Lab 3: EM Algorithm

Course Title: Image Processing II (Fall 2021)

Course Number: ECE 69400

Instructor: Prof. Charles A. Bouman

Author: **Zhankun Luo**

2. Parameter Estimation for Multivariate Gaussian Distributions

Question 2.1

Show that the density function of (10) with parameters θ of (11)

$$f(\vec{y}, \vec{x} \mid \theta) = \prod_{n=0}^{N-1} \frac{\pi_{x_n}}{(2\pi)^{M/2}} \left|R_{x_n}\right|^{-1/2} \exp\left\{-\frac{1}{2}(y_n - \mu_{x_n})^\top R_{x_n}^{-1}(y_n - \mu_{x_n})\right\}$$

$$\theta = [\pi_0, \mu_0, R_0, \cdots, \pi_{K-1}, \mu_{K-1}, R_{K-1}]$$

forms an exponential family with natural sufficient statistics given in (12), (13), and (14).

$$N_k = \sum_{n=0}^{N-1} \delta(x_n - k)$$

$$t_{1,k} = \sum_{n=0}^{N-1} y_n \delta(x_n - k)$$

$$t_{2,k} = \sum_{n=0}^{N-1} y_n y_n^\top \delta(x_n - k)$$

solution

If and only if we can write PDF in such a form below, $\vec{T}$ are the **natural sufficient statistics** of the exponential family

$$f(\vec{y}, \vec{x} \mid \theta) = \exp\left(\vec{\eta}^\top(\theta) \cdot \vec{T} - \psi(\theta)\right) \cdot h(\vec{x}, \vec{y})$$

Where $\vec{y} \equiv [y_0, \cdots, y_{N-1}]$, $\vec{x} = [x_0, \cdots x_{N-1}]$ and $x_n \in \{0, \cdots, K-1\}$ for $n = 0, \cdots, N-1$

Firstly, we rewrite the PDF of (11) in such a form, where $\delta(x-k)$ represents such an indicator function $I_{x=k}$

$$
\begin{aligned}
&f(\vec{y},\vec{x}\mid\theta)\\
=&\prod_{n=0}^{N-1}(2\pi)^{-\frac{M}{2}}\left\{\prod_{k=0}^{K-1}\left(\pi_k|R_k|^{-1/2}\right)^{\delta(x_n-k)}\exp\left\{-\frac{1}{2}\delta(x_n-k)(y_n-\mu_k)^\top R_k^{-1}(y_n-\mu_k)\right\}\right\}\\
=&(2\pi)^{-\frac{MN}{2}}\prod_{k=0}^{K-1}\left[\left(\pi_k|R_k|^{-1/2}\right)^{\sum\limits_{n=0}^{N-1}\delta(x_n-k)}\right.\\
&\cdot\exp\left\{-\frac{1}{2}\sum_{n=0}^{N-1}\delta(x_n-k)\,\mathrm{tr}\left(y_n^\top R_k^{-1}y_n\right)\right.\\
&\left.\left.+\mu_k^\top R_k^{-1}\left[\sum_{n=0}^{N-1}y_n\delta(x_n-k)\right]-\frac{1}{2}\mu_k^\top R_k^{-1}\mu_k\left[\sum_{n=0}^{N-1}\delta(x_n-k)\right]\right\}\right]\\
=&(2\pi)^{-\frac{MN}{2}}\prod_{k=0}^{K-1}\left[\left(\pi_k|R_k|^{-1/2}\right)^{\sum\limits_{n=0}^{N-1}\delta(x_n-k)}\right.\\
&\cdot\exp\left\{-\frac{1}{2}\mathrm{tr}\left(R_k^{-1}\sum_{n=0}^{N-1}y_ny_n^\top\delta(x_n-k)\right)\right.\\
&\left.\left.+\mu_k^\top R_k^{-1}\left[\sum_{n=0}^{N-1}y_n\delta(x_n-k)\right]-\frac{1}{2}\mu_k^\top R_k^{-1}\mu_k\left[\sum_{n=0}^{N-1}\delta(x_n-k)\right]\right\}\right]
\end{aligned}
$$

Let $h(\vec{x},\vec{y})\equiv(2\pi)^{-\frac{MN}{2}}I_{\vec{y}\in\mathbb{R}^{M\times N},x_n\in\{0,\cdots,K-1\}\text{ for }n=0,\cdots,N-1}$ and $\exp\left(\vec{\eta}^\top(\theta)\cdot\vec{T}-\psi(\theta)\right)$ to be

$$
\begin{aligned}
&\exp\left(\vec{\eta}^\top(\theta)\cdot\vec{T}-\psi(\theta)\right)\\
=&\prod_{k=0}^{K-1}\left[\left(\pi_k|R_k|^{-1/2}\right)^{\sum\limits_{n=0}^{N-1}\delta(x_n-k)}\right.\\
&\cdot\exp\left\{-\frac{1}{2}\mathrm{tr}\left(R_k^{-1}\sum_{n=0}^{N-1}y_ny_n^\top\delta(x_n-k)\right)\right.\\
&\left.\left.+\mu_k^\top R_k^{-1}\left[\sum_{n=0}^{N-1}y_n\delta(x_n-k)\right]-\frac{1}{2}\mu_k^\top R_k^{-1}\mu_k\left[\sum_{n=0}^{N-1}\delta(x_n-k)\right]\right\}\right]\\
=&\prod_{k=0}^{K-1}\left[\left(\pi_k|R_k|^{-1/2}\right)^{N_k}\cdot\exp\left\{-\frac{1}{2}\mathrm{tr}\left(R_k^{-1}t_{2,k}\right)+\mu_k^\top R_k^{-1}t_{1,k}-\frac{N_k}{2}\mu_k^\top R_k^{-1}\mu_k\right\}\right]\\
=&\exp\left(\sum_{k=1}^{K-1}\left[\log\left(\pi_k\right)-\frac{1}{2}\log|R_k|-\frac{1}{2}\mu_k^\top R_k^{-1}\mu_k\right]N_k+\left[\mu_k^\top R_k^{-1}\right]t_{1,k}+\mathrm{tr}\left(-\frac{1}{2}R_k^{-1}t_{2,k}\right)\right)\\
=&\exp\left(\sum_{k=1}^{K-1}\left[\log\left(\pi_k\right)-\frac{1}{2}\log|R_k|-\frac{1}{2}\mu_k^\top R_k^{-1}\mu_k\right]N_k+\left[\mu_k^\top R_k^{-1}\right]t_{1,k}+\eta_{2,k}^\top\tilde{t}_{2,k}\right)
\end{aligned}
$$

Where $\theta=[\pi_0,\mu_0,R_0,\cdots,\pi_{K-1},\mu_{K-1},R_{K-1}]$ and $\psi(\theta)\equiv 0$, and $\eta_{2,k}\equiv[\underbrace{-\frac{1}{2}r_{11}^{(k)},\cdots,-\frac{1}{2}r_{MM}^{(k)}}_{\text{length }=M},\underbrace{-\frac{1}{2}r_{12}^{(k)},\cdots,-\frac{1}{2}r_{1M}^{(k)},-\frac{1}{2}r_{23}^{(k)},\cdots,-\frac{1}{2}r_{(M-1)M}^{(k)}}_{\text{length}=M(M-1)/2}]^\top$, here

$r_{ij}^{(k)}$ represents the element in the i-th row and j-th column of the inverse matrix R_k^{-1}

$\tilde{t}_{2,k}$ is the corresponding rearranged form of $t_{2,k}$

$$\vec{\eta}(\theta)\equiv\left[\log\left(\pi_k\right)-\frac{1}{2}\log|R_k|-\frac{1}{2}\mu_k^\top R_k^{-1}\mu_k,\quad \mu_k^\top R_k^{-1},\quad \eta_{2,k}\text{ for }k=0,\cdots,K-1\right]$$

$$\vec{T}\equiv[N_k,t_{1,k},\tilde{t}_{2,k}\text{ for }k=0,\cdots,K-1]$$

So, we prove that we can write PDF in such a form

$$f(\vec{y}, \vec{x} \mid \theta) = \exp\left(\vec{\eta}^\top(\theta) \cdot \vec{T} - \psi(\theta)\right) \cdot h(\vec{x}, \vec{y})$$

Furthermore, the corresponding **natural sufficient statistics** are
$\vec{T} \equiv [N_k, t_{1,k}, \tilde{t}_{2,k}$ for $k = 0, \cdots, K-1]$

Question 2.2

Show that the ML estimate of θ given (Y, X) is formed by the expressions in (15), (16), and (17). (Hint: This is not easy. You will probably need some matrix trace identities from a good text book.)

$$\hat{\mu}_k = \frac{t_{1,k}}{N_k}$$

$$\hat{R}_k = \frac{t_{2,k}}{N_k} - \frac{t_{1,k}t_{1,k}^\top}{N_k^2}$$

$$\hat{\pi}_k = \frac{N_k}{N}$$

solution

Let's write down the logarithm likelihood function, and only consider the θ part, ignore $h(\vec{x}, \vec{y})$

$$\log f(\vec{y}, \vec{x} \mid \theta) = \log \left[\exp\left(\vec{\eta}^\top(\theta) \cdot \vec{T} - \psi(\theta)\right) \cdot h(\vec{x}, \vec{y})\right] = \vec{\eta}^\top(\theta) \cdot \vec{T} - \psi(\theta) + \text{const}$$

With the derivation in Problem 2.1, we have

$$\log f(\vec{y}, \vec{x} \mid \theta) = \sum_{k=0}^{K-1} \left[\log(\pi_k) - \frac{1}{2}\log|R_k| - \frac{1}{2}\mu_k^\top R_k^{-1}\mu_k\right] N_k + \left[\mu_k^\top R_k^{-1}\right] t_{1,k} + \text{tr}\left(-\frac{1}{2}R_k^{-1}t_{2,k}\right)$$

Then consider the critical point $\theta = \hat{\theta}$ of the logarithm likelihood function under the constrain $\sum_{k=0}^{K-1} \pi_k - 1 = 0$.

