INF 552 Programming Assignment 2

K Means and GMM

4472-6221-33 Yo Shuan Liu

I. Introduction

I wrote this homework in Python. And in both K means and GMM algorithm implementations, I take advantages of numpy array to make calculations between data array, mean array and covariance matrix smoother.
In addition, I use WCSS, within-cluster sum of square, and log-likelihood to measure the quality of clustering result in K Means and GMM, respectively.

In this report, I will introduce both implementations in the order of
(1) Notation
(2) Data Structure
(3) Challenges I Face

I will then compare the result of K Means and GMM, and also provide the comparison between my output and output of python library functions.

At the very last of this report, I will briefly introduce two interesting applications which take advantages of K Means and GMM algorithms.

II. Implementation of K Means

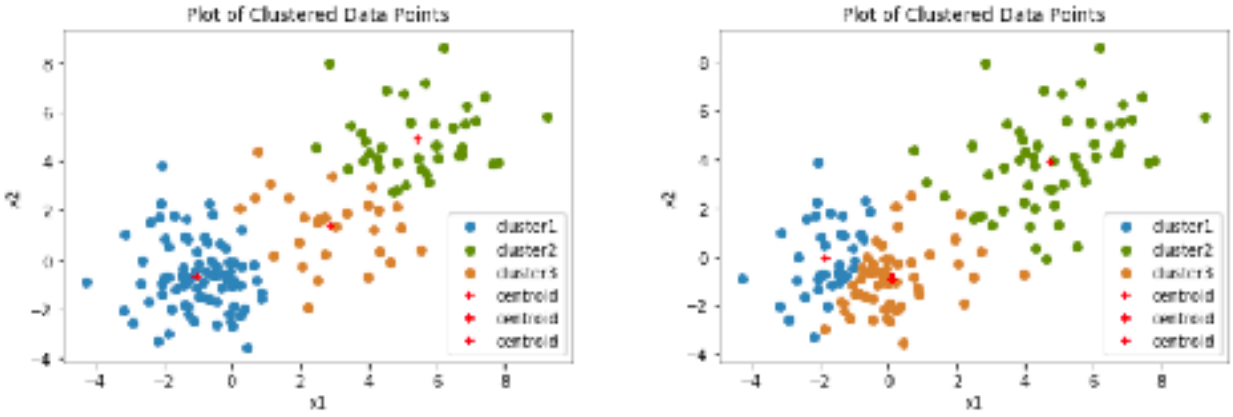
Notations
- Total rows of input data = n
- Dimension = m
- Total clusters = k

Data Structure
- Input data: a (n*m) array
- Centroids: a (m*k) array
- Distance matrix: a (k*n) array, record the distance between each data point and cluster
- Final result: a list with length k, data points classified as cluster C stored in position (C-1)

Parameters
Reinitialize Centroid: 20
Max iteration: 30
Convergence tolerance: 0.0001

Challenges I Face: Unstable result due to random start centroids
The first step of implementing K Means is picking random centroids from data points. The choice of initial centroids will sometimes dominate the clustering result, even after introducing the criteria of convergence.

I attach two clustering result as below as examples. I used epsilon = 0.000001 as convergence criteria in both clustering process, and both were stopped because they hit the criteria, however, the clustering result in the left was better than the one at the right, with WCSS equals to 510.1492200357534 and 570.1649153203446, respectively.



I noticed that my learning algorithm was stuck in a local minimum. I then decided to repeat the K Means several times (in my implementation, 20 times), in order to obtain different initial centroid. And then picked the result with the smallest WCSS.

III. Implementation of GMM

Notation

- Total rows of input data = n
- Dimension = m
- Total clusters = k

Data Structure

- Input data: a (n*m) array
- Cluster information: a list of dictionaries with length k. I choose to store information of cluster in dictionary because I need to store not only the mean, but also the covariance matrix, pi and gamma. It would be more organized and also less parameters to pass between functions to store these in dictionary, comparing to separated lists.
- Final result: a list with length k, data points classified as cluster C stored in position (C-1).

Parameters

Initial Centroid: centroid from K Means output
GMM Epoch: 100
Convergence tolerance: 0.0001

Challenges I Face: Unfamiliar with the math idea behind the algorithm

At first, I am not proficient with the math behind GMM enough, which makes my progress really slow. I can get the big picture, but not small details. Therefore, I searched and studied online and proof the expectation and maximization steps. And also added some comments in my code to make each step clearer.

