计算机科学与技术学院2018机器学习—实验报告

- 课程名称： 机器学习
- 课程类型： 选修
- 实验题目： K-Means聚类和GMM模型
- 学号：1160100626
- 姓名：单心茹

实验目的

实现一个k-means算法和混合高斯模型，并且用EM算法估计模型中的参数。

实验要求求及实验环境

实验要求

用高斯分布产生k个高斯分布的数据（不同均值和方差）（其中参数自己设定）。

- 用k-means聚类，测试效果；
- 用混合高斯模型和你实现的EM算法估计参数，看看每次迭代后似然值变化情况，考察EM算法是否可以获得正确的结果（与你设定的结果比较）。

应用：可以UCI上找一个简单问题数据，用你实现的GMM进行聚类。

实验环境

Win10 + 8GB RAM + python 3.6

设计思想与实现

K-Means

K-means算法是很典型的基于距离的聚类算法，采用距离作为相似性的评价指标，即认为两个对象的距离越近，其相似度就越大。**K-Means**是一种**无监督学习**聚类算法，该算法的目标是在给定数据中查找簇。

Algorithm：

1. Initialize **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.
2. Repeat until convergence: $\{$

   For every i, set

   $$c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$$

   For each j, set

   $$\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$$

   $\}$

循环内执行两个步骤：

1. 为样本归类：对每一个样本计算它到各个中心的距离，并将其归类为距离最近的类。
2. 更新中心位置：对每个中心，将其调整到该类样本点位置的均值处。

代价函数为：

$$J(c, \mu) = \sum_{i=1}^{m} ||x^i - \mu_{c^{(i)}}||^2$$

所以，K-means的实质优化对象是**样本到中心的距离。**

但是，J是非凸函数，所以**K-means不能保证全局收敛，可能会得到局部最优解**，要避免陷入局部优化，可以**多次运行K-means，使用不同的初始中心，使得J最小。**

其中K-Means的核心实现部分如下：

```
def k_means(self):
    """
    clustering and update centroids
    :return: clustering result
    """
    centroids = self.random_centroids()
    count = 0
    for i in range(0, self.loop_max):
        count += 1
        self.get_labels()    # 更新样本的类别
        former_centroids = centroids
        # 更新中心的位置
        centroids = self.update_centroids()

        diff = centroids - former_centroids
        if diff.any() < self.epsilon:
            break
     return self.labels
```

GMM

高斯混合模型也是一种无监督学习算法，在聚类问题中各个类别的尺寸不同，聚类间有相关关系时，使用GMM较为合适，是多个高斯模型的线性叠加。

高斯混合模型的概率分布模型如下：

$$P(x) = \sum_{k=1}^{K} \alpha_k \phi(x; \mu_k, \Sigma_k)$$

GMM的似然函数为：

$$\sum_{i=1}^{N} log(\sum_{k=1}^{K} \alpha_k \phi(x; \mu_k, \Sigma_k))$$

其中，K表示模型的个数，α_k是第k个模型的系数，为该模型的概率，$\phi(x; \mu_k, \Sigma_k)$是第k个高斯模型的概率分布。

假定现有的数据是由GMM生成出来的，那么**只需要根据数据推出GMM的概率分布，则可以调用EM算法来迭代求出GMM的概率分布。**

GMM的核心实现部分如下（采用EM算法）：

```python
def gmm_em(self):
    """
    GMM algorithm, estimate mu, cov, alpha of GMM
    :return: estimate of mu, cov, alpha
    """
    self.x = self.scaling()
    oldlikelyhood = 1
    for i in range(0, self.loop_max):
        gamma, likelyhood = self.e_step(self.mu, self.cov, self.alpha)  # e-step
        self.mu, self.cov, self.alpha = self.m_step(gamma)  # m-step
        if np.abs(np.sum(likelyhood-oldlikelyhood)) < self.epsilon:
            print(i)
            break
        else:
            oldlikelyhood = likelyhood
    print('------------------result-------------------')
    print("mu:", self.mu)
    print("cov:", self.cov)
    print("alpha:", self.alpha)
    print('-------------------------------------------')
    return self.mu, self.cov, self.alpha
```

EM

EM算法首先需要给出初始化的模型参数，然后通过不断迭代来求得最佳参数。初始化每个模型的μ为随机值，Σ单位矩阵，α为$\frac{1}{K}$，即为每个模型初始时都是等概率出现的。其中需要先把样本进行归一化处理。