Let the Lagrangian to be $L(\theta, \lambda) \equiv \log f(\vec{y}, \vec{x} \mid \theta) - \lambda(\sum\limits_{k=0}^{K-1} \pi_k - 1)$ it must satisfy that,

(note: $R_k^\top = R_k$)

$$\begin{aligned}
\frac{\partial L(\theta,\lambda)}{\partial \pi_k}\Big|_{\theta=\hat{\theta}} &= \frac{N_k}{\hat{\pi}_k} - \lambda = 0 \\
\frac{\partial L(\theta,\lambda)}{\partial \mu_k}\Big|_{\theta=\hat{\theta}} &= -\hat{R}_k^{-1}\hat{\mu}_k N_k + \hat{R}_k^{-1} t_{1,k} = \hat{R}_k^{-1}\left[-\hat{\mu}_k N_k + t_{1,k}\right] = \vec{0} \\
\frac{\partial L(\theta,\lambda)}{\partial R_k^{-1}}\Big|_{\theta=\hat{\theta}} &= -\frac{1}{2}N_k\hat{R}_k^\top - \frac{1}{2}(N_k\hat{\mu}_k\hat{\mu}_k^\top)^\top + (t_{1,k}\hat{\mu}_k^\top)^\top - \frac{1}{2}t_{2,k}^\top \\
= \frac{1}{2}[N_k\hat{R}_k - N_k\hat{\mu}_k\hat{\mu}_k^\top + 2t_{1,k}\hat{\mu}_k^\top - t_{2,k}]^\top &= 0_{M\times M} \\
\frac{\partial L(\theta,\lambda)}{\partial \lambda}\Big|_{\theta=\hat{\theta}} &= \sum_{k=0}^{K-1}\hat{\pi}_k - 1 = 0
\end{aligned}$$

With the first and the last equations above, we conclude

$$\lambda = \frac{1}{N} \Leftarrow 1 = \sum_{k=0}^{K-1}\hat{\pi}_k = \lambda\sum_{k=0}^{K-1} N_k = \lambda N$$

$$\hat{\pi}_k = \lambda N_k = \frac{N_k}{N}$$

Since we assume $\hat{R}_k^{-1}$ to be full rank, we conclude that from the second equation

$$\hat{\mu}_k = \frac{t_{1,k}}{N_k}$$

Use the equation above and consider the third equation, we have

$$\hat{R}_k = \frac{1}{N_k}\left(N_k \hat{\mu}_k \hat{\mu}_k^\top - 2t_{1,k}\hat{\mu}_k^\top + t_{2,k}\right) = \frac{t_{2,k}}{N_k} - \frac{t_{1,k}t_{1,k}^\top}{N_k^2}$$

In the end, we conclude that ML estimate of θ given (Y, X) is formed by the expressions below

$$\begin{aligned}
\hat{\pi}_k &= \frac{N_k}{N} \\
\hat{\mu}_k &= \frac{t_{1,k}}{N_k} \\
\hat{R}_k &= \frac{t_{2,k}}{N_k} - \frac{t_{1,k}t_{1,k}^\top}{N_k^2}
\end{aligned}$$

3. Parameter Estimation for Gaussian Mixture Distributions

Question 3.1

Use the Matlab program `mk_data.m` to create 500 samples from a Gaussian mixture with $M = 2, K = 3, \pi = [0.4, 0.4, 0.2]$

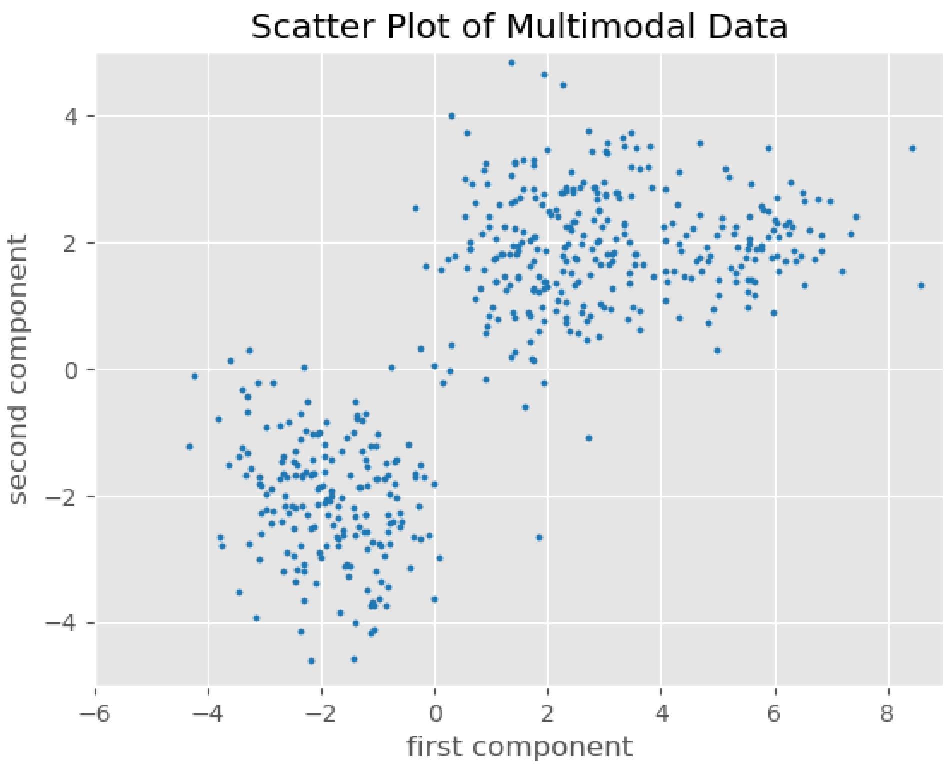
$$\begin{aligned} \mu_0 &= [2, 2]^\top \\ \mu_1 &= [-2, -2]^\top \\ \mu_2 &= [5.5, 2]^\top \end{aligned}$$

and

$$\begin{aligned} R_0 &= \begin{bmatrix} 1 & 0.1 \\ 0.1 & 1 \end{bmatrix} \\ R_1 &= \begin{bmatrix} 1 & -0.1 \\ -0.1 & 1 \end{bmatrix} \\ R_2 &= \begin{bmatrix} 1 & 0.2 \\ 0.2 & 0.5 \end{bmatrix} \end{aligned}$$

solution

The generated data is displayed below:



We generate data `data.txt` and labels `label.txt` by running the following python script **mk_data.py**

The python script **mk_data.py** is

```python
import numpy as np
from numpy import sqrt, diag, transpose
from numpy.linalg import eig
from numpy.random import seed, randn, random_sample
from os.path import join, abspath, dirname
import matplotlib.pyplot as plt

def generate_data(R, mu, N, se=0):
    # generate N points for each cluster
    # V[:,i] is the eigenvector corresponding to the eigenvalue D[i]
    # X~N(0, D), and Y = V X <=> V^T Y = X then Y~N(0, V D V^T)
    # W~N(0, I), sqrt(D) W~N(0, D), V sqrt(D) W~N(0, V D V^T)~N(0, R)
    # with decomposition: R = V D V^T
    seed(se)
    D, V = eig(R)
    return transpose(V @ diag(sqrt(D)) @ randn(2, N) + mu)

def choose_data(X, pi, se=0):
    N = len(X[0])
    seed(se); switch = random_sample(N)
    def get_category(x):
        sum = 0
        for c, p in enumerate(pi):
            if sum <= x and x < sum + p:
                return c
            sum += p
    label = list(map(get_category, switch))
    return np.asarray([X[c][i] for i, c in enumerate(label)]), label

def plot_data(data):
    x1, x2 = data[:, 0], data[:, 1]
    plt.style.use('ggplot')
    plt.scatter(x1, x2, s=5, marker='o', c='#1f77b4')
    plt.title('Scatter Plot of Multimodal Data')
    plt.xlim([-6, 9]); plt.ylim([-5, 5])
    plt.xlabel('first component')
    plt.ylabel('second component')
    path_save = join(dirname(abspath(__file__)), 'data.png')
    plt.savefig(path_save,
                bbox_inches='tight',
                pad_inches=0)
    plt.show()

def save_data(data, label):
    path_data  = join(dirname(abspath(__file__)), 'data.txt')
    path_label = join(dirname(abspath(__file__)), 'label.txt')
    np.savetxt(path_data, data, fmt='%16.7e', delimiter='',
newline='\n')
```

