hw2-template

June 15, 2019

1 Homework 2: Clustering

1.0.1 Deadline: June 18, 11:59 pm

1.0.2 Dr. Mahdi Roozbahani

In this homework, you will have the chance to implement two algorithms for clustering, namely the KMeans and Gaussian Mixture Model (GMM). Specifically:

- Implement K-Means algorithm, as well as the helper function like fast pairwise distance.
- Implement GMM, as well as the numerical stable helper functions.

1.1 0. Setup

This notebook is tested under python 3.6.8, and the corresponding packages can be downloaded from miniconda. You may also want to get yourself familiar with several packages:

- jupyter notebook
- numpy
- matplotlib

1.2 Instructions about the assignment

- For coding questions: you are asked to fill in the blanks in the code cells.
- After you have completed your coding, please restart your kernel and run your notebook from start to finish before you submit (so we don't have to re-run it!)
- Your notebook should not take more than 10 minutes to run (ours runs in < 2 minutes)

You are only allowed to use the packages imported below. Other packages like scipy, sklearn are not allowed in this homework. The purpose is to let you write most part of the algorithms from scratch, so you can have a better understanding about them.

- For writing questions: you are asked to answer them in the markdown cells.

You can also type the Latex equations in the markdown cell. To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type

```
In [1]: ################################
        ### DO NOT CHANGE THIS CELL ###
        ################################

        from __future__ import absolute_import
        from __future__ import print_function
        from __future__ import division

        %matplotlib inline

        import sys
        import matplotlib
        import numpy as np
        import matplotlib.pyplot as plt
        import imageio
        from tqdm import tqdm_notebook as tqdm

        print('Version information')

        print('python: {}'.format(sys.version))
        print('matplotlib: {}'.format(matplotlib.__version__))
        print('numpy: {}'.format(np.__version__))

        # Set random seed so output is all same
        np.random.seed(1)
```

```
Version information
python: 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
matplotlib: 3.0.2
numpy: 1.15.4
```

2 1. Helper functions [20 pts]

In this section, you are asked to implement several mathematical functions that will be used later in your implementation for KMeans and GMM.

You are expected to receive the exact same output with your implementation.

2.1 pairwise distance [7 pts]

Given $X \in \mathbb{R}^{NxD}$ and $Y \in \mathbb{R}^{MxD}$, obtain the pairwise distance matrix $dist \in \mathbb{R}^{NxM}$, where $dist_{i,j} = ||X_i - Y_j||_2$.

DO NOT USE A FOR LOOP in your implementation -- they are slow and will make your code too slow to pass our grader. Use array broadcasting instead.

```
In [2]: def pairwise_dist(X, Y):
            """
            Args:
```

```
            X: N x D numpy array
            Y: M x D numpy array
        Return:
            dist: N x M array, where dist2[i, j] is the euclidean distance between
            X[i, :] and Y[j, :]
        """
        X1 = X[:, None]   # (N,D) --> (N,1,D)
        Y1 = Y[None, :]   # (M,D) --> (1,M,D)
        square_distance = np.sum((X1-Y1)**2, axis = -1) #(N,1,D) - (1,M,D) = (N,M,D) --> s
        return np.sqrt(square_distance)


# Check answer
np.random.seed(1)
x = np.random.randn(2, 2)
y = np.random.randn(3, 2)
print("*** Expected Answer ***")
print("""==x==
[[ 1.62434536 -0.61175641]
 [-0.52817175 -1.07296862]]
==y==
[[ 0.86540763 -2.3015387 ]
 [ 1.74481176 -0.7612069 ]
 [ 0.3190391  -0.24937038]]
==dist==
[[1.85239052 0.19195729 1.35467638]
 [1.85780729 2.29426447 1.18155842]]""")


print("\n*** My Answer ***")
print("==x==")
print(x)
print("==y==")
print(y)
print("==dist==")
print(pairwise_dist(x, y))
```

```
*** Expected Answer ***
==x==
[[ 1.62434536 -0.61175641]
 [-0.52817175 -1.07296862]]
==y==
[[ 0.86540763 -2.3015387 ]
 [ 1.74481176 -0.7612069 ]
 [ 0.3190391  -0.24937038]]
==dist==
[[1.85239052 0.19195729 1.35467638]
 [1.85780729 2.29426447 1.18155842]]
```

```
*** My Answer ***
==x==
[[ 1.62434536 -0.61175641]
 [-0.52817175 -1.07296862]]
==y==
[[ 0.86540763 -2.3015387 ]
 [ 1.74481176 -0.7612069 ]
 [ 0.3190391  -0.24937038]]
==dist==
[[1.85239052 0.19195729 1.35467638]
 [1.85780729 2.29426447 1.18155842]]
```

2.2 softmax [7 pts]

Given $logit \in \mathbb{R}^{NxD}$, calculate $prob \in \mathbb{R}^{NxD}$, where $prob_{i,j} = \frac{\exp(logit_{i,j})}{\sum_{d=1}^{D} exp(logit_{i,k})}$.

