(PSL) Coding Assignment 4

Contents

Part I: Gaussian Mixtures

Objective

Implement the EM algorithm **from scratch** for a p-dimensional Gaussian mixture model with G components:

$$\sum_{k=1}^{G} p_k \cdot \mathsf{N}(x; \mu_k, \Sigma).$$

Requirements

Your implementation should consists of **four** functions.

- **`Estep`** function: This function should return an n-by-G matrix, where the (i, j)th entry represents the conditional probability $P(Z_i = k \mid x_i)$. Here i ranges from 1 to n and k ranges from 1 to G.
- **`Mstep`** function: This function should return the updated parameters for the Gaussian mixture model.
- **`loglik`** function: This function computes the log-likelihood of the data given the parameters.
- **`myEM`** function (main function): Inside this function, you can call the `Estep` and `Mstep` functions. The function should take the following inputs and return the estimated parameters and log-likelihood (via the `loglik` function):
  - **Input**:
    * data: The dataset.
    * G: The number of components. Although your code will be tested with $G = 2$ and $G = 3$, it should be able to handle any value of G. (You can, of course, ignore the case where $G > n$.)
    * Initial parameters.
    * `itmax`: The number of iterations.
  - **Output**:
    * `prob`: A G-dimensional probability vector $(p_1, \ldots, p_G)$
    * `mean`: A p-by-G matrix with the k-th column being μ_k, the p-dimensional mean for the k-th Gaussian component.
    * `Sigma`: A p-by-p covariance matrix Σ shared by all G components;

  * `loglik`: A number equal to $\sum_{i=1}^{n} \log \Big[\sum_{k=1}^{G} p_k \cdot \mathsf{N}(x; \mu_k, \Sigma) \Big]$.

Implementation Guidelines:

The requirements are very similar to Coding Assignment 1. “No loops” means no explicit loops such as `for` or `while`, and no use of functions like `apply` or `map`.

- `Estep` function: No loops.
- `Mstep` function: You may only loop over G when updating `Sigma`.
- `loglik` function: You may only loop over G.
- You are not allowed to use pre-existing functions or packages for evaluating normal densities. However, you may use built-in functions to compute the inverse of a matrix or perform SVD.

Testing

Test your code with the provided dataset, [faithful.dat], with both $G = 2$ and $G = 3$.

For the case when $G = 2$, set your initial values as follows:

- $p_1 = 10/n$, $p_2 = 1 - p_1$.
- $\mu_1 =$ the mean of the first 10 samples; $\mu_2 =$ the mean of the remaining samples.
- Calculate Σ as

$$\frac{1}{n}\Big[\sum_{i=1}^{10}(x_i-\mu_1)(x_i-\mu_1)^t + \sum_{i=11}^{n}(x_i-\mu_2)(x_i-\mu_2)^t\Big].$$

  Here $(x_i - \mu_i)$ is a 2-by-1 vector and the **superscript** t **denotes the transpose**. so the resulting Σ matrix is a 2-by-2 matrix.

Run your EM implementation with **20** iterations. Your results from `myEM` are expected to look like the following. (Even though the algorithm has not yet reached convergence, matching the expected results below serves as a validation that your code is functioning as intended.)

```
prob
[1] 0.04297883 0.95702117

mean
               [,1]      [,2]
eruptions  3.495642  3.48743
waiting   76.797892 70.63206

Sigma
          eruptions   waiting
eruptions  1.297936  13.92434
waiting   13.924336 182.58009

loglik
[1] -1289.569
```

For the case when $G = 3$, set your initial values as follows:

- $p_1 = 10/n$, $p_2 = 20/n$, $p_3 = 1 - p_1 - p_2$
- $\mu_1 = \frac{1}{10}\sum_{i=1}^{10} x_i$, the mean of the first 10 samples; $\mu_2 = \frac{1}{20}\sum_{i=11}^{30} x_i$, the mean of next 20 samples; and $\mu_3 =$ the mean of the remaining samples.
- Calculate Σ as

$$\frac{1}{n}\Big[\sum_{i=1}^{10}(x_i-\mu_1)(x_i-\mu_1)^t + \sum_{i=11}^{30}(x_i-\mu_2)(x_i-\mu_2)^t + \sum_{i=31}^{n}(x_i-\mu_3)(x_i-\mu_3)^t\Big].$$