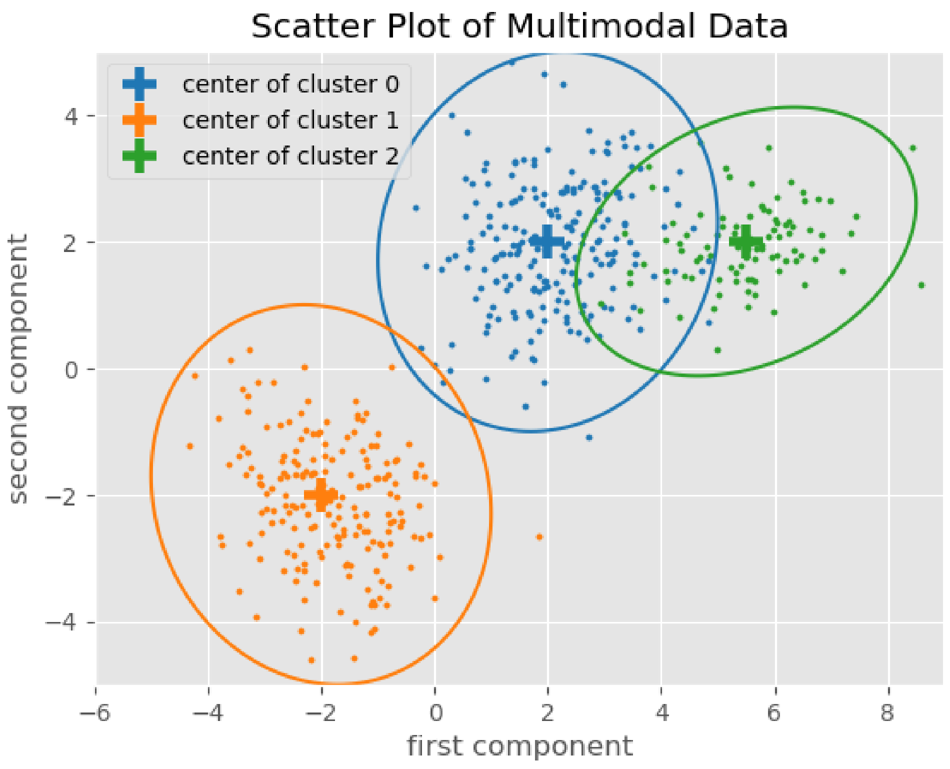
```
    np.savetxt(path_label, label, fmt='%d', delimiter=' ',
newline='\n')

if __name__ == "__main__":
    N = 500 # total number of generated points
    R0 = np.asarray([[1, 0.1],[0.1, 1]])
    mu0 = np.asarray([[2], [2]])
    R1 = np.asarray([[1,-0.1],[-0.1,1]])
    mu1 = np.asarray([[-2], [-2]])
    R2 = np.asarray([[1,0.2],[0.2,0.5]])
    mu2 = np.asarray([[5.5], [2]])
    X_all = [generate_data(R, mu, N, se) \
            for R, mu, se in \
            list(zip([R0, R1, R2], [mu0, mu1, mu2], [0, 1, 2]))]
    pi = [0.4, 0.4, 0.2]
    data, label = choose_data(X_all, pi, se=19)
    plot_data(data)
    save_data(data, label)
```

Question 3.2

Print out a scatter plot of the samples generated in step 1. Circle and label each of the three clusters in the mixture distribution.

solution

The circled and labeled three clusters in the mixture distribution are displayed below, see `soln_3_2.png`:



We circle and label each of the three clusters in the mixture distribution by running the following python script **soln_3_2.py**, thus generate image `soln_3_2.png`

The python script **soln_3_2.py** is

```
import sys
from os.path import join, abspath, dirname
sys.path.insert(0, dirname(dirname(__file__)))
import numpy as np
import matplotlib.pyplot as plt
from src.utils import circle_cluster, plot_data_label,
plot_cluster_center


if __name__ == "__main__":
    path_data  = join(dirname(dirname(abspath(__file__))),
'data/data.txt')
    path_label = join(dirname(dirname(abspath(__file__))),
'data/label.txt')
    data = np.loadtxt(path_data, dtype='float', delimiter=None)
    label = np.loadtxt(path_label, dtype=np.int32, delimiter=None)
    plot_data_label(data, label)
    R0 = np.asarray([[1, 0.1],[0.1, 1]])
    mu0 = np.asarray([[2], [2]])
    R1 = np.asarray([[1,-0.1],[-0.1,1]])
    mu1 = np.asarray([[-2], [-2]])
    R2 = np.asarray([[1,0.2],[0.2,0.5]])
    mu2 = np.asarray([[5.5], [2]])
    list(map(circle_cluster, [R0, R1, R2], [mu0, mu1, mu2], [0, 1,
2]))
    list(map(plot_cluster_center, [0, 1, 2], [mu0, mu1, mu2]))
    path_save = join(dirname(abspath(__file__)), 'soln_3_2.png')
    plt.savefig(path_save,
                bbox_inches='tight',
                pad_inches=0)
    plt.show()
```

Question 3.3

Derive an explicit expression for $P\left\{X_n = k \mid Y = y, \hat{\theta}\right\}$ used in in the E-step

solution

We can write the conditional PDF with Bayesian formula

$$P\left\{X_n = k \mid Y = y, \hat{\theta}\right\} \equiv \frac{f_{X,Y}(k, y \mid \hat{\theta})}{f_Y(y \mid \hat{\theta})} = \frac{f_{X,Y}(k, y \mid \hat{\theta})}{\sum\limits_{X} f_{X,Y}(k, y \mid \hat{\theta})}$$

$$= \frac{f_{Y|X}(y \mid k, \hat{\theta}) \cdot p_X(k \mid \hat{\theta})}{\sum\limits_{k=0}^{K-1} f_{Y|X}(y \mid k, \hat{\theta}) \cdot p_X(k \mid \hat{\theta})}$$

$$= \frac{f_{Y|X}(y \mid \hat{\mu}_k, \hat{R}_k) \cdot p_X(k \mid \hat{\pi}_k)}{\sum\limits_{k=0}^{K-1} f_{Y|X}(y \mid \hat{\mu}_k, \hat{R}_k) \cdot p_X(k \mid \hat{\pi}_k)}$$

notice that

$$\hat{\pi}_k \equiv p_X(k \mid \hat{\pi}_k)$$

$$f_{Y|X}(y \mid \hat{\mu}_k, \hat{R}_k) = (2\pi)^{-\frac{M}{2}} |\hat{R}_k|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(y - \hat{\mu}_k)^\top \hat{R}_k^\top (y - \hat{\mu}_k)\right\}$$

In the end, we Derive an explicit expression for $P\left\{X_n = k \mid Y = y, \hat{\theta}\right\}$

$$P\left\{X_n = k \mid Y = y, \hat{\theta}\right\} = \frac{\hat{\pi}_k |\hat{R}_k|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(y - \hat{\mu}_k)^\top \hat{R}_k^\top (y - \hat{\mu}_k)\right\}}{\sum\limits_{k=0}^{K-1} \hat{\pi}_k |\hat{R}_k|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(y - \hat{\mu}_k)^\top \hat{R}_k^\top (y - \hat{\mu}_k)\right\}}$$

Question 3.4

Implement the EM algorithm for computing the ML estimate of θ. Use the initial value for θ of $\pi \leftarrow [1/3, 1/3, 1/3]$

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ for } k = 0, 1, 2$$

and select μ_1, μ_2, and μ_3 as the first three sample vectors produced in step 1 above.

solution

We implement the EM algorithm with Gaussian mixture Model in `class GaussianMixture` of **GMM.py**

