



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2020 年春季学期
计算学部《机器学习》课程

Lab 3 实验报告

姓名	赵仁杰
学号	1180300113
班号	1803001
电子邮件	1579974122@qq.com
手机号码	15122925619

目录

1 实验目的	2
2 实验要求、环境.....	3
2.1 实验要求	3
2.2 实验环境	3
3 算法原理	3
3.1 k-means 聚类.....	3
3.2 GMM:	4
3.3 高斯混合模型参数估计的 EM 算法:	4
4 算法实现:	6
4.1 k-means 聚类实现:	6
4.1.1 随机选取样本作为初始均值向量.....	6
4.1.2 利用最大化初始均值向量之间距离方式进行选择.....	7
4.2 GMM 算法实现.....	7
5 : 实验结果分析:	7
5.1 : 按照生成的数据测试.....	7

1 实验目的

理解 k-means 聚类过程

理解混合高斯模型(GMM)用 EM 估计参数的实现过程

掌握 k-means 和混合高斯模型的联系

学会 EM 估计参数的方法和代码实现

学会用 GMM 解决实际问题

2 实验要求、环境

2.1 实验要求

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。

测试：用高斯分布产生的 k 个高斯分布的数据。

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，管擦和 EM 算法是否可以获得正确结果。

应用：在 UCI 上找一个简单问题数据，用你实现的 GMM 进行聚类。

2.2 实验环境

Win10, x86-64

Pycharm 2019.1

python 3.7.1

使用 python 库：numpy（计算），sklearn(用于聚类验证), matplotlib.pyplot(用于可视化)

3 算法原理

3.1 k-means 聚类

K-Means 算法的思想很简单，对于给定的样本集，按照样本之间的距离大小，将样本集划分为 K 个簇。让簇内的点尽量紧密的连在一起，而让簇间的距离尽量大。

k-means 聚类就是根据某种度量方式(常用欧氏距离，如欧氏距离越小，相关性越大)，将相关性较大的一些样本点聚集在一起，一共聚成 k 个堆，每一个堆我们称为一个“类”。k-means 的过程为：先在样本点中选取 k 个点作为暂时的聚类中心，然后依次计算每一个样本点与这 k 个点的距离，将每一个与距离这个点最近的中心点聚在一起，这样形成 k 个类“堆”，求每一个类的期望，将求得的期望作为这个类的新的中心点。一直不停地将所有样本点分为 k 类，直至中心点不再改变停止。

给定样本集 $D = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$ 和划分聚类的数量 k ，给出一个簇划分 $C = \{C_1, C_2, \dots, C_k\}$ ，使得该簇划分的平方误差 E 最小化，其中 E 如式(1)

$$E = \sum_{i=1}^k \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \mu_i\|_2^2 \quad (1)$$

式(1)中， $\mu_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}$ 是簇 C_i 的均值向量。 E 刻画了簇内样本的内聚的紧密程度，其值越小，则簇内样本的相似度越高。

3.2 GMM:

首先给出 维样本空间中的随机变量 服从高斯分布的密度函数:

$$p(\mathbf{x}|\mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)^T \Sigma^{-1}(\mathbf{x} - \mu)\right)$$

其中 $\mu = \{\mu_1, \mu_2, \dots, \mu_n\}$ 为 n 维的均值向量, Σ 是 $N \times N$ 的协方差阵。

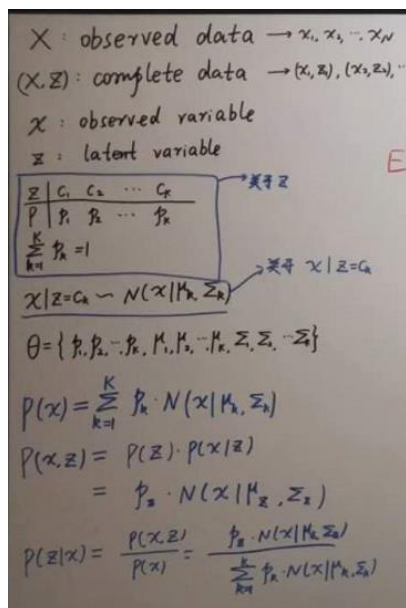
之后可以引出:

$$p_M = \sum_{i=1}^k \alpha_i p(\mathbf{x}|\mu_i, \Sigma_i)$$

这个分布由 k 个混合成分构成, 每个混合成分对应一个高斯分布。由 $1 \rightarrow k$ 的系数和为 1。

通常将 (α, Σ, μ) 记为 θ , 是高斯混合模型中的隐变量, 由于隐变量的存在, 高斯混合模型无法求出解析解, 但是可以用 EM 算法迭代求解隐变量, 并完成分类。

