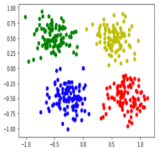
MIXTURE MODELS AND EM ALGORITHM

TOPIC 9

Name: Mirhady Dorodjatun

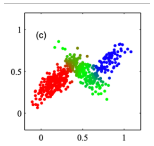
UID: u6474009

Contents



1. k-means algorithm

- Overview
- k-means optimization algorithm
- Why k-means always converge
- Choosing k and initialization method
- Python implementation



2. Gaussian Mixture

- Overview
- Latent variable modeling
- Maximum likelihood and responsibilities
- EM algorithm for Gaussian Mixtures
- Python implementation

3. Summary

Overview - k-means algorithm

- Separating dataset into k ”clusters” by minimizing cost function

$$\mathrm{J}(\boldsymbol{x}, \boldsymbol{\mu}) = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2$$

data point, centroid

- “hard” assignment to clusters (non-probabilistic)

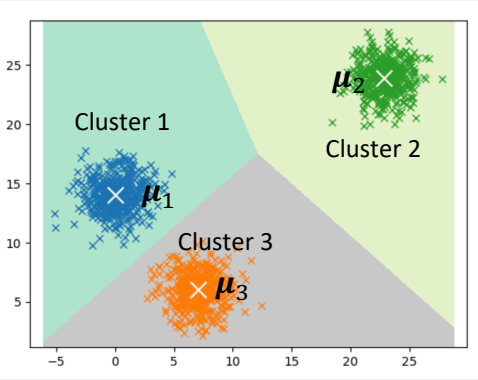


Figure 1

Goal: Minimize $\mathrm{L}(\boldsymbol{x}, \boldsymbol{\mu})$

Method: Optimization algorithm (coordinate descent/EM)

1. initialize value for k clusters (initialization step)
2. Repeat until convergence
   a) Update which cluster is assigned to each data point
   b) Update the mean of the cluster

- Remember Euclidean distance represents the similarity between data points

$$\mathrm{dist}(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_2 := \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Overview - k-means algorithm

- Importance of k-means:
  - Classify groups of similar points without need for labeling (unsupervised learning)
  - Guaranteed to converge (but may be converging to local minima) in finite steps
  - Not difficult to implement but useful: top 10 algorithm in data mining (Wu et al. 2008)
  - Can be used for image segmentation and compression

- Drawbacks:
  - Computationally expensive (Euclidean distance)
  - Time Complexity $O(n^2)$
  - Non-linearly separable dataset

k-means optimization

To optimize cost function

$$\mathrm{J}(\boldsymbol{x}, \boldsymbol{\mu}) = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2$$

Steps:

1. Initialization: randomly choose *k* data points from $\boldsymbol{x}$ as initial centroids $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$ (simplest one)
2. Iterate until convergence:
   a) For every $n \in \{1, \ldots, N\}$, **assign to the closest centroid**

$$r_{nk} = \begin{cases} 1, \text{if } k = \arg\min_j \|\boldsymbol{x}_n - \boldsymbol{\mu}_j\|^2 \\ 0, \text{ otherwise.} \end{cases}$$

Corresponds to **E-step** (expectation) in EM

From derivative of J to r_{nk} while $\boldsymbol{\mu}_k$ constant

r_{nk} determines whether x_n belong to cluster *k*, and follows "1-of-*k*" coding scheme

   b) For every $j \in \{1, \ldots, k\}$, **update the mean** (centroid) **of each cluster**.

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk} \boldsymbol{x}_n}{\sum_n r_{nk}}$$

Corresponds to **M-step** (maximization) in EM

From derivative of J to $\boldsymbol{\mu}_k$ while r_{nk} constant

Illustration of k-means algorithm

E-step:
assign all data points to the closest centroid

M-step:
update the mean (centroid) of each cluster

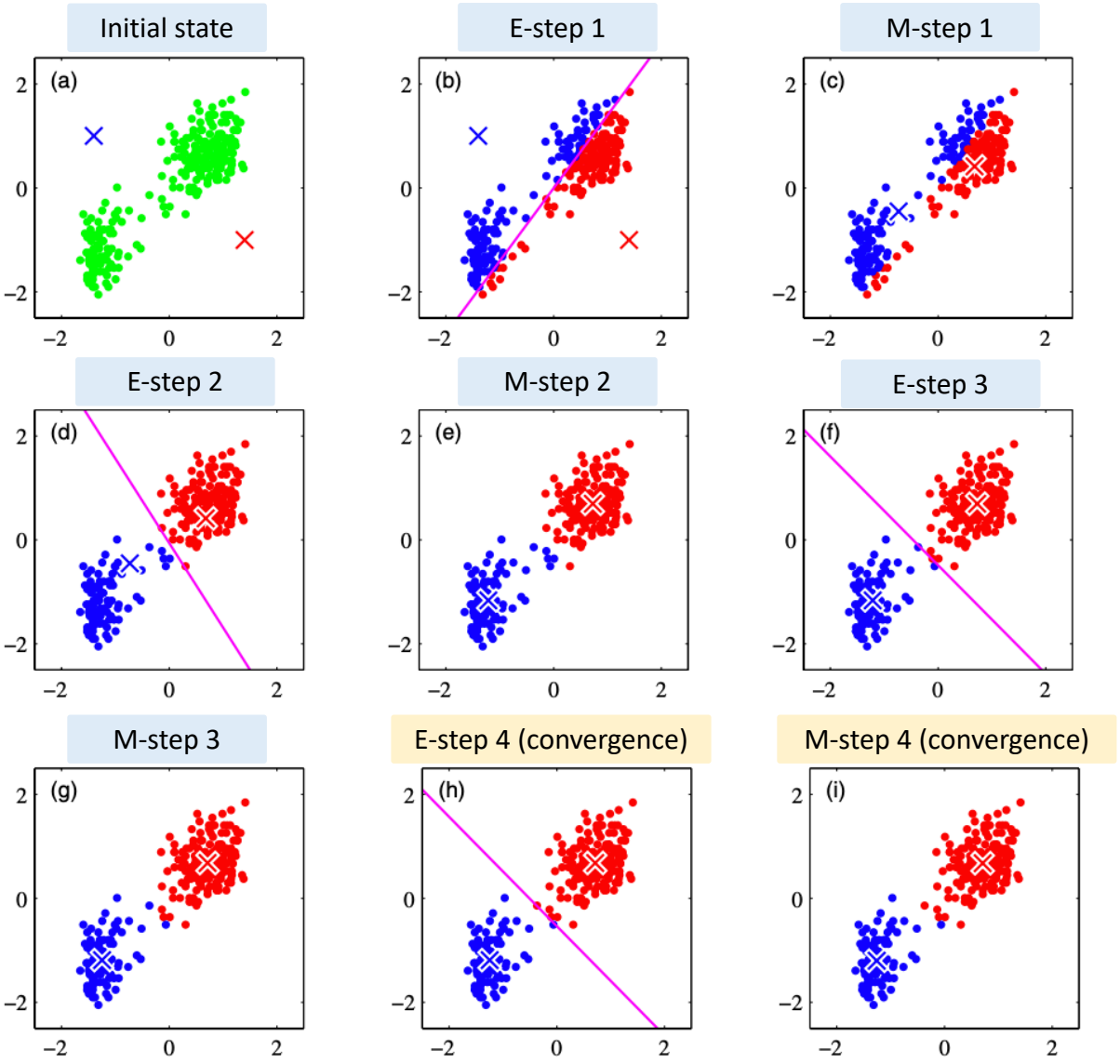


Figure 2

Cost on Convergence

- Cost (J) monotonically decrease in each EM step
- Always converge to the direction of the local minimum