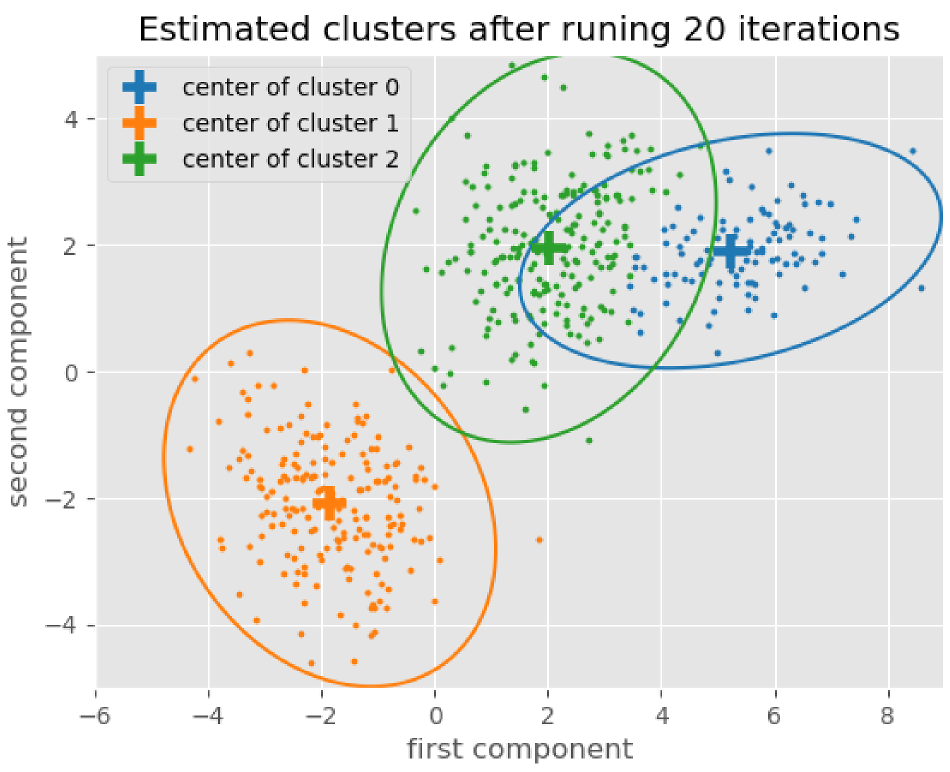
```
import numpy as np
from numpy import empty, eye, log, exp, square, pi, inf, squeeze,
amax, mean
from scipy.linalg import cholesky, solve_triangular, LinAlgError
import warnings


class GaussianMixture(object):
    def __init__(self,
                n_clusters=1,
                tol=None,
                max_iter=1000,
                weights_init=None,
                means_init=None,
                precisions_init=None,
                random_state=None,
                ignore_converged=False):
        # random number generator instance controls the random seed
        # used for the method chosen to initialize the parameters.
        self.n_clusters = n_clusters
        self.tol_init = tol
        self.max_iter = max_iter
        self.weights_init = weights_init
        self.means_init = means_init
        self.precisions_init = precisions_init
        self.random_state = random_state
        self.ingore_converged=ignore_converged
        self.converged = False
        self.iteration_mdl = []


    def _initialize_params(self, X):
        n_samples, n_features = X.shape
        self.tol = 1e-6 * ((1 + n_features + n_features*
(n_features+1)/2) / 100) \
            * log(n_samples * n_features) if self.tol_init is None
else self.tol_init
        assert((self.weights_init is None) \
            or (self.weights_init.shape == (self.n_clusters,)))
        assert((self.means_init is None) \
            or (self.means_init.shape == (self.n_clusters,
n_features)))
        assert((self.precisions_init is None) \
            or (self.precisions_init.shape == (self.n_clusters,
n_features, n_features)))
        if self.random_state is None:
            random_state = np.random.mtrand._rand
        if isinstance(self.random_state, int):
            random_state = np.random.RandomState(random_state)
        if isinstance(self.random_state, np.random.RandomState):
            random_state = self.random_state
        resp = random_state.rand(n_samples, self.n_clusters) # N*K
        resp /= resp.sum(axis=1, keepdims=True)
        weights, means, covariances =
self._estimate_gaussian_params(X, resp)
        self.weights = weights if self.weights_init is None else
self.weights_init
        self.means = means if self.means_init is None else
self.means_init
        self.precisions_cholesky =
self._compute_precision_cholesky(covariances) \
            if self.precisions_init is None else \
                np.array([cholesky(prec_init, lower=True)
                for prec_init in self.precisions_init])
        self.covariances = covariances if self.precisions_init is
None else None


    def _get_parameters(self):
        return (
            self.weights,
            self.means,
            self.covariances,
            self.precisions_cholesky,
        )


    def _set_parameters(self, params):
        (
            self.weights,
            self.means,
            self.covariances,
            self.precisions_cholesky,
        ) = params
        # Attributes computation
        self.precisions = empty(self.precisions_cholesky.shape)
        for k, prec_chol in enumerate(self.precisions_cholesky):
            self.precisions[k] = prec_chol @ prec_chol.T


    def _n_parameters(self):
        """Return the number of free parameters in the model."""
        _, n_features = self.means.shape
        cov_params = self.n_clusters * n_features * (n_features + 1)
/ 2.0
        mean_params = n_features * self.n_clusters
        return int(cov_params + mean_params + self.n_clusters - 1)


    def _compute_log_det_cholesky(self, matrix_chol, n_features):
        n_clusters, _, _ = matrix_chol.shape
        return np.sum(log(matrix_chol.reshape(n_clusters, -1)[:, ::
n_features + 1]), 1)


    def _estimate_log_gaussian_prob(self, X, means, precisions_chol):
        n_samples, n_features = X.shape
        n_clusters, _ = means.shape
        # log(det(precision_chol)) is half of log(det(precision))
        log_det = self._compute_log_det_cholesky(precisions_chol,
n_features)
        log_prob = empty((n_samples, n_clusters))
        for k, (mu, prec_chol) in enumerate(zip(means,
precisions_chol)):
            y = (X-mu) @ prec_chol
            log_prob[:, k] = np.sum(square(y), axis=1)
        return -0.5 * (n_features * log(2 * pi) + log_prob) + log_det


    def _estimate_log_prob(self, X):
        """Estimate the log-probabilities, log P(X | Z)"""
        return self._estimate_log_gaussian_prob(X, self.means,
self.precisions_cholesky)


    def logsumexp(self, a, axis=None):
        a_max = amax(a, axis=axis, keepdims=True)
        tmp = exp(a - a_max)
        # suppress warnings about log of zero
        with np.errstate(divide='ignore'):
            s = np.sum(tmp, axis=axis)
            out = log(s)
        a_max = squeeze(a_max, axis=axis)
        out += a_max
        return out


    def score(self, X):
        """Compute log P(X | sigma) log-likelihood of the given data
X."""
        # ``np.log(np.sum(np.exp(a)))`` calculated in a numerically
stable way
        return self.logsumexp(self._estimate_log_prob(X) +
log(self.weights), axis=1).sum()


    def aic(self, X):
        return -2 * self.score(X) + 2 * self._n_parameters()


    def mdl(self, X):
        return -self.score(X) + 0.5 * self._n_parameters() *
log(X.shape[0] * X.shape[1])


    def _get_iteration_mdl(self):
        return self.iteration_mdl


    def _estimate_log_prob_resp(self, X):
        weighted_log_prob = self._estimate_log_prob(X) +
log(self.weights)
        log_prob_norm = self.logsumexp(weighted_log_prob, axis=1)
        with np.errstate(under="ignore"):
            # ignore underflow
            log_resp = weighted_log_prob - log_prob_norm[:,
np.newaxis]
        return log_prob_norm, log_resp


    def _e_step(self, X):
        log_prob_norm, log_resp = self._estimate_log_prob_resp(X)
        return mean(log_prob_norm), log_resp


    def _m_step(self, X, log_resp):
        self.weights, self.means, self.covariances \
            = self._estimate_gaussian_params(X, exp(log_resp))
        self.precisions_cholesky \
            = self._compute_precision_cholesky(self.covariances)


    def fit(self, X):
        self._initialize_params(X)
        lower_bound = -inf
        self.iteration_mdl.append(self.mdl(X))
        for n_iter in range(1, self.max_iter+1):
            lower_bound_prev = lower_bound
            log_prob_norm, log_resp = self._e_step(X)
            self._m_step(X, log_resp)
            lower_bound = log_prob_norm
            self.iteration_mdl.append(self.mdl(X))
            if (not self.ingore_converged) \
                and abs(lower_bound - lower_bound_prev) < self.tol:
                self.converged = True
                break
        if not (self.ingore_converged or self.converged):
            warnings.warn(
                "Did not converge. "
                "Try different init parameters, "
                "or increase max_iter, tol "
                "or check for degenerate data."
            )
        self.n_iter_ = n_iter
        self.lower_bound_ = lower_bound
        _, log_resp = self._e_step(X)
        return log_resp.argmax(axis=1)


    def predict(self, X):
        msg = (
            "This instance is not fitted yet. Call 'fit' with "
            "appropriate arguments before using this estimator."
        )
        if not (self.ingore_converged or self.converged):
            raise ValueError(msg)
        return (self._estimate_log_prob(X) +
log(self.weights)).argmax(axis=1)


    def predict_prob(self, X): # return P(Z | X)
        msg = (
            "This instance is not fitted yet. Call 'fit' with "
            "appropriate arguments before using this estimator."
        )
        if not (self.ingore_converged or self.converged):
            raise ValueError(msg)
        _, log_resp = self._estimate_log_prob_resp(X)
        return exp(log_resp)


    def _estimate_gaussian_params(self, X, resp): # X N*d, resp N*k
        weights = resp.sum(axis=0) / resp.sum()
        means = (resp.T @ X) / resp.sum(axis=0)[:, np.newaxis]
        n_clusters, n_features = means.shape
        covariances = empty((n_clusters, n_features, n_features))
        for k, mu, gamma in zip(range(n_clusters), means, resp.T):
            d = X - mu
            covariances[k]  = ((d.T * gamma) @ d) / gamma.sum(axis=0)
            covariances[k].flat[:: n_features+1] += 1e-6 # add eps to
diag element
        return weights, means, covariances


    def _compute_precision_cholesky(self, covariances):
        estimate_precision_error_message = (
            "Fitting the mixture model failed because some components
have "
            "ill-defined empirical covariance (for instance caused by
singleton "
            "or collapsed samples). Try to decrease the number of
components, "
            "or increase reg_covar."
        )
        n_clusters, n_features, _ = covariances.shape
        precisions_chol = empty((n_clusters, n_features, n_features))
        for k, covariance in enumerate(covariances):
            try:
                # return L, where Sigma = L * L^T
                # conv_chol is L, covariance is Sigma
                # cov_chol*cov_chol^T = covariance
                cov_chol = cholesky(covariance, lower=True)
            except LinAlgError:
                raise ValueError(estimate_precision_error_message)
            # return L^{-T}
            # precisions_chol is L^{-T}
            precisions_chol[k] \
                = solve_triangular(cov_chol, eye(n_features),
lower=True).T #
        return precisions_chol
```

