MGSC695 OPTIMIZATION FOR DATA SCIENCE
Final Project Report

McGill University
Desautels Faculty of Management

Gaussian Mixture Model Report

Author: Anqi (Angela) Chen
Student Number: 261044081
Professor: Sanjith Gopalakrishnan

February 25, 2022

1 Introduction

Gaussian Mixture Model (GMM) is a type of clustering algorithm that aims to optimize normally distributed subgroups within an overall population [2]. It is a mixture of Gaussian distributions, which means the multivariate normal distribution. Mixture models offer a distributed-based clustering approach, and it will have greater power to cluster data in a more complex setting [5].

In this project, I implemented the Gaussian Mixture Model from scratch using Python and compared the results with the Gaussian Mixture black-box algorithm from *sklearn* through four unit test cases. Unit tests are expanded across datasets with different numbers of dimensions and features, as well as the use case of image compression problem.

Overall, my implementation provides a highly accurate result compared to the results from the black-box algorithm, despite some randomness from the initialization step. However, the use case of the image compression problem will require future improvements.

2 Gaussian Mixture Model in detail

As mentioned above, Gaussian Mixture Model provides a distributed based clustering approach, where it optimizes normally distributed subgroups. Gaussian represents multivariate normal distribution $N(\mu, \Sigma)$. And mixture model represents a mix of several distributions, each with a different mean vector μ and a covariance matrix Σ. Overall, it represents a distribution as [1]:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \tag{1}$$

With π_k the mixing coefficients, where

$$\sum_{k=1}^{K} \pi_k = 1 \text{ and } \pi_k \geqslant 0 \ \forall k \tag{2}$$

Within one class, maximum likelihood estimation can be used to estimate the mean and variance of the corresponding normal distribution. Since the formulation of the normal distribution is the following [4]:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{3}$$

The maximum likelihood estimation can be written as follows:

$$\ln p(X|\pi,\mu,\Sigma) = \sum_{n=1}^{N} \ln(\sum_{k=1}^{K} \pi_k \mathcal{N}(x^{(n)}|\mu_k,\Sigma_k)) \text{ w.r.t } \Theta = \{\pi_k,\mu_k,\Sigma_k\} \tag{4}$$

In this setting, this is a closed form expression, where it uses a finite number of standard operations.

Since there is no labeled dataset as it is an unsupervised learning method, a latent variable (z) is introduced [3]. It is a hidden variable that represents which class any of the observations belong to. It makes the assumption that observations are caused by some underlying unobservable variable, which can be inferred from other variables or parameters. It can be represented as: $z \sim Categorical(\pi)$, (where $\pi_k \geq 0, _k \pi_k = 1$)

And thus the distribution will be [1]:

$$p(x) = \sum_{k=1}^{K} p(x,z=k) = \sum_{k=1}^{K} p(z=k)p(x|z=k) \tag{5}$$

The latent variables are introduced to model complex dependencies between variables to simplify the problem.

The Gaussian Mixture Model incorporates the Expectation-Maximization(EM) algorithm. Formally, the EM algorithm tries to find maximum likelihood estimates for models with latent variables. Roughly speaking, it iterates through updating posterior probabilities and updating new mean vector and covariance matrices. To look into the detail of the EM algorithm, there are four steps [6]:

1. Initialization step
   In this step, the mean vectors μ_j and covariance matrices Σ_j are initialized as the starting point of all the gaussians.

2. Expectation step
   The centroid μ_j and covariance matrix Σ_j are holding to be fixed. This step will update the likelihood or weight of each observation being observed in each of the Gaussians. More specifically, the expectation step will calculate the probability that X_i came from each class and normalizing it:

$$E[z_{i,j}] = \frac{P(x=x_i|\mu=\mu_j)}{\sum_{m=1}^{k} P(x=x_i|\mu=\mu_m)} \tag{6}$$

3. Maximization step
   Once the expectation step is finished, the mean μ_j and variance Σ_j of each Gaussian distribution will be re-calculated, holding the new weights from previous step

constant. In another word, each Gaussian gets a certain amount of posterior probability for each datapoint. The formulation is the following:

$$\mu_j = \frac{\sum_i E[z_{i,j}]x_i}{\sum_i E[z_{i,j}]} \tag{7}$$

$$\sigma_j = \frac{\sum_i E[z_{i,j}](x_i - \mu_j)(x_i - \mu_k)}{\sum_i E[z_{i,j}]} \tag{8}$$

4. Iterating step
   Set the tolerance to be some small ϵ. Iterate step 2 and 3 until the change in log-likelihood gets smaller than the tolerance.

