CS 182: Introduction to Machine Learning, Fall 2022
Homework 4

(Due on Wednesday, Dec. 7 at 11:59pm (CST))

Notice:

- Please submit your assignments via Gradescope. The entry code is G2V63D.
- Please make sure you select your answer to the corresponding question when submitting your assignments.
- Each person has a total of five days to be late without penalty for all the assignments. Each late delivery less than one day will be counted as one day.

1. [20 points] *[Clustering and Mixture Models]*
   (a) Describe the k-means clustering algorithm step-by-step; [10 points]
   (b) Given a set of 5 samples

$$X = \begin{bmatrix} 0 & 0 & 1 & 5 & 5 \\ 2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

   try the k-means clustering algorithm to cluster the samples into 2 clusters. (You should write a detailed derivation) [10 points]

Solution:

(a) Given a training set $\{x_i\}_{i=1}^N$, find k clusters
   (1) randomly choose k points as the initial cluster centroids
   (2) for all data points, find the closest centroid, get k clusters
   (3) compute the means of k clusters as the new centroids
   (4) if converged then output, otherwise return (2)

(b) Denote the five data points as $a = (0, 2)$, $b = (0, 0)$, $c = (1, 0)$, $d = (5, 0)$ and $e = (5, 2)$. Suppose we choose a and b as initial points.
   (1) at the first iteration, we choose cluster centers as a and b. After computing the distance between cluster centers and each data point, we obtain the clustering result

$$\mathcal{C}_1^{(1)} = \{a, e\}$$
$$\mathcal{C}_2^{(1)} = \{b, c, d\}$$

   and new cluster centers $\mathcal{G}_1^{(1)} = (2.5, 2)$ and $\mathcal{G}_2^{(1)} = (2, 0)$.
   (2) at the first iteration, after computing the distance between cluster centers and each data point, we obtain the clustering result

$$\mathcal{C}_1^{(2)} = \{a, e\}$$
$$\mathcal{C}_2^{(2)} = \{b, c, d\}$$

   The algorithm converges.
   – hence, we obtain two clusters

$$\mathcal{C}_1 = \{a, e\}$$
$$\mathcal{C}_2 = \{b, c, d\}$$

Choosing different initial centers may result in different clustering results. For example, if we choose a and e as initial points, we will obtain two clusters

$$\mathcal{C}_1 = \{a, b, c\}$$
$$\mathcal{C}_2 = \{d, e\}$$

2. [20 points] *[Clustering and Mixture Models]* A Gaussian mixture model (GMM) is a model of the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

with model parameters π_k, $\boldsymbol{\mu}_k$, and $\boldsymbol{\Sigma}_k$, where $\pi_k \geq 0$ and $\sum_k \pi_k = 1$.

(a) Discuss the advantages of the GMM and why it can be used for clustering; [5 points]

(b) Given a training set $\{\mathbf{x}_i \in \mathbb{R}^D\}_{i=1}^N$, try the expectation-maximization (EM) algorithm to estimate the parameters of the GMM. [15 points]

Solution:

(a) – The multivariate Gaussian distribution model can only approximate Gaussian distributed data, while the multivariate Gaussian mixture model has the ability to approximate many naturally occurring real-world data thanks to the law of large numbers.

– The clustering ability is natural obtained for mixture model as it classified the data into K components, i.e., clusters.

(b) Let $\boldsymbol{\Theta} = \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K$. The log likelihood function is

$$\ell(\boldsymbol{\Theta} \mid \mathbf{x}) = \sum_{i=1}^N \log p(\mathbf{x}_i \mid \boldsymbol{\Theta}) = \sum_{i=1}^N \log \left\{ \sum_{k=1}^K \pi_k \cdot p(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

(1) define latent variable $\gamma_{ik} \in \{0,1\}$ as the responsibility of cluster k for data point $\mathbf{x}_i$, and complete data log likelihood is

$$\begin{aligned}
\ell(\boldsymbol{\Theta} \mid \mathbf{x}, \gamma) &= \log \prod_{i=1}^N p(\mathbf{x}_i, \gamma_i \mid \boldsymbol{\Theta}) \\
&= \log \prod_{i=1}^N \left\{ \prod_{k=1}^K [\pi_k \cdot p(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)]^{\gamma_{ik}} \right\} \\
&= \sum_{k=1}^K \left\{ \log \pi_k \cdot \sum_{i=1}^N \gamma_{ik} + \sum_{i=1}^N \gamma_{ik} \log p(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}
\end{aligned}$$

