香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

K-means and GMM Implementation from scratch

DDA3020 Machine Learning HW4 Report

Jingquan Li
119010148

Date: May 8, 2024

1 Implement K-means

For $\mathbf{x} \in \mathbb{R}^D$ and finite set $\mathcal{C} \subset \mathbb{R}^D$ define $d(\mathbf{x}, \mathcal{C}) := \min_{\mathbf{c} \in \mathcal{C}} \|\mathbf{x} - \mathbf{c}\|_2$.

For finite set $\mathcal{X} \subset \mathbb{R}^D$ and finite set $\mathcal{C} \subset \mathbb{R}^D$ define $\phi(\mathcal{X}, \mathcal{C}) := \sum_{\mathbf{x} \in \mathcal{X}} d^2(\mathbf{x}, \mathcal{C})$.

Given $\mathcal{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\} \subset \mathbb{R}^D$, k-means optimizes $\mathcal{C} \subset \mathbb{R}^D$, where $|\mathcal{C}| = K$, to minimize the following loss:

$$\mathcal{L}(\mathcal{C}) = \phi(\mathcal{X}, \mathcal{C}) = \sum_{\mathbf{x} \in \mathcal{X}} \min_{\mathbf{c} \in \mathcal{C}} \|\mathbf{x} - \mathbf{c}\|_2^2$$

1.1 Implement the solve_k_means() function

1.1.1 Code Overview

The provided solve_k_means() function takes a dataset x and an initial set of cluster centers c, and iteratively updates the cluster centers until convergence or until a maximum number of iterations (max_step) is reached. The goal is to group the dataset into K clusters, where each point belongs to the cluster with the nearest mean.

1.1.2 Function Signature

Inputs:

- x (`ndarray[float]`): An array with shape $[N, D]$, where N is the number of data samples and D is the dimensionality of each sample.
- c (`ndarray[float]`): An array with shape $[K, D]$, representing the initial K cluster centers.
- `max_step` (`int`, optional): The maximum number of iterations to perform, defaulting to 10,000.

Outputs:

- c (`ndarray[float]`): The updated cluster centers after the algorithm has converged.
- `index` (`ndarray[int]`): An array indicating the nearest cluster center for each data sample.
- `losses` (`list[float]`): A list tracking the loss at each iteration, where the loss is defined as the within-cluster sum of squares.

1.1.3 Implementation Details

Initialization:

- N and D are derived from the shape of x, and K from c.
- `index` is initialized to store the index of the nearest cluster center for each sample.
- `losses` is a list to track the loss over iterations.

Main Iterative Process:

- **Distance Calculation:** For each data point, the Euclidean distance to each cluster center is calculated using vectorized operations, avoiding explicit loops over dimensions or samples.
- **Re-assignment:** Each data point is assigned to the cluster center to which it has the smallest distance.
- **Cluster Center Update:** New cluster centers are computed as the mean of all data points assigned to each cluster.
- **Loss Calculation:** At each step, the loss is calculated as the sum of squared distances of each point to its nearest cluster center.
- **Convergence Check:** The algorithm checks if the updated cluster centers are close to the old centers using `np.allclose`, which compares the arrays element-wise within a tolerance.

1.1.4 Hyperparameter Choices

- **max_step**: Set to a default of 10,000 as a balance between allowing sufficient iterations for convergence and limiting computational cost.
- **Convergence tolerance in np.allclose**: Default settings are used, striking a balance between precision and performance.

1.2 Plot loss function $\mathcal{L}$

The function compute_wcss() computes the Within-Cluster Sum of Squares (WCSS) to evaluate the performance of the k-means clustering algorithm. The WCSS measures the compactness of the clusters and is calculated by summing the squared distances between each point and its nearest centroid.

