
runs if the loop finishes without break
        print(f"\nMaximum number of iterations ({max_iterations})
reached without convergence.")

# --- Final Results ---
print("\n--- Final Learned Parameters ---")
print(f"Final Means (mu):\n{mu}")
print(f"Final Std Devs (sigma):\n{sigma}")
print(f"Final Priors (pk):\n{pk}")
```

```
--- Initial Guesses for parametrs---
Initial Means (mu):
[[2.17662611 2.3922087 ]
```

```
 [3.75694927 2.91898309]]
Initial Std Devs (sigma):
[[1.15470054 1.15470054]]
Initial Priors (pk):
[[0.5 0.5]]
Iteration 1: Change in means = 0.684225
Iteration 2: Change in means = 0.596798
Iteration 3: Change in means = 0.107111
Iteration 4: Change in means = 0.000001

Convergence reached after 4 iterations.

--- Final Learned Parameters ---
Final Means (mu):
[[1.  2.5]
 [4.  2.5]]
Final Std Devs (sigma):
[[0.35355339 0.35355339]]
Final Priors (pk):
[[0.5 0.5]]
```

Observations

The EM algorithm performed exceptionally well on this dataset. It demonstrated perfect convergence solution in just 4 iterations.

Assigning each data point to a cluster

Let's conclude the clustering conceptually first.

1. **Take the Final Parameters:** Use the converged mu, sigma, and pk values. These represent the "perfect" model of our data's structure.
2. **Calculate Final Likelihoods:** For each data point, calculate its likelihood of belonging to each of the two learned clusters.

- Likelihood_A = P(point | Cluster_A) calculated using g(point, mu_A, sigma_A).
- Likelihood_B = P(point | Cluster_B) calculated using g(point, mu_B, sigma_B).

1. **Calculate Final Probabilities (Responsibilities):** Convert these likelihoods into true probabilities by also considering the mixing weights (pk).

- Responsibility_A = pk_A * Likelihood_A
- Responsibility_B = pk_B * Likelihood_B
- Then, normalize these so they sum to 1 for each point.

1. **Make the Hard Assignment:** For each data point, assign it to the cluster for which it has the highest final probability (responsibility).

Clustering Implementation (Code)

The algorithm perfectly partitioned the data into two distinct, non-overlapping groups based on the values in the first feature."

```
# --- Concluding the Clustering ---
print("\n--- Final Cluster Assignments ---")

# The final learned parameters from the loop
final_mu = mu
final_sigma = sigma
final_pk = pk

# Create an empty list to store the cluster assignments for each data
point
assignments = []

# Iterate through each data point to assign it to a cluster
for point in x:
    # Calculate the unnormalized probability (likelihood * prior) for
each cluster
    prob_in_cluster_0 = final_pk[0, 0] * g(point, final_mu[0],
final_sigma[0, 0])
    prob_in_cluster_1 = final_pk[0, 1] * g(point, final_mu[1],
final_sigma[0, 1])

    # Assign the point to the cluster with the higher probability
    if prob_in_cluster_0 > prob_in_cluster_1:
        assignments.append(0) # Assign to Cluster 0
    else:
        assignments.append(1) # Assign to Cluster 1

# Print the results in a clear format
print("Data Point\t ->\tAssigned Cluster")
print("---------------------------------------")
for i, point in enumerate(x):
    print(f"{point}\t ->\tCluster {assignments[i]}")
```

```
--- Final Cluster Assignments ---
Data Point  ->  Assigned Cluster
---------------------------------------
[1 2]  ->  Cluster 0
[4 2]  ->  Cluster 1
[1 3]  ->  Cluster 0
[4 3]  ->  Cluster 1
```

Part 2: Analyzing Runtime

1. Import Python's `time` module and measure the runtime of each section of the code:
   - **Initialization**: Measure the time to compute the initial `mu`, `sigma`, and `pk`.
   - **E-Step**: Measure the time to compute probabilities and numerators.
   - **M-Step**: Measure the time to update the parameters.
2. Add comments next to each runtime measurement to document how long the operation took.
3. Using the runtimes, analyze the algorithm's complexity:
   - Consider (n), the number of data points.
   - Consider (k), the number of Gaussian components.
   - Derive the overall runtime complexity as a function of (n) and (k).