Algorithm：

E-step：根据当前模型参数，计算模型k对观测数据x_i的响应度\gamma_{i}^{k}：

$$\gamma_i^k = \frac{\alpha_k \phi(x_i; \mu_k, \Sigma_k)}{\sum_{k=1}^{K} \phi(x_i; \mu_k, \Sigma_k)}$$

M-step：计算新一轮迭代的模型参数：

$$\mu_k = \frac{\sum_{i=1}^{N} \gamma_{ik} x_i}{\sum_{i=1}^{N} \gamma_{ik}}$$

$$\Sigma_k = \frac{\sum_{i=1}^{N} \gamma_{ik} (x_i - \mu_k)^2}{\sum_{i=1}^{N} \gamma_{ik}}$$

$$\alpha_k = \frac{\sum_{i=1}^{N} \gamma_{ik}}{N}$$

重复E-step和M-step直到算法收敛。

E-step实现如下：

```python
def e_step(self, mu, cov, alpha):
    """
    e-step of EM algorithm
    :param mu: mean
    :param cov: Covariance matrix
    :param alpha: probability of  each model
    :return: responsibility for each sample
    """
    # gamma为响应度矩阵，行代表样本，列表示对第k个模型
    gamma = np.mat(np.zeros((self.sample, self.k)))
    probility = np.zeros((self.sample, self.k))
    # 计算k个模型中，样本的概率
    for k in range(0, self.k):
        probility[:, k] = self.phi(self.x, mu[k], cov[k])
    probility = np.mat(probility)
    # 计算响应度矩阵
    for k in range(0, self.k):
        gamma[:, k] = alpha[k] * probility[:, k]
    for i in range(0, self.sample):
        gamma[i, :] = gamma[i, :] / np.sum(gamma[i, :])
    return gamma
```

M-step实现如下：

```python
def m_step(self, gamma):
    """
    m-step of EM algorithm
    :param gamma: responsibility for each sample
    :return: the update mu, cov, alpha
    """
    mu = np.zeros((self.k, self.dimension))
```

```
    alpha = np.zeros(self.k)
    cov = []

    for k in range(self.k):
        gamma_sum = np.sum(gamma[:, k])      # 计算响应度之和
        # 更新mu
        for d in range(self.dimension):
            mu[k, d] = np.sum(np.multiply(gamma[:, k], self.x[:, d])) / gamma_sum
        # 更新cov
        cov_k = np.mat(np.zeros((self.dimension, self.dimension)))
        for i in range(self.sample):
            cov_k += gamma[i, k] * (self.x[i] - mu[k]).T * (self.x[i] - mu[k]) / gamma_sum
        cov.append(cov_k)
        # 更新 alpha
        alpha[k] = gamma_sum / self.sample
    cov = np.array(cov)
    return mu, cov, alpha
```