Note that it is possible that $logit_{i,j}$ is very large, making $\exp(\cdot)$ of it to explode. To make sure it is numerical stable, you may need to subtract the maximum for each row of *logits*. As in calculating pairwise distances, DO NOT USE A FOR LOOP.

```
In [3]: def softmax(logits):
            logits -= np.max(logits, axis = 1, keepdims=True) # (N,d) --sum, axis = 1-->(N,)--
            return np.exp(logits)/np.sum(np.exp(logits), axis = 1, keepdims=True)      #(N,d)

        logits = np.array([[1000, 1000],
                           [1, 2]], dtype=np.float32)
        print("""Correct answer:
        ===softmax===
        [[0.5        0.5       ]
         [0.26894143 0.7310586 ]]""")

        print("My answer:")
        print(softmax(logits))
```

```
Correct answer:
===softmax===
[[0.5        0.5       ]
 [0.26894143 0.7310586 ]]
My answer:
[[0.5        0.5       ]
 [0.26894143 0.7310586 ]]
```

2.3 logsumexp [6 pts]

Given $logit \in \mathbb{R}^{NxD}$, calculate $s \in \mathbb{R}^{N}$, where $s_i = \log\left(\sum_{j=1}^{D} \exp(logit_{i,j})\right)$. Again, pay attention to the numerical problem. You may want to use similar trick as in the softmax function. DO NOT USE A FOR LOOP.

```
In [4]: def logsumexp(logits):
            """
            Args:
                logits: N x D numpy array
            Return:
                s: N x 1 array where s[i,0] = logsumexp(logits[i,:])
            """
            logits_max = np.max(logits, axis =1)    # (N,D) --sum,axis=1--> (N,)
            logits_normalized = logits - logits_max[:,None]  # (N,) --[:,None]--> (N,1) //// (
            sum_exp_logits_normalized = np.log(np.sum(np.exp(logits_normalized), axis = 1)) #(
            return (sum_exp_logits_normalized + logits_max)[:,None]                  #(N,) -->

        logits = np.array([[1000, 1000],
                           [1, 2]], dtype=np.float32)
        print("""Correct Answer:
        ===logsumexp===
        [[1000.6932   ]
         [   2.3132617]]""")

        print("My answer: ")
        print(logsumexp(logits))
```

```
Correct Answer:
===logsumexp===
[[1000.6932   ]
 [   2.3132617]]
My answer:
[[1000.6932   ]
 [   2.3132617]]
```

```
In [5]: # below are some helper functions for plot.
        # you don't have to modify them.

        def plot_images(img_list, title_list, figsize=(11, 6)):
            assert len(img_list) == len(title_list)
            fig, axes = plt.subplots(1, len(title_list), figsize=figsize)
            for i, ax in enumerate(axes):
                ax.imshow(img_list[i] / 255.0)
                ax.set_title(title_list[i])
                ax.axis('off')

        def plot_scatter(samples, ids):
            colors = np.zeros((len(ids), 3))
            choices = [[0, 0, 1], [0, 1, 0], [1, 0, 0]]
            num_points = []
            for i in range(3):
                num_points.append(np.sum(ids == i))
```

```
maps = np.argsort(num_points)
for i in range(3):
    colors[np.where(ids == maps[i]), :] = choices[i]
plt.scatter(samples[:, 0], samples[:, 1], s=1, color=colors)
plt.axis('equal')
```

3 2. KMeans implementation [30 pts]

KMeans is trying to solve the following optimization problem:

$$\arg\min_{S} \sum_{i=1}^{K} \sum_{x_j \in S_i} ||x_j - \mu_i||^2 \tag{1}$$

where one needs to partition the N observations into K sets: $S = \{S_1, S_2, \ldots, S_K\}$ and each set has μ_i as its center.

In this section, you are asked to implement the kmeans algorithm. Below is a template which requires you to implement some member functions that haven't been implemented yet.