Run your EM implementation with **20** iterations. Your results from `myEM` are expected to look like the following.

```
prob
[1] 0.04363422 0.07718656 0.87917922

mean
                [,1]      [,2]      [,3]
eruptions  3.510069  2.816167  3.545641
waiting   77.105638 63.357526 71.250848

Sigma
          eruptions   waiting
eruptions  1.260158  13.51154
waiting   13.511538 177.96419

loglik
[1] -1289.351
```

Derivation

Partial results for the derivation of the EM algorithm are given below. Note that the **`faithful`** data are two-dimensional, therefore $d = 2$, μ_k's are 2-by-1 vectors and Σ is a 2-by-2 matrix.

1. The (marginal) likelihood function:

$$\begin{aligned}&\prod_{i=1}^n p(x_i \mid p_{1:G}, \mu_{1:G}, \Sigma)\\ =&\prod_{i=1}^n \big[p_1 N(x_i; \mu_1, \Sigma) + \cdots + p_G N(x_i; \mu_G, \Sigma)\big]\\ =&\prod_{i=1}^n \Big[p_1 \frac{\exp(-\frac{1}{2}(x_i-\mu_1)^t\Sigma^{-1}(x_i-\mu_1))}{\sqrt{(2\pi)^d|\Sigma|}} + \cdots + p_G \frac{\exp(-\frac{1}{2}(x_i-\mu_G)^t\Sigma^{-1}(x_i-\mu_G))}{\sqrt{(2\pi)^d|\Sigma|}}\Big]\end{aligned}$$

where $|\Sigma|$ denotes the determinant of matrix Σ. Your **`loglik`** function needs to compute the log of this function.

2. The complete likelihood function $\sum_{i=1}^n p(x_i, Z_i \mid p_{1:G}, \mu_{1:G}, \Sigma)$ or its log, which is the function we work with in the EM algorithm.

$$\begin{aligned}&\prod_{i=1}^n p(x_i, Z_i \mid p_{1:G}, \mu_{1:G}, \Sigma)\\ =&\prod_{i=1}^n \prod_{k=1}^G \Big[p_k \frac{\exp(-\frac{1}{2}(x_i-\mu_k)^t\Sigma^{-1}(x_i-\mu_k))}{\sqrt{(2\pi)^d|\Sigma|}}\Big]^{1_{\{Z_i=k\}}}\end{aligned}$$

3. Find the distribution of Z_i at the E-step. Given data and the current parameter value $(p^{(0)}_{1:G}, \mu^{(0)}_{1:G}, \Sigma^{(0)})$, Z_i follows a discrete distribute taking values from 1 to G with probabilities

$$\begin{aligned}w_{ik} :=& P(Z_i = k \mid x_i, p^{(0)}_{1:G}, \mu^{(0)}_{1:G}, \Sigma^{(0)})\\ \propto& P(x_i \mid Z_i = k, \mu^{(0)}_{1:G}, \Sigma^{(0)}) \times P(Z_i = k \mid p^{(0)}_{1:G})\end{aligned}$$

4. The objective function you aim to maximize (or minimize) at the M-step. At the M-step, we optimize the following objective function (where the expectation is taken over $Z_1, \dots, Z_n$ with respect to the probabilities computed at Step 3):

$$\begin{aligned} g(p_{1:G}, \mu_{1:G}, \Sigma) =& \mathbb{E} \log \prod_{i=1}^{n} p(x_i, Z_i \mid p_{1:G}, \mu_{1:G}, \Sigma) \\ =& \mathbb{E} \sum_{i=1}^{n} \sum_{k=1}^{G} 1_{\{Z_i = k\}} \log \left[p_k \frac{\exp(-\frac{1}{2}(x_i - \mu_k)^t \Sigma^{-1} (x_i - \mu_k))}{\sqrt{(2\pi)^d |\Sigma|}} \right] \\ =& \sum_{i=1}^{n} \sum_{k=1}^{G} w_{ik} \log \left[p_k \frac{\exp(-\frac{1}{2}(x_i - \mu_k)^t \Sigma^{-1} (x_i - \mu_k))}{\sqrt{(2\pi)^d |\Sigma|}} \right] \end{aligned}$$

where the last step is due to the fact that $\mathbb{E}[1_{\{Z_i=k\}}] = \mathbb{P}(Z_i = k) = w_{ik}$. You need to find the updating formulas for $p_{1:G}, \mu_{1:G}, \Sigma$ at the M-step.

