

JAEWON YOUM

12/10/2024



COSE362-MACHINE LEARNING

ASSIGNMENT 1

EXPECTATION MAXIMIZATION (EM) ON GAUSSIAN MIXTURE MODEL (GMM)

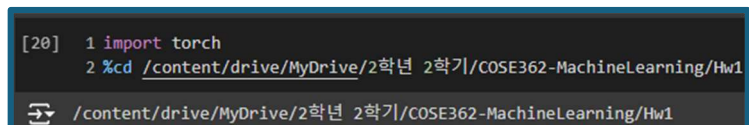
ASSIGNMENT 1 – EM OF GAUSSIAN MIXTURE MODEL

Note that specific details of the code are mostly unpresented here for the sake of legibility. Check the comments in the submitted notebook for detailed explanation on the implementation. It is strongly recommended that this document is read side-by-side with the relevant code.

ENVIRONMENT SETUP

First create a folder in Google Drive that contains the “train.txt”, “test.txt” data files and the provided source code. The implementation was written in python using Google Colab. To run the source code (.ipynb file, henceforth referred as notebook) in Colab, one must set the “runtime type” option as “Python 3” and the “Hardware accelerator” option as one of the provided GPUs.

Under the [Preparation - Importing Relevant Libraries and moving to location of data file](#) section in the notebook (shown on the right), one must edit the directory to the aforementioned file.



```
[20] 1 import torch
      2 %cd /content/drive/MyDrive/2학년 2학기/COSE362-MachineLearning/Hw1
```

/content/drive/MyDrive/2학년 2학기/COSE362-MachineLearning/Hw1

Make sure everything is set up correctly by checking the output of [Preparation - Checking GPU available for acceleration](#)

GMM CLASS EXPLANATION

INITIALIZATION

The initialization of the GMM class is rather simple and straightforward.

- The weights of the mixture components (different Normal distributions) were considered even.
e.g.) for 5 mixture components, the five weights were all set equally to 0.2.
- The means were chosen randomly from the given data. This adds a stochastic nature to the EM algorithm.
e.g.) for 5 mixture components, five observations were randomly chosen from the training data.
- The covariance matrices were all initialized to the covariance matrix of the given dataset. This seemed like a good way to “kickstart” the EM algorithm and lead to faster convergence.

Interesting observations

When covariance matrices were initialized as identity matrices, convergence was noticeably slower. I assume this is due to the data having correlated features. Moreover, when the mean vectors were initialized by the actual mean of the data points, this led to unexpected behaviors in convergence. It seemed like some sort of stochastic aspect existing within the initialization helped with fast convergence. This was an interesting find.

EXPECTATION

The .expectation() method accounts for the Expectation part of the EM algorithm. It calculates the responsibility matrix of the current parameters using the formula below.

$$P(g_k | x^t, \theta^i) = \frac{P(x^t | g_k, \theta^i)P(g_k | \theta^i)}{\sum_{\bar{k}} P(x^t | g_{\bar{k}}, \theta^i)P(g_{\bar{k}} | \theta^i)} \equiv w_k^t$$

To resolve numerical instability, the logarithm of the responsibility was calculated using the “Logarithm of a sum” method, and then exponentiated. All calculations were vectorized to lower computational costs.

The expectation method also returns the log likelihood of the dataset. This is later used to determine the convergence of the EM algorithm.

MAXIMIZATION

The .maximize() method accounts for the Maximization part of the EM algorithm. It calculates the new parameters (weights, means, covariance matrices) of the GMM using the following formulas from the slides:

$$\pi_k = \frac{\sum_t w_k^t}{N}, \mu_k = \frac{\sum_t w_k^t x^t}{\sum_t w_k^t}, \Sigma_k = \frac{\sum_t w_k^t (x^t - \mu_k)(x^t - \mu_k)^T}{\sum_t w_k^t}$$

Like the expectation method, all calculations were vectorized, and the “Logarithm of a sum” method was utilized whenever possible. But due to numerical stability, sometimes the calculations led to non-positive definite covariance matrices.

To account for these errors, the implementation “forces” positive definiteness by ensuring symmetry and adding a small epsilon value to the diagonals of the covariance matrices. More details can be found in the notebook.

The maximize method also returns the log likelihood of the dataset, calculated identically as the log likelihood in the Expectation step. This is later used to determine the convergence of the EM algorithm.

LOG LIKELIHOOD

The .log_likelihood(input_data) method returns the separate log likelihood of each observation given in the input dataset. This is used to later classify test data. Calculations follow the same pattern as before, except that the final weighted sum of log likelihoods are not added up but returned directly.

EM ALGORITHM

The .EM(max_iter=1000, tol=1e-4) method is used to perform EM on the GMM class until convergence.

The convergence was determined by using the difference in log likelihoods of the previous and new GMM parameters regarding the training data. The tolerance (upper bound to difference) was set to 1e-4 by default. Experimentation showed that GMMs with mixture components up to 20 essentially never reach the default max iteration of 1000 when tolerance is set to 1e-4. This number was not decreased as the algorithm itself was computationally cheap, hence taking little time to converge.