手写推导如下:



X : observed data $\rightarrow x_1, x_2, \dots, x_N$
 (X, Z) : complete data $\rightarrow (x_1, z_1), (x_2, z_2), \dots$
 x : observed variable
 z : latent variable

z	c_1	c_2	\dots	c_k
p	p_1	p_2	\dots	p_k
μ	μ_1	μ_2	\dots	μ_k
Σ	Σ_1	Σ_2	\dots	Σ_k

$\sum_{k=1}^K p_k = 1$
 $x|z=c_k \sim N(x|\mu_k, \Sigma_k)$
 $\theta = \{p_1, p_2, \dots, p_k, \mu_1, \mu_2, \dots, \mu_k, \Sigma_1, \Sigma_2, \dots, \Sigma_k\}$
 $p(x) = \sum_{k=1}^K p_k \cdot N(x|\mu_k, \Sigma_k)$
 $p(x, z) = p(z) \cdot p(x|z)$
 $= p_k \cdot N(x|\mu_k, \Sigma_k)$
 $p(z|x) = \frac{p(x, z)}{p(x)} = \frac{p_k \cdot N(x|\mu_k, \Sigma_k)}{\sum_{k=1}^K p_k \cdot N(x|\mu_k, \Sigma_k)}$

3.3 高斯混合模型参数估计的 EM 算法:

初始化响应度矩阵 γ , 协方差矩阵, 均值和 α

E 步: 定义响应度矩阵 γ , 其中 γ_{jk} 表示第 j 个样本属于第 k 类的概率, 计算式如下

$$\gamma_{jk} = \frac{\alpha_k \varphi(y_j | \theta_k)}{\sum_{k=1}^K \alpha_k \varphi(y_j | \theta_k)}, j = 1, 2, \dots, N; k = 1, 2, \dots, K$$

M 步：将响应度矩阵视为定值，更新均值，协方差矩阵和 α

$$\mu_k = \frac{\sum_{j=1}^N \gamma_{jk} y_j}{\sum_{j=1}^N \gamma_{jk}}, k = 1, 2, \dots, K$$

$$\Sigma_k = \frac{\sum_{j=1}^N \gamma_{jk} (y_j - \mu_k)(y_j - \mu_k)^T}{\sum_{j=1}^N \gamma_{jk}}, k = 1, 2, \dots, K$$

重复：EM 步-直至收敛。

具体推到过程如下：

E-step

$$\begin{aligned} Q(\theta, \theta^{(t)}) &= \int_Z \log P(X, Z | \theta) \cdot P(Z | X, \theta^{(t)}) dZ \\ \text{E-step} \Rightarrow &= \sum_{i=1}^N \sum_{z_i} \log \left[\underbrace{p_{z_i} \cdot N(x_i | \mu_{z_i}, \Sigma_{z_i})}_{\theta} \right] \cdot \underbrace{\frac{p_{z_i} \cdot N(x_i | \mu_{z_i}, \Sigma_{z_i})}{\sum_{k=1}^K p_k \cdot N(x_i | \mu_k, \Sigma_k)}}_{\theta^{(t)}} \\ &= \sum_{i=1}^N \sum_{z_i} \log [p_{z_i} \cdot N(x_i | \mu_{z_i}, \Sigma_{z_i})] \cdot P(z_i | x_i, \theta^{(t)}) \\ &= \sum_{z_i} \sum_{i=1}^N \log p_{z_i} \cdot N(x_i | \mu_{z_i}, \Sigma_{z_i}) \cdot P(z_i | x_i, \theta^{(t)}) \\ &= \sum_{k=1}^K \sum_{i=1}^N \log [p_k \cdot N(x_i | \mu_k, \Sigma_k)] P(z_i = G_k | x_i, \theta^{(t)}) \\ &= \sum_{k=1}^K \sum_{i=1}^N [\log p_k + \log N(x_i | \mu_k, \Sigma_k)] P(z_i = G_k | x_i, \theta^{(t)}) \end{aligned}$$