Question 3.5

Run 20 iterations of the EM algorithm, and print out the values of the estimated parameters. Do the indices k of the estimated and true parameters correspond?

solution

The estimated clusters are shown below, see `soln_3_4.png`:



The values of the estimated parameters are printed as follows by running **soln_3_4.py** script. Obviously, the indices k of the estimated and true parameters **don't correspond**

```
pi estimated:
[0.21477052 0.38189303 0.40333645]

mu estimated:
[[ 5.22579585  1.90021554]
 [-1.84617794 -2.0906876 ]
 [ 2.01958642  1.95505247]]

R estimated:
[[[ 1.53730153  0.22464551]
  [ 0.22464551  0.38172451]]

 [[ 0.95367803 -0.23517409]
  [-0.23517409  0.93092094]]

 [[ 0.96988047  0.23086724]
  [ 0.23086724  1.05438756]]]
```

We run 20 iterations of the EM algorithm, and print out the values of the estimated parameters by running the following python script **soln_3_4.py**, thus generate image `soln_3_4.png`. The python script **soln_3_4.py** is

```python
import sys
from os.path import join, abspath, dirname
sys.path.insert(0, dirname(dirname(__file__)))
import numpy as np
from numpy import ones, eye, array
import matplotlib.pyplot as plt
from src.GMM import GaussianMixture
from src.utils import plot_data_label, circle_cluster,
plot_cluster_center

if __name__ == '__main__':
    path_data  = join(dirname(dirname(abspath(__file__))),
'data/data.txt')
    data = np.loadtxt(path_data, dtype='float', delimiter=None)
    n_samples, n_features = data.shape
    n_clusters, n_iterations = 3, 20
    weights_init = ones(n_clusters) / n_clusters
    means_init = data[:n_clusters, :]
    precisions_init = array([eye(n_features) for i in
range(n_clusters)])
    gm = GaussianMixture(n_clusters=n_clusters,
                         max_iter=n_iterations,
                         weights_init=weights_init,
                         means_init=means_init,
                         precisions_init=precisions_init,
                         ignore_converged=True)
    gm.fit(data)
    label_pred = gm.predict(data)
    weights, means, covariances, _ = gm._get_parameters()
    print("pi:\n{}\n".format(weights))
    print("mu:\n{}\n".format(means))
    print("R:\n{}\n".format(covariances))
    plot_data_label(data, label_pred)
    list(map(circle_cluster, covariances, means, range(n_clusters)))
    list(map(plot_cluster_center, range(n_clusters), means))
    plt.title("Estimated clusters after runing {}
iterations".format(n_iterations))
    path_save = join(dirname(abspath(__file__)), 'soln_3_4.png')
    plt.savefig(path_save,
                bbox_inches='tight',
                pad_inches=0)
    plt.show()
```

Question 3.6

Find the best correspondence of the true and estimated parameters, and print them out in a tabular form comparing their values.

solution

We find the best correspondence of the true and estimated parameters with function `get_correspondece()` in **soln_3_6.py**, then print them out in a tabular form comparing their values.

```
pi estimated:
[0.40333645 0.38189303 0.21477052]

pi true:
[0.4 0.4 0.2]

mu estimated:
[[ 2.01958642  1.95505247]
 [-1.84617794 -2.0906876 ]
 [ 5.22579585  1.90021554]]

mu true:
[[ 2.   2. ]
 [-2.  -2. ]
 [ 5.5  2. ]]

R estimated:
[[[ 0.96988047  0.23086724]
  [ 0.23086724  1.05438756]]

 [[ 0.95367803 -0.23517409]
  [-0.23517409  0.93092094]]

 [[ 1.53730153  0.22464551]
  [ 0.22464551  0.38172451]]]

R true:
[[[ 1.   0.1]
  [ 0.1  1. ]]

 [[ 1.  -0.1]
  [-0.1  1. ]]

 [[ 1.   0.2]
  [ 0.2  0.5]]]
```

We run 20 iterations of the EM algorithm to get the values of the estimated parameters, and find the best correspondence of the true and estimated parameters, and print them out in a tabular form comparing their values by running the following python script **soln_3_6.py**. The python script **soln_3_6.py** is

```
import sys
from os.path import join, abspath, dirname
sys.path.insert(0, dirname(dirname(__file__)))
import numpy as np
from numpy import ones, eye, array
import matplotlib.pyplot as plt
from src.GMM import GaussianMixture

def get_correspondece(means_esitimated, means_true):
    n_clusters = len(means_true)
    means_e, means_t = means_esitimated, means_true
    def _get_correspondence(ind_estimated, ind_true):
        if len(ind_true) <= 1:
            return ind_estimated, ind_true
        record_min = [np.inf, 0, 0]
        for ind_e, mean_e in list(zip(ind_estimated,
means_e[ind_estimated])):
            distances = np.square(means_t[ind_true] -
mean_e).sum(axis=1)
            ind = np.argmin(distances)
            d_min, ind_t = distances[ind], ind_true[ind]
            if d_min < record_min[0]:
                record_min = [d_min, ind_e, ind_t]
        ind_estimated_new, ind_true_new = ind_estimated.copy(),
ind_true.copy()
        ind_estimated_new.remove(record_min[1])
        ind_true_new.remove(record_min[2])
        ind_e_correspond, ind_t_correspond = _get_correspondence(
            ind_estimated_new,
            ind_true_new)
        ind_e_correspond.append(record_min[1])
        ind_t_correspond.append(record_min[2])
        return ind_e_correspond, ind_t_correspond
    ind_estimated, ind_true =
_get_correspondence(list(range(n_clusters)), list(range(n_clusters)))
    ind_combine = list(zip(ind_estimated, ind_true))
    ind_combine.sort(key=lambda item: item[1])
    indices_estimated, _ = zip(*ind_combine)
    return list(indices_estimated)


if __name__ == '__main__':
    path_data  = join(dirname(dirname(abspath(__file__))),
'data/data.txt')
    data = np.loadtxt(path_data, dtype='float', delimiter=None)
    n_samples, n_features = data.shape
    n_clusters, n_iterations = 3, 20
    weights_init = ones(n_clusters) / n_clusters
    means_init = data[:n_clusters, :]
    precisions_init = array([eye(n_features) for i in
range(n_clusters)])
    gm = GaussianMixture(n_clusters=n_clusters,
max_iter=n_iterations,
                        weights_init=weights_init,
                        means_init=means_init,
                        precisions_init=precisions_init,
                        ignore_converged=True)
    gm.fit(data)
    label_pred = gm.predict(data)
    weights, means, covariances, _ = gm._get_parameters()
    mu0 = np.asarray([2, 2])
    mu1 = np.asarray([-2, -2])
    mu2 = np.asarray([5.5, 2])
    means_true = np.stack((mu0, mu1, mu2), axis=0)
    indices = get_correspondece(means, means_true)
    R0 = np.asarray([[1, 0.1],[0.1, 1]])
    R1 = np.asarray([[1,-0.1],[-0.1,1]])
    R2 = np.asarray([[1,0.2],[0.2,0.5]])
    covariances_true = np.stack((R0, R1, R2), axis=0)
    weights_true = np.array([0.4, 0.4, 0.2])
    print("pi estimated:\n{}\n".format(weights[indices]))
    print("pi true:\n{}\n".format(weights_true))
    print("mu estimated:\n{}\n".format(means[indices]))
    print("mu true:\n{}\n".format(means_true))
    print("R estimated:\n{}\n".format(covariances[indices]))
    print("R true:\n{}\n".format(covariances_true))
```

4. Order Identification for Gaussian Mixture Distributions

Question 4.1

Use the data generated in problem 1 of the previous Section 3.

solution

In the code of **soln_4_2.py**, we read data from `data.txt`, and the corresponding part in **soln_4_2.py** is

```
import sys
from os.path import join, abspath, dirname
sys.path.insert(0, dirname(dirname(__file__)))
import numpy as np
from numpy import ones, eye, array
import matplotlib.pyplot as plt
from src.GMM import GaussianMixture, OrderIndentification
from src.utils import plot_data_label, circle_cluster, \
    plot_cluster_center_merged, plot_cluster_center

if __name__ == '__main__':
    path_data  = join(dirname(dirname(abspath(__file__))),
'data/data.txt')
    data = np.loadtxt(path_data, dtype='float', delimiter=None)
```