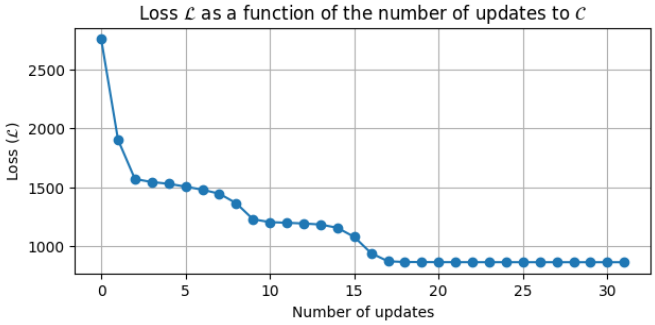

Figure 1: Loss function of the number of updates to c

1.3 Scatter plot of the training samples and cluster centers

1.3.1 Standard Normal Initialization:

This function initializes K cluster centers using random values drawn from a standard normal distribution.

- Input: K (number of cluster centers)
- Output: ndarray of shape $[K, 2]$ representing the initial cluster centers.

1.3.2 Cluster Initialization and Update

- `init_c = standard_normal_initialization(8)`: Initializes 8 cluster centers using the standard normal initialization function.
- `updated_c, index, losses = solve_k_means(train_x, init_c, max_step=1000)`: Solves the K-Means problem using the initial cluster centers, training data (`train_x`), and a maximum number of steps (`max_step=1000`). It returns the updated cluster centers (`updated_c`), index of cluster assignments for each data point (`index`), and the loss values during optimization (`losses`).

1.3.3 Choice of Hyperparameters

- `Number of Clusters (K)`: The number of clusters (K) is set to 8.
- `Maximum Steps (max_step)`: The maximum number of steps for the optimization process is set to 1000. This parameter determines the maximum number of iterations the algorithm will run before converging. Setting it too low may result in premature convergence, while setting it too high may increase computational overhead.

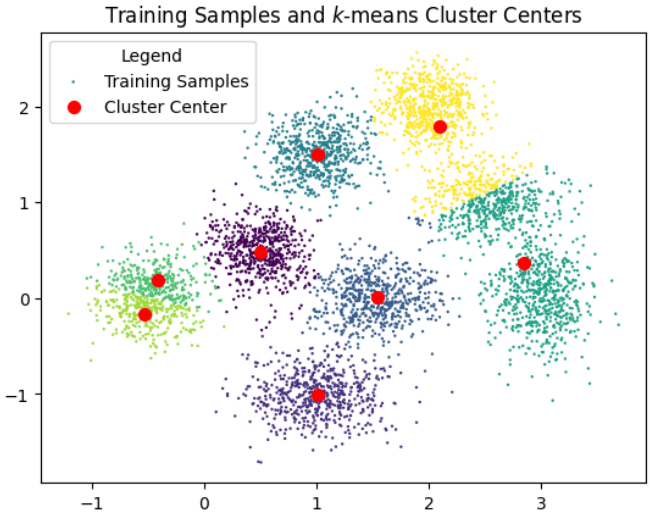

Figure 2: Training Samples and k-means Cluster Centers

1.4 Run k-means multiple times

Conduct a convergence analysis by running the K-Means algorithm multiple times (`num_runs`) with random initializations.

- `num_runs`: The number of times the K-Means algorithm is run with different initializations. In this case, it is set to 10.
- `converge_count`: A counter variable initialized to zero, which keeps track of the number of times the algorithm converges to the same solution as the initial one.
- `init_c`: The initial cluster centers are generated using the `standard_normal_initialization` function.
- `for _ in range(num_runs)`: Iterates through `num_runs` and runs the K-Means algorithm each time.
- `updated_c, index, losses = solve_k_means(train_x, init_c, max_step=1000)`: Solves the K-Means problem with the current initial cluster centers (`init_c`) and training data (`train_x`). It returns the updated cluster centers (`updated_c`), index of cluster assignments for each data point (`index`), and the loss values during optimization (`losses`).
- `if np.allclose(init_c, updated_c)`: Checks if the updated cluster centers are equal to the initial ones. If they are close enough, it implies that the algorithm has converged to the same solution.
- `converge_count += 1`: Increments the `converge_count` if convergence is detected.
- `init_c = updated_c`: Updates the initial cluster centers for the next iteration.
- `freq = converge_count / num_runs`: Calculates the frequency of convergence among all runs.