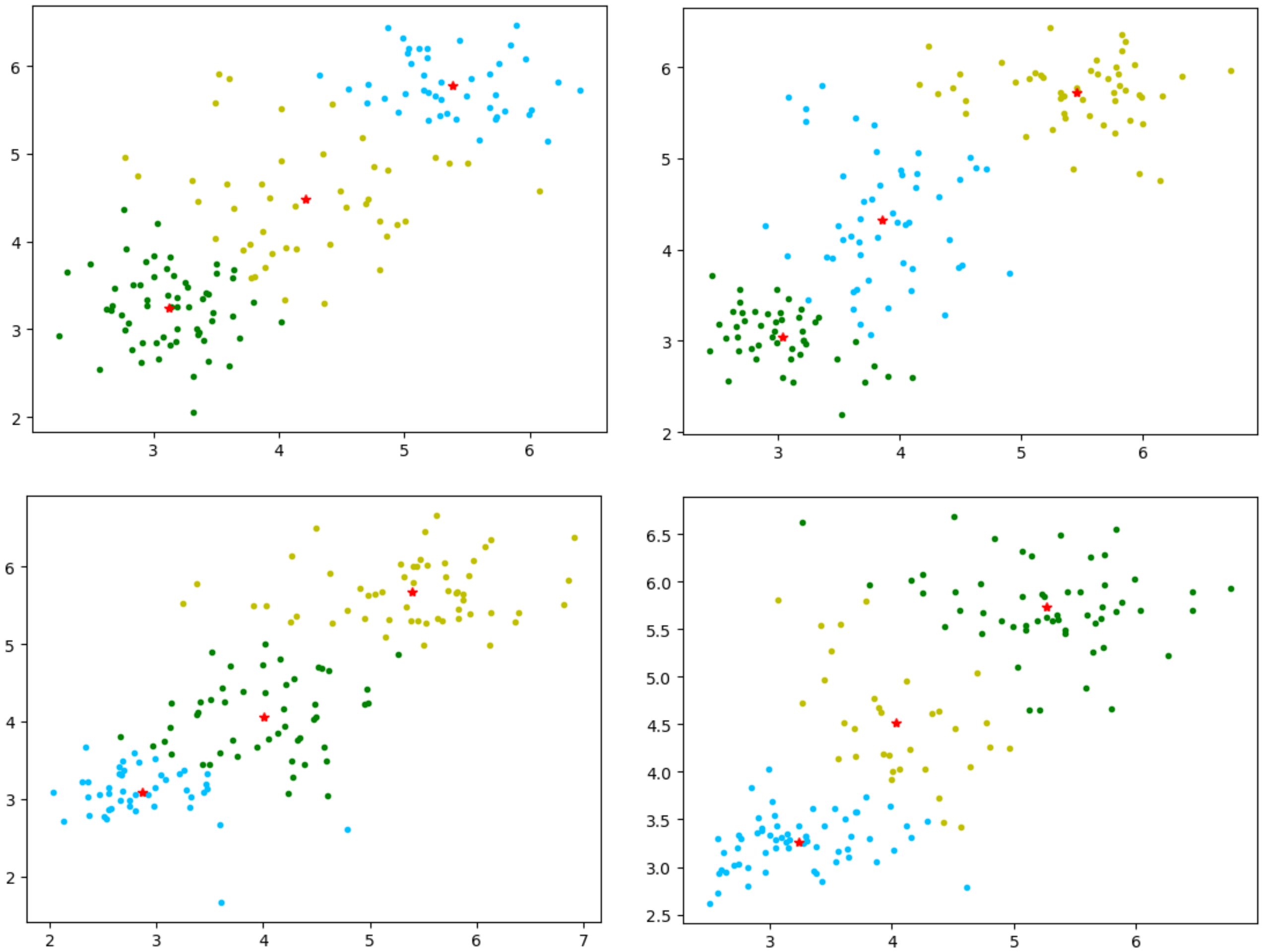
实验结果与分析

K-Means

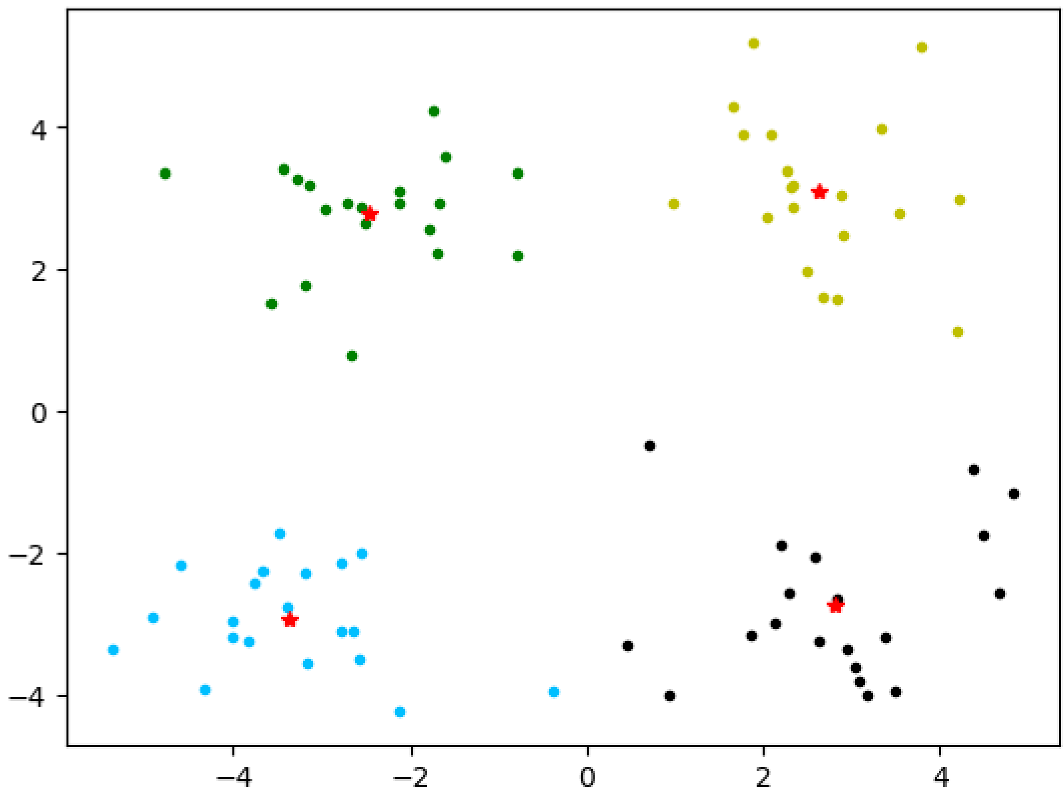
- 利用**高斯分布**生成3类数据，两个特征，自定义高斯分布实现如下：

```
def create_data():
    np.random.seed()
    s1 = np.random.normal(3, 0.4, 50)
    s2 = np.random.normal(3.2, 0.3, 50)
    s3 = np.random.normal(4, 0.5, 50)
    s4 = np.random.normal(4.3, 0.8, 50)
    s5 = np.random.normal(6, 1, 50)
    s6 = np.random.normal(5.7, 0.8, 50)
    with open('testGaussian.txt', 'w') as f:
        for i in range(0, len(s1)):
            f.write(str(s1[i]) + ' ' + str(s2[i]) + '\n')
            f.write(str(s3[i]) + ' ' + str(s4[i]) + '\n')
            f.write(str(s5[i]) + ' ' + str(s6[i]) + '\n')
        f.close()
```