```
import time # Import the time module

init_start_time = time.time() # Start timer for initialization

# Initialize means (mu) based on the data
mu = ((column_means.reshape(1, -1) * np.array([1, 1]).reshape(-1, 1))
+
      (column_stddevs.reshape(1, -1) * np.array([-0.1867,
0.7257]).reshape(-1, 1)))

# Initialize standard deviations (sigma) for cluster
sigma = std_deviations.reshape(1, -1) * np.array([1, 1]).reshape(1, -
1)

# Initialize the prior probabilities (pk) i.e. Mixing Weights
pk = np.array([1, 1]).reshape(1, -1) / 2

init_end_time = time.time() # End timer for initialization
print(f"--- Parameter initialization took: {init_end_time -
init_start_time:.6f} seconds ---")


# Define iteration controls
max_iterations = 100
convergence_threshold = 1e-5 # Stop when the change in means is very
small

print("--- Initial Guesses for parametrs---")
print(f"Initial Means (mu):\n{mu}")
print(f"Initial Std Devs (sigma):\n{sigma}")
print(f"Initial Priors (pk):\n{pk}")

# Main loop for the E-M algorithm
total_e_step_time = 0
```

```
total_m_step_time = 0

for iteration in range(max_iterations):
    # Store old means to check for convergence later
    old_mu = mu.copy()

    # --- E-Step ---
    e_step_start_time = time.time() # Start E-Step timer

    # (The logic here is identical to the original code)
    numerators = []
    # Note: mu.shape[0] correctly gets the number of clusters (k)
    for j in range(0, mu.shape[0]): # Runtime complexity: Run k times
        for i in range(0, x.shape[0]): # Runtime complexity: Run n
times
            value = g(x[i], mu[j], sigma[0][j]) * pk[0][j] # Run time
complexity: Run d times (size of feature)
            numerators.append(value)
    # Runtime complexity for E step: O(k*n*d)
    # Denominators are calculated based on the number of data points
    num_points = x.shape[0]
    denominators = sum_every_kth_value_list(numerators, num_points)

    # Update probabilities for each data point and each component
    new_p = []
    for i in range(0, len(numerators)):
        new_p.append(numerators[i] / denominators[i % num_points])

    # Compute p_k_n (updated prior probabilities)
    p_k_n = sum_every_k_values(new_p, num_points)

    e_step_end_time = time.time() # NEW: End E-Step timer
    total_e_step_time += (e_step_end_time - e_step_start_time)


    # --- M-Step ---
    m_step_start_time = time.time() # Start M-Step timer
    # (The logic here is identical to the original code, but we use
variables for clarity)
    new_mu = []
    new_std = []
    num_clusters = mu.shape[0]

    # Note: This loop was hardcoded for 2 clusters. A more general
solution
    # would be `for i in range(num_clusters):`
    for i in range(1, num_clusters + 1): # Run time complexity: Run k
times
        temp = 0
        temp2 = 0
```

```
        for j in range(0, x.shape[0]): # Run time complexity: Run n
times
            # Note: The '4' here was hardcoded for 4 data points.
            temp += get_nth_set_of_k_values(new_p, num_points, i)[j] *
x[j] # run d times
        temp = temp / p_k_n[i - 1]
        new_mu.append(temp)

        for k in range(0, x.shape[0]): # Run n times
            # Note: The '4' here was hardcoded for 4 data points.
            temp2 += get_nth_set_of_k_values(new_p, num_points, i)[k]
* (np.linalg.norm(x[k] - new_mu[i - 1]) ** 2) # Run d times
        # Note: The '2' here is hardcoded for 2 dimensions.
        temp2 = (temp2 / (2 * p_k_n[i - 1])) ** 0.5
        new_std.append(temp2)