The calculated `freq` represents the proportion of runs in which the K-Means algorithm converged to the same solution as the initial one. In this case, the calculated frequency that the algorithm converges to a global minimum is 0.9.

2 Implement K-means++

Given $\mathcal{X}$, k-means++ generates a set of k vectors $\mathcal{I}$ in $\mathbb{R}^D$ using the following steps: 1. First, randomly pick a sample $\mathbf{x}$ from $\mathcal{X}$. And set $\mathcal{I} := \{\mathbf{x}\}$. 2. While $|\mathcal{I}| < k$. Sample $\mathbf{x} \in \mathcal{X}$ with probability $p(\mathbf{x}) := d^2(\mathbf{x}, \mathcal{I})/\phi(\mathcal{X}, \mathcal{I})$. And add the sampled $\mathbf{x}$ to set $\mathcal{I}$.

2.1 K-Means++ Initialization Method

This function initializes K cluster centers using the K-Means++ method.

2.1.1 Input

- x (`ndarray`): An array of shape $[N, D]$, storing N data samples with D as the feature dimension.
- K (`int`): Number of cluster centers to generate.

2.1.2 Output

- `centers` (`ndarray`): An array of shape $[K, D]$, representing the initial cluster centers generated by K-Means++.

2.1.3 Steps in K-Means++ Initialization

1. Randomly Select First Center: Choose the first center randomly from the data points.
2. Initialize Distance Array: Initialize an array to store the squared distances from each data point to the closest center.
3. Choose Remaining Centers:
   (a) Update the distance from each data point to the closest center.
   (b) Calculate probabilities for each data point based on the squared distances.
   (c) Choose the next center based on weighted probabilities.

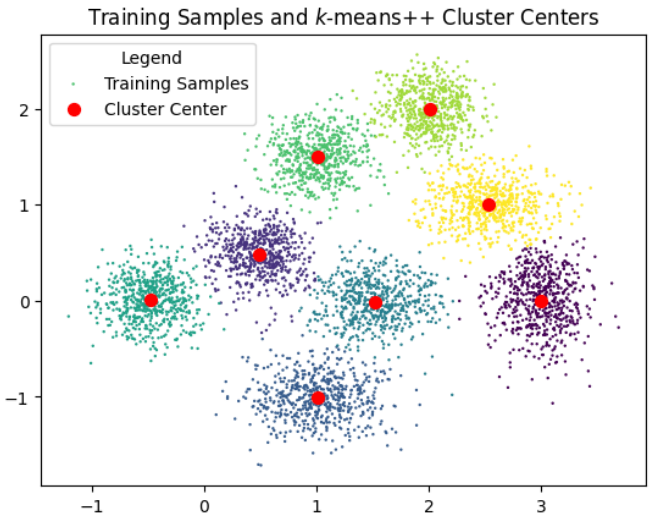

Figure 3: Training Samples and k-means++ Cluster Centers

2.2 Compare the loss and convergence speed

Put the loss function graph with standard normal initialization and k-means++ initialization, and compare their loss.We can see that the loss with k-means++ initialization is largely lower than that with standard normal initialization, which means the k-means++ initialization performs better and has higher efficiency while doing the clustering task.

We can also compare the convergence speed with standard normal initialization and k-means++ initialization. We can see from the graph that the convergence speed with k-means++ initialization is larger that that with standard normal initialization. To be specific, the clustering with k-means++ initialization converged at about the 7th update, but the clustering with standard normal initialization converges at about the 17th update.