For the function signature, please see the corresponding doc strings.

In [6]:
```
class KMeans(object):

    def __init__(self): #No need to implement
        pass

    def _init_centers(self, points, K, **kwargs):
        """
        Args:
            points: NxD numpy array, where N is # points and D is the dimensionality
            K: number of clusters
            kwargs: any additional arguments you want
        Return:
            centers: K x D numpy array, the centers.
        """

        K_indexes = np.random.choice(len(points),K)
        return points[K_indexes]


    def _update_assignment(self, centers, points):
        """
        Args:
            centers: KxD numpy array, where K is the number of clusters, and D is the
            points: NxD numpy array, the observations
        Return:
            cluster_idx: numpy array of length N, the cluster assignment for each poin

        Hint: You could call pairwise_dist() function.
```

```
        """
        pairwise_distance = pairwise_dist(points, centers)    # (N,D) pairwise (K,D) --
        return np.argmin(pairwise_distance, axis = 1)         # (N,K) --axis=1 --> (N,)

    def _update_centers(self, old_centers, cluster_idx, points):
        """
        Args:
            old_centers: old centers KxD numpy array, where K is the number of cluster
            cluster_idx: numpy array of length N, the cluster assignment for each poin
            points: NxD numpy array, the observations
        Return:
            centers: new centers, K x D numpy array, where K is the number of clusters
        """
        centers = np.zeros((len(old_centers),points.shape[1]))  #(K,D)
        for k in range(len(old_centers)):
            center_k_points = points[cluster_idx==k]  #(Nk, D) where Nk <= N
            centers[k] = np.mean(center_k_points, axis=0)    #(Nk,D) --axis=0-->(D,)

        return centers

    def _get_loss(self, centers, cluster_idx, points):
        """
        Args:
            centers: KxD numpy array, where K is the number of clusters, and D is the
            cluster_idx: numpy array of length N, the cluster assignment for each poin
            points: NxD numpy array, the observations
        Return:
            loss: a single float number, which is the objective function of KMeans.

        Hint: The loss equals to the average squared distance for all the observation
              You could call pairwise_dist() function which you have implemented to ge
        """

        pairwise_distance = pairwise_dist(points, centers)    # (N,D) pairwise (K,D) --
        distance_to_centers = pairwise_distance[np.arange(len(points)),cluster_idx]  #
        return np.mean(np.sum(distance_to_centers**2))

    def __call__(self, points, K, max_iters=100, abs_tol=1e-16, rel_tol=1e-16, **kwargs
        """
        Args:
            points: NxD numpy array, where N is # points and D is the dimensionality
            K: number of clusters
            max_iters: maximum number of iterations (Hint: You could change it when de
            abs_tol: convergence criteria w.r.t absolute change of loss
            rel_tol: convergence criteria w.r.t relative change of loss
            kwargs: any additional arguments you want
        Return:
            cluster assignments: Nx1 int numpy array
```

```
            cluster centers: K x D numpy array, the centers

        Hint: You do not need to change it. For each iteration, we update the centers
        If the loss between two iterations qualify our two conditions, then we will st
        """
        centers = self._init_centers(points, K, **kwargs)
        pbar = tqdm(range(max_iters))
        for it in pbar:
            cluster_idx = self._update_assignment(centers, points)
            centers = self._update_centers(centers, cluster_idx, points)
            loss = self._get_loss(centers, cluster_idx, points)
            K = centers.shape[0]
            if it:
                diff = np.abs(prev_loss - loss)
                if diff < abs_tol and diff / prev_loss < rel_tol:
                    break
            prev_loss = loss
            pbar.set_description('iter %d, loss: %.4f' % (it, loss))
        return cluster_idx, centers
```

Let's try a simple image pixel clustering, which tries to compress the image using 5 colors:

In [7]:
```
image = imageio.imread('fruit.bmp')
im_height, im_width, im_channel = image.shape

flat_img = np.reshape(image, [-1, im_channel]).astype(np.float32)

cluster_ids, centers = KMeans()(flat_img, K=5)
# cluster_ids = (H*W ,)  with the values ranges from 0-(K-1)  ---> need to reshape to
# centers = (K,3)  - K `k-means` colors
# centers[cluster_ids] = (H,W,3) --> we only need to reshape cluster_ids

kmeans_img = centers[cluster_ids.reshape(im_height, im_width)]
# centers[cluster_idx] such a good numpy indexing
# Questions: how does the intensity be transformed in this algorithm?
# Answer: there is only 5 unique values in the kmeans picture. But it looks brighter a


# A very good visualization of the 5 different classes.
# This visualization of K-means result is much better than drawing the decision bounda
# where majority of k-means lectures normally do it

# I also like the fact that the k-means here acts on 3-D data,
# and again, this is a good 2-D representation of a 3-D structure

plot_images([image, kmeans_img], ['origin', 'kmeans'])
```