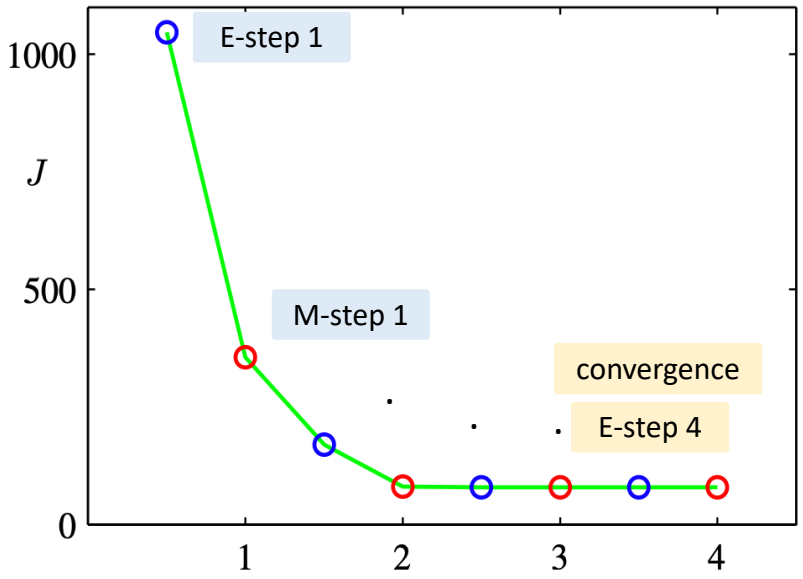


Figure 3

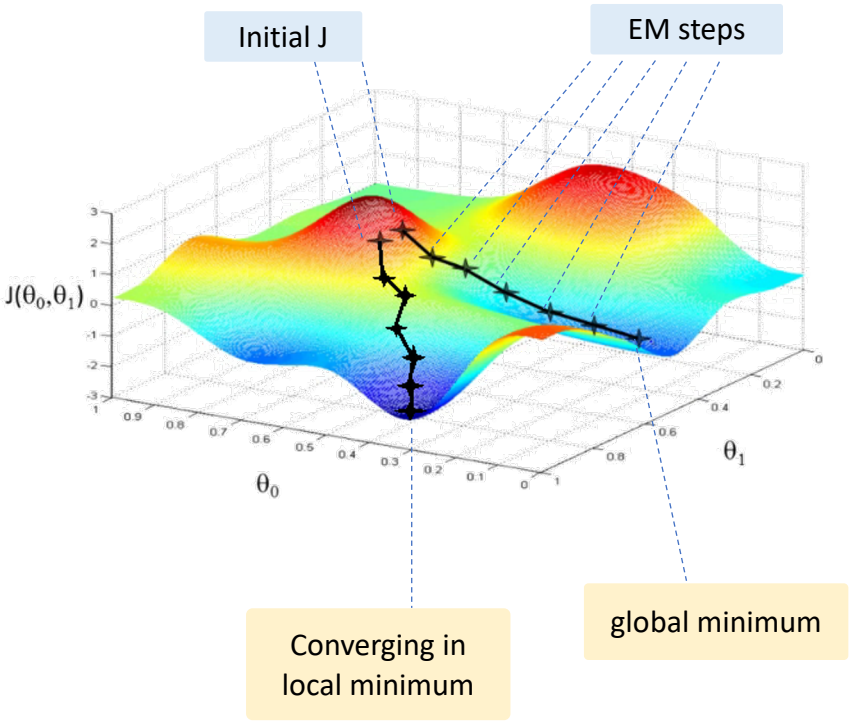


Figure 4

Why cost function monotonically decrease

Monotonically decreasing cost function will always lead to convergence.

Proof:

- E-step: assign all data points to the closest centroid

previous assignment: $C_1, C_2, \ldots, C_k$

new assignment: $C'_1, C'_2, \ldots, C'_k$

$$\begin{aligned} cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) &\geq \min_{C_1,\ldots,C_k} cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) \\ &= cost(C'_1, C'_2, \ldots, C'_k, z^{(1)}, \ldots, z^{(k)}) \end{aligned}$$

- M-step: update the mean (centroid) of each cluster

previous centroid: $z_1, z_2, \ldots, z_k$

new centroid: $z'_1, z'_2, \ldots, z'_k$

$$\begin{aligned} cost(C'_1, C'_2, \ldots, C'_k, z^{(1)}, \ldots, z^{(k)}) &\geq \min_{z^{(1)},\ldots,z^{(2)}} cost(C'_1, C'_2, \ldots, C'_k, z^{(1)}, \ldots, z^{(k)}) \\ &= cost(C'_1, C'_2, \ldots, C'_k, z'^{(1)}, \ldots, z'^{(k)}) \end{aligned}$$

- General idea:
  During E-step, each x_n is reassigned only when it is closer to another cluster. Therefore, E-step will either reduce or maintain the cost function (never increase it). The same holds for M-step.