(2) E-step: compute the posterior probability ($\boldsymbol{\Theta}$ is known, infer γ)

$$\begin{aligned}
\mathbb{E}[\gamma_{ik} \mid \mathbf{x}_i, \boldsymbol{\Theta}_k] &= p(\gamma_{ik} = 1 \mid \mathbf{x}_i, \boldsymbol{\Theta}_k) \\
&= \frac{p(\gamma_{ik} = 1 \mid \boldsymbol{\Theta}_k) p(\mathbf{x}_i \mid \gamma_{ik} = 1, \boldsymbol{\Theta}_k)}{\sum_{k=1}^K p(\gamma_{ik} = 1 \mid \boldsymbol{\Theta}_k) p(\mathbf{x}_i \mid \gamma_{ik} = 1, \boldsymbol{\Theta}_k)} \\
&= \frac{\pi_k \cdot p(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k=1}^K \pi_k \cdot p(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}
\end{aligned}$$

expected log likelihood of complete data

$$Q(\boldsymbol{\Theta} \mid \boldsymbol{\Theta}^t) = \mathbb{E}_{\gamma \mid \mathbf{x}, \boldsymbol{\Theta}^t} [\ell(\boldsymbol{\Theta} \mid \mathbf{x}, \gamma)] = \sum_{k=1}^K \left\{ \log \pi_k \cdot \sum_{i=1}^N \mathbb{E}[\gamma_{ik}] + \sum_{i=1}^N \mathbb{E}[\gamma_{ik}] \log p(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

(3) M-step:maximizing the expected log likelihood (γ is known, infer $\boldsymbol{\Theta}$)

$$\boldsymbol{\Theta}^{t+1} = \arg\max Q(\boldsymbol{\Theta} \mid \boldsymbol{\Theta}^t)$$

yields

$$\begin{aligned}
\boldsymbol{\mu}_k &= \frac{\sum_{i=1}^N \gamma_{ik} \mathbf{x}_i}{\sum_{i=1}^N \gamma_{ik}} \\
\boldsymbol{\Sigma}_k &= \frac{\sum_{i=1}^N \gamma_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_i)(\mathbf{x}_i - \boldsymbol{\mu}_i)^T}{\sum_{i=1}^N \gamma_{ik}} \\
\pi_k &= \frac{\sum_{i=1}^N \gamma_{ik}}{N}
\end{aligned}$$

3. [20 points] *[Nonparametric Density Estimation]* Let the i.i.d. sample $\mathcal{X} = \{x_i\}_{i=1}^n$ be drawn from some unknown probability density $p(x)$ with $x \in [0,1]$. We can obtain the histogram estimator $\hat{p}(x)$ for $p(x)$ with bin width specified as h. We define the loss of estimation error in the L^2 space:

$$L(h) = \int_0^1 \left((\hat{p}(x) - p(x)\right)^2 \mathsf{d}x = \int_0^1 \hat{p}^2(x)\mathsf{d}x - 2\int_0^1 \hat{p}(x)p(x)\mathsf{d}x + \int_0^1 p^2(x)\mathsf{d}x.$$

Considering the last term $\int p^2(x)\mathsf{d}x$ is uncorrelated with $\hat{p}(x)$ and replacing the integral with the average, we get

$$L'(h) = \int_0^1 \hat{p}^2(x)\mathsf{d}x - \frac{2}{n}\sum_{i=1}^n \hat{p}(x_i).$$

Please derive

(a) the expression of $\hat{p}(x)$; [6 points]

(b) the expression of $L'(h)$ based on the histogram estimator $\hat{p}(x)$; [6 points]

(c) the h that minimizes $L'(h)$. [8 points]

Solution:

(a) The expression of $\hat{p}(x)$ is

$$\hat{p}(x) = \frac{\#\left\{x^{(\ell)} \text{ in the same bin as } x\right\}}{nh}.$$