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

and 16 colors:

```
In [8]: cluster_ids, centers = KMeans()(flat_img, K=16, max_iters=500)

        kmeans_img = np.reshape(centers[cluster_ids], (im_height, im_width, im_channel))

        plot_images([image, kmeans_img], ['origin', 'kmeans'])
```

```
HBox(children=(IntProgress(value=0, max=500), HTML(value='')))
```

To get an intuition for what a multivariate Gaussian is, consider the simple case where $n = 2$, and where the covariance matrix Σ is diagonal, i.e.,

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} \qquad \Sigma = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}$$

In this case, the multivariate Gaussian density has the form,

$$\begin{aligned} p(x;\mu,\Sigma) &= \frac{1}{2\pi \begin{vmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{vmatrix}^{1/2}} \exp\left(-\frac{1}{2}\begin{bmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{bmatrix}^T \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}^{-1} \begin{bmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{bmatrix}\right) \\ &= \frac{1}{2\pi(\sigma_1^2\cdot\sigma_2^2 - 0\cdot 0)^{1/2}} \exp\left(-\frac{1}{2}\begin{bmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{bmatrix}^T \begin{bmatrix} \frac{1}{\sigma_1^2} & 0 \\ 0 & \frac{1}{\sigma_2^2} \end{bmatrix} \begin{bmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{bmatrix}\right), \end{aligned}$$

where we have relied on the explicit formula for the determinant of a 2×2 matrix[3], and the fact that the inverse of a diagonal matrix is simply found by taking the reciprocal of each diagonal entry. Continuing,

$$\begin{aligned} p(x;\mu,\Sigma) &= \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{1}{2}\begin{bmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{bmatrix}^T \begin{bmatrix} \frac{1}{\sigma_1^2}(x_1-\mu_1) \\ \frac{1}{\sigma_2^2}(x_2-\mu_2) \end{bmatrix}\right) \\ &= \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{1}{2\sigma_1^2}(x_1-\mu_1)^2 - \frac{1}{2\sigma_2^2}(x_2-\mu_2)^2\right) \\ &= \frac{1}{\sqrt{2\pi}\sigma_1} \exp\left(-\frac{1}{2\sigma_1^2}(x_1-\mu_1)^2\right) \cdot \frac{1}{\sqrt{2\pi}\sigma_2} \exp\left(-\frac{1}{2\sigma_2^2}(x_2-\mu_2)^2\right). \end{aligned}$$

title

4 3. GMM implementation [30 pts]

GMM is trying to do MLE (maximum likelihood estimation). It approximates the distribution of data using a set of gaussian distributions. Here we assume that each gaussian component has diagonal covariance matrix, which makes it easier to calculate the density.

Given N samples $X = [x_1, x_2, \ldots, x_N]$, we are asked to find K diagonal gaussian distributions to model the data X:

$$\max_{\{\mu_k,\sigma_k\}_{k=1}^K} \sum_{i=1}^{N} \log\Big(\sum_{k=1}^{K} \pi(k)\mathcal{N}(x_i|\mu_k,\sigma_k)\Big) \tag{2}$$

Hints

1. Here $\pi(\cdot)$ is the prior of the latent variable. To make it simple, we assume $\pi(k) = \frac{1}{K}, \forall k = 1, 2, \ldots, K$.

2. As we create our model, we will need to use a multivariate Gaussian since our pixels are 3-dimensional vectors corresponding to red, green, and blue color intensities. For simplicity, we will also assume that our components are independent (i.e. the red intensity of a pixel is independent from its blue intensity, etc). This allows us to simplify the computation.

