Homework 4

20213073 Donggyu Kim

May 24, 2021

1 Equation

Since $\boldsymbol{\pi}$ is under the constraint $\sum_{j=1}^{k} \pi_j = 1$, we introduce the Lagrangian multiplier,

$$\begin{aligned}\tilde{\mathcal{L}}(\boldsymbol{\theta}, \lambda) &= \tilde{\mathcal{L}}(\theta) + \lambda \left(1 - \sum_{j=1}^{k} \pi_j\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{k} r_{i,j}^{(t)}(\log \pi_j + \log \mathcal{N}(\boldsymbol{x}_i; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)) + \lambda \left(1 - \sum_{j=1}^{k} \pi_j\right)\end{aligned}$$

We take the gradient w.r.t. $\boldsymbol{\pi}$ and get

$$\frac{\partial \tilde{\mathcal{L}}(\boldsymbol{\theta}, \lambda)}{\partial \pi_j} = \sum_{i=1}^{n} \frac{r_{i,j}^{(t)}}{\pi_j} - \lambda = 0 \;\Rightarrow\; \pi_j^{(t)} = \frac{1}{\lambda}\sum_{i=1}^{n} r_{i,j}^{(t)}$$

Using the constraint $\sum_{j=1}^{k} \pi_j = 1$ again, we finally have

$$\begin{aligned}\sum_{j=1}^{k} \pi_j^{(t)} &= \frac{1}{\lambda}\sum_{j=1}^{k}\sum_{i=1}^{n} r_{i,j}^{(t)} \\ &= \frac{1}{\lambda}\sum_{i=1}^{n}\sum_{j=1}^{k} \frac{\pi_j \mathcal{N}(\boldsymbol{x}_i; \boldsymbol{\mu}_j^{(t-1)}, \boldsymbol{\Sigma}_j^{(t-1)})}{\sum_{l=1}^{k} \pi_l \mathcal{N}(\boldsymbol{x}_i; \boldsymbol{\mu}_l^{(t-1)}, \boldsymbol{\Sigma}_l^{(t-1)})} \\ &= \frac{1}{\lambda}\sum_{i=1}^{n} 1 = \frac{n}{\lambda} = 1 \;\Rightarrow\; \lambda = n \\ \therefore \pi_j^{(t)} &= \frac{1}{n}\sum_{i=1}^{n} r_{i,j}^{(t)}\end{aligned}$$

By taking the gradient w.r.t. $\boldsymbol{\mu}$, we get

$$\begin{aligned}\frac{\partial \tilde{\mathcal{L}}(\boldsymbol{\theta}, \lambda)}{\partial \boldsymbol{\mu}_j} &= \sum_{i=1}^{n} r_{i,j}^{(t)} \frac{\partial}{\partial \boldsymbol{\mu}_j}\left(-\frac{1}{2}(\boldsymbol{x}_i - \boldsymbol{\mu}_j)^\top \boldsymbol{\Sigma}_j^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu}_j)\right) \\ &= \sum_{i=1}^{n} r_{i,j}^{(t)}(\boldsymbol{x}_i - \boldsymbol{\mu}_j)^\top \boldsymbol{\Sigma}_j^{-1} = \mathbf{0} \qquad (\because \boldsymbol{\Sigma}_j = \boldsymbol{\Sigma}_j^\top) \\ \therefore \boldsymbol{\mu}_j^{(t)} &= \frac{1}{\sum_{i=1}^{n} r_{i,j}^{(t)}}\sum_{i=1}^{n} r_{i,j}^{(t)} \boldsymbol{x}_i\end{aligned}$$

This time, taking the gradient w.r.t. $\boldsymbol{\Sigma}^{-1}$, we get

$$\begin{aligned}\frac{\partial\tilde{\mathcal{L}}(\boldsymbol{\theta},\lambda)}{\partial\boldsymbol{\Sigma}_j^{-1}} &= \sum_{i=1}^{n} r_{i,j}^{(t)} \frac{\partial}{\partial\boldsymbol{\Sigma}_j^{-1}}\left(\frac{1}{2}\log(\det(\boldsymbol{\Sigma}_j^{-1})) - \frac{1}{2}(\boldsymbol{x}_i-\boldsymbol{\mu}_j)^\top\boldsymbol{\Sigma}_j^{-1}(\boldsymbol{x}_i-\boldsymbol{\mu}_j)\right)\\ &= \sum_{i=1}^{n}\frac{r_{i,j}^{(t)}}{2}\left(\boldsymbol{\Sigma}_j - (\boldsymbol{x}_i-\boldsymbol{\mu}_j)(\boldsymbol{x}_i-\boldsymbol{\mu}_j)^\top\right) = \mathbf{0} \qquad (\because \boldsymbol{\Sigma}_j = \boldsymbol{\Sigma}_j^\top)\\ \therefore \boldsymbol{\Sigma}_j^{(t)} &= \frac{1}{\sum_{i=1}^{n} r_{i,j}^{(t)}}\sum_{i=1}^{n} r_{i,j}^{(t)}\left(\boldsymbol{x}_i-\boldsymbol{\mu}_j^{(t)}\right)\left(\boldsymbol{x}_i-\boldsymbol{\mu}_j^{(t)}\right)^\top\end{aligned}$$