Selecting value of *k*

- Value of k can determine result, but manually inputted

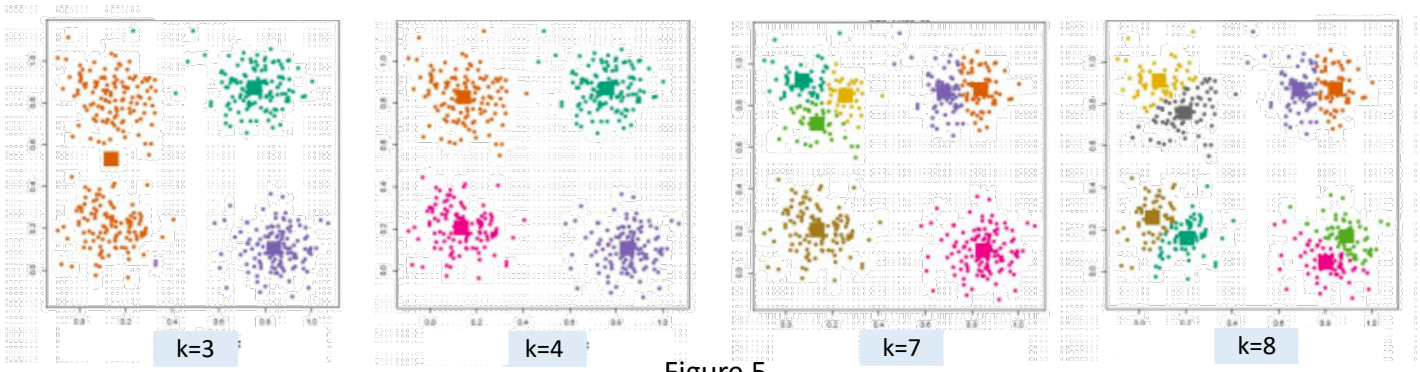


Figure 5

- Elbow method
  - Observing the “elbow” from cost function result for each number of clusters.

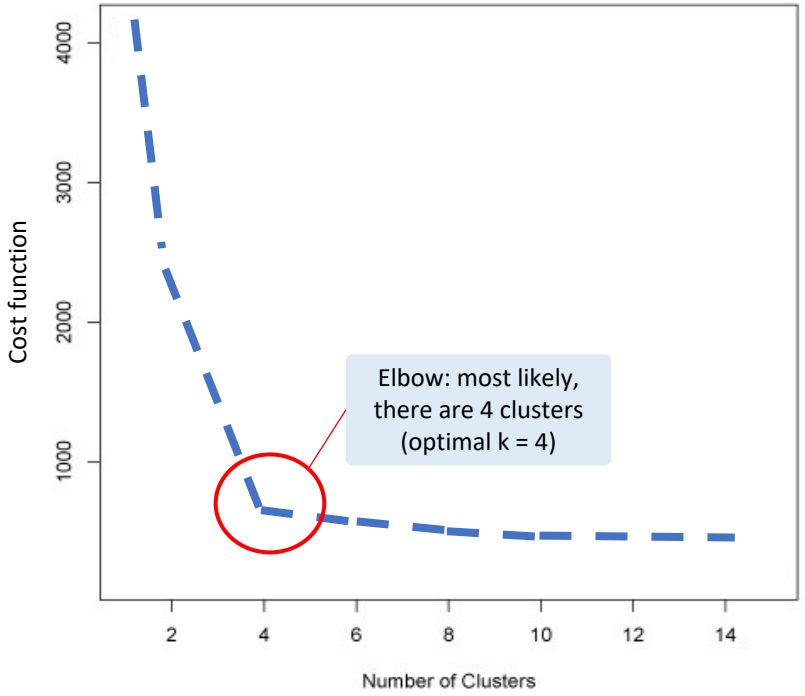


Figure 6

Importance of Initialization in k-means

- Different initialization can drastically alter the result
- Less optimal initialization method can result in empty clusters

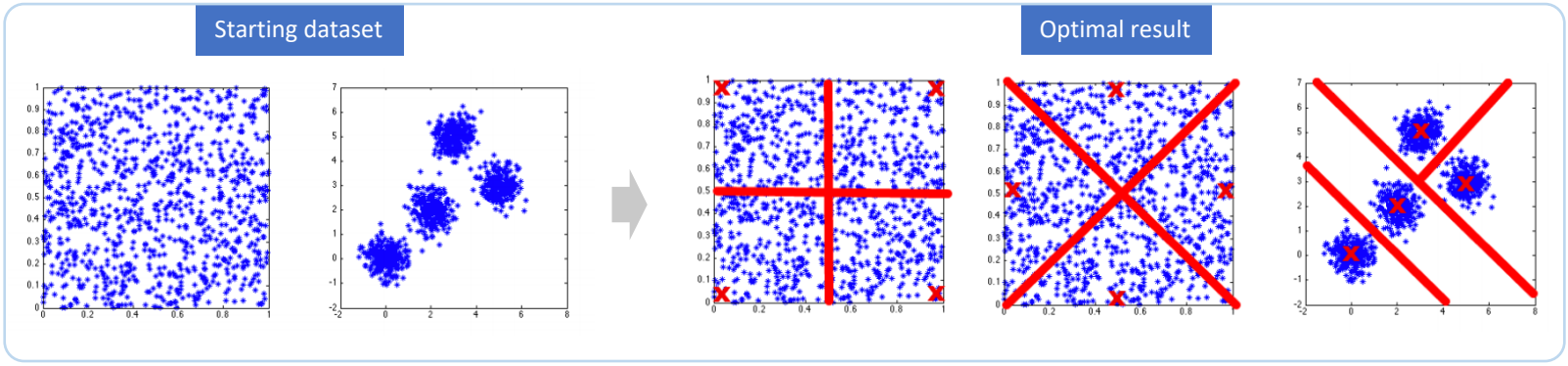


Figure 7.a

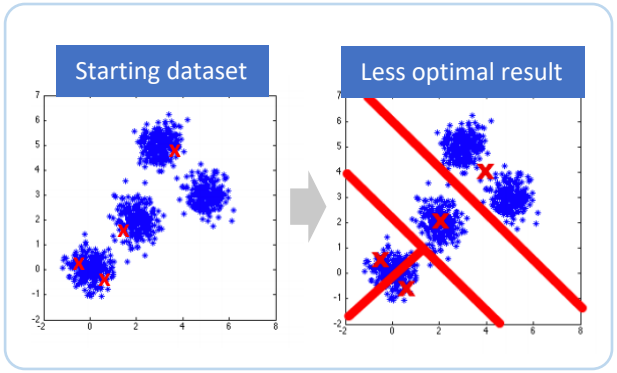


Figure 7.b

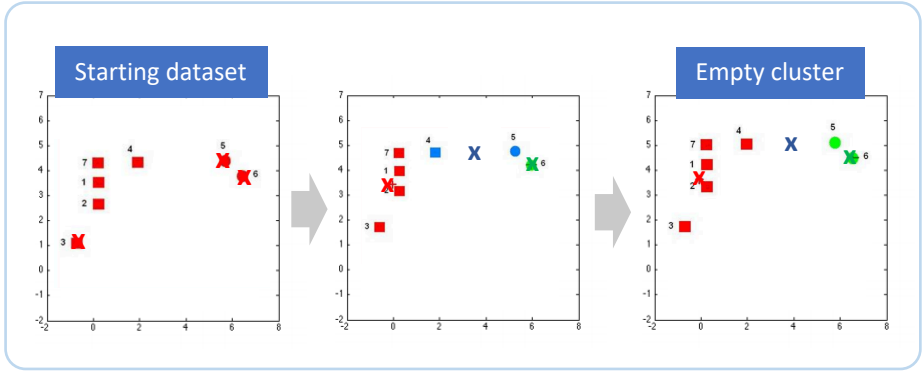


Figure 7.c

Importance of Initialization in k-means

- Example of initialization methods in k-means:
  1. Random selection (simplest)
  2. K-means++
     a) Select 1 data point randomly as the first centroid
     b) For the next centroids, choose the data point with the highest sum distance from all existing centroids
     c) Repeat step b) until the count of centroids reaches k