IV. Clustering Result and Software Familiarization

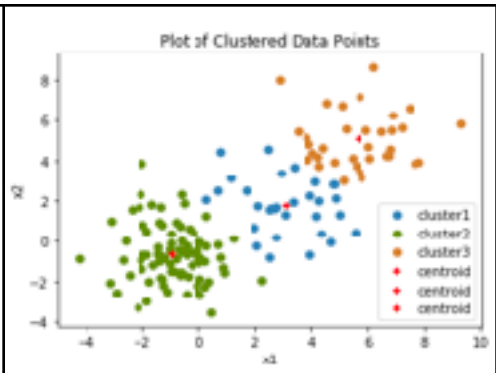
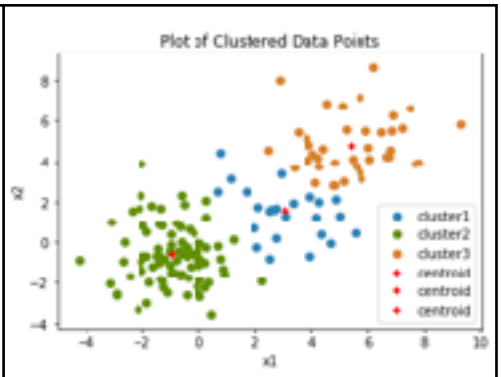
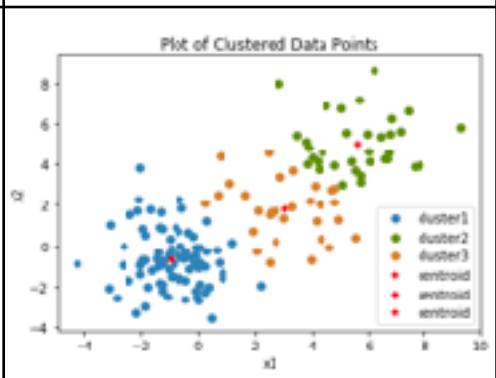
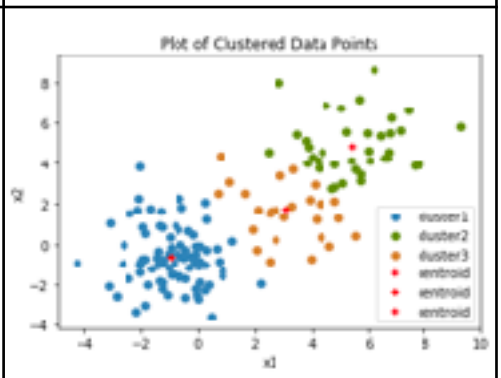
Please find below the comparison between K Means and GMM, and also the comparison between my and python's library implementation. There are a few highlights a would like to mention:

(1) My clustering result (statistics as well as data labels) of K Means is **exactly the same** as the result of python's K Means library.

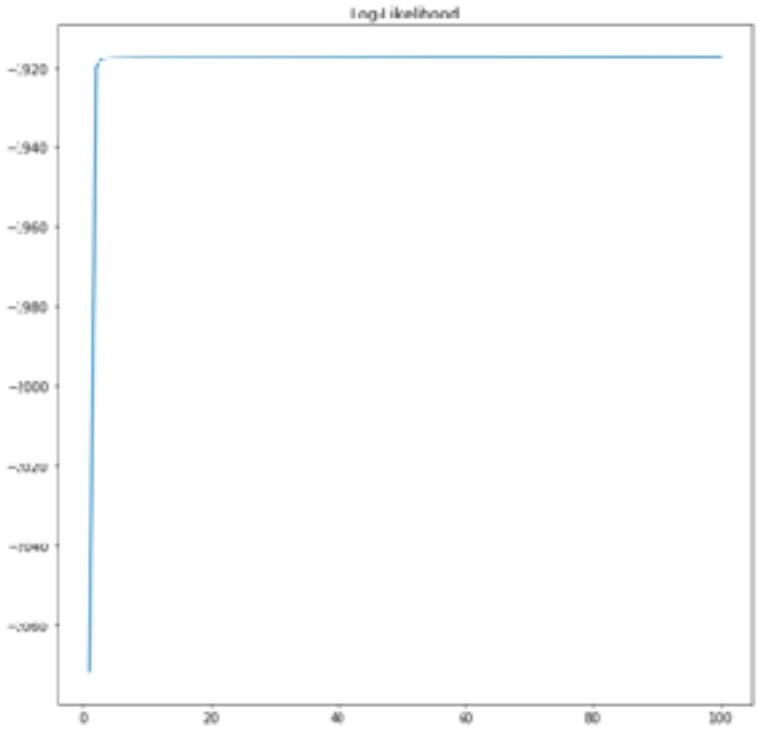
(2) I realize that GMM can do both soft and hard clustering, in order to be able to visualize the result, I classified the data points to the cluster with highest probability.