The following example from a machine learning textbook may be helpful:

3. At EM steps, gamma means $\tau\,(z_{nk})$ at our slide of GMM.

4. For E steps, we already get the log-likelihood at ll_joint() function. For the fomula at our slide:

$$\tau\left(z_{nk}\right)=\frac{\pi_k N\left(x_n|\mu_k,\Sigma_k\right)}{\sum_{j=1}^{K}\pi_j N\left(x_n|\mu_j,\Sigma_j\right)},$$

ll_joint equals to the N here. Thus, You should be able to finish E steps with just a few lines of code by using ll_joint() and softmax() defined above.

```
In [9]: class GMM(object):
            def __init__(self): # No need to implement
                pass

            def _init_components(self, points, K, **kwargs):
                """
                Args:
                    points: NxD numpy array, the observations
                    K: number of components
                    kwargs: any other args you want
                Return:
                    pi: numpy array of length K, prior
                    mu: KxD numpy array, the center for each gaussian.
                    sigma: KxD numpy array, the diagonal standard deviation of each gaussian.

                Hint: You could use the K-means results to initial GMM. It will help to convery
                For instance, you could use ids, mu = KMeans()(points, K)  to initialize.
                """
                pi = np.ones(K)/K                                  # (K,)
                _, mu = KMeans()(points, K=K, max_iters=100)    # mu: KxD
                sigma = np.ones((K, points.shape[1]))/points.shape[1]  # (K,D) standard deviat

                return pi, mu, sigma

            def _ll_joint(self, points, pi, mu, sigma):
                """
                Args:
                    points: NxD numpy array, the observations
                    pi: np array of length K, the prior of each component
                    mu: KxD numpy array, the center for each gaussian.
                    sigma: KxD numpy array, the diagonal standard deviation of each gaussian.
                Return:
                    ll(log-likelihood): NxK array, where ll(i, j) = log pi(j) + log NormalPDF(

                Hint: Assume that the three dimensions of our multivariate gaussian are indepe
                      This allows you to write treat it as a product of univariate gaussians.
                """
                # First, I need to minus points = (N,D) - mu = (K,D) to a vector diff =  (N,D)

                N, D = points.shape
```

```
        log_pi = np.log(pi)                    # pi (K,) -- Ready to broadcast

        minus_log_d = -0.5*D*np.log(np.pi)     # (1)     -- Ready to broadcast

        minus_log_product_sigma_square = -0.5*np.log(np.product(sigma**2, axis = 1))
        # (K,D) --axis=1--> (K,) --Ready to broadcast

        diff = points[:,None] - mu[None:]   # (N,1,D) - (1,K,D) = (N,K,D)
        minus_sum_of_square = -0.5*np.sum((diff**2/sigma), axis=-1) #(N,K,D)/(K,D) = (

        return log_pi + minus_log_d + minus_log_product_sigma_square + minus_sum_of_squ

    def _E_step(self, points, pi, mu, sigma):
        """
        Args:
            points: NxD numpy array, the observations
            pi: np array of length K, the prior of each component
            mu: KxD numpy array, the center for each gaussian.
            sigma: KxD numpy array, the diagonal standard deviation of each gaussian.
        Return:
            gamma: NxK array, the posterior distribution (a.k.a, the soft cluster assig

        Hint: You should be able to do this with just a few lines of code by using _ll
        """
        return softmax(self._ll_joint(points, pi, mu, sigma))          #(N,K)

    def _M_step(self, points, gamma):
        """
        Args:
            points: NxD numpy array, the observations
            gamma: NxK array, the posterior distribution (a.k.a, the soft cluster assig
        Return:
            pi: np array of length K, the prior of each component
            mu: KxD numpy array, the center for each gaussian.
            sigma: KxD numpy array, the diagonal standard deviation of each gaussian.

        Hint:  There are formulas in the slide.
        """
        pi = np.mean(gamma, axis = 0)             # (N,K) --axis=0-->(K,)
        sum_gamma_K = np.sum(gamma, axis = 0)[:,None]     # (N,K) --axis=0-->(K,) --[:,

        product_gamma_points = np.sum(gamma[:,:,None]*points[:,None,:], axis = 0)  #(N
        mu = product_gamma_points/sum_gamma_K   #(K,D)/(K,1) = (K,D)

        square_diff = (points[:,None] - mu[None:])**2   # (N,1,D) - (1,K,D) = (N,K,D)
        product_gamma_square_diff = np.sum(square_diff*gamma[:,:,None],axis=0)  #(N,K,
        sigma = product_gamma_square_diff/sum_gamma_K  #(K,D)/(K,1) =(K,D)
```

```
        return pi, mu, sigma


    def __call__(self, points, K, max_iters=100, abs_tol=1e-16, rel_tol=1e-16, **kwargs
        """
        Args:
            points: NxD numpy array, where N is # points and D is the dimensionality
            K: number of clusters
            max_iters: maximum number of iterations
            abs_tol: convergence criteria w.r.t absolute change of loss
            rel_tol: convergence criteria w.r.t relative change of loss
            kwargs: any additional arguments you want
        Return:
            gamma: NxK array, the posterior distribution (a.k.a, the soft cluster assi
            (pi, mu, sigma): (1xK np array, KxD numpy array, KxD numpy array), mu and

        Hint: You do not need to change it. For each iteration, we process E and M ste
        """
        pi, mu, sigma = self._init_components(points, K, **kwargs)

        pbar = tqdm(range(max_iters))
        for it in pbar:
            # E-step

            gamma = self._E_step(points, pi, mu, sigma)

            # M-step
            pi, mu, sigma = self._M_step(points, gamma)

            # calculate the negative log-likelihood of observation
            joint_ll = self._ll_joint(points, pi, mu, sigma)
            loss = -np.sum(logsumexp(joint_ll))
            if it:
                diff = np.abs(prev_loss - loss)
                if diff < abs_tol and diff / prev_loss < rel_tol:
                    break
            prev_loss = loss
            pbar.set_description('iter %d, loss: %.4f' % (it, loss))
        return gamma, (pi, mu, sigma)
```