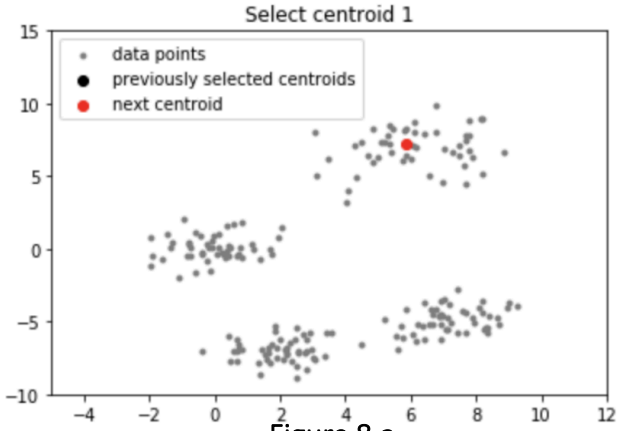


Figure 8.a

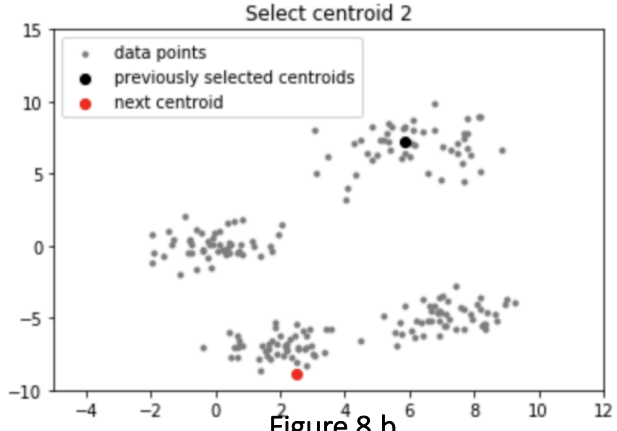


Figure 8.b

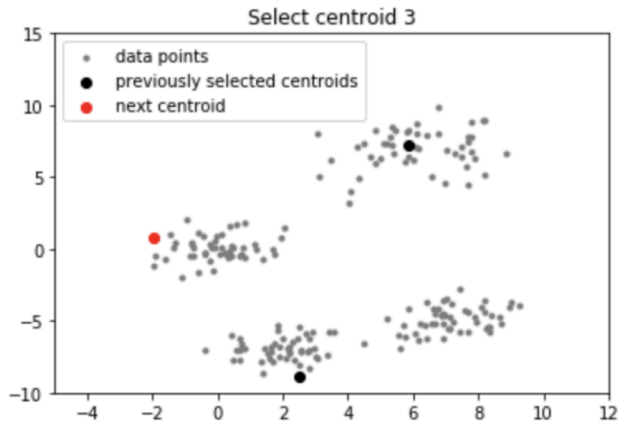


Figure 8.c

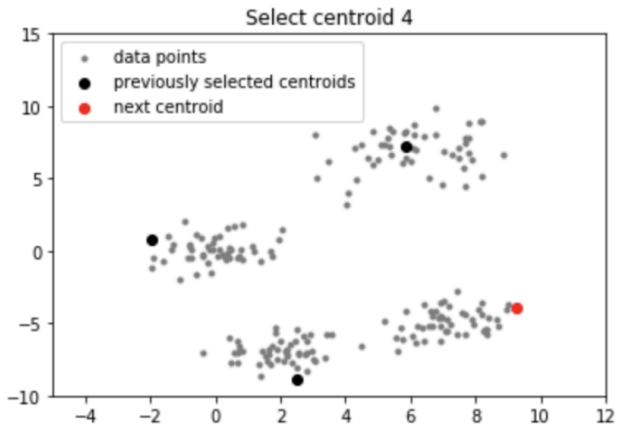


Figure 8.d

The general idea is to place the centroids furthest apart with each other

K-means implementation in Python

- Initialization: random selection

```
def initialise_parameters(m, n, X):
    C = np.zeros((m, n))

    for c_index in range(0, m):
        C[c_index] = X[np.random.randint(0,X[:,0].size)]

    return C
```

- Initialization: k-means++

```
import sys

def euc_distance(p1, p2):
    return np.sum((p1 - p2)**2)

def initialise_parameters(k, n, X):
    C = []

    # randomly select one data point as first centroid
    C.append(X[np.random.randint(0,X[:,0].size)])

    # then, for the remaining k-1 centroids:
    for c_index in range(1,k):

        distance = []
        for c_index in range(X.shape[0]):
            xn = X[c_index, :]
            d = sys.maxsize

            # compute distance of each xn from each existing centroids
            for j in range(len(C)):
                temp_d = euc_distance(xn, C[j])
                d = min(d, temp_d)
            distance.append(d)

        # select data point with maximum distance as other centroids
        distance = np.array(distance)
        next_C = X[np.argmax(distance), :]
        C.append(next_C)
        distance = []

    return C
```

K-means implementation in Python

- E-step

```
def E_step(C, X):
    L = np.zeros(X.shape)

    # assign all data points in X to the closest centroid
    for x_row in range(0, len(X[:,0])):
        for c_row in range(0, len(C[:,0])):
            if ( (np.linalg.norm(np.subtract(X[x_row], C[c_row]))
                  < np.linalg.norm(np.subtract(X[x_row], L[x_row])) )
                or np.array_equal(L[x_row], np.zeros(L[x_row].shape))):
                L[x_row] = C[c_row]

    return L

L = E_step(C, X)
```

- M-step

```
def M_step(C, X, L):
    new_C = np.zeros(C.shape)

    # updates the mean (centroid) of each cluster
    for c_row in range(0, len(C[:,0])):
        n_assigned_samples = 0
        for l_row in range(0, len(X[:,0])):
            if (np.array_equal(C[c_row], L[l_row])):
                n_assigned_samples += 1
                new_C[c_row] += X[l_row]

        if (n_assigned_samples != 0):
            new_C[c_row] /= n_assigned_samples

    return new_C
```

K-means implementation in Python

- Iterating E-M steps

```
def kmeans(X, m, i):
    # calls previously defined methods
    L = np.zeros(X.shape)
    C = initialise_parameters(m, len(X[0,:]), X)

    # iterates E-step and M-step for i times
    for i in range(0, i):
        L = E_step(C, X)
        C = M_step(C, X, L)

    return C, L

C_final, L_final = kmeans(X, 4, 10)
```