In terms of the underlying optimization problem for the Gaussian Mixture Model, the model essentially is a mixture of probabilities of each datapoint to each Gaussian. It utilizes normal distribution and uses the mean to represent the centroid of each cluster. The objective is to use maximum likelihood estimation to estimate the mean and variance of the corresponding normal distributions. Therefore, the underlying optimization is a convex optimization problem that can be solved by minimizing the negative log-likelihood function. Another perspective is that the goal is to minimize the loss function for the normal distributions.

3 Technical Approaches

As the details of the Gaussian Mixture Model mathematical formulation are illustrated in the above section, this section will focus on the approach I used when I constructed the Gaussian Mixture Model using Python [7], and discuss the underlying optimization problem from a technical perspective.

I constructed the Gaussian Mixture Model using Python in a class format with four unit test cases. Each unit test is a different dataset, and it would be interesting to compare the output from my algorithm to the ones using the black-box implementation. Initially, I installed *pandas* and *numpy* packages to help with data structures and various mathematical calculations.

Then, I created the *gaussian_mixture_model* class with three parameters: *k*, *max_iter*, and *precision*. *k* is the number of gaussian distributions, i.e., the number of clusters. *Max_iter* is the maximum number of iterations, which is the first stopping criteria. And *precision* is the second stopping criteria, indicating a threshold for the algorithm to stop once the change in likelihood function is smaller than it. These three parameters are then initialized, with a default maximum iteration of 100 and a precision of 0.00001.

Next, I constructed a function for initialization. The mean vector is chosen randomly using the random package, and it is stored in the form of a list. The covariance matrix is created using the *NumPy cov* function, which estimates a covariance matrix, given data and weights. The length of the mean and covariance depends on the value of k, which is the number of Gaussians. Moreover, I initialized a vector of ϕ and a vector of weight. The vector of ϕ represents the cluster weight assigned to each cluster. And the vector of weight is the expectation or the weight, meaning the likelihood of i^{th} observation belongs to cluster j. These two vectors are assigned with the initial value of 1/k for simplicity.

I also defined a function to calculate the log-likelihood. The function will return the weight and the likelihood. Say the data is a m * n dimensional matrix, then the likelihood is a m * k dimensional matrix. Each row of the likelihood matrix will be for one observation, and each column is the likelihood of that observation for each cluster. The likelihood is calculated using the *multivariate_normal* and the pdf function under the *scipy.stats* package. Once the likelihood is calculated, weight or the expectation is calculated according to the formula (6).

The next step is to define the expectation step. The goal is to update ϕ and weight by holding μ and Σ of each Gaussian distribution fixed. It is straightforward to update the weight by calling the log-likelihood function I defined above. The ϕ is the average of the weight across each observation. The function will again return the likelihood.

Now it is time to define the maximization step. The objective is to update the mean and variance of each Gaussian distribution by holding ϕ and weights constant. According to the formula of mean (7) and variance (8), the mean vector and covariance matrices are updated by iterating through each cluster. This function does not return anything.

I also created two functions to visualize the changes if the dataset is a two-dimensional dataset. The first function init_glance is just an ordinary function to show the standard initial scatter plot of the data points. The second function, visualization, shows the change in mean and variance through iterations. Each cluster will be displayed in a different color, and the function will show as many changes in visualization as the number of iterations. There will be a sample visualization output in the following result section.

Last but not least, I created a final gmm function to perform the iteration step. Using an initial current iteration of 0 and a delta of 50, the function calls the initialization function to instantiate. Then, if the data is two-dimensional, it will plot the 2-D scatter plot both initially and iteratively. I used a while loop, holding the condition of the current iteration is smaller than the maximum iteration and the current delta is greater than precision. Then the function calls the expectation step and maximization step and updates the current iteration and delta. Eventually, the function will print out the

final mean vector and covariance matrix, and show the visualization of optimization if the dataset is two-dimensional. Otherwise, the function will still show the results numerically, but will not show any visualization. Also, the function will return the final labels of each datapoint, indicating which Gaussian cluster each data point is assigned to.

4 Results

Overall, I used four datasets for unit tests. The datasets are different from each other in terms of dimension and use cases, so I selected them to examine the accuracy across different cases.