Interesting observations

Initially, convergence was determined by the upper bounding of the Euclidean distance between the old and new parameters. However, this did not produce satisfactory results. The reason for this may be the fact that the parameters have different scales, hence needing different “weights” to properly measure convergence. However, this seemed overly tedious compared to the more traditional way of using log-likelihood difference, which was chosen as the final method.

DATA PREPARATION

Both **train.txt** and **test.txt** contain 14 columns, where the first 13 columns are for the different features, and the last being the label. Each row is an observation (data).

Since K-fold validation was used to find the optimal number of components for the GMM, training data were sliced into K overlapping data sets, where each data set did not contain one section of the original data.

The test data for each K-fold was the one missing data section of the corresponding training data.

FINDING OPTIMAL NUMBER OF MIXTURE COMPONENTS (C)

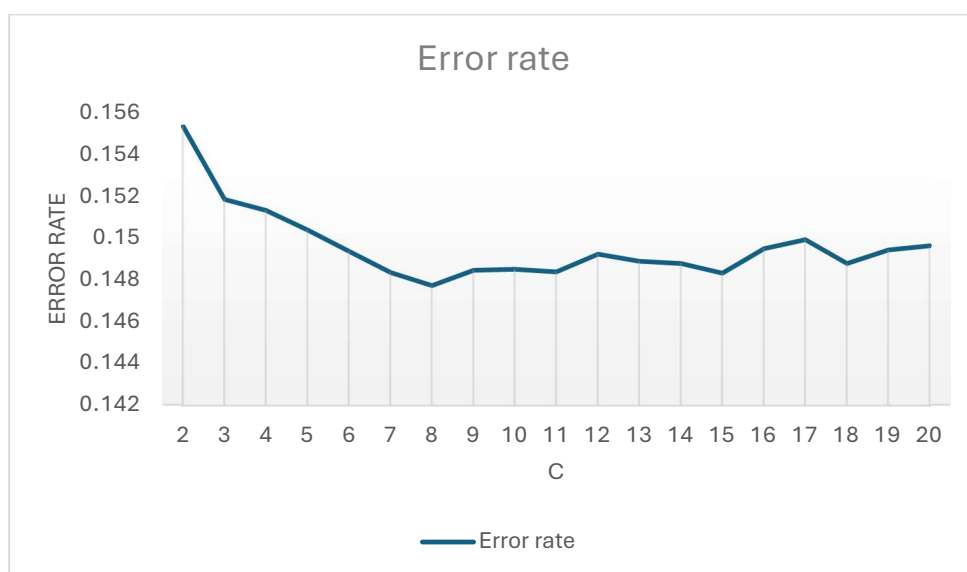
Under section *Using GMM class to perform classification-Finding optimal c* is a code snippet that runs only **a single K-fold validation** of c starting from 2 and ending at 20. The upper bound of c was set to 20 as the error rate seemed to minimize usually around 7-12 mixture components and consequently oscillate at a slightly higher value than the minimum.

To account for the stochastic nature of the EM algorithm, K-fold validation was done twenty times, and the errors were averaged to find the optimal value of c. This is easily done by using an array to keep track of the error values for each trial while iterating the code snippet twenty times.

For each c, a K-fold validation where K = 5 was performed. Larger values of K, such as 10, made computation exponentially costly. The process is as follows for each of the two GMMs for a binary classification problem:

- 1) GMM is initialized
- 2) GMM is trained using the .EM() method
- 3) Likelihood of the GMM regarding the training dataset is calculated
- 4) Likelihood is used to classify test dataset
- 5) Error is calculated by calculating the absolute value of the elementwise-subtraction of the predicted class vector and true class vector, and then averaging over the resulting vector. Note that this only applies to binary classification, as the difference will always be either 0 (correct classification) or 1 (wrong classification).

Details on how this process is implemented can be found in the notebook.



Hence the optimal c value can be evaluated to 8, at a minimum moderately lower than other c values.

FINAL ERROR REPORT ON GMM MODEL WITH 8 MIXTURE COMPONENTS

Using the optimal c value, a final GMM model for class 0/1 can be trained. This time, unlike previous training, we utilize the entire training dataset. Then the GMM models were used to classify the test dataset, and the error was calculated. The process follows the same procedures mentioned before.

The final error rate of a Gaussian Mixture Model, with 8 mixture components learned from data in **train.txt**, when used for the binary classification of data in **test.txt** was:

$$0.1467820405960083 \approx 0.147 = 14.7\%$$

Interesting observations

By mistake, when training the final GMMs, the error was calculated for GMMs that never underwent the EM algorithm; the error for merely initialized GMMs were calculated. The result was surprisingly “good”, where the error rate was approximately 0.20. This means that the EM algorithm reduces error percentage by merely 5%. Though I am not sure as to why this happens, the following reasons may be the cause:

- The nature of the data
- Good initialization methods
- Ineffectiveness of EM

Further investigation may be done in the future to reveal the true cause.