- Result

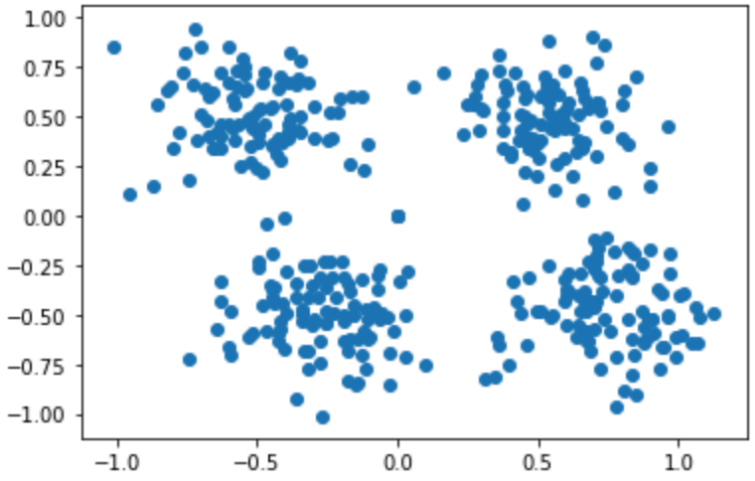


Figure 9

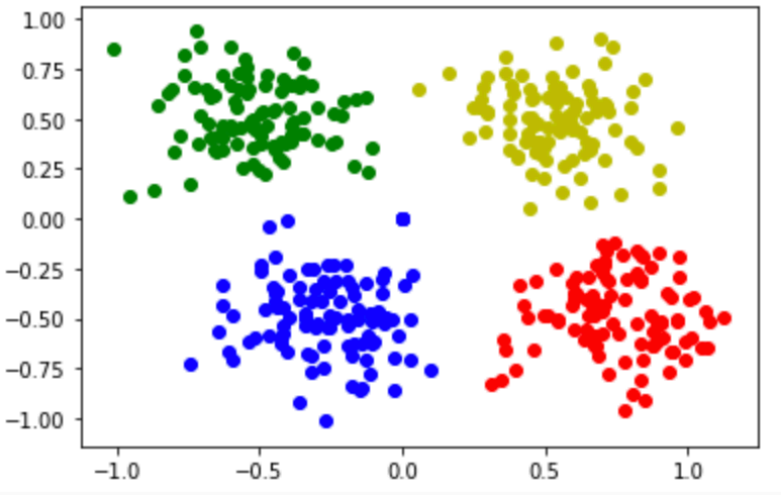


Figure 10

K-means implementation in Python

- What about data that are not linearly separable?

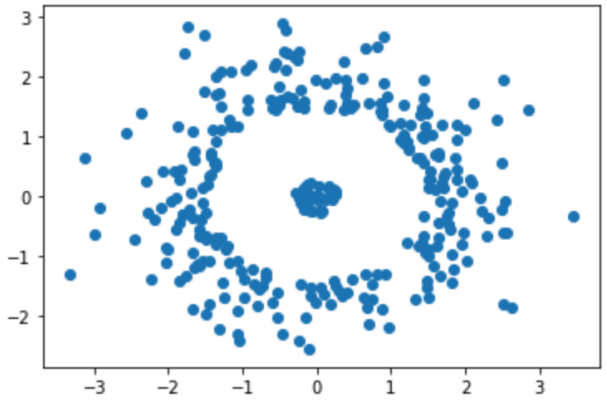
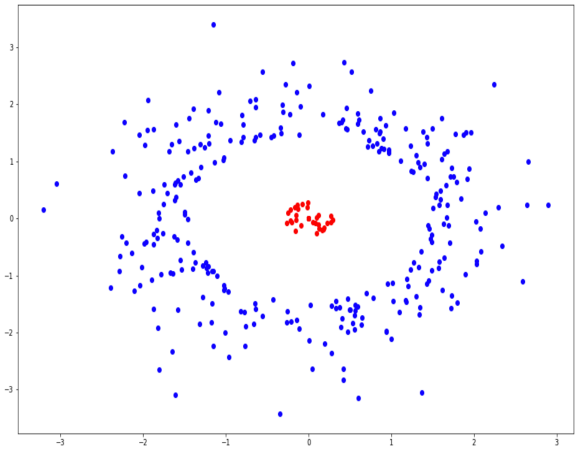


Figure 11



- Solution: add dimension to the data to make it linearly separable using k-means

Lossless reconstruction

```
def add_dimension(X):
    m = len(X[:,0])
    n = len(X[0,:])
    L = np.zeros(X.shape)
    X_new = np.zeros((m,n+1))

    #make X_new equals to X on every row-col of X
    for x_row in range(0, m):
        for x_col in range(0,n):
            X_new[x_row,x_col] = X[x_row,x_col]

    #make 3rd dimension equal to square of elements in X times a scalar
    for x_row in range(0, m):
        X_new[x_row, n] = (1* np.power(X_new[x_row, 0], 2))
        + (1* np.power(X_new[x_row, 1], 2))

    return X_new

Z = add_dimension(Z)
```

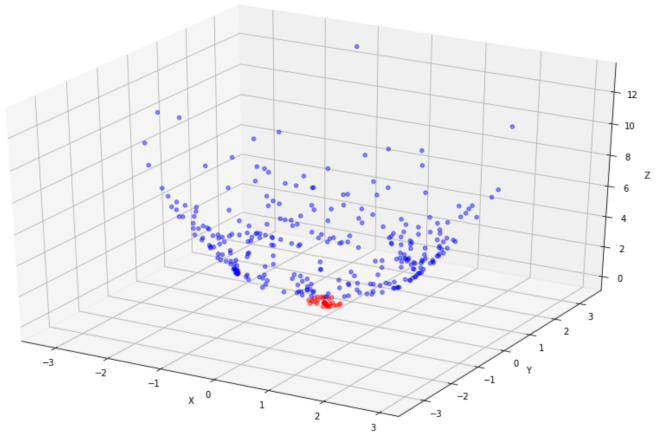


Figure 12

Image segmentation

- Segmenting image into *k* segments
- K-means is run on each pixels until convergence, then replace all pixel value with the centroid/mean of the cluster assigned to each pixel