2 Implementation

```python
import numpy as np
from scipy.stats import multivariate_normal as mn
import matplotlib.pyplot as plt
```

I used some functionalities of NumPy and SciPy to implement the EM algorithm, and Matplotlib for visualization.

```python
X=np.loadtxt('X.txt')
```

First, $\boldsymbol{X}$ is set to be 2D data written in `X.txt`.

```python
n,d=X.shape
k=5 # number of components
random_seed=0
```

$\boldsymbol{X}$ is composed of n observations, where each observation has d values. In the case of `X.txt`, $n = 2000$ and $d = 2$. The number of components is fixed $k = 5$. I additionally used `random_seed` hyperparameter. This is to analyze the sensitiviy of the EM algorithm with respect to the initialization and for reproducibility of the results.

```python
np.random.seed(random_seed)
pi=np.random.rand(k)
pi/=sum(pi) #normalization
mu=np.random.randn(k,d)
sigma=[]
for _ in range(k):
    A=np.random.randn(d,d)
    sigma.append(A@A.T)
```

Before running the EM algorithm, parameter $\boldsymbol{\theta}$ should be initialized. I used `np.random.rand()` to get $\boldsymbol{\pi}$. Sicne $\boldsymbol{\pi}$ is under the constraint $\sum_{j=1}^{k} \pi_j = 1$, it requires normalization.
While $\boldsymbol{\mu}$ can be easily initialized by using `np.random.randn`, $\boldsymbol{\Sigma}$ needs a few steps. Since each covariance matrix $\boldsymbol{\Sigma}_j$ should be symmetric and positive semi-definite, the easiest way is to initialize $\boldsymbol{\Sigma}_j$ to $\boldsymbol{A}\boldsymbol{A}^\top$ where $\boldsymbol{A} \in R^{d \times d}$ is just a random matrix.

```
def E_step(pi,mu,sigma):
    # Responsibility values
    r=np.array([[pi[j]*mn.pdf(X[i],mu[j],sigma[j])\
                for j in range(k)] for i in range(n)])
    for i in range(n):
        s=sum(r[i])
        for j in range(k):
            r[i][j]/=s
    # Complete-data log-likelihood
    L=sum(sum(r[i][j]*(np.log(pi[j])+mn.logpdf(X[i],mu[j],sigma[j]))\
                                for j in range(k)) for i in range(n))
    return r,L
```

In the E-step, the algorithm computes the responsibility values $\{r_{i,j}^t\}$ and the complete-data log-likelihood $\tilde{\mathcal{L}}$. The function gets $\boldsymbol{\pi}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as the inputs and outputs $\boldsymbol{r}$ and $\tilde{\mathcal{L}}$. I used `ndarray` of NumPy for the parameters, and this enables such compact implementation.

```
def M_step(r):
    pi=np.array([np.mean(r[:,j]) for j in range(k)])
    mu=np.array([sum(r[i][j]*X[i] for i in range(n))/(n*pi[j])\
                                                for j in range(k)])
    sigma=np.array([sum(r[i][j]*np.outer(X[i]-mu[j],X[i]-mu[j])\
                for i in range(n))/(n*pi[j]) for j in range(k)])
    return pi,mu,sigma
```

In the M-step, the algorithm updates the parameters. The function just computes the new parameter values according to the equations that we derived in the Q.1. It only takes $\boldsymbol{r}$ as the input and outputs $\boldsymbol{\pi}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$.

```
t=1
Ls=[]
L_prev=0
eps=1e-3
while True:
    r,L=E_step(pi,mu,sigma)
    Ls.append(L)
    if t>1 and abs(L_prev-L)<eps:
```

```
        break
    L_prev=L
    pi,mu,sigma=M_step(r)
    # Logging
    if t%5==1:
        print(L)
    t+=1
```

Finally, this is the main part of the EM algorithm. Using `E_step` and `M_step`, the algorithm runs until the log-likelihood converges. I set `eps=1e-3` as the threshold of convergence.

```
print(Ls[0],Ls[-1])
```

```
-33997.18908764336 -4436.270807069354
```

(a) `random_seed` = 0

```
print(Ls[0],Ls[-1])
```

```
-12894.590885700829 -4436.270727270894
```

(b) `random_seed` = 1

```
print(Ls[0],Ls[-1])
```

```
-21665.12917769737 -5666.830818981626
```

(c) `random_seed` = 42

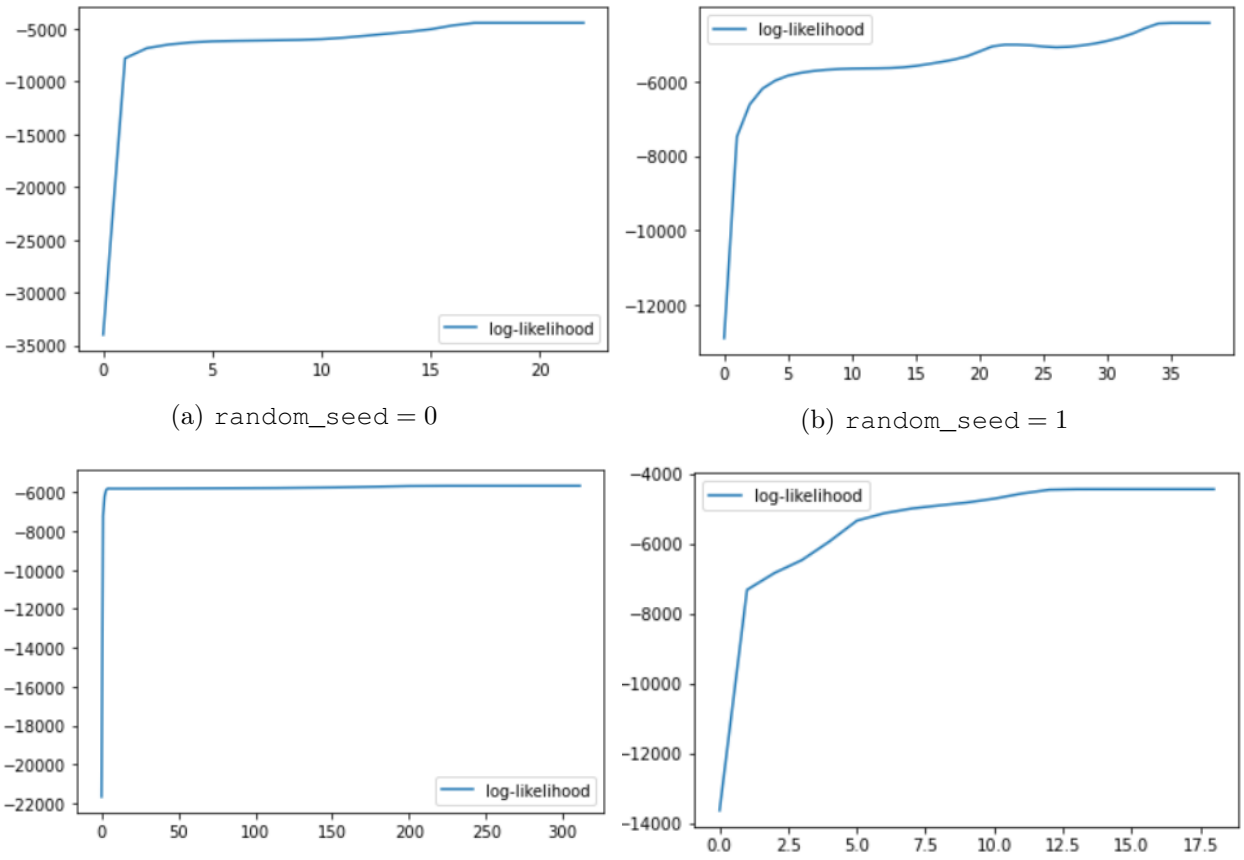
```
print(Ls[0],Ls[-1])
```

```
-13643.352022656489 -4436.270817079326
```

(d) `random_seed` = 777

Figure 1: The first(left) and last(right) log-likelihood values

Fig. 1 and 2 are the results. The results show that the EM algorithm is sensitive with respect to the initialization. It is quite interesting that the algorithm reached to the almost same value with `random_seed` of 0, 1 and 777. However, when `random_seed` is 42, the algorithm converged to very different value. Also, it took 10 times more iterations to converge, which looks quite wrong.



(a) `random_seed` = 0

(b) `random_seed` = 1

(c) `random_seed` = 42

(d) `random_seed` = 777

Figure 2: The trace plot of the log-likelihood