M-step:

$$\begin{aligned} \theta^{(t+1)} &= \arg \max_{\theta} Q(\theta, \theta^{(t)}) \\ \text{求 } p_k^{(t+1)} & \\ p_k^{(t+1)} &= \arg \max_{p_k} \sum_{k=1}^K \sum_{i=1}^N \log p_k \cdot P(z_i = G_k | x_i, \theta^{(t)}), \text{ st } \sum_{k=1}^K p_k = 1 \end{aligned}$$

即每个高斯成分的混合系数由样本属于该成分的平均后验概率确定。

4 算法实现:

4.1 k-means 聚类实现:

算法步骤:

1. (随机) 选择 k 个聚类的初始中心;
2. 对任意一个样本点, 求其到 k 个聚类中心的距离, 将样本点归类到距离最小的中心的聚类, 如此迭代 n 次;
3. 每次迭代过程中, 利用均值等方法更新各个聚类的中心点(质心);
4. 对 k 个聚类中心, 利用 2,3 步迭代更新后, 如果位置点变化很小(可以设置阈值), 则认为达到稳定状态, 迭代结束, 对不同的聚类块和聚类中心可选择不同的颜色标注。

优点:

- 1) 原理比较简单, 实现也是很容易, 收敛速度快。
- 2) 聚类效果较优。
- 3) 算法的可解释度比较强。
- 4) 主要需要调参的参数仅仅是簇数 k 。

缺点:

- 1) k 值的选取不好把握
- 2) 对于不是凸的数据集比较难收敛
- 3) 如果各隐含类别的数据不平衡, 比如各隐含类别的数据量严重失衡, 或者各隐含类别的方差不同, 则聚类效果不佳。
- 4) 最终结果和初始点的选择有关, 容易陷入局部最优。
- 5) 对噪音和异常点比较敏感。

有关 k-means 的优化这里就不具体阐述了, 下面介绍这次实验使用的两种方法。

4.1.1 随机选取样本作为初始均值向量

从样本 D 中随机选择 k 个样本作为初始化的假设均值向量 $\{\mu_1, \mu_2, \dots, \mu_k\}$
重复迭代直至收敛:

1. 初始化 $C_i = \emptyset, i = 1, 2, \dots, k$
2. 对 $\mathbf{x}_j, j = 1, 2, \dots, m$ 标记为 λ_j , 使得 $\lambda_j = \arg \min_i \|\mathbf{x}_j - \mu_i\|$, 即使得每个 \mathbf{x}_j 都是属于距离其最近的均值向量所在的簇
3. 将样本 \mathbf{x}_j 划分到相应的簇 $C_{\lambda_j} = C_{\lambda_j} \cup \{\mathbf{x}_j\}$
4. 重新计算每个簇的均值向量 $\hat{\mu}_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}$
5. 如果对于所有的 $i \in 1, 2, \dots, k$, 均有 $\hat{\mu}_i = \mu_i$, 则终止迭代; 否则将重新赋值 $\mu_i = \hat{\mu}_i$ 进行迭代

这样产生的聚类结果会严重依赖于初始化的中心, 所以如果初始化的中心不算很好的时候, 可能会产生局部最优解。

4.1.2 利用最大化初始均值向量之间距离方式进行选择

因此用如下算法进行优化：

- 首先随机选择一个样本作为均值向量
- 进行迭代，直到选择到 k 个均值向量：
 - 假设当前已经选择到 i 个均值向量 $\{\mu_1, \mu_2, \dots, \mu_i\}$ ，则在 D
 $\{\mu_1, \mu_2, \dots, \mu_i\}$ 选择距离已选出的 i 个均值向量距离最远的样本
 - 将其加入初始均值向量，得到 $\{\mu_1, \mu_2, \dots, \mu_i, \mu_{i+1}\}$ 通过这种初始化均值向量的方式，能够有效降低初始簇中心的“集中程度”，在一定程度上避免结果陷入局部最优解。