Question 4.2

Implement the MDL order estimation method as described above using and initial value of $K = 9$. Use the initial value for θ of $\pi \leftarrow [1/9, \cdots, 1/9]$

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ for } k = 0, \cdots, 9$$

and select $\mu_1, \mu_2, \cdots, \mu_9$ as the first nine sample vectors produced in step 1 above. For each value of K, run 20 iterations of the EM algorithm

solution

We implement MDL order estimation method with Gaussian Mixture Model in `class OrderIndentification` of **GMM.py**

```
import numpy as np
from numpy import empty, eye, log, exp, square, pi, inf, squeeze,
amax, mean
from scipy.linalg import cholesky, solve_triangular, LinAlgError
import warnings

class OrderIndentification(object):
    def __init__(self,
                weights,
                means,
                covariances,
                precisions_cholesky):
        self.weights = weights
        self.means = means
        self.covariances = covariances
        self.precisions_cholesky = precisions_cholesky

    def _compute_merged_params(self, l, m):
        weight = self.weights[l] + self.weights[m]
        mean = (self.weights[l]*self.means[l] \
            + self.weights[m]*self.means[m]) / weight
        dl = (self.means[l] - mean)[np.newaxis, :]
        dm = (self.means[m] - mean)[np.newaxis, :]
        covariance = self.weights[l] * (self.covariances[l] + dl.T @
dl)\
            + self.weights[m] * (self.covariances[m] + dm.T @ dm)
        covariance /= weight
        return weight, mean, covariance

    def _compute_distance_cluster(self, l, m):
        _, _, covariance = self._compute_merged_params(l, m)
        # log(det(cov_chol)) is half of log(det(covariance)), note L
* L^T = R
```

```
        n_features, _ = covariance.shape
        cov_chol = cholesky(covariance, lower=True)
        log_det_cov_chol = log(cov_chol.flat[:: n_features +
1]).sum()
        log_det_precision_chol_l \
            = log(self.precisions_cholesky[l].flat[:: n_features +
1]).sum()
        log_det_precision_chol_m \
            = log(self.precisions_cholesky[m].flat[:: n_features +
1]).sum()
        return self.weights[l] * (log_det_cov_chol +
log_det_precision_chol_l)\
            + self.weights[m] * (log_det_cov_chol +
log_det_precision_chol_m)

    def merge(self):
        d_min = inf
        n_clusters = self.weights.shape[0]
        if n_clusters <= 1:
            return
        for l in range(n_clusters-1):
            for m in range(l+1, n_clusters):
                d = self._compute_distance_cluster(l, m)
                if (d < d_min):
                    d_min = d
                    ind_cluster = [l, m]
        self.l, self.m = ind_cluster
        self.weight_merged, self.mean_merged, self.covariance_merged
\
            = self._compute_merged_params(self.l, self.m)
        n_features, _ = self.covariance_merged.shape
        cov_chol = cholesky(self.covariance_merged, lower=True)
        self.precision_chol_merged \
                = solve_triangular(cov_chol, eye(n_features),
lower=True).T
        self.weights = np.delete(self.weights, m, axis=0)
        self.means = np.delete(self.means, m, axis=0)
        self.covariances = np.delete(self.covariances, m, axis=0)
        self.precisions_cholesky =
np.delete(self.precisions_cholesky, m, axis=0)
        self.weights[l] = self.weight_merged
        self.means[l] = self.mean_merged
        self.covariances[l] = self.covariance_merged
        self.precisions_cholesky[l] = self.precision_chol_merged

    def _get_params_merged(self):
        return (
            self.weight_merged,
            self.mean_merged,
            self.covariance_merged,
            self.precision_chol_merged,
        )

    def _get_index_cluster_merged(self):
        return (self.l, self.m)

    def _get_parameters(self):
        return (
            self.weights,
            self.means,
            self.covariances,
            self.precisions_cholesky,
        )
```

We test the MDL order estimation method in **soln_4_2.py**, for each value of $K = 9, 8, \cdots, 1$, run 20 iterations of the EM algorithm. The python script **soln_4_2.py** is

```
import sys
from os.path import join, abspath, dirname
sys.path.insert(0, dirname(dirname(__file__)))
import numpy as np
from numpy import ones, eye, array
import matplotlib.pyplot as plt
from src.GMM import GaussianMixture, OrderIndentification
from src.utils import plot_data_label, circle_cluster, \
    plot_cluster_center_merged, plot_cluster_center, \
    plot_iteration_mdl, plot_list_mdl

if __name__ == '__main__':
    path_data  = join(dirname(dirname(abspath(__file__))),
'data/data.txt')
    data = np.loadtxt(path_data, dtype='float', delimiter=None)
    dict_gm, list_mdl, iteration_mdl = {}, [], []
    n_samples, n_features = data.shape
    n_clusters_init, n_iterations = 9, 20
    weights_init = ones(n_clusters_init) / n_clusters_init
    means_init = data[:n_clusters_init, :]
    precisions_init = array([eye(n_features) for i in
range(n_clusters_init)])
    for k in range(n_clusters_init, 0, -1):
        dict_gm[k] = GaussianMixture(n_clusters=k,
                                     max_iter=n_iterations,
                                     weights_init=weights_init,
                                     means_init=means_init,
                                     precisions_init=precisions_init,
                                     ignore_converged=True)
        dict_gm[k].fit(data)
        iteration_mdl += dict_gm[k]._get_iteration_mdl()
        # assert(dict_gm[k].converged)
        list_mdl.append((k, dict_gm[k].mdl(data)))
        order = OrderIndentification(*dict_gm[k]._get_parameters())
        order.merge()
        weights_init, means_init, _, precisions_init =
order._get_parameters()
        # predict and plot merged clusters
        label_pred = dict_gm[k].predict(data)
        plot_data_label(data, label_pred)
        plt.title("Estimated {} clusters after runing {}
iterations".format(k, n_iterations))
        _, means, covariances, _ = dict_gm[k]._get_parameters()
        list(map(circle_cluster, covariances, means, range(k)))
        if k >= 2:
            l, m = order._get_index_cluster_merged()
            mean_l, mean_m = means[l], means[m]
            _, mean_merged, covariance_merged, _ =
order._get_params_merged()
            circle_cluster(covariance_merged, mean_merged, k,
linestyles='dashed')
            plot_cluster_center_merged(l, m, k, mean_l, mean_m,
mean_merged)
        img_name = 'soln_4_4_cluster' + str(k) + '.png'
        path_save = join(dirname(abspath(__file__)), img_name)
        plt.savefig(path_save, bbox_inches='tight',
                    pad_inches=0)
        plt.show()
    plot_iteration_mdl(iteration_mdl)
    path_save = join(dirname(abspath(__file__)), 'soln_4_4.png')
    plt.savefig(path_save, bbox_inches='tight',
                pad_inches=0)
    plt.show()
    plot_list_mdl(list_mdl)
    path_save = join(dirname(abspath(__file__)), 'soln_4_5.png')
    plt.savefig(path_save, bbox_inches='tight',
                pad_inches=0)
    plt.show()
    n_clusters, mdl = min(list_mdl, key=lambda item: item[1])
    label_pred = dict_gm[n_clusters].predict(data)
    plot_data_label(data, label_pred)
    plt.title("Estimated clusters with order indentification K=
{}".format(n_clusters))
    _, means, covariances, _ = dict_gm[n_clusters]._get_parameters()
    list(map(circle_cluster, covariances, means, range(n_clusters)))
    list(map(plot_cluster_center, range(n_clusters), means))
    path_save = join(dirname(abspath(__file__)), 'soln_4_6.png')
    plt.savefig(path_save, bbox_inches='tight',
                pad_inches=0)
    plt.show()
```

Question 4.3

Implement a Matlab subroutine that computes the MDL criteria for specified values of K, θ, and y

solution

We implemented MDL in the method `mdl()` of `class GaussianMixture` in **GMM.py**