(b) Denote $m = \frac{1}{h}$ and Z_j as the number of x_i in the j bin. Since

$$\frac{1}{n}\sum_{i=1}^n \hat{p}(x_i) = \frac{1}{n}\sum_{j=1}^m \sum_{i\in B_j} \hat{p}(x_i) = \frac{1}{n}\sum_{j=1}^m Z_j\hat{p}(x_i) = \frac{1}{n}\sum_{j=1}^m \frac{Z_j^2}{nh},$$

we have

$$\begin{aligned} L'(h) &= \int_0^1 \hat{p}^2(x)\mathsf{d}x - \frac{2}{n}\sum_{i=1}^n \hat{p}(x_i) \\ &= \sum_{j=1}^m \int_{(j-1)h}^{jh} \hat{p}^2(x)\mathsf{d}x - \frac{2}{n}\sum_{j=1}^m \frac{Z_j^2}{nh} \\ &= \sum_{j=1}^m \int_{(j-1)h}^{jh} (\frac{Z_j}{nh})^2\mathsf{d}x - \frac{2}{n}\sum_{j=1}^m \frac{Z_j^2}{nh} \\ &= -\frac{1}{n^2h}\sum_{j=1}^m Z_j^2. \end{aligned}$$

(c) Because $\sum_{j=1}^m Z_j^2 \geq \sum_{j=1}^m Z_j = n$ and $\sum_{j=1}^m Z_j^2 \leq \sum_{j=1}^m Z_j n = n^2$, we have

$$-\frac{1}{h} \leq L'(h) \leq -\frac{1}{nh}.$$

Therefore, when $h \to 0$, $L'(h) \to -\infty$.

4. [20 points] *[Nonparametric Regression]* Given a set of n examples, $(\mathbf{x}_i, y_i), i = 1, \ldots, n$, a linear smoother is defined as follows. For any $\mathbf{x}$, there exists a vector $\ell(\mathbf{x}) = (\ell_1(\mathbf{x}), \ldots, \ell_n(\mathbf{x}))^\top$ such that the estimated output $\hat{y}$ of $\mathbf{x}$ is $\hat{y} = \sum_{i=1}^n \ell_i(\mathbf{x}) y_i = \ell(\mathbf{x})^\top Y$ where Y is a $n \times 1$ vector, $Y_i = y_i$.

   (a) In linear regression with basis functions h, the data is assumed to be generated by $y_i = \sum_{j=1}^m w_j h_j(\mathbf{x}_i) + \epsilon_i$. The least squares estimate for the coefficient vector $\mathbf{w}$ is given by $\mathbf{w}^* = \left(H^\top H\right)^{-1} H^\top Y$, where H is a $n \times m$ matrix, $H_{ij} = h_j(\mathbf{x}_i)$. Given an input $\mathbf{x}$, please derive the estimated output $\hat{y}$ (The solution should be in matrix form, and you may use the vector $h(\mathbf{x}) = [h_1(\mathbf{x}), \ldots, h_m(\mathbf{x})]^\top$). Furthermore, is linear regression a linear smoother? [10 points]

   (b) In kernel regression, if we use the kernel $K(\mathbf{x}_i, \mathbf{x}) = \exp\left\{\frac{-\|\mathbf{x}_i - \mathbf{x}\|^2}{2\sigma^2}\right\}$, given an input $\mathbf{x}$, please derive the estimated output $\hat{y}$. Furthermore, is this kernel regression a linear smoother? [10 points]

   Solution:

   (a) $\hat{y} = \mathbf{w}^{*T} h(\mathbf{x}) = h^T(\mathbf{x}) \left(H^T H\right)^{-1} H^T Y = \left(H \left(H^T H\right)^{-1} h(\mathbf{x})\right)^T Y$. Let $l(\mathbf{x}) = H\left(H^T H\right)^{-1} h(\mathbf{x})$, then $\hat{y} = l(\mathbf{x})^\top y$. Therefore, linear regression is a linear smoother.