4.2 GMM 算法实现

GMM 因为隐变量的存在，无法求出解析解，所以 EM 算法进行迭代优化求解是一种十分“优美”的解决方式，其中每次迭代---先根据当前参数计算每个样本数据每个高斯分布的后验概率（E-step），之后进行参数的更新（M-step）。

给定样本集 D 和高斯混合成分数 k ：

- 1：随机初始化参数
- 2：迭代到（迭代次数或者参数值不变化）
E/M
- 3：加入相对应簇

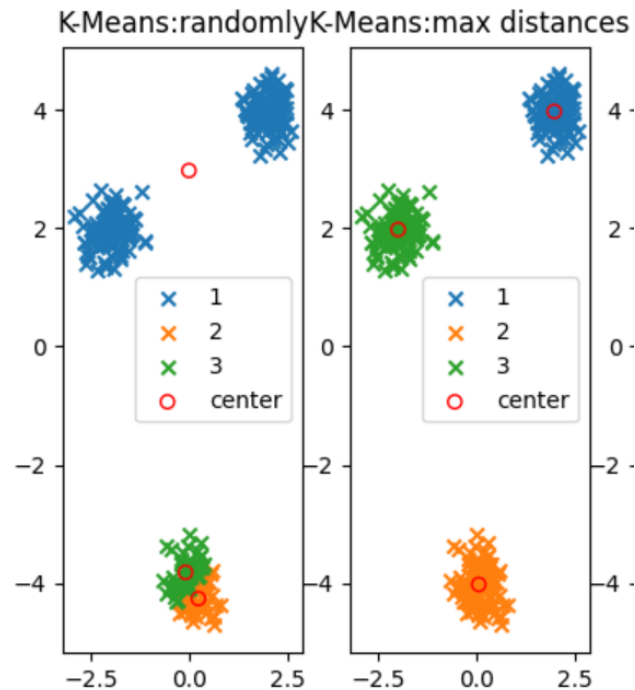
5：实验结果分析：

5.1：按照生成的数据测试

在生成数据的时候，我们使用的是二维空间的数据，便于数据可视化；利用高斯分布，按照给定的均值生成相对应的数据。

5.1.1 在上述 K-means 用两种方法设置初始值对比

因为上面已经阐述了相关的情况，如果初始值随机选取，会产生局部最优解的情况，使得聚类的效果变得很差-----解决方法便是重新运行即可。



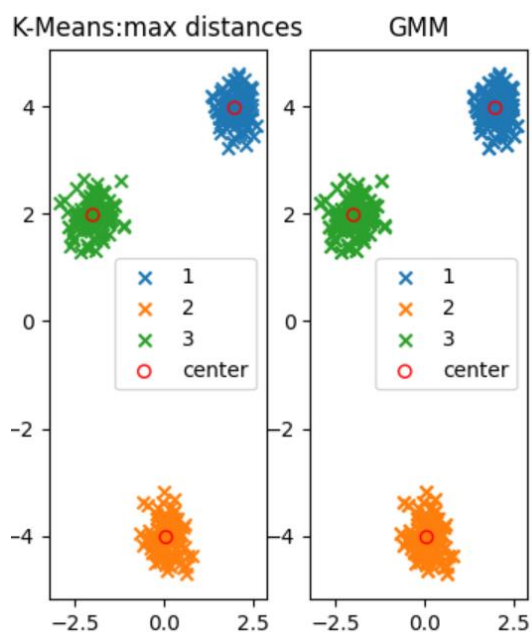
左面的图便是随机选取的，可以观察到由于初始中心的问题，产生了局部最优解的情况，聚类的结果与实际相差甚远。

作为对比的右图，便是采用了选择最远的 k 个初始化中心，将产生的点有效的分为实际的三种聚类。

解决方法：优化方法可以二分-k-means 即可，亦或者重新运行随机取初始中心簇。

5.1.2 K-means AND GMM

采用相同的二维数据，两种方法对比结果如下：



如上图所示：两种方法在生成数据的表现类似，可以实现想要的结果。

EM 的参数如下:

```
cov: [[ [ 1.30173831 -0.00645367]
        [-0.00645367  0.73356964]]

       [[ 0.84010918  0.30144325]
        [ 0.30144325  1.41324357]]

       [[ 1.02343547 -0.20477765]
        [-0.20477765  1.02313077]]

       [[ 0.79549756  0.33641743]
        [ 0.33641743  1.10474447]]]
```