The corresponding codes in `class GaussianMixture` are

```
    def mdl(self, X):
        return -self.score(X) + 0.5 * self._n_parameters() *
log(X.shape[0] * X.shape[1])


    def _n_parameters(self):
        """Return the number of free parameters in the model."""
        _, n_features = self.means.shape
        cov_params = self.n_clusters * n_features * (n_features + 1)
/ 2.0
        mean_params = n_features * self.n_clusters
        return int(cov_params + mean_params + self.n_clusters - 1)


    def _compute_log_det_cholesky(self, matrix_chol, n_features):
        n_clusters, _, _ = matrix_chol.shape
        return np.sum(log(matrix_chol.reshape(n_clusters, -1)[:, ::
n_features + 1]), 1)


    def _estimate_log_gaussian_prob(self, X, means, precisions_chol):
        n_samples, n_features = X.shape
        n_clusters, _ = means.shape
        # log(det(precision_chol)) is half of log(det(precision))
        log_det = self._compute_log_det_cholesky(precisions_chol,
n_features)
        log_prob = empty((n_samples, n_clusters))
        for k, (mu, prec_chol) in enumerate(zip(means,
precisions_chol)):
            y = (X-mu) @ prec_chol
            log_prob[:, k] = np.sum(square(y), axis=1)
        return -0.5 * (n_features * log(2 * pi) + log_prob) + log_det


    def _estimate_log_prob(self, X):
        """Estimate the log-probabilities, log P(X | Z)"""
        return self._estimate_log_gaussian_prob(X, self.means,
self.precisions_cholesky)
```

```
    def logsumexp(self, a, axis=None):
        a_max = amax(a, axis=axis, keepdims=True)
        tmp = exp(a - a_max)
        # suppress warnings about log of zero
        with np.errstate(divide='ignore'):
            s = np.sum(tmp, axis=axis)
            out = log(s)
        a_max = squeeze(a_max, axis=axis)
        out += a_max
        return out

    def score(self, X):
        """Compute log P(X | sigma) log-likelihood of the given data
X."""
        # ``np.log(np.sum(np.exp(a)))`` calculated in a numerically
stable way
        return self.logsumexp(self._estimate_log_prob(X) +
log(self.weights), axis=1).sum()
```

Question 4.4

Plot the value of MDL given in (35) as a function of the total number of EM iterations.

$$MDL(K,\theta) = -\sum_{n=0}^{N-1} \log\left(\sum_{i=0}^{K-1} p_{y_n|x_n}(y_n \mid \theta_k)\pi_k\right) + \frac{1}{2}L\log(NM)$$

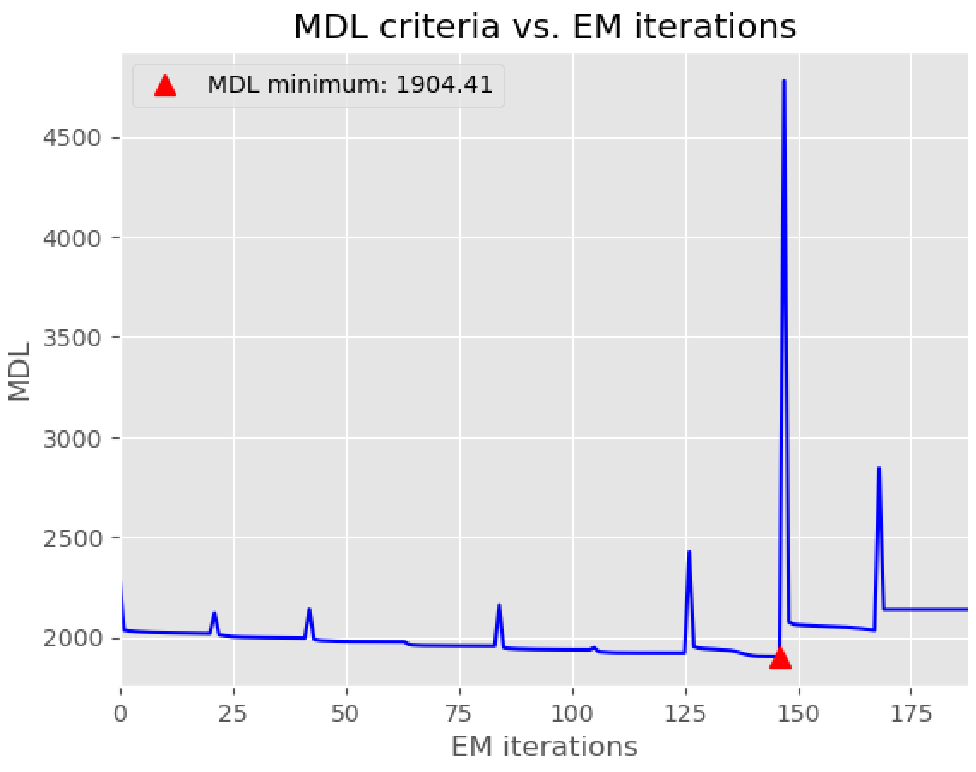
For each EM iteration, you should plot the value of the MDL criteria after the iteration is complete. In addition, if the EM iteration is followed by a cluster merging, then you should also plot the value of the MDL criteria after the clusters are merged, and the parameters are updated according to (36), (37), and (38).

$$\begin{aligned}
\pi_{(l,m)} &= \hat{\pi}_l + \hat{\pi}_m \\
\mu_{(l,m)} &= \frac{\hat{\pi}_l\hat{\mu}_l + \hat{\pi}_m\hat{\mu}_m}{\hat{\pi}_l + \hat{\pi}_m} \\
R_{(l,m)} &= \frac{\hat{\pi}_l\left(\hat{R}_l + \left(\hat{\mu}_l - \mu_{(l,m)}\right)\left(\hat{\mu}_l - \mu_{(l,m)}\right)^\top\right) + \hat{\pi}_m\left(\hat{R}_m + \left(\hat{\mu}_m - \mu_{(l,m)}\right)\left(\hat{\mu}_m - \mu_{(l,m)}\right)^\top\right)}{\hat{\pi}_l + \hat{\pi}_m}
\end{aligned}$$

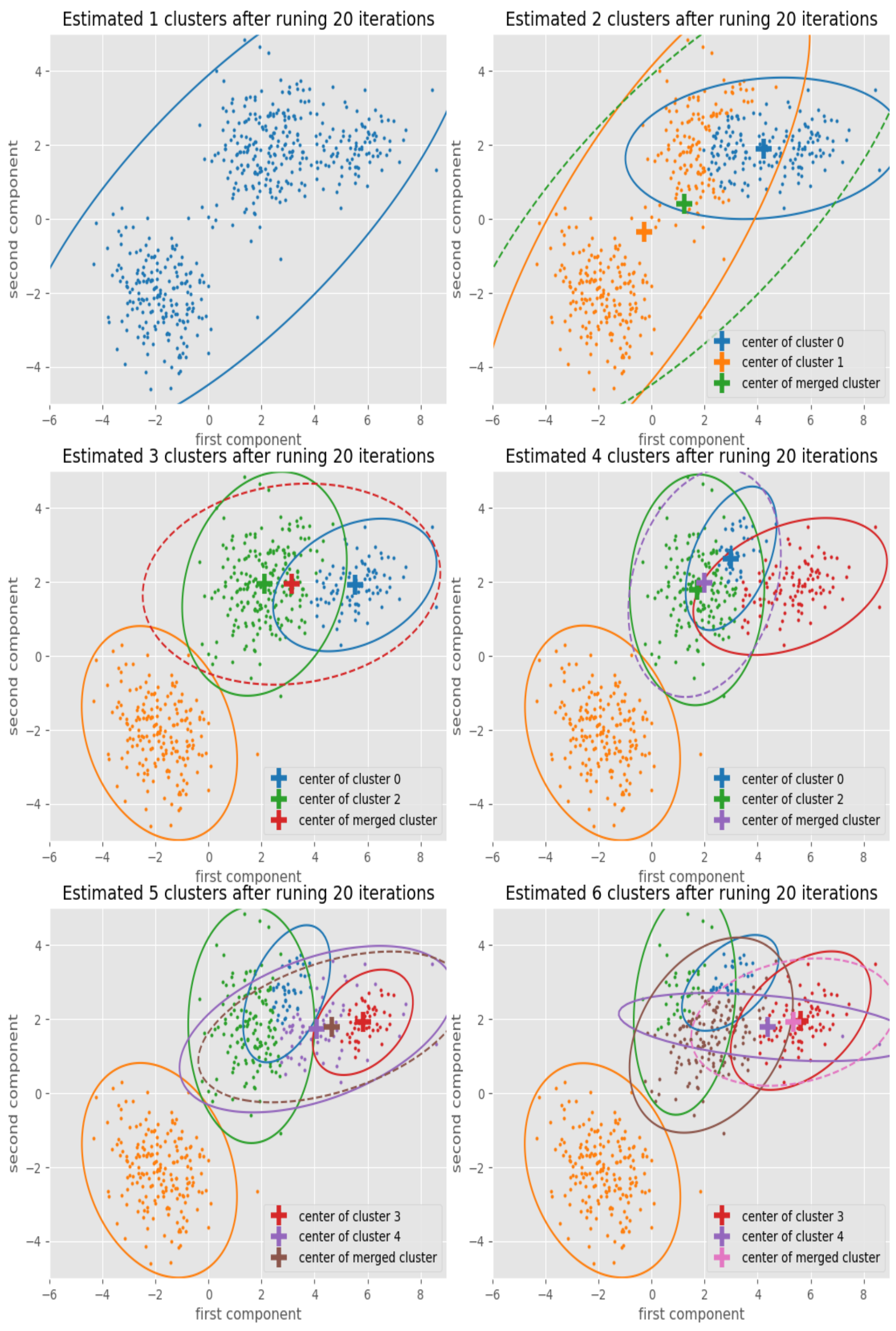
Noticed that for cluster merging iterations, there will be two ordinate points plotted. Mark the locations on the plot that correspond to the merging of clusters