下图是几次分类情况，由于高斯分布样本的产生具有随机性，每次运行结果也具有很大的随机性：

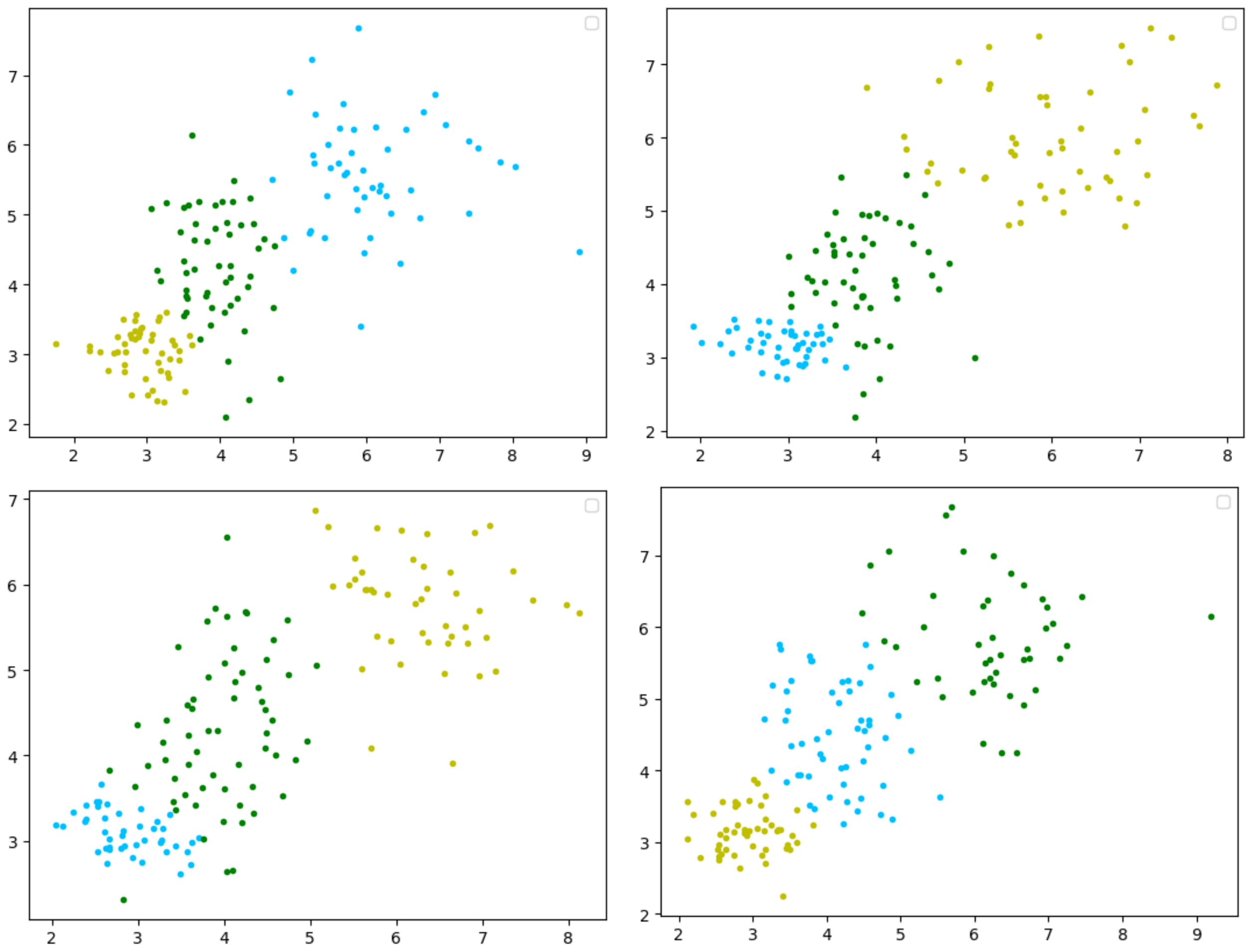


- 利用test.txt测试分类情况如下:



GMM

- 利用**高斯分布**生成3类数据，两个特征，自定义高斯分布实现同KMeans，结果如下：



- 在利用Guassin生成的数据时观察似然函数的变化，可以看到似然函数增大，且最后保证收敛性：

GMM

```
likelyhood:  3.085296511681428
likelyhood:  5.988774210342251
likelyhood:  5.99545675412898
likelyhood:  6.021775436788563
likelyhood:  6.083982285040935
likelyhood:  6.179438541288814
likelyhood:  6.296753608062412
likelyhood:  6.395664373903428
likelyhood:  6.444942868812104
likelyhood:  6.461961558683946
likelyhood:  6.467638139257662
likelyhood:  6.470824566240885
likelyhood:  6.47329211067583
likelyhood:  6.474860563379607
likelyhood:  6.475565124887228
likelyhood:  6.475789786809695
likelyhood:  6.475911439372398
likelyhood:  6.476153080036989
likelyhood:  6.476618531495132
likelyhood:  6.477353975844728
likelyhood:  6.478384070640672
likelyhood:  6.47972600243873
likelyhood:  6.4813926356574 9
likelyhood:  6.483391767700783
likelyhood:  6.485724530342664
likelyhood:  6.488383583531865
```

- UCI数据测试：

seeds Data Set

Download: Data Folder, Data Set Description

Abstract: Measurements of geometrical properties of kernels belonging to three different varieties of wheat. A soft X-ray technique and GRAINS package were used to construct all seven, real-valued attributes.

Data Set Characteristics:	Multivariate	**Number of Instances:**	210	**Area:**	Life
Attribute Characteristics:	Real	**Number of Attributes:**	7	**Date Donated**	2012-09-29
Associated Tasks:	Classification, Clustering	**Missing Values?**	N/A	**Number of Web Hits:**	203783

包括属于三种不同品种小麦的籽粒：Kama，Rosa和Canadian，每种70个元素。

数据为3类，7个特征

Attribute Information:

To construct the data, seven geometric parameters of wheat kernels were measured:
1. area A,
2. perimeter P,
3. compactness C = 4*pi*A/P^2,
4. length of kernel,
5. width of kernel,
6. asymmetry coefficient
7. length of kernel groove.

All of these parameters were real-valued continuous.