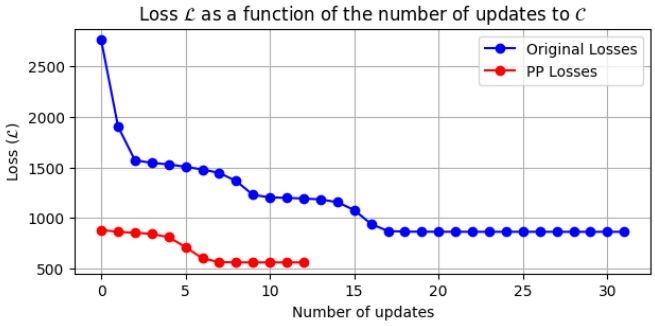

Figure 4: Loss $\mathcal{L}$ as a function of the number of updates to $\mathcal{C}$

3 Implement a Gaussian Mixture Model

In this task, I will implement the Expectation-Maximization (EM) algorithm for learning a Gaussian Mixture Model (GMM). Our GMM in $\mathbb{R}^2$ has a density given by:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k), \quad \mu_k \in \mathbb{R}^2, \quad \Sigma_k \in \mathbb{R}^{2\times 2}$$

3.1 Implemant EM algorithm

3.1.1 Code Overview

The GaussianMixtureModel class encapsulates an implementation of the Gaussian Mixture Model (GMM), a probabilistic model for representing normally distributed subpopulations within an overall population. The implementation uses the Expectation-Maximization (EM) algorithm to estimate the parameters that maximize the likelihood of the data given the model.

3.1.2 Class Structure

- **n_components**: The number of Gaussian distributions in the mixture.
- **n_iter**: Maximum number of iterations for the EM algorithm.
- **tol**: Convergence threshold for the log likelihood improvement.
- **means**: Current means of the Gaussian components.
- **init_means**: Initial means used for the EM algorithm.
- **weights**: Mixture weights for each Gaussian component.
- **covariances**: Covariance matrices for each component.
- **responsibilities**: Posterior probabilities of each component given the data.

3.1.3 Methods and their functionalities

- **initialize_parameters(self, X):** Initializes the model parameters using a k-means++ like initialization for means, sets equal weights, and uses the empirical covariance of the dataset for each component.
- **e_step(self, X):** Computes the responsibilities using the current model parameters. This step uses the multivariate normal PDF to evaluate the density of each data point under each component, weighted by the component's current weight.
- **m_step(self, X):** Updates the model's parameters based on the newly computed responsibilities. Calculates new weights as the average of the responsibilities across all data points. Updates the means as the weighted

average of the data points, with weights given by the responsibilities. Recomputes covariance matrices as the weighted outer product of the data deviations.

- **fit(self, X):** Coordinates the running of the EM algorithm, including initializing parameters and iteratively performing the E and M steps. Monitors convergence by checking if the change in log likelihood between iterations falls below the tolerance.
- **compute_log_likelihood_init(self, X)** and **compute_log_likelihood(self, X):** Computes the log likelihood of the data under the current model parameters, important for monitoring convergence and for model selection.
- **plot(self, X):** Utility function for visualizing the results of the Gaussian mixture model.

3.1.4 Hyperparameter choices

- **n_components**: Determines the complexity of the model by specifying the number of Gaussian distributions.
- **n_iter**: Sets a cap on the number of iterations to prevent excessive computation.
- **tol**: Controls the precision of convergence; smaller values result in more precise fits at the cost of potentially longer run times.

3.2 Generate a 2D contour plot of the GMM density

The resulting plot provides a clear visual representation of how well the GMM components fit the data distribution. It highlights areas of high probability density and how these areas correspond to the clustering of data points.