   (b) $\hat{y} = \frac{\sum_{i=1}^n w_i y_i}{\sum_{i=1}^n w_i}$, where $w_i = \exp\left(-\frac{D(\mathbf{x}_i, \mathbf{x})^2}{K_w^2}\right)$. Let $l_i(\mathbf{x}) = \frac{w_i}{\sum_{i=1}^n w_i}$. Then $\hat{y} = l(\mathbf{x})^\top y$. Therefore, kernel regression is a linear smoother.

5. [20 points] [*Coding: EM*] Complete "HW4-Coding.ipynb". After completion, you should convert your notebook to PDF, and concatenate the writing part and the coding part into one PDF which is the file to submit.

Sol4-Coding

November 25, 2022

1 Fit GMM by EM algorithm

Please convert your coding notebook from .ipynb into .pdf and concatenate it with your writing part.

You need to implement the EM algorithm, using closed-forms derived in './HW4-Writing.pdf/2(b)'.

[1]:
```python
# Do NOT change this cell.
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```python
# E-step: compute posterior probabilities h_{li}
def probabilities(data, weights, means, covariances):
    '''
    :param data: size: (N, D)
    :param weights: corresponds to \pi, size: (K,)
    :param means: corresponds to \mu, size: (K, D)
    :param covariances: corresponds to \sigma, size: (K, D, D)
    :return prob_matrix: a matrix filled by posterior probability h_{ik}, size␣
 ↪(N, K)
    '''
    num_data = len(data)
    num_clusters = len(means)
    prob_matrix = np.zeros((num_data, num_clusters))
    ########## START YOUR CODE HERE ##########
    for k in range(num_clusters):
        for i in range(num_data):
            unbiased = data - means[k]
            power = -0.5 * np.dot(unbiased[i].T, np.dot(np.linalg.
 ↪inv(covariances[k]), unbiased[i]))
            prob_matrix[i, k] = 1 / np.sqrt(np.linalg.det(covariances[k])) * np.
 ↪exp(power) * weights[k]
    row_sums = np.sum(prob_matrix, axis=1)[:, np.newaxis]
    prob_matrix = prob_matrix / row_sums
    ########## END YOUR CODE HERE ##########
    return prob_matrix
```

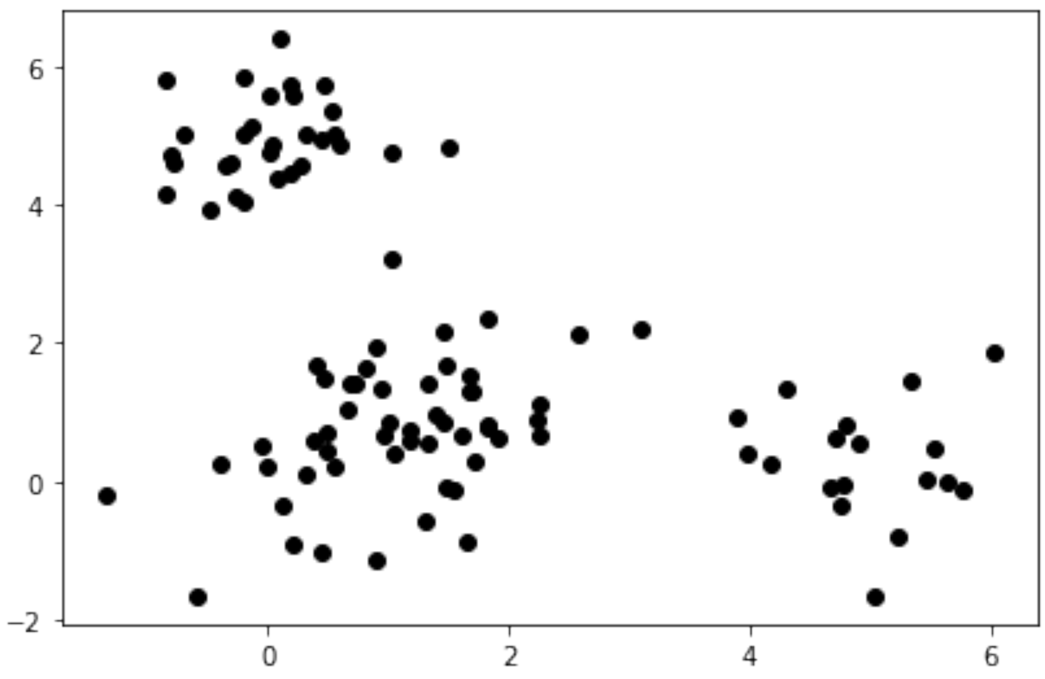
[3]:
```python
# M-step: update weights \pi, means \mu, and covariances \sigma
def updates(data, prob_matrix, weights, means, covariances):
    num_clusters = len(means)
    num_data = len(data)
    dim = data.shape[1]
    ########## START YOUR CODE HERE ##########
    col_sums = np.sum(prob_matrix, axis=0)
    for k in range(num_clusters):
        weights[k] = col_sums[k] / num_data
        means[k] = np.sum(np.tile(prob_matrix[:, k], (dim, 1)).T * data,␣
 ↪axis=0) / col_sums[k]
        up = np.zeros(covariances[0].shape)
        for i in range(num_data):
            up += prob_matrix[i][k] * np.outer(data[i] - means[k], data[i] -␣
 ↪means[k])
        covariances[k] = up / col_sums[k]
    ########## END YOUR CODE HERE ##########
    return weights, means, covariances
```