4.1 *Clustering_gmm* dataset from Homework 6

This is dataset has 500 rows and 2 columns, with each column representing weight and height. Using the class function to run through this dataset, the dataset converged at the 61^{st} iteration if the k is equal to 4. Here is the output of the final converged visualization:

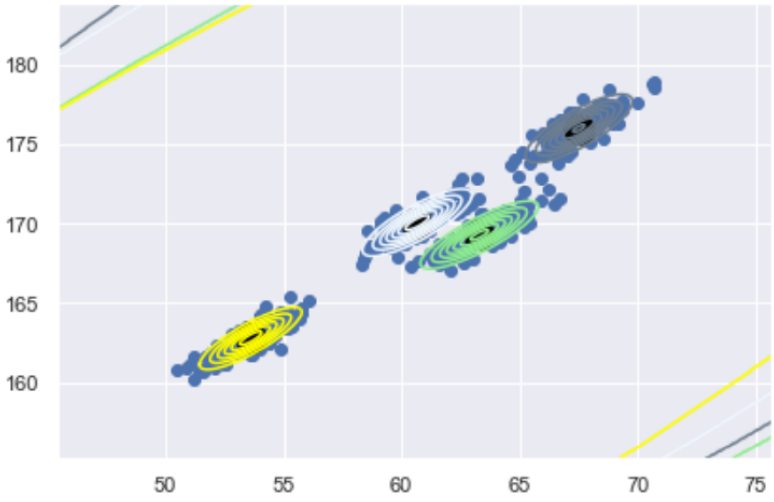

Figure 1: Final clustering result using *Clustering_gmm* dataset when k = 4

As we can see, each color represents a Gaussian distribution, and the black dot in the middle is the centroid of that cluster. The ovals represent the variance of each distribution. It is clear that all the data points are assigned to one Gaussian, and the oval of each Gaussian can cover the majority of the data points in each cluster.

If we were to compare it with black-box implementation, the data points are clustered in the same way if we set the same k. Similarly, if we were to compare how the algorithm

labels the datapoints to each cluster, these two algorithms will have the exact same results. Overall, my method from scratch produces the result as the black-box method according to this use case.

4.2 *Iris* data from *sklearn* dataset package

The Iris dataset has 150 rows and 4 columns. The reason to use this dataset is that it has four features instead of two. Therefore, it will not provide a visualization, but instead, we can compare the labels to see how the algorithm performs.

Overall, the model from scratch has a good performance. However, the final label could be slightly different at each run due to different initialization. From this use case, we can say that the model from scratch can also perform well with greater features.

4.3 *Faith* dataset from Homework 6

This dataset also comes from Homework 6. It has a shape of 272 observations and 2 features. Therefore, we can visualize the result for this use case. By setting the k equal to 2, the algorithm stopped at iteration 18. Here is the output visualization:

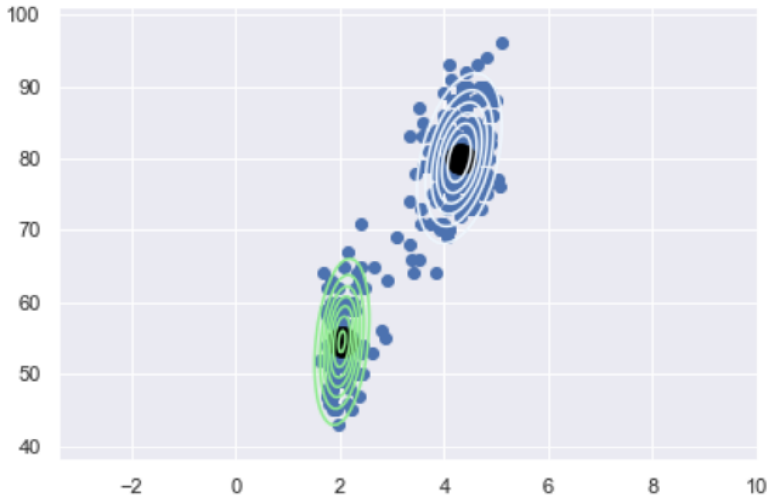

Figure 2: Final clustering result using *Faith* dataset when k = 2

As we can see, the algorithm separates the data points into two clear clusters, with the black dot in the middle being the centroids. The number of iterations it takes also varies for each run due to different initial assignments.