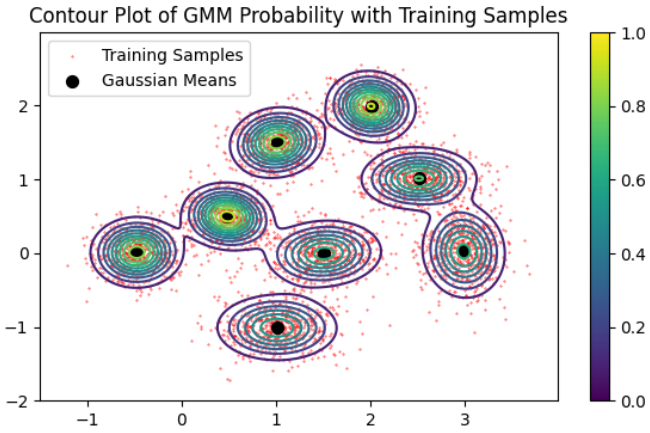

Figure 5: Contour Plot of GMM Probability with Training Samples

3.3 Report the log-likelihood on the training set and development set

The function gmm_log_prob computes the log probabilities of given data points under a Gaussian Mixture Model (GMM) using PyTorch's distribution libraries.

3.3.1 The log probability of the training set

```
[-1.0763839, -1.54256438, -1.09837806, ..., -1.37186707, -1.0407084, -2.96304398]
```

3.3.2 The log probability of the development set

```
[-3.01289871, -4.88739487, -1.07392199, -2.01393966, -2.14995957, -1.21085158,
-3.38868284, -1.37573343, -1.26428815, -2.95509965, -1.68947912, -2.55407091,
-4.98363017, -1.26859155, -1.76087422, -1.87492613, -0.96467143, -2.09768647,
```

```
-1.88104261, -2.79928273, -2.24732301, -1.56953075, -2.3042276, -1.22961184,
-1.51911158, -1.39717186, -2.70263953, -1.2115244, -1.5618419, -4.03277753,
-2.70584515, -1.10858013, -1.61158534, -1.68130234, -1.45925973, -2.68950916,
-2.79277004, -1.88262046, -1.40505877, -2.21963341, -2.04068837, -3.00133505,
-2.21992579, -3.12856726, -2.58021589, -1.62513441, -1.38604897, -1.51362573,
-2.25566364, -1.61406873, -1.0153317, -1.85360387, -1.80630432, -2.07345289,
-1.93103654, -2.49862097, -1.76349746, -2.1298992, -1.80916879, -3.16832383,
-5.59332091, -1.14429242, -1.87010351, -2.59564254, -1.47673363, -1.2922715,
-0.96234884, -1.91109783, -2.22535778, -1.80271351, -2.59757653, -2.11488784,
-2.79615348, -1.7536742, -1.47598131, -5.83516238, -3.48589668, -1.17864577,
-1.43565769, -2.26288939, -3.30993108, -1.17586267, -2.75953804, -2.4461565,
-1.36348829, -1.43773212, -2.48290689, -2.04786243, -2.72534538, -2.27232864,
-2.08175001, -1.8345003, -2.3468446, -2.92265787, -1.49162983, -1.60214014,
-3.08305393, -0.99900345, -3.1781902, -1.15244191, -5.04278189, -0.96535059,
-1.41211923, -2.15288795, -2.30826709, -4.20806702, -2.37590817, -1.43207572,
-1.91537347, -0.95009799, -2.5980243, -2.45175154, -3.92277177, -2.85671865,
-1.58637683, -2.5764404, -2.08453045, -1.33623315, -1.35174576, -2.78978737,
-5.08351666, -1.933459, -0.99620505, -1.00624576, -1.74215811, -8.01509903,
-1.08825007, -2.0378159, -2.72353339, -1.9166893, -1.3763935, -2.94626297,
-2.61224128, -1.43918016, -2.33608867, -1.21240287, -2.51972785, -2.38931699,
-2.22677916, -2.20828412, -1.41079291, -1.1147447, -3.38543888, -4.6671017,
-2.25474493, -1.91855516, -1.23912108, -2.26197829, -1.5899188, -3.52805363,
...
-1.10124684, -1.08899235, -0.99726571, -2.85781293, -7.2339675, -3.47261716,
-3.98460437, -1.0371552, -1.35785851, -2.80623583, -1.57769946, -1.06214429,
-2.07913459, -1.78010699, -3.1686996, -1.66845292, -1.96161519, -2.62642614,
-1.24109268, -1.68223608]
```