由于多特征数据不适合采用画图进行直观呈现，故只呈现计算结果，在GMM中运行结果如下：

均值μ:

```
mu:
[[0.735745   0.77247143 0.69425801 0.70372761 0.75494207 0.37025743
  0.73520652]
 [0.36550612 0.42072466 0.52452552 0.40934343 0.4152109  0.30426661
  0.39616639]
 [0.20622005 0.24858681 0.50553318 0.23211176 0.26779679 0.40566646
  0.26520559]]
```

协方差矩阵Σ:

```
cov:
[[[ 1.62974434e-02  1.51968190e-02  2.75798693e-03  1.59503270e-02
   1.29535971e-02 -2.61052827e-04  1.23964885e-02]
 [ 1.51968190e-02  1.49388027e-02 -1.10433279e-03  1.67329539e-02
   1.05793611e-02  3.12392858e-05  1.31770933e-02]
 [ 2.75798693e-03 -1.10433279e-03  1.82522381e-02 -6.20671317e-03
   9.38783489e-03 -2.09872106e-03 -5.66511317e-03]
 [ 1.59503270e-02  1.67329539e-02 -6.20671317e-03  2.24654510e-02
   8.68912568e-03 -7.55730328e-04  1.81496222e-02]
 [ 1.29535971e-02  1.05793611e-02  9.38783489e-03  8.68912568e-03
   1.40382610e-02  4.16350094e-04  5.97919232e-03]
 [-2.61052827e-04  3.12392858e-05 -2.09872106e-03 -7.55730328e-04
   4.16350094e-04  2.35651284e-02 -7.34676281e-04]
 [ 1.23964885e-02  1.31770933e-02 -5.66511317e-03  1.81496222e-02
   5.97919232e-03 -7.34676281e-04  1.69118971e-02]]

[[ 2.94556948e-02  2.53358229e-02  4.53853993e-02  1.58963871e-02
   3.61933004e-02 -3.05552742e-02  5.32109455e-03]
 [ 2.53358229e-02  2.23496319e-02  3.62920387e-02  1.46394964e-02
   2.99658543e-02 -2.59339600e-02  6.39769064e-03]
 [ 4.53853993e-02  3.62920387e-02  8.43035198e-02  1.94478880e-02
   6.18034497e-02 -4.85880313e-02 -1.33604186e-03]
 [ 1.58963871e-02  1.46394964e-02  1.94478880e-02  1.15958463e-02
   1.73386033e-02 -1.31654050e-02  7.81956620e-03]
 [ 3.61933004e-02  2.99658543e-02  6.18034497e-02  1.73386033e-02
   4.72620207e-02 -3.82141320e-02  2.25966261e-03]
 [-3.05552742e-02 -2.59339600e-02 -4.85880313e-02 -1.31654050e-02
  -3.82141320e-02  4.71269345e-02  2.10340364e-03]
 [ 5.32109455e-03  6.39769064e-03 -1.33604186e-03  7.81956620e-03
   2.25966261e-03  2.10340364e-03  1.20856266e-02]]

[[ 1.44906548e-02  1.40906521e-02  1.53017868e-02  1.07323048e-02
   1.68737329e-02 -9.17279569e-03  1.17422065e-03]
 [ 1.40906521e-02  1.44438859e-02  1.07159345e-02  1.21673362e-02
   1.50643886e-02 -8.61156057e-03  2.94782981e-03]
 [ 1.53017868e-02  1.07159345e-02  4.00069477e-02  1.60474411e-03
   2.55226027e-02 -1.16838305e-02 -8.90982975e-03]
 [ 1.07323048e-02  1.21673362e-02  1.60474411e-03  1.28212617e-02
   9.07066338e-03 -5.57971748e-03  5.76779792e-03]
 [ 1.68737329e-02  1.50643886e-02  2.55226027e-02  9.07066338e-03
   2.28494675e-02 -1.06706421e-02 -2.35365014e-03]
 [-9.17279569e-03 -8.61156057e-03 -1.16838305e-02 -5.57971748e-03
  -1.06706421e-02  4.28938608e-02  4.53255976e-03]
 [ 1.17422065e-03  2.94782981e-03 -8.90982975e-03  5.76779792e-03
  -2.35365014e-03  4.53255976e-03  9.54436930e-03]]]
```

概率α：

```
alpha:
[0.33352659 0.12054896 0.54592446]
```

结论

- K-Means 往往迭代次数较少，但容易陷入局部最优，或者结果受到初值的影响。
- GMM-EM迭代次数较多，GMM实质上与K-Means有很多相同之处，优点在于EM算法最大化似然函数，可以保证收敛性。

参考文献

斯坦福机器学习课程K-Means，GMM

李航《统计学习方法》

K-Means Clustering in Python

斯坦福ML公开课笔记12——K-Means、混合高斯分布、EM算法

GMM的EM算法实现

高斯混合模型 EM 算法的 Python 实现

附录：源代码

详见压缩包内k-means.py, GMM.py