And if we were to compare the labels and result with the black-box implementation, we can see that the two algorithms have the same labels. Therefore we can conclude that

the model from scratch performs very well on datasets with two features. And as the number of features increases, the performance will decrease slightly depending on each run.

4.4 Image compression problem from Homework 6

The dataset for the image compression problem has a shape of (427, 640, 3). It is a three-dimensional data array. To simplify the problem, it is transformed into a two-dimensional array, where the first two dimensions are multiplied. So the data array that will be used has a shape of (273280, 3). It is interesting to see how will the algorithm from scratch perform on such a large dataset.

This use case causes some issues during different runs. Sometimes it will run successfully, and sometimes it will encounter a singular matrix error. And if we look closely, the delta does not necessarily decrease for each iteration, instead, it is floating up and down for each run. This indicates that the algorithm is having difficulty converging using the image compression dataset. This will be discussed more in the challenges sections.

Throughout the four use cases, we can conclude that the algorithm I implemented from scratch does a very good job on datasets with two features, and it does a relatively good job on datasets with more than two features. In terms of the image compression problem, there are still some future improvements to do.

5 Challenges

There were many challenges throughout the whole project. I have overcome most of them, but some parts can still be improved.

5.1 The use of class or function format

Writing the algorithm in class or function format has its distinct pros and cons. A class can share common parameters and attributes, so there is no need for repetition. On the other hand, it is more difficult to test out a specific function within the class in this case. On the other hand, writing the algorithm in separate functions is easier to be tested. However, since attributes such as mu vector and covariance matrices are shared across all the functions, it would be burdensome to repetitively input these variables in each function. Eventually, I decided to write it as a format of class for professionalism and clarity.

5.2 Different symbols and formulation across different resources

The mathematical formulation of the Gaussian Mixture Model can be rigorous. There are different formulations across different resources. I took some time to read over all the resources I can find and combined the formulation section in an easily to understand, but detailed enough format.

5.3 Challenges on data structure

The algorithm extensively used darray and array format from the *NumPy* package. In some functions, it was relatively difficult to record and calculate the shape of the matrix multiplication. It could be time-consuming to accurately calculate the matrix size and to design the matrix multiplications.

5.4 Image compression problem

This problem was my last use case, where I import the same image data from Homework 6. However, it encounters a singular matrix issue during each execution. Also, the delta for each iteration tends to decrease in the first few iterations, and increase and eventually cause the error.

According to resources online, the problem could be caused when the algorithm tries to take the inverse of a matrix for which the determinant is zero. In future studies, there should be a checkpoint of whether the determinant of the matrix is zero. This issue has to be solved before a successful execution of the algorithm.

In conclusion, Gaussian Mixture Model is an unsupervised learning algorithm to perform soft clustering that incorporated the EM algorithm. It is a powerful clustering algorithm that is able to provide soft probabilities to the data point, and it can provide better clustering results compared to hard clustering methods such as K-means [8]. There is still some limitation of this algorithm, such as it requires a parameter of k, and there is no hard metrics like AUC score to compare accuracy. The algorithm I implemented from scratch does a good job on simpler use cases. However, it requires additional improvements in terms of the image compression problem in the future.

References

[1] csc 411: introduction to machine learning - lecture 16: gmm. https://www.cs.utoronto.ca/~mren/teach/csc411_19s/lec/lec16.pdf. pages 2, 3

[2] Gaussian mixture model. https://brilliant.org/wiki/gaussian-mixture-model/. pages 2

[3] Latent variable models, part 1. https://krasserm.github.io/2019/11/21/latent-variable-models-part-1/. pages 3

[4] Table of contents. http://ethen8181.github.io/machine-learning/clustering/GMM/GMM.html. pages 2

[5] Matt Bonakdarpour. https://stephens999.github.io/fiveMinuteStats/intro_to_em.html, Jan 2016. pages 2

[6] Oran Looney. Ml from scratch, part 5: Gaussian mixture models. http://www.oranlooney.com/post/ml-from-scratch-part-5-gmm/, Jun 2019. pages 3

[7] Siddharth Vadgama. Gaussian mixture model(gmm) using em algorithm from scratch. https://medium.com/@siddharthvadgama/gaussian-mixture-model-gmm-using-em-algorithm-from-scratch-6b7c764aac9f, Nov 2019. pages 4

[8] Jake VanderPlas. In depth: Gaussian mixture models. https://jakevdp.github.io/PythonDataScienceHandbook/05.12-gaussian-mixtures.html. pages 9