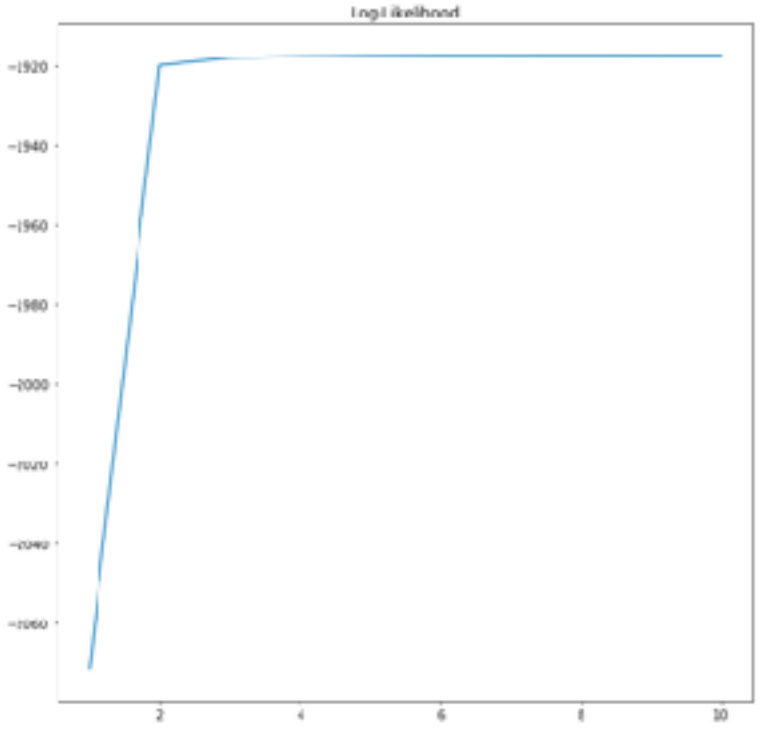
(3) My clustering result (statistics) of GMM is **slightly different** from the result of python's GMM library. And also there are some points where two implementations don't reach an agreement.

	K Means	**GMM**
Statistics of my Implementation	**Centroids**: [[-0.97476572 -0.68419304] [3.08318256 1.77621374] [5.62016573 5.02622634]]	**Means**: [[3.04739151 1.58852894] [-0.97470451 -0.64780723] [5.38301008 4.74096413]] **Amplitude**: [[0.18021134] [0.56677239] [0.25301627]] **Covariance Matrix**: [[[1.97420953 -0.11409818] [-0.11409818 2.25338464]] [[1.21557139 -0.10711132] [-0.10711132 2.00196522]] [[2.38778289 0.44092664] [0.44092664 2.34885555]]]
Library	sklearn.cluster.KMeans	sklearn.mixture.GaussianMixture
Statistics of Library Implementation	**Centroids**: [[-0.97476572 -0.68419304] [3.08318256 1.77621374] [5.62016573 5.02622634]]	**Means**: [[-0.96967895 -0.64450687] [5.45263907 4.82010088] [3.11928459 1.68884231]] **Amplitude**: [0.56831945 0.24140394 0.19027661] **Covariance Matrix**: [[[1.22511022 -0.10371187] [-0.10371187 2.00739399]] [[2.32795809 0.34778225] [0.34778225 2.25012269]] [[1.95112174 -0.05006407] [-0.05006407 2.32917397]]]

Visualization of my Implementation		
Visualization of Library Implementation		

Moreover, below is the progress of log-likelihood in 100 epochs. This plot tells us that the GMM converges after 10 epochs. Therefore, I lower the epoch and run the code again and find out that the GMM converges after 3 epochs! I believe this can be credit to the initial point which provides by the K Means.





However, if I use 10 as the max epoch, I will get warnings as below sometimes. Therefore, I choose to remain max epoch of GMM as 100, even it converges mostly in 10 epochs.

/anaconda3/lib/python3.6/site-packages/sklearn/mixture/base.py:237: ConvergenceWarning: Initialization 1 did not converge. Try different init parameters, or increase max_iter, tol or check for degenerate data.

V. Code Level Optimization

Before:

I input data as a 2D list, but later found out that I had to write lots of for loops to perform calculations. For example, distances between points and clusters, and calculation of the argmin.

After:

- I finally chose to use numpy library, which made the calculations easier and more efficient. And enabled me to delete some nested for loops in my code.

Before:

I store the data points instead of the index of those data points as clustering result. This way of storing will be space consuming when the input data has high dimension.

After:

- I revise the code in training step and visualization step to store only indices in clustering result. And use map function in python to convert data indices to data points when I need to calculate means or plot the result.

I also organized my code in an object-oriented manner, which contains three classes, KMeans, GMM, and VisualizationCluster. And I also put all adjustable parameters at top of my code to make everything clear.

VI. Application

One interesting application I found is using K Means algorithm to classified documents. It uses TF-IDF, term frequency-inverse document frequency, to describe documents in the Vector Space Model, which identify commonly used terms that helps classify the document. And this application also uses similarity score to identify the similarity between documents, which could be seen as the distance between different documents. Document classification is an example of machine learning in the form of natural language processing, which is often used in information science and library science.

Application regarding Gaussian Mixtures is to approximate empirical distributions (the distribution function of a discrete variable) of log daily differences of financial indices, such as DAX, Dow Jones, Nikkei, RTSI, S&P 500. It turns out that EM-algorithm is quite effective and the approximations based on Gaussian mixtures can be used to improve methods for financial risk analysis.

Reference:
https://www.codeproject.com/Articles/439890/Text-Documents-Clustering-using-K-Means-Algorithm
https://arxiv.org/pdf/1607.01033.pdf