

**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](mailto:1579974122@qq.com) |
| 手机号码 | 15122925619 |

**目录**

[1 实验目的 2](#_Toc55749423)

[2 实验要求、环境 3](#_Toc55749424)

[2.1 实验要求 3](#_Toc55749425)

[2.2 实验环境 3](#_Toc55749426)

[3 算法原理 3](#_Toc55749427)

[3.1 k-means聚类 3](#_Toc55749428)

[3.2 GMM： 4](#_Toc55749429)

[3.3 高斯混合模型参数估计的EM算法： 4](#_Toc55749430)

[4 算法实现： 6](#_Toc55749431)

[4.1 k-means聚类实现： 6](#_Toc55749432)

[4.1.1 随机选取样本作为初始均值向量 6](#_Toc55749433)

[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](#_Toc55749434)

[4.2 GMM算法实现 7](#_Toc55749435)

[5 ：实验结果分析： 7](#_Toc55749436)

[5.1 ：按照生成的数据测试 7](#_Toc55749437)

# 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

# 实验要求、环境

## 实验要求

实现一个k-means算法和混合高斯模型，并且用EM算法估计模型中的蚕食。

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。

## 实验环境

Win10，x86-64

Pycharm 2019.1

python 3.7.1

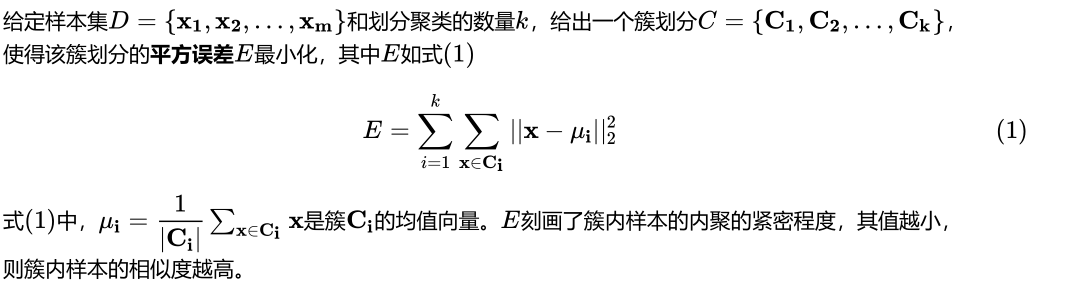
使用python库：numpy（计算），sklearn(用于聚类验证), matplotlib.pyplot(用于可视化)

# 算法原理

## k-means聚类

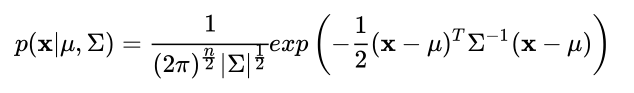
K-Means算法的思想很简单，对于给定的样本集，按照样本之间的距离大小，将样本集划分为K个簇。让簇内的点尽量紧密的连在一起，而让簇间的距离尽量的大。

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“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。



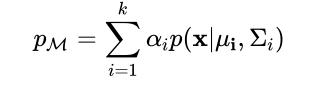
## GMM：

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：



其中 为n维的均值向量， 是N\*N 的协方差阵。

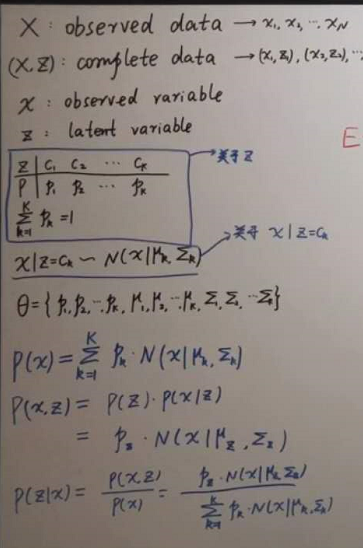
之后可以引出：



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->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。