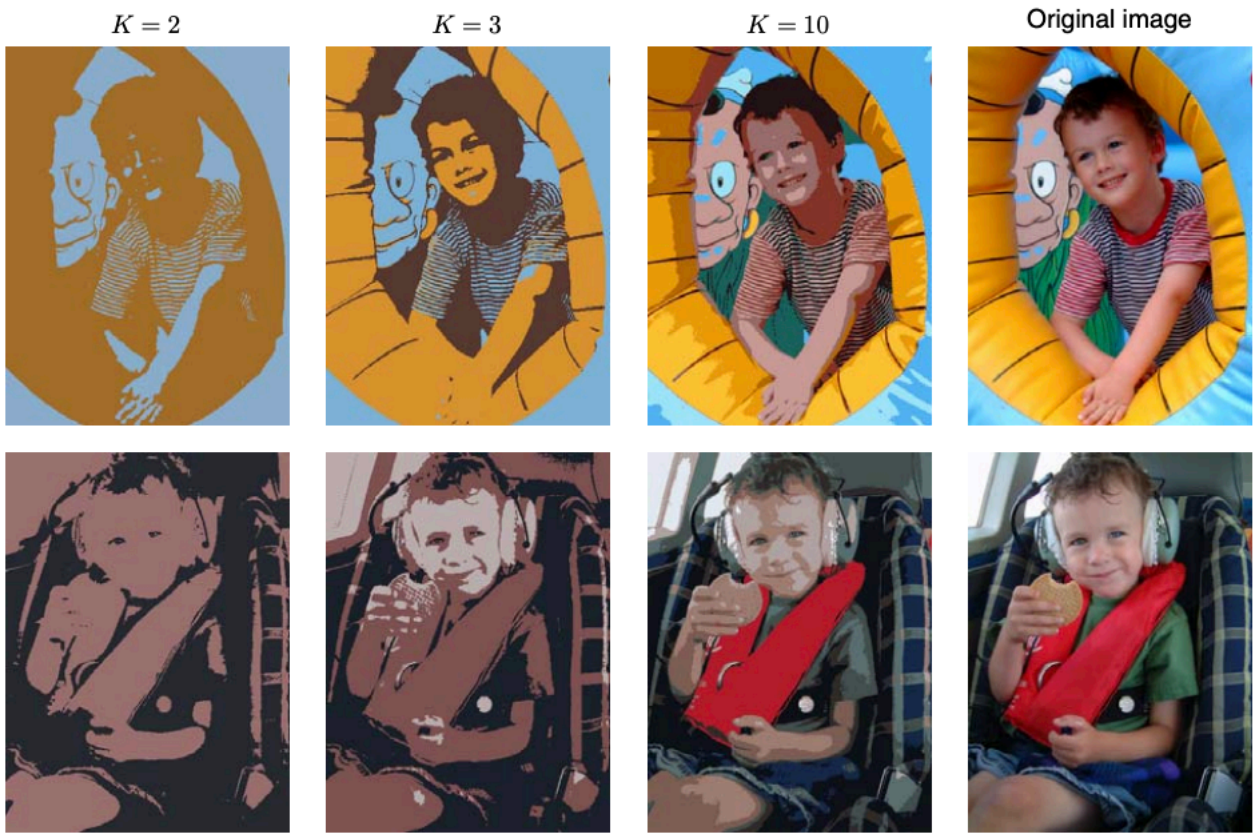


Figure 13

Image compression

- K-means as a compression algorithm (vector quantization)
- Lossy compression (for reconstruction)
- K-means is run on each pixels until convergence
- Store μ_k as code-book vectors, then store the labels (value of k assigned to each data point) to the code-book.

Figure 14.a

- Edge detection (using Sobel Filter):

```
im_face = cv2.imread('1.png', 0)

cv2_sobel_x = cv2.Sobel(im_face,cv2.CV_64F,1,0,ksize=3)
cv2_sobel_y = cv2.Sobel(im_face,cv2.CV_64F,0,1,ksize=3)

cv2_edge = cv2_sobel_x + cv2_sobel_y
plt.title("k=10 + sobel filter")
plt.imshow(cv2_edge,cmap='gray')
plt.show()
```

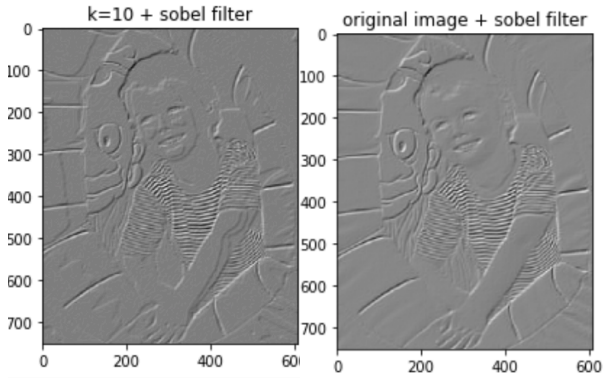


Figure 14.b

Overview – Gaussian Mixture

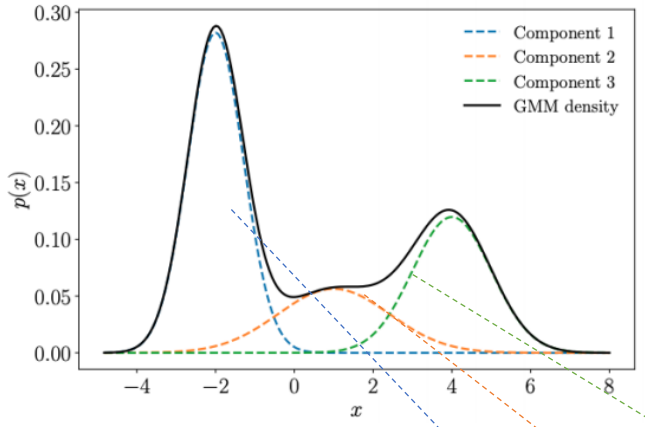


Figure 15

$$p(x) = \sum_{k=1}^{K} \pi_k p_k(x)$$ → complex to work with

π_k: mixture weight
(weight of the gaussian component)

introduce latent variable θ : $p(x, \theta) = p(x|\theta)p(\theta)$ → (product rule)

knowing $p(\theta) = \prod_{k=1}^{K} \pi_k{}^{\theta k}$, → (by 1-of-k)

we then derive: $p(x|\theta) = \prod_{k=1}^{K} N\,(x|\mu_k, \Sigma_k)^{\theta_k}$

Therefore, we get the formula of Gaussian Mixture Model (GMM):

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

in figure 15

$$p(x|\boldsymbol{\theta}) = 0.5\mathcal{N}\left(x\left|-2, \frac{1}{2}\right.\right) + 0.2\mathcal{N}(x|1, 2) + 0.3\mathcal{N}(x|4, 1)$$

Also, introducing the latent variable θ will allow us to **identify** (by probability) **which mixture component a data point comes from**

Overview – Gaussian Mixture

- We get the log-likelihood by: $\mathcal{L} = \log p(\boldsymbol{X}|\boldsymbol{\theta}) = \sum_{n=1}^{N} \log p(\boldsymbol{x}_n|\boldsymbol{\theta}) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$