    # Update prior probabilities
    updated_p = []
    for i in range(0, num_clusters):
        updated_p.append(p_k_n[i] / num_points)

    m_step_end_time = time.time() # End M-Step timer
    total_m_step_time += (m_step_end_time - m_step_start_time)

    # --- Convergence Check & Parameter Update ---

    # Calculate the change in means. np.linalg.norm computes the
    #      Euclidean distance between the flattened old and new mean
matrices.
    change = np.linalg.norm(np.array(new_mu) - old_mu)

    print(f"Iteration {iteration + 1}: Change in means =
{change:.6f}")

    # Update the parameters for the next iteration
    # We must convert the lists back to numpy arrays with the correct
shapes
    mu = np.array(new_mu)
    sigma = np.array([new_std])
    pk = np.array([updated_p])

    # NEW: Check if the change is below the threshold
    if change < convergence_threshold:
        print(f"\nConvergence reached after {iteration + 1}
iterations.")
        break
    if iteration == max_iterations: # This belongs to the 'for' loop,
runs if the loop finishes without break
        print(f"\nMaximum number of iterations ({max_iterations})
reached without convergence.")
```

```python
# --- Final Results ---
print("\n--- Final Learned Parameters ---")
print(f"Final Means (mu):\n{mu}")
print(f"Final Std Devs (sigma):\n{sigma}")
print(f"Final Priors (pk):\n{pk}")

# NEW: Print runtime summary
print("\n--- Runtime Analysis ---")
print(f"Total E-Step time over {iteration+1} iterations:
{total_e_step_time:.6f} seconds")
print(f"Total M-Step time over {iteration+1} iterations:
{total_m_step_time:.6f} seconds")
```

```
--- Parameter initialization took: 0.000373 seconds ---
--- Initial Guesses for parametrs---
Initial Means (mu):
[[2.17662611 2.3922087 ]
 [3.75694927 2.91898309]]
Initial Std Devs (sigma):
[[1.15470054 1.15470054]]
Initial Priors (pk):
[[0.5 0.5]]
Iteration 1: Change in means = 0.684225
Iteration 2: Change in means = 0.596798
Iteration 3: Change in means = 0.107111
Iteration 4: Change in means = 0.000001

Convergence reached after 4 iterations.

--- Final Learned Parameters ---
Final Means (mu):
[[1.  2.5]
 [4.  2.5]]
Final Std Devs (sigma):
[[0.35355339 0.35355339]]
Final Priors (pk):
[[0.5 0.5]]

--- Runtime Analysis ---
Total E-Step time over 4 iterations: 0.000780 seconds
Total M-Step time over 4 iterations: 0.000742 seconds
```

Runtime Analysis

When running the code on given dataset (4 points), it measured total time as below for E & M Steps.

```
Total E-Step time over 4 iterations: 0.000787 seconds
Total M-Step time over 4 iterations: 0.000615 seconds
```

Algorithm Complexity

Let's define following variables

- n: The number of data points (x.shape[0]).
- k: The number of Gaussian components, or clusters (mu.shape[0]).
- d: The number of dimensions, or features, of each data point (x.shape[1]).
- I : The number of iterations until convergence.

E-Step Complexity (per iteration):

We have three loops

- For for components k
- Second for data points n
- Third for size of features for PDF function `g` d

Therefore overall time complexity would be O(k.n.d)

M-Step Complexity (per iteration):

We have three loops

- For for components k
- Second for data points n
- Third for size of features for function `get_nth_set_of_k_values` d

Therefore overall time complexity would be O(k.n.d)

Overall Runtime Complexity

Since iterative E-M code can run maximum `I` times therefore overall time complexity will be O(I.k.n.d)