与库函数的聚类进行对比, 差距不大。

5.2 UCI 数据测试

这里使用的 UCI 数据是 Github 上学长的数据, 有关鸢尾花, 其中包括 (花萼长度、花萼宽度、花瓣长度、花瓣宽度) 来预测属于哪一种聚类 (共三种)。

将测试产生的结果与原数据 label 进行对比, 得出正确率:

```
0.7888888812888
0.81666666666666
```

, 上面为 k-means, 下面为 GMM

除此之外, 在 GMM 算法中会产生迭代变化, GMM 初始参数的初始化与 k-means 类似, 选取 k 个距离最远的均值, 协方差初始化为 $N \times N$ 的对角阵, 对角元素为 0.1, 可以观察到似然值随着迭代次数的增多而变大。

6 结论分析

1: GMM-EM 以及 k-means 算法可以达到很高的准确度, 与库函数的聚类算法非常接近。

2: 有关 k-means 的随机取初始簇中心, 会产生局部最优解, 我们可以通过一些求均值或者使用最大化初始均值向量改良寻找初始点来尽量减少这种影响; 除此之外, GMM-EM 对初始值也比较敏感, 某些值的时候也会使得聚类结果比较奇怪, 我们可以用 k-means 的结果作为初值。

3: K-means 与 GMM 进行对比:

我们其实可以吧 K-means 看作一种比较特殊的 GMM，假设每种聚类在样本中出现的概率都一样 $=1/K$ （高斯模型中的每一个变量是独立的，and 变量间的协方差矩阵是对角阵），因此欧氏距离作为 K-means 的协方差去衡量相似性；

K-means 对其中系数做了简化，对于样本来说，每一个样本都只属于一个类，if 属于，则系数为 1，其余都为 0.

对于 GMM，每个类的数据出现在样本中的概率为 a ，用协方差代替上述的欧氏空间去度量相似度，系数也由 0.1 变成了所谓的全概率公式计算。

GMM 没有增加 k-means 那么多的假设，分类比 k-means 优秀，但是容易被噪声影响，作为优化算法更好。

7 参考文献:

统计学习方法-李航

西瓜书-周志华

8 源代码:

```
'''读取数据'''
import numpy as np
import pandas as pd
import itertools

class IrisProcessing(object):
    def __init__(self):
        self.data_set = pd.read_csv("./iris.csv")
        # self.data_set['class'] = self.data_set['class'].map(
        #     {'Iris-setosa': 1, 'Iris-versicolor': 2, 'Iris-
virginica': 3}).astype(int)
        self.x = self.data_set.drop('class', axis=1)
        self.y = self.data_set['class']
        self.classes = list(itertools.permutations(['Iris-setosa',
'Iris-versicolor', 'Iris-virginica'], 3))

    def get_data(self):
        return np.array(self.x, dtype=float)

    def acc(self, y_label):
        """ 用于测试聚类的正确率 """
        number = len(self.y)
        counts = []
```

```
for i in range(len(self.classes)):
    count = 0
    for j in range(number):
        if self.y[j] == self.classes[i][y_label[j]]:
            count += 1
    counts.append(count)
return np.max(counts) * 1.0 / number
```