- Why log-likelihood instead of likelihood?
  - Log is monotonically increasing so both works equivalently for maximization
  - Adding log will help in simplifying mathematical calculations
  - Likelihood can be a very small number, which can lead to floating point error (underflow)

- Importance of Gaussian Mixture
  - Modeling complex distribution with gaussian components
  - Classify groups of similar points without need for labeling (unsupervised learning)
  - Richer representation of r_{nk} than k-means

- Drawbacks:
  - Even more expensive than k-means
  - Time Complexity $O(NKD^3)$ for N data points, K gaussians and D dimensions

Responsibilities

- We can get responsibilities r_{nk} using Bayes Theorem:

$$r_{nk} = p(\theta = 1|x) = \frac{p(\theta = 1)p(x|\theta = 1)}{\sum_{j=1}^{K} p(\theta = 1)p(x|\theta = 1)}$$

$$r_{nk} := \frac{\pi_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}\left(\boldsymbol{x}_n \middle| \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j\right)}$$

- Responsibility represents soft assignment in GMM, which contrasts with k-means' hard assignment:

GMM

$$r_{nk} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 \\ 0.057 & 0.943 & 0.0 \\ 0.001 & 0.999 & 0.0 \\ 0.0 & 0.066 & 0.934 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \in \mathbb{R}^{N \times K}$$

k-means

$$r_{nk} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \in \mathbb{R}^{N \times K}$$

- Plotted:

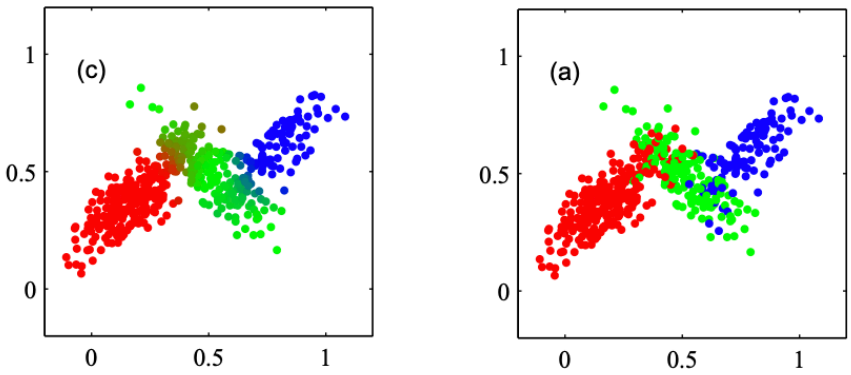


Figure 16

EM Algorithm in Gaussian Mixtures

To optimize log-likelihood function: $\mathcal{L} = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$

Steps

1. Initialization:
   - π_k = 1/k for all k
   - μ_k = centroids, same as k-means algorithm
   - Σ_k = sample variance for all k
2. Repeat until convergence:
   - E-step: For every $n \in \{1, \dots, N\}$, update r_{nk} using current π_k, μ_k and Σ_k

$$r_{nk} := \frac{\pi_k \mathcal{N}(\boldsymbol{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\boldsymbol{x}_n | \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

From Bayes Theorem In previous slide

   - M-step: For every $j \in \{1, \dots, k\}$, update parameters π_k, μ_k and Σ_k using the updated r_{nk}

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} \boldsymbol{x}_n$$

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} (\boldsymbol{x}_n - \boldsymbol{\mu}_k)(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}$$

$$\pi_k = \frac{N_k}{N}$$

From partial derivative of log-likelihood function to π_k, μ_k and Σ_k

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}_k} = \mathbf{0}^{\mathrm{T}} \Leftrightarrow \sum_{n=1}^{N} \frac{\partial \log p(\boldsymbol{x}_n | \boldsymbol{\theta})}{\partial \boldsymbol{\mu}_k} = \mathbf{0}^{\mathrm{T}}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\Sigma}_k} = 0 \Leftrightarrow \sum_{n=1}^{N} \frac{\partial \log p(\boldsymbol{x}_n | \boldsymbol{\theta})}{\partial \boldsymbol{\Sigma}_k} = \mathbf{0}$$

$$\frac{\partial \mathcal{L}}{\partial \pi_k} = 0 \Leftrightarrow \sum_{n=1}^{N} \frac{\partial \log p(\boldsymbol{x}_n | \boldsymbol{\theta})}{\partial \pi_k} = 0$$

GMM EM algorithm implementation in Python

- Initialization

```
def initialise_parameters(X, K):
    m = X.shape[1]
    N = X.shape[0]
    sigma = []
    mu = []
    pi = []

    # Calculate mean of data points X (the relative center) to use as the first centroid
    mean_X = X.shape[0]
    for n in range (N):
        mean_X += X[n]
    mean_X /= N

    # Calculate stdev_X to represent the standard deviation of the datapoints X
    stdev_X = 0
    for n in range (N):
        stdev_X += np.linalg.norm((X[n] - mean_X))
    stdev_X /= N

    # Now we initialize mu, first we use the mean of X as the first k
    mu.append(mean_X)

    # and then choose the next one(s) randomly, but retry if too close to mean_X
    # we choose (2 * stdev_X) from mean_X to determine the "forbidden" range
    for i in range(1,K):
        isFound = False
        while (isFound == False):
            mu_candidate = np.asarray(X[np.random.randint(0,N),:])
            if (np.linalg.norm((mu_candidate - mean_X)) > (2 * stdev_X)):
                mu.append(mu_candidate)
                isFound = True

    for i in range (K):
        # assign pi uniform for each k
        pi.append(1/K)

        # for simple positive semidefinite matrix sigma, use identity matrix
        sigma.append(np.asarray(np.eye((m))))

    return sigma, mu, pi
```

GMM EM algorithm implementation in Python

- E-step

```
def E_step(pi, mu, sigma, X):
    N = X.shape[0]
    m = X.shape[1]
    K = len(pi)
    responsibilities = np.zeros((N, K))
    pdf = np.zeros((N,K))

    # compute pdf of gaussian matrices
    for n in range (N):
        for k in range (K):
            x = X[n]
            pdf[n][k] = multivariate_normal.pdf(x, mean=np.asarray(mu[k]),
                                                cov=sigma[k], allow_singular=True)

    # compute rnk with respect to each gaussian matrix
    for n in range (N):
        rnk_denom = np.dot(np.asarray(pi).T, pdf[n])
        for k in range (K):
            responsibilities[n][k] = pi[k] * pdf[n][k] / rnk_denom

    return responsibilities
```

GMM EM algorithm implementation in Python

- M-step

```
def M_step(responsibilities, X):
    K = responsibilities.shape[1]
    N = X.shape[0]
    m = X.shape[1]
    mu = []
    sigma = []
    pi = []

    # count Nk for each gaussian distribution
    # starting with one to prevent problems with zero value
    # (then divided by N + 2**k later to average out the value)
    Nk = np.ones((K,))
    for k in range (K):
        for n in range (N):
            Nk[k] += responsibilities[n,k]
    print("Nk :",Nk)

    # updates mu
    b = np.ones((responsibilities[0][0] * X[0]).shape)
    for k in range (K):
        for n in range (N):
            a = np.ones((responsibilities[n][k] * X[n]).shape)
            a = np.dot(responsibilities[n][k], X[n])
            b += a
        mu.append(np.asarray(b / (Nk[k])))

    # updates sigma
    c=0
    for k in range (K):
        for n in range (N):
            c += responsibilities[n][k] * np.outer((X[n]) - (mu[k]), ((X[n]) - (mu[k])).T)
        sigma.append(c / (Nk[k]))

    # make all ij in sigma positive since sigma need to be positive semidefinite
    for k in range (K):
        for i in range (m):
            for j in range (m):
                if (sigma[k][i,j] < 0):
                    sigma[k][i,j] = sigma[k][i,j] * -1

    # updates pi, where we divide by N + 2**k to average out the 1 added in Nk
    for k in range (K):
        pi.append(Nk[k]/N + (2**k))

    return mu, sigma, pi
```

GMM EM algorithm implementation in Python

- Iterating E-M steps

```
def EM(X, K, iterations):
    N = X.shape[0]

    # initialization step
    sigma, mu, pi = initialise_parameters(X, K)

    # do i iterations of EM step, by calling the predefined E_step and M_step in each iteration
    for i in range (iterations):
        print("iteration ",i, end=', ')
        responsibilities = E_step(pi, mu, sigma, X)
        mu, sigma, pi = M_step(responsibilities, X)

    return mu, sigma, pi
```

EM implementation in Python

- Result:

r_{nk} after 1 iteration

```
[[1.00000000e+000 9.43602925e-139 9.43602925e-139]
 [1.00000000e+000 6.72742056e-159 6.72742056e-159]
 [1.00000000e+000 2.29466786e-140 2.29466786e-140]
 ...
 [1.00000000e+000 5.29990521e-033 5.29990521e-033]
 [1.00000000e+000 1.65488827e-031 1.65488827e-031]
 [1.29224280e-024 5.00000000e-001 5.00000000e-001]]
```

r_{nk} after 10 iterations

```
[[2.02899408e-29 1.02437345e-02 9.89756266e-01]
 [8.70385773e-53 9.40156692e-04 9.99059843e-01]
 [9.06979976e-34 6.46505220e-03 9.93534948e-01]
 ...
 [2.99280419e-08 1.83245471e-01 8.16754499e-01]
 [1.56620844e-05 2.31746288e-01 7.68238050e-01]
 [5.79161322e-04 4.08990529e-01 5.90430310e-01]]
```

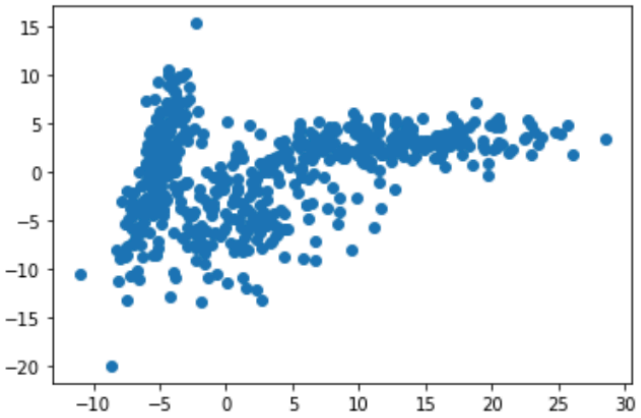


Figure 16

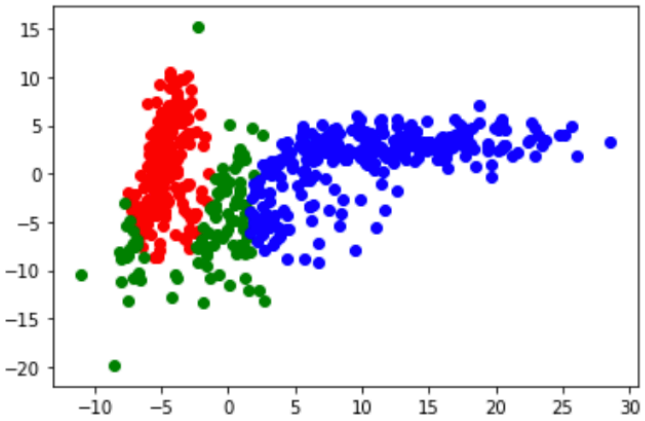


Figure 17

Limitation: Plotting the 'intermediate' colors

Summary

- General overview, Importance and drawback of each topic
- K-means:
  - Clustering: time complexity $O(n^2)$
  - K-means' EM algorithm and where each EM steps are derived from
  - responsibilities r_{nk} (hard assignment - nonprobabilistic)
  - Illustration of how k-means progresses with each EM steps
  - How cost function monotonically decrease but does not guarantee global minimum
  - Why the cost function monotonically decrease with each EM steps
  - Selecting value of k with elbow method
  - Importance of initialization
  - 2 initialization methods: random selection and k-means++
  - K-means for image segmentation and compression
  - Example of how the compressed image can be used, with edge detection using Sobel Filter
  - Python implementation of k-means: 2 examples, one of which with (initially) non-linearly separable dataset
- Gaussian Mixture Model:
  - Clustering: time complexity $O(NKD^3)$
  - How adding latent variable would be useful in modeling the data
  - EM algorithm and where each EM steps are derived from
  - responsibilities r_{nk} (soft assignment - probabilistic)
  - Why use log-likelihood instead of likelihood
  - Python implementation of clustering with GMM

References:

- Bishop, CM. 2006, “Pattern Recognition and Machine Learning”, New York :Springer
- Deisenroth, MP, Faisal, AA, Ong, CS. 2019, “Mathematics for Machine Learning”, Cambridge: Cambridge University Press
- COMP8600 lecture slides
- COMP6670 lecture slides

Some figures are downloaded from online sources:

- Figure 1: https://medium.com/@saahil1292/machine-learning-103-k-means-clustering-linearly-separable-maximum-likelyhood-principal-3b903bf3817c
- Figure 8: https://www.geeksforgeeks.org/ml-k-means-algorithm/
- Other figures are taken from PRML book or lecture slides, except the output of python implementation

Link to ppt slide, python code implementation and video backup:

https://www.github.com/rbdm/comp8600-video-material-topic-9

THANK YOU

u6474009