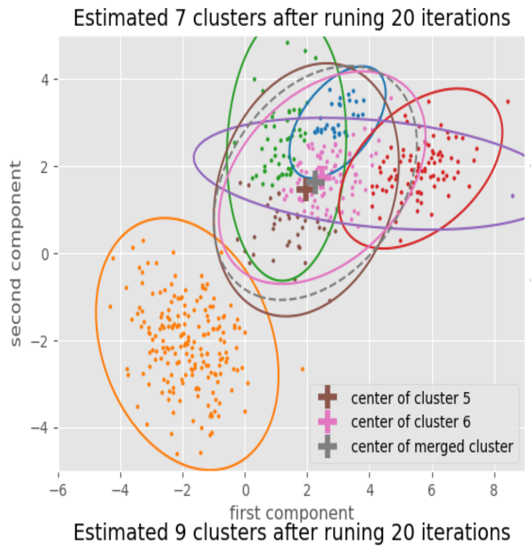
solution

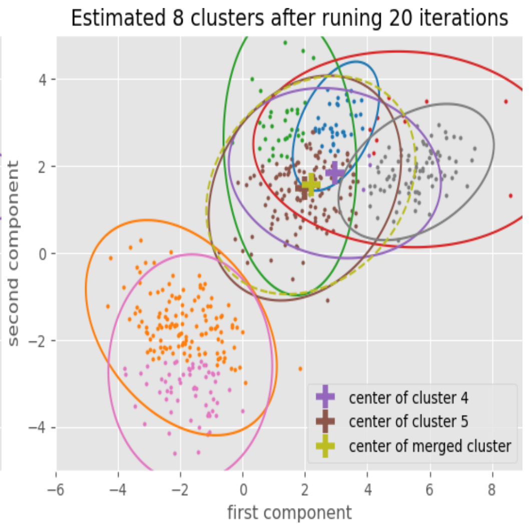
We plot the value of MDL given in (35) as a function of the total number of EM iterations, and also plot the value of the MDL criteria after the clusters are merged by running the python script **soln_4_2.py**

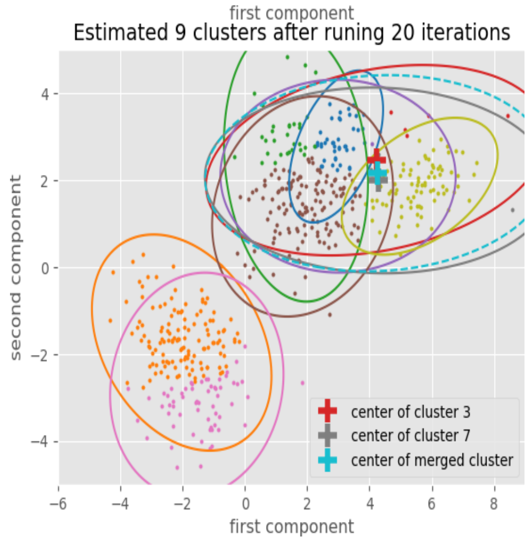


For cluster merging iterations, there are two ordinate points plotted. We mark the locations on the plot that correspond to the merging of clusters




Estimated 7 clusters after runing 20 iterations
second component
first component
center of cluster 5
center of cluster 6
center of merged cluster


Estimated 8 clusters after runing 20 iterations
second component
first component
center of cluster 4
center of cluster 5
center of merged cluster

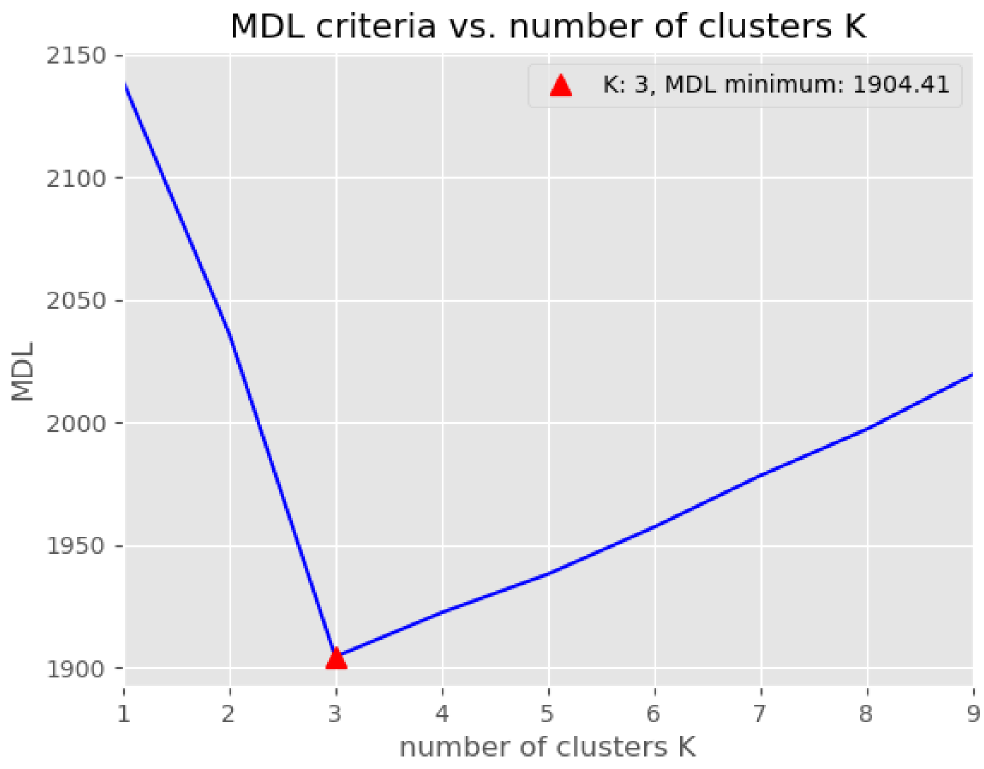

Estimated 9 clusters after runing 20 iterations
second component
first component
center of cluster 3
center of cluster 7
center of merged cluster

Question 4.5

Plot the MDL value versus K. For each value of K ranging from K = 1 to 9, plot the minimum value of the MDL obtained for that number of clusters. Label the value of K corresponding to the minimum observed value of MDL

solution

We plot the minimum value of the MDL obtained for that number of clusters, and label the value of K corresponding to the minimum observed value of MDL with python script **soln_4_2.py**



We obtain the **minimum** observed value of MDL **1904.41** when the number of clusters $K = 3$

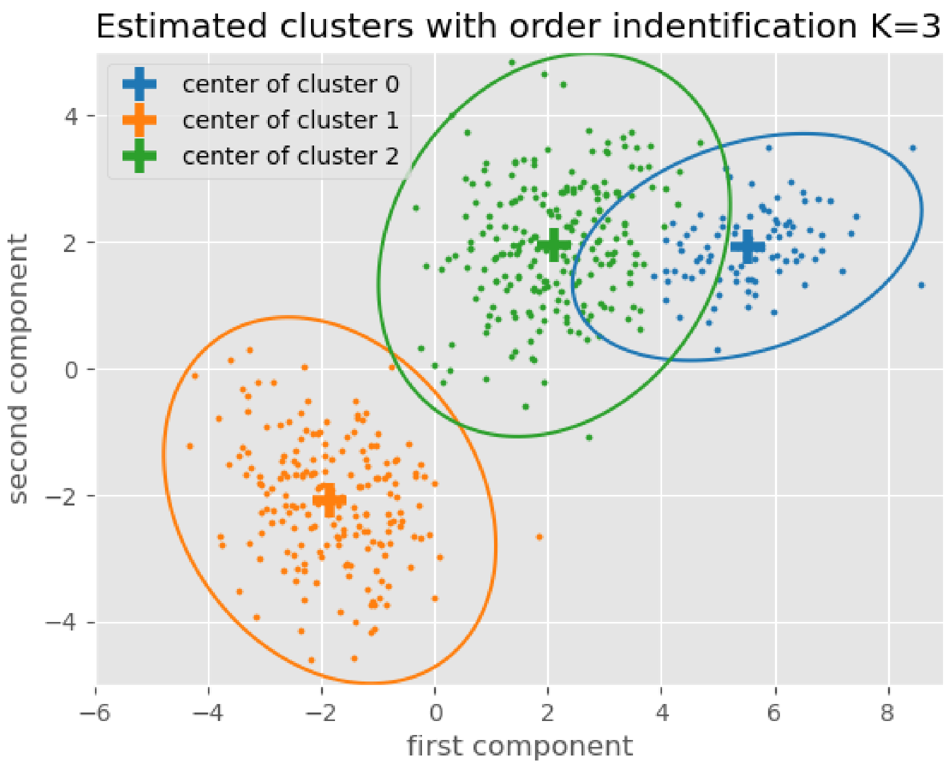
Question 4.6

Does the estimated value of $\hat{K}$ correspond to the true value of $K = 3$?

solution

Yes, the estimated value of $\hat{K}$ is exactly the true value $K = 3$

The final predicted results with the **minimal MDL** value are shown below


Estimated clusters with order indentification K=3
center of cluster 0
center of cluster 1
center of cluster 2
second component
first component