Let's try a simple image pixel clustering, which tries to compress the image using 5 colors:

```
In [10]: image = imageio.imread('fruit.bmp')
         im_height, im_width, im_channel = image.shape
         flat_img = np.reshape(image, [-1, im_channel]).astype(np.float32)
         gamma, (pi, mu, sigma) = GMM()(flat_img, K=5, max_iters=200)
         cluster_ids = np.argmax(gamma, axis=1)
         centers = mu
```

```
        gmm_img = np.reshape(centers[cluster_ids], (im_height, im_width, im_channel))

        plot_images([image, gmm_img], ['origin', 'gmm'])
```

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, max=200), HTML(value='')))
```

and 16 colors

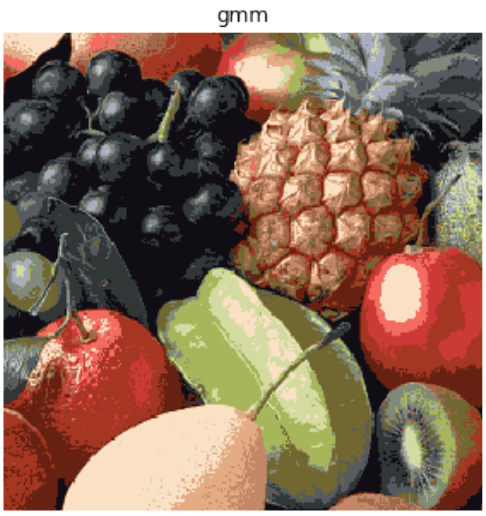
```
In [11]: gamma, (pi, mu, sigma) = GMM()(flat_img, K=16, max_iters=200)
         cluster_ids = np.argmax(gamma, axis=1)
         centers = mu

         gmm_img = np.reshape(centers[cluster_ids], (im_height, im_width, im_channel))

         plot_images([image, gmm_img], ['origin', 'gmm'])
```

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, max=200), HTML(value='')))
```



5 3. Compare KMeans with GMM [10 pts]

In this section, we are going to have a comparison between the two algorithms.

5.1 mixture model

We first create a dataset that is sampled from the mixture model. To do so, you first need a sampling method, that can sample from this mixture model. The sampling procedure looks like this:

- first sample a component id according to prior distribution $\pi(\cdot)$
- then choose the corresponding gaussian distribution, and sample from that gaussian

```
In [12]: def gmm_sampling(num_samples, pi, mu, sigma):
             """
             Args:
                 num_samples: number of samples required
                 pi: np array of length K, the prior distribution, where K is # components
                 mu: KxD np array, the center for each gaussian, where D is data dimension
                 sigma: KxD np array, the standard deviation for each gaussian
             Return:
                 samples: NxD np array, the result samples
                 ids: np array of length N, the component id for each sample

             Hint: You could use np.random.randn() function.
             """
             K, D = mu.shape

             ids = np.random.choice(K,num_samples,p=pi)
```