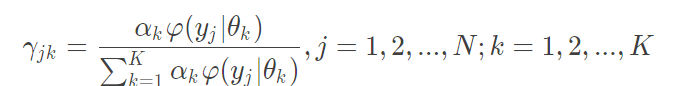
手写推导如下：



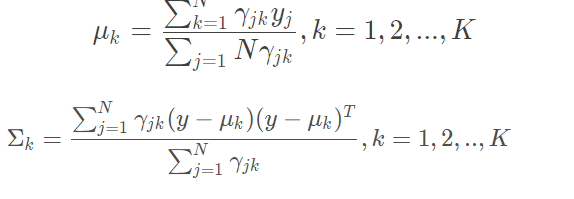
## 高斯混合模型参数估计的EM算法：

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



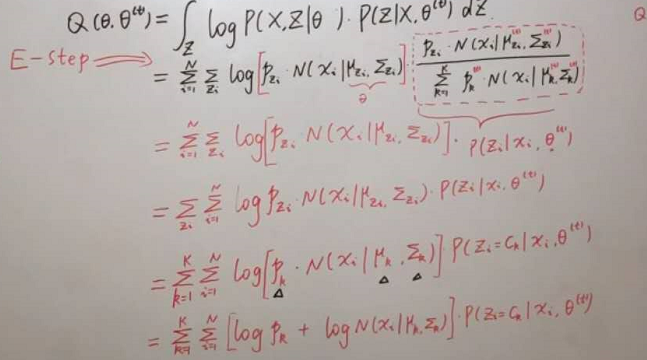
M步：将响应度矩阵视为定值，更新均值，协方差矩阵和 α



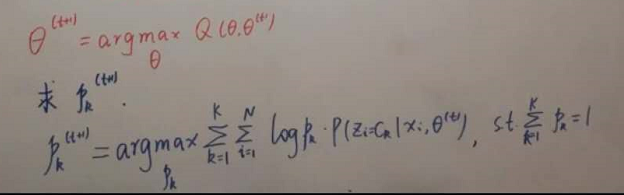
重复：EM步-直至收敛。

具体推到过程如下：

E-step



M-step：



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。

# 算法实现：

## 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的敏感。

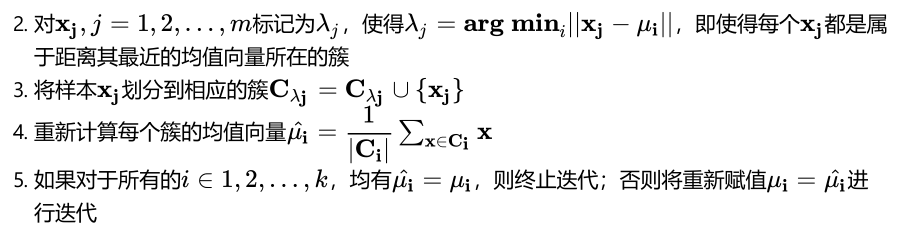
有关k-means的优化这里就不具体阐述了，下面介绍这次实验使用的两种方法。

### 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

重复迭代直至收敛：

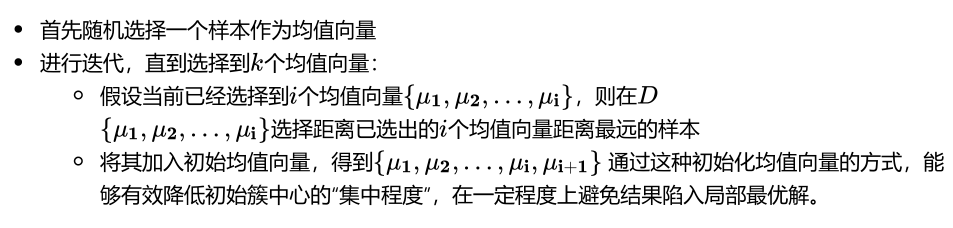




这样产生的聚类结果会严重依赖于初始化的中心，所以如果初始化的中心不算很好的时候，可能会产生局部最优解。

### 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：



## 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

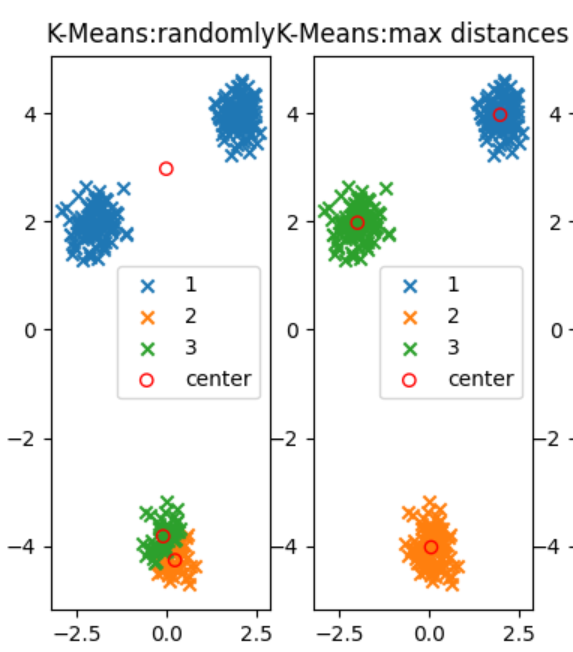
# ：实验结果分析：

## ：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。

### 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。



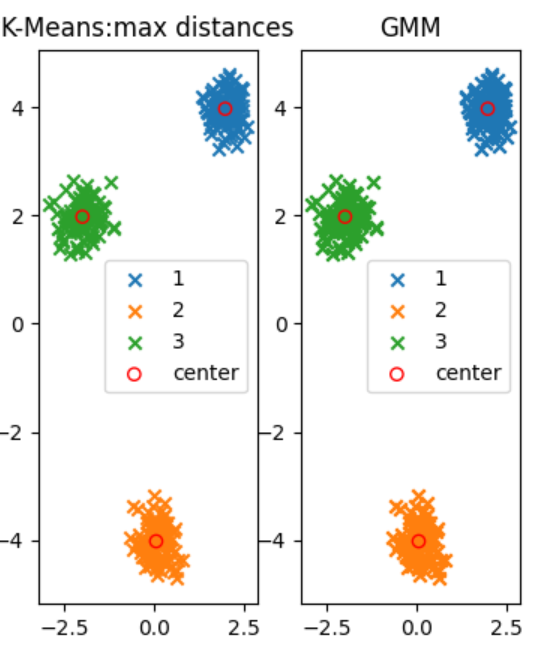
左面的图便是随机选取的，可以观察到由于初始中心的问题，产生了局部最优解的情况，聚类的结果与实际相差甚远。

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。

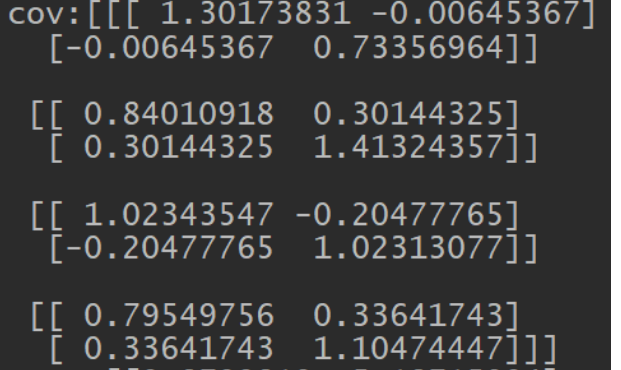
### K-means AND GMM

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：



与库函数的聚类进行对比，差距不大。

## 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：

，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\*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。

# 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那麽多的假设，分类比k-means优秀，但是容易被噪声影响，作为优化算法更好。

# 参考文献：

统计学习方法-李航

西瓜书-周志华

# 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], 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  
0.8166666666666  
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  
import random  
  
''':k-means类'''  
class KMeans(object):  
 def \_\_init\_\_(self, data, k, delta=1e-6):  
 self.data = data  
 self.k = k  
 self.delta = delta  
 self.\_\_data\_rows, self.\_\_data\_columns = data.shape  
 self.\_\_mu = self.\_\_initial\_center\_not\_random()  
 self.sample\_assignments = [-1] \* self.\_\_data\_rows  
  
 # 计算范数，默认为2范数  
 @staticmethod  
 def \_\_euclidean\_distance(x1, x2):  
 return np.linalg.norm(x1 - x2)  
  
 """ 选择彼此距离尽可能远的K个点 """  
 def \_\_initial\_center\_not\_random(self):  
 # 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 \_\_k\_means(self):  
 times = 0  
 while True:  
 c = collections.defaultdict(list)  
 for i in range(self.\_\_data\_rows):  
 dij = [self.\_\_euclidean\_distance(  
 self.data[i], self.\_\_mu[j]) for j in range(self.k)]  
 lambda\_j = np.argmin(dij)  
 c[lambda\_j].append(self.data[i].tolist())  
 self.sample\_assignments[i] = lambda\_j  
  
 new\_mu = np.array([np.mean(c[i], axis=0).tolist()  
 for i in range(self.k)])  
  
 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