Part II: HMM

Objective

Implement the Baum-Welch (i.e., EM) algorithm and the Viterbi algorithm **from scratch** for a Hidden Markov Model (HMM) that produces an outcome sequence of discrete random variables with three distinct values.

A quick review on parameters for Discrete HMM:

- `mx`: Count of distinct values X can take.
- `mz`: Count of distinct values Z can take.
- `w`: An mz-by-1 probability vector representing the initial distribution for Z_1.
- `A`: The mz-by-mz transition probability matrix that models the progression from Z_t to Z_{t+1}.
- `B`: The mz-by-mx emission probability matrix, indicating how X is produced from Z.

Focus on updating the parameters `A` and `B` in your algorithm. The value for `mx` is given and you'll specify `mz`.

For `w`, initiate it uniformly but refrain from updating it within your code. The reason for this is that `w` denotes the distribution of Z_1 and we only have a single sample. It's analogous to estimating the likelihood of a coin toss resulting in heads by only tossing it once. Given the scant information and the minimal influence on the estimation of other parameters, we can skip updating it.

Baum-Welch Algorihtm

The Baum-Welch Algorihtm is the EM algorithm for the HMM. Create a function named **`BW_onestep`** designed to carry out the E-step and M-step. This function should then be called iteratively within **`myBW`**.

`BW_onestep`:

- **Input**:
  - data: a T-by-1 sequence of observations
  - Current parameter values
- **Output**:
  - Updated parameters: `A` and `B`

Please refer to formulas provided on Pages 7, 10, 14-16 in [lec_W7.2_HMM]

Viterbi Algorihtm

This algorithm outputs the most likely latent sequence considering the data and the MLE of the parameters.

`myViterbi`:

- **Input**:
  - data: a T-by-1 sequence of observations
  - parameters: `mx`, `mz`, `w`, `A` and `B`
- **Output**:
  - `Z`: A T-by-1 sequence where each entry is a number ranging from 1 to `mz`.

Please refer to formulas provided on Pages 18-20 in [lec_W7.2_HMM]

Note on Calculations in Viterbi:

Many computations in HMM are based on the product of a sequence of probabilities, resulting in extremely small values. At times, these values are so small that software like R or Python might interpret them as zeros. This poses a challenge, especially for the Viterbi algorithm, where differentiating between magnitudes is crucial. If truncated to zero, making such distinctions becomes impossible. Therefore, it's advisable to evaluate these probabilities on a logarithmic scale in the Viterbi algorithm.

Testing

1. Test your code with the provided data sequence: [Coding4_part2_data.txt]. Set `mz = 2` and start with the following initial values

$$w = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}, \quad A = \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix}, \quad B = \begin{pmatrix} 1/9 & 3/9 & 5/9 \\ 1/6 & 2/6 & 3/6 \end{pmatrix}$$

Run your implementation with **100** iterations. The results from your implementation of the Baum-Welch algorithm should match with the following:

```
A: the 2-by-2 transition matrix

    0.49793938 0.50206062
    0.44883431 0.55116569

B: the 2-by-3 emission matrix
    0.22159897 0.20266127 0.57573976
    0.34175148 0.17866665 0.47958186
```

The output from your Viterbi algorithm implementation should align with the following benchmarks. Please cross-check your results against the complete binary sequence available in [Coding4_part2_Z.txt]

```
1 1 1 1 1 1 1 2 1 1 1 1 1 2 2 1 1 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1
1 1 2 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 2 2 2 1 1 1 1 2 2 2 2 1 1
......
2 1 1 1 1 1 1 1
```

2. Initialize matrix `B` such that each entry is 1/3, and run your Baum-Welch algorithm for **20** and **100** iterations. Examine the resulting `A` and `B` matrices, and explain why you obtained these outcomes. Based on your findings, you should understand why we cannot initialize our parameters in a way that makes the latent states indistinguishable.