To help us develop and test our implementation, we will generate some observations from a mixture of Gaussians and then run our EM algorithm to discover the mixture components. We'll begin with a function to generate the data, and a quick plot to visualize its output for a 2-dimensional mixture of three Gaussians.

[4]:
```
# Generate dataset

def generate_MoG_data(num_data, means, covariances, weights):
    """ Creates a list of data points """
    num_clusters = len(weights)
    data = []
    for i in range(num_data):
        #  Use np.random.choice and weights to pick a cluster id greater than
        #  or equal to 0 and less than num_clusters.
        k = np.random.choice(len(weights), 1, p=weights)[0]
        # Use np.random.multivariate_normal to create data from this cluster
        x = np.random.multivariate_normal(means[k], covariances[k])
        data.append(x)
    return data

# Model parameters
data_means = np.array([
    [5, 0], # mean of cluster 1
    [1, 1], # mean of cluster 2
    [0, 5]  # mean of cluster 3
])
data_covariances = np.array([
    [[.5, 0.], [0, .5]], # covariance of cluster 1
    [[.92, .38], [.38, .91]], # covariance of cluster 2
    [[.5, 0.], [0, .5]]  # covariance of cluster 3
])
data_weights = np.array([1/4., 1/2., 1/4.])  # weights of each cluster

np.random.seed(4)
data = np.array(generate_MoG_data(100, data_means, data_covariances,␣
 ↪data_weights))
```

Now plot the data you created above. The plot should be a scatterplot with 100 points that appear to roughly fall into three clusters.

[5]:
```
plt.figure()
d = np.vstack(data)
plt.plot(d[:,0], d[:,1],'ko')
plt.rcParams.update({'font.size':16})
plt.tight_layout()
plt.show()
```



Now we'll fit a mixture of Gaussians to this data using our implementation of the EM algorithm. As with k-means, it is important to ask how we obtain an initial configuration of mixing weights and component parameters. In this simple case, we'll take three random points to be the initial cluster means, use the empirical covariance of the data to be the initial covariance in each cluster (a clear overestimate), and set the initial mixing weights to be uniform across clusters.

[6]:
```
# Initialization of parameters
np.random.seed(4)
```

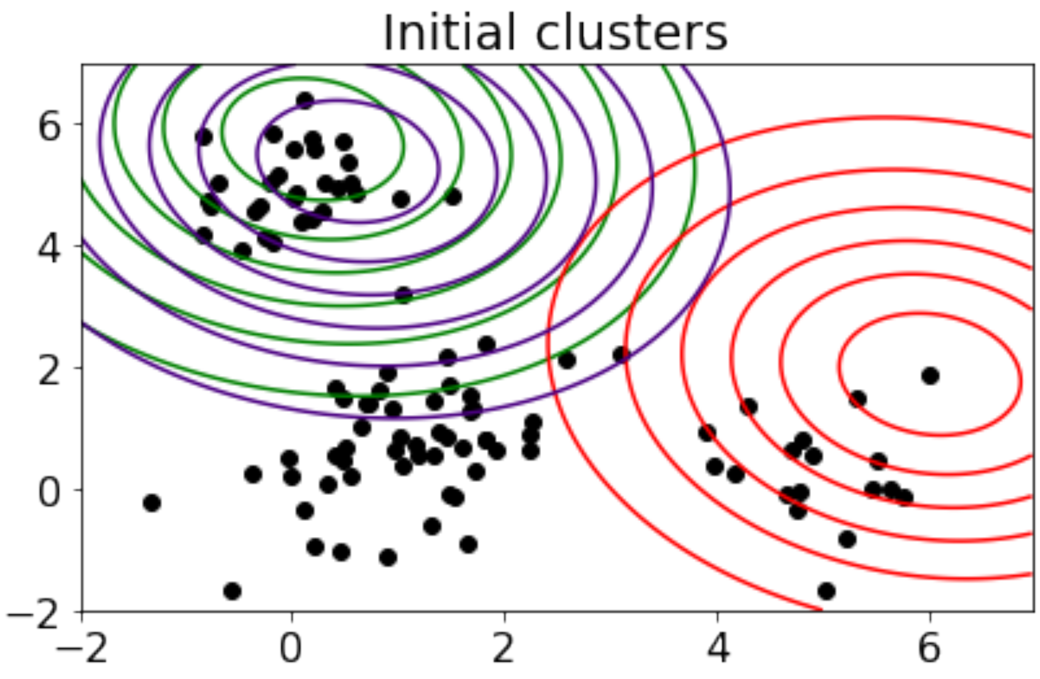
```python
chosen = np.random.choice(len(data), 3, replace=False)
initial_means = np.array([data[x] for x in chosen])
initial_covariances = np.array([np.cov(data, rowvar=0)] * 3)
initial_weights = np.array([1/3., 1/3., 1/3.])
```

We will use the following plot_contours() function to visualize the Gaussian components over the data at three different points in the algorithm's execution:

1. At initialization (using initial_means, initial_covariances, and initial_weights)
2. After running the algorithm to completion
3. After 20 iterations

[7]:
```python
def bivariate_normal(X, Y, sigmax=1.0, sigmay=1.0,
                 mux=0.0, muy=0.0, sigmaxy=0.0):
    Xmu = X-mux
    Ymu = Y-muy
    rho = sigmaxy/(sigmax*sigmay)
    z = Xmu**2/sigmax**2 + Ymu**2/sigmay**2 - 2*rho*Xmu*Ymu/(sigmax*sigmay)
    denom = 2*np.pi*sigmax*sigmay*np.sqrt(1-rho**2)
    return np.exp(-z/(2*(1-rho**2))) / denom

def plot_contours(data, means, covs, title):
    plt.figure()
    plt.plot([x[0] for x in data], [y[1] for y in data],'ko') # data

    delta = 0.025
    k = len(means)
    x = np.arange(-2.0, 7.0, delta)
    y = np.arange(-2.0, 7.0, delta)
    X, Y = np.meshgrid(x, y)
    col = ['green', 'red', 'indigo']
    for i in range(k):
        mean = means[i]
        cov = covs[i]
        sigmax = np.sqrt(cov[0][0])
        sigmay = np.sqrt(cov[1][1])
        sigmaxy = cov[0][1]/(sigmax*sigmay)
        Z = bivariate_normal(X, Y, sigmax, sigmay, mean[0], mean[1], sigmaxy)
        plt.contour(X, Y, Z, colors = col[i])
        plt.title(title)
    plt.rcParams.update({'font.size':16})
    plt.tight_layout()
    plt.show()
```

[8]:
```python
# Parameters after initialization
plot_contours(data, initial_means, initial_covariances, 'Initial clusters')
```

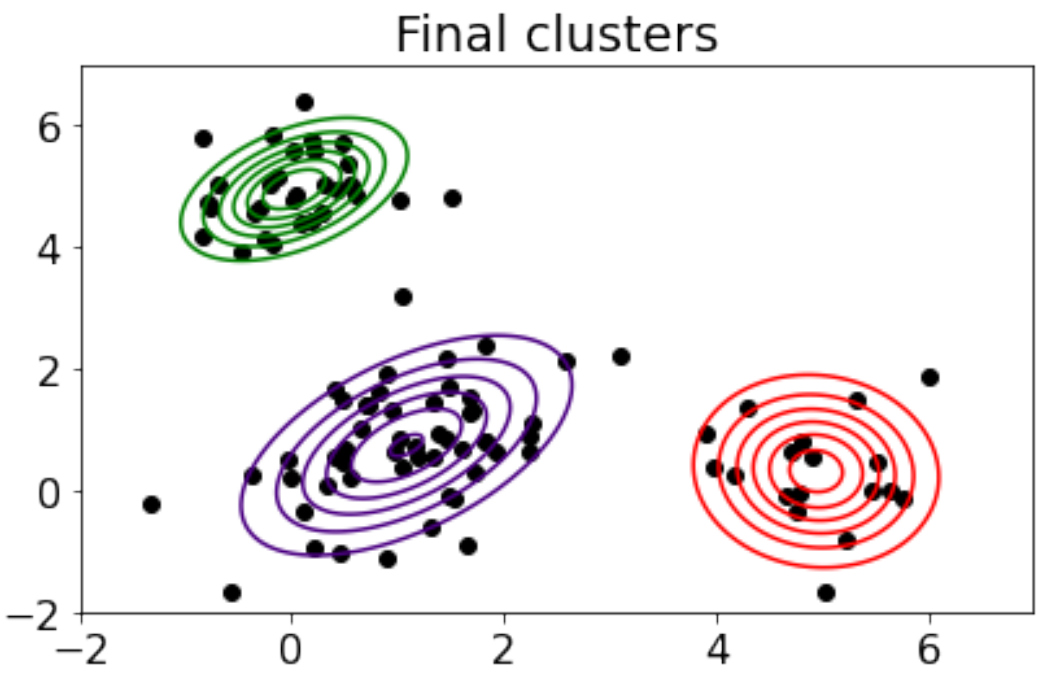


Now run the EM algorithm and plot contours afterwards.

[9]:
```
# Run EM
my_means = initial_means
my_weights = initial_weights
my_covariances = initial_covariances
iters = 20
for iter in range(iters):
    # Here we run 20 iterations. You can use a more strict convergence␣
 ↪criterion,
    # such as that the increment of the log-likelihood function is less than␣
 ↪1e-3
    prob_matrix = probabilities(data, my_weights, my_means, my_covariances)
    [my_weights, my_means, my_covariances] = updates(data, prob_matrix,␣
 ↪my_weights, my_means, my_covariances)
print("my_weights =\n{},\n\nmy_means =\n{},\n\nmy_covariances =\n{}".
 ↪format(my_weights, my_means, my_covariances))
```

```
my_weights =
[0.30100743 0.17993525 0.51905733],

my_means =
[[0.02238127 4.94603136]
 [4.94240545 0.31364134]
 [1.08184868 0.73761235]],
my_covariances =
[[[ 0.29398303  0.04872587]
  [ 0.04872587  0.35540776]]

 [[ 0.35562857 -0.01493194]
  [-0.01493194  0.66694074]]

 [[ 0.67155363  0.3308503 ]
  [ 0.3308503   0.90125037]]]
```

```
[10]: # Parameters after running EM to convergence
      plot_contours(data, my_means, my_covariances, 'Final clusters')
```



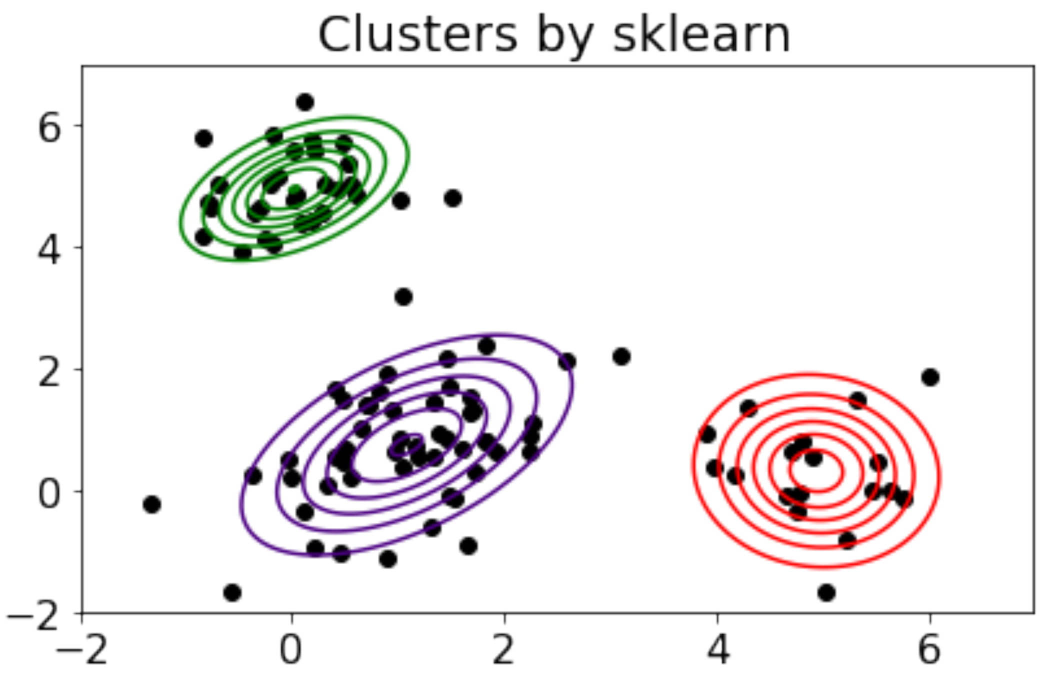
Now call the GaussianMixture() method in sklearn package, and see its performance.

```
[11]: from sklearn.mixture import GaussianMixture

      gmm_sklearn = GaussianMixture(n_components=3, covariance_type='full',
                                    max_iter=20, weights_init=initial_weights,
                                    means_init=initial_means).fit(data)
      print("weights_sklearn =\n{},\n\nmeans_sklearn =\n{},\n\ncovariance_sklearn␣
       ↪=\n{}".format(gmm_sklearn.weights_, gmm_sklearn.means_, gmm_sklearn.
       ↪covariances_))
```

```
weights_sklearn =
[0.30084512 0.17993617 0.51921871],

means_sklearn =
[[0.02183646 4.94695903]
 [4.94239893 0.31364722]
 [1.08182854 0.73838913]],

covariance_sklearn =
[[[ 0.29359094  0.04968644]
  [ 0.04968644  0.35399255]]

 [[ 0.35563708 -0.01494031]
  [-0.01494031  0.66694647]]

 [[ 0.67133505  0.33070759]
  [ 0.33070759  0.90291275]]]
```

outputs of sklearn can be different, but outputs of the hand-coded algorithm should be as same as that in this pdf.

[12]:
```
# Parameters after running EM to convergence
plot_contours(data, gmm_sklearn.means_, gmm_sklearn.covariances_, 'Clusters by␣
↪sklearn')
```



Now see the differences of parameters of your algorithm and that of sklearn. You should see the precision is at least 1e-3.

```
[13]: print("After 20 iterations, differences between our algorithm and sklearn:\n")
      print("delta_weights =\n{},\n\ndelta_means =\n{},\n\ndelta_covariances=\n{}".
        ↪format(my_weights-gmm_sklearn.weights_, my_means-gmm_sklearn.means_,␣
        ↪my_covariances-gmm_sklearn.covariances_))
```

```
After 20 iterations, differences between our algorithm and sklearn:

delta_weights =
[ 1.62311085e-04 -9.25287076e-07 -1.61385798e-04],

delta_means =
[[ 5.44806553e-04 -9.27674821e-04]
 [ 6.52261871e-06 -5.88055378e-06]
 [ 2.01444642e-05 -7.76783993e-04]],

delta_covariances=
[[[ 3.92087535e-04 -9.60570037e-04]
  [-9.60570037e-04  1.41521696e-03]]

 [[-8.51688022e-06  8.37099172e-06]
  [ 8.37099172e-06 -5.73057291e-06]]

 [[ 2.18584933e-04  1.42708918e-04]
  [ 1.42708918e-04 -1.66238509e-03]]]
```

```
[ ]:
```

```
[ ]:
```