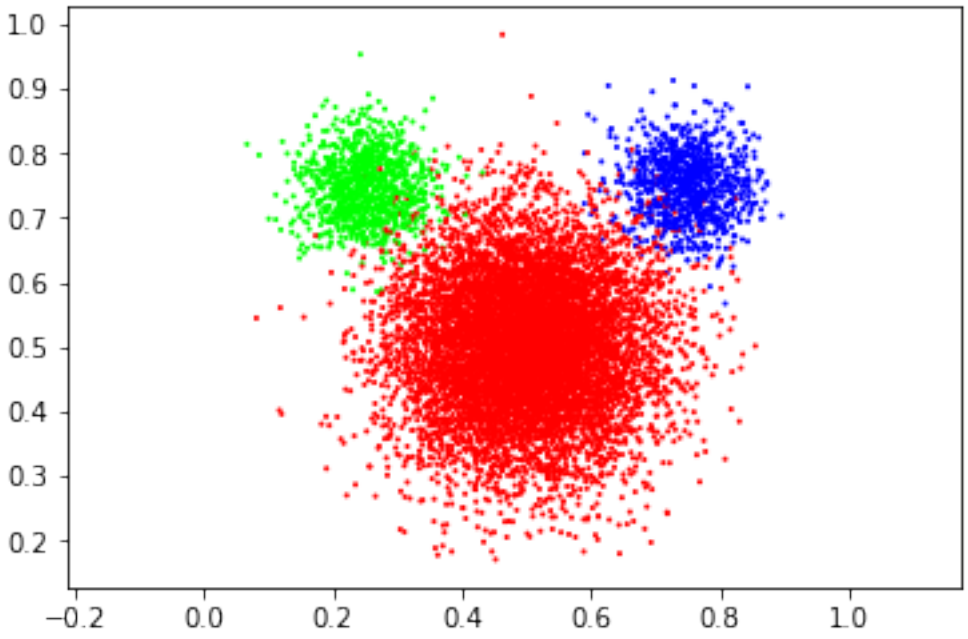
```
    data = sigma[ids]*np.random.randn(num_samples,D) + mu[ids]

    return data, ids

    '''
    data, ids = np.zeros((num_samples,D)), np.zeros(num_samples)
    for idx in range(num_samples):
        class_k = np.random.choice(K,1,p=pi)
        ids[idx] = class_k
        data[idx] = sigma[class_k] * np.random.randn(D) + mu[class_k]
    return data, ids.astype(int)
    '''

pi = np.array([0.8, 0.1, 0.1])
mu = np.array([[0.5, 0.5],
               [0.25, 0.75],
               [0.75, 0.75]], dtype=np.float32)
sigma = np.array([[0.1, 0.1],
                  [0.05, 0.05],
                  [0.05, 0.05]], dtype=np.float32)
samples, ids = gmm_sampling(10000, pi, mu, sigma)

plot_scatter(samples, ids)
```

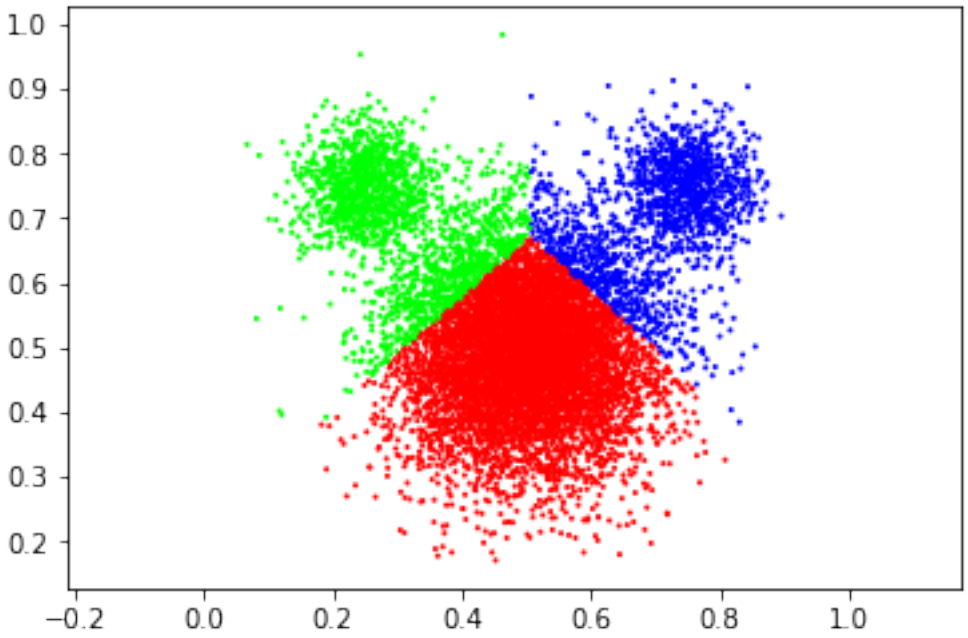


Let's see how KMeans does in this case

```
In [13]: ids, centers = KMeans()(samples, K=3)
```

```
        plot_scatter(samples, ids)
```