'''作为主函数调用'''

```
import numpy as np
from matplotlib import pyplot as plt

import k_means
import GMM
import read

def watermelon_data():
    watermelon = np.array([[0.697, 0.46],
                           [0.774, 0.376],
                           [0.634, 0.264],
                           [0.608, 0.318],
                           [0.556, 0.215],
                           [0.403, 0.237],
                           [0.481, 0.149],
                           [0.437, 0.211],
                           [0.666, 0.091],
                           [0.243, 0.267],
                           [0.245, 0.057],
                           [0.343, 0.099],
                           [0.639, 0.161],
                           [0.657, 0.198],
                           [0.36, 0.37],
                           [0.593, 0.042],
                           [0.719, 0.103],
                           [0.359, 0.188],
                           [0.339, 0.241],
                           [0.282, 0.257],
                           [0.748, 0.232],
                           [0.714, 0.346],
                           [0.483, 0.312],
                           [0.478, 0.437],
                           [0.525, 0.369],
```

```

        [0.751, 0.489],
        [0.532, 0.472],
        [0.473, 0.376],
        [0.725, 0.445],
        [0.446, 0.459]])

    return watermelon

def generate_data(sample_means, sample_number, number_k):
    """ 生成 2 维数据
    :argument number_k k 类
    :argument sample_means k 类数据的均值 以 list 的形式给出 如[[1, 2],[-1,
-2], [0, 0]]
    :argument sample_number k 类数据的数量 以 list 的形式给出 如[10, 20, 30]
    """
    assert number_k > 0
    assert len(sample_means) == number_k
    assert len(sample_number) == number_k
    cov = [[0.1, 0], [0, 0.1]]
    sample_data = []
    for index in range(number_k):
        for times in range(sample_number[index]):
            sample_data.append(np.random.multivariate_normal(
                [sample_means[index][0], sample_means[index][1]],
                cov).tolist())
    return np.array(sample_data)

k = 3
means = [[2, 4], [0, -4], [-2, 2]]
number = [100, 100, 100]
data = generate_data(means, number, k)

km = k_means.KMeans(data, k)
mu_random, c_random = km.k_means_random_center()
mu_normal, c_normal = km.k_means_not_random_center()
# watermelon = watermelon_data()
# mu, c = k_means(watermelon, k)
plt.subplot(131)
plt.title("K-Means:randomly")
for i in range(k):
    plt.scatter(np.array(c_random[i])[:, 0], np.array(c_random[i])[:,
1], marker="x", label=str(i + 1))
plt.scatter(mu_random[:, 0], mu_random[:, 1], facecolor="none",

```

```

edgecolor="r", label="center")
plt.legend()

plt.subplot(132)
plt.title("K-Means:max distances")
for i in range(k):
    plt.scatter(np.array(c_normal[i])[:, 0], np.array(c_normal[i])[:, 1],
1], marker="x", label=str(i + 1))
plt.scatter(mu_normal[:, 0], mu_normal[:, 1], facecolor="none",
edgecolor="r", label="center")
plt.legend()

gmm = GMM.GaussianMixtureModel(data, k=k)
mu_gmm, c_gmm = gmm.predict()

plt.subplot(133)
plt.title("GMM")
for i in range(k):
    plt.scatter(np.array(c_gmm[i])[:, 0], np.array(c_gmm[i])[:, 1],
marker="x", label=str(i + 1))
plt.scatter(mu_gmm[:, 0], mu_gmm[:, 1], facecolor="none",
edgecolor="r", label="center")
plt.legend()

plt.show()

iris = read.IrisProcessing()
iris_data = iris.get_data()
0.7888888812888
0.81666666666666
gmm_iris = GMM.GaussianMixtureModel(iris_data, k)
mu_iris, c_iris = gmm_iris.predict()
print(mu_iris)

km_iris = k_means.KMeans(iris_data, 3)
km_mu_iris, km_c_iris = km_iris.k_means_not_random_center()
print(km_mu_iris)
print(iris.acc(gmm_iris.sample_assignments))
print(iris.acc(km_iris.sample_assignments))
# TODO 对iris数据集, 使用GMM得到的正确率低于使用k-means正确率

```

```

'''k-means'''
import numpy as np
import collections

```



```

        loss = np.sum(self.__euclidean_distance(
            self.__mu[i], new_mu[i]) for i in range(self.k))
        if loss > self.delta:
            self.__mu = new_mu
        else:
            break

        print("K-means", times)
        times = times + 1
        print(self.__mu)
    return self.__mu, c

""" 随机选择 k 个顶点作为初始簇中心点 """
def k_means_random_center(self):

    self.__mu = self.data[random.sample(range(self.__data_rows),
self.k)]
    return self.__k_means()

""" 随机选择第一个簇中心点 再选择彼此距离最大的 k 个顶点作为初始簇中心点 """
def k_means_not_random_center(self):

    self.__mu = self.__initial_center_not_random()
    return self.__k_means()

```

```

'''GMM 的 EM 实现'''
import numpy as np
import random
from scipy.stats import multivariate_normal
import collections

class GaussianMixtureModel(object):
    """ 高斯混合聚类 EM 算法 """

    def __init__(self, data, k=3, delta=1e-12, max_iteration=1000):
        self.data = data
        self.k = k
        self.delta = delta
        self.max_iteration = max_iteration
        self.data_rows, self.data_columns = self.data.shape
        self.__alpha = np.ones(self.k) * (1.0 / self.k)
        self.__mu, self.__sigma = self.__init_params()

```

```

        self.sample_assignments = None
        self.c = collections.defaultdict(list)
        self.__last_alpha = self.__alpha
        self.__last_mu = self.__mu
        self.__last_sigma = self.__sigma
        self.__gamma = None

    @staticmethod
    def __euclidean_distance(x1, x2):
        return np.linalg.norm(x1 - x2)

    def __initial_center_not_random(self):
        """ 选择彼此距离尽可能远的 k 个点 """
        # 随机选第 1 个初始点
        mu_0 = np.random.randint(0, self.k) + 1
        mu = [self.data[mu_0]]
        # 依次选择与当前 mu 中样本点距离最大的点作为初始簇中心点
        for times in range(self.k-1):
            temp_ans = []
            for i in range(self.data_rows):
                temp_ans.append(np.sum([self.__euclidean_distance(
                    self.data[i], mu[j]) for j in range(len(mu))]))
            mu.append(self.data[np.argmax(temp_ans)])
        return np.array(mu)

    def __init_params(self):
        # mu = np.array(self.data[random.sample(range(self.data_rows),
self.k)])
        # 随机选择 k 个点作为初始点 极易陷入局部最小值
        mu = self.__initial_center_not_random()
        sigma = collections.defaultdict(list)
        for i in range(self.k):
            sigma[i] = np.eye(self.data_columns, dtype=float) * 0.1
        return mu, sigma

    # def __gaussian(self, mean, cov):
    #     det = np.linalg.det(cov)
    #     cov_i = np.linalg.pinv(cov)
    #     temp_x = np.math.pow(2 * np.pi, 0.5 * self.data_columns) *
np.math.pow(det, 0.5)
    #     temp_y = np.exp(-0.5 * (self.data[5] -
mean).T.dot(cov_i).dot(self.data[5] - mean))
    #     return 1.0 * temp_y / temp_x

```



```

def __likelihoods(self):
    likelihoods = np.zeros((self.data_rows, self.k))
    for i in range(self.k):
        likelihoods[:, i] = multivariate_normal.pdf(self.data,
self.__mu[i], self.__sigma[i])
    return likelihoods

def __expectation(self):
    # 求期望 E
    weighted_likelihoods = self.__likelihoods() * self.__alpha #
(m, k)
    sum_likelihoods = np.expand_dims(np.sum(weighted_likelihoods,
axis=1), axis=1) # (m, 1)
    print(np.log(np.prod(sum_likelihoods))) # 输出似然值
    self.__gamma = weighted_likelihoods / sum_likelihoods #
(m, k)
    self.sample_assignments = self.__gamma.argmax(axis=1) # (m,)
    for i in range(self.data_rows):
self.c[self.sample_assignments[i]].append(self.data[i].tolist())

def __maximization(self):
    # 最大化 M
    for i in range(self.k):
        gamma = np.expand_dims(self.__gamma[:, i], axis=1) # 提取
每一列 作为列向量 (m, 1)
        mean = (gamma * self.data).sum(axis=0) / gamma.sum()
        covariance = (self.data - mean).T.dot((self.data - mean) *
gamma) / gamma.sum()
        self.__mu[i], self.__sigma[i] = mean, covariance # 更新参
数
    self.__alpha = self.__gamma.sum(axis=0) / self.data_rows

def __converged(self):
    # 迭代终止条件 参数 sigma mu 和 alpha 几乎不变化
    diff = np.linalg.norm(self.__last_alpha - self.__alpha) \
        + np.linalg.norm(self.__last_mu - self.__mu) \
        + np.sum([np.linalg.norm(self.__last_sigma[i] -
self.__sigma[i]) for i in range(self.k)])
    if diff > self.delta:
        self.__last_sigma = self.__sigma
        self.__last_mu = self.__mu
        self.__last_alpha = self.__alpha
    return False

```

```
        else:
            return True

def predict(self):
    print("GMM")
    for i in range(self.max_iteration):
        print(i)
        self.__expectation()
        self.__maximization()
        if self.__converged():
            break
    self.__expectation()
    return self.__mu, self.c
```