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

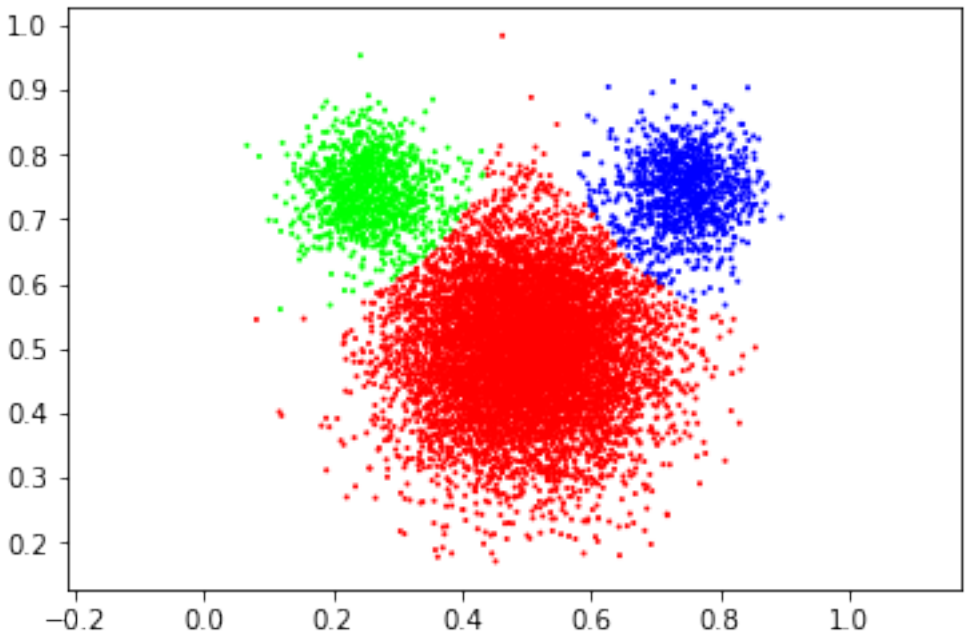


And GMM

```
In [14]: gamma, (pi, mu, sigma) = GMM()(samples, K=3, max_iters=200)
         ids = np.argmax(gamma, axis=1)
         plot_scatter(samples, ids)
```

```
HBox(children=(IntProgress(value=0), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, max=200), HTML(value='')))
```



6 4. Written Questions [10 pts]

In this section, you are asked to answer several question. You are expected to write down the answers in the notebook markdown cells (similar to the current one you are reading). You can also type Latex equations if necessary.

6.1 4.1 Initialization.

- What initialization method you used for KMeans and GMM, respectively?
- Did you see different results after different runs of the same code? Why?

Your answer goes here:

- I use the `np.random.choice()` function to **randomly select K indexes** from the set of index for the `Kmeans`. This guarantees that the centers are initilized inside the cluster.
- For GMM, I set the **prior probability** to be the same for all classes (equals to `1/K`). I use the `means of Kmeans` as the initial values for the **GMM centers** `mu`. I set all the **standard deviations** to be the same and equals to `1/K` as intital value for the standard deviation.

I see the results are `more or less the same` for different runs of the same code. Sometimes, it takes a longer number of iterations for the algorithm to converge, but the final clustering results does not considerably change. The reason is that we set the `fixed seed` for the `random` function

6.2 4.2 Convergence.

- Is it guaranteed that GMM with EM always converges to the optimal solution?
- Is KMeans always guaranteed to converge (to some local optima)?

Your answer goes here:

- It is NOT guaranteed that GMM and EM always converges to GLOBAL optimal solution. This is a `NP-Hard` problem, and the proposed techniques can only achieves LOCAL optima.
- KMEANs always converge to some local optima. After each iteration, the `loss function` we want to optimize CAN NOT grow bigger, and can only `decrease`. After certain iterations, the loss function will be stable, or change within our expected tolerance.

In [ ]: