《模式识别与机器学习A》实验报告

实验题目：实现k-means聚类方法和混合高斯模型

学号： 2021110683

姓名： 徐柯炎

# 实验报告内容

## 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。

## 实验内容

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）。

1. 用k-means聚类，测试效果；
2. 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）。
3. 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

Windows10; python3.9;PyCharm 2021.2.2

## 实验过程、结果及分析（包括代码截图、运行结果截图及必要的理论支撑等）

### Kmeans实验原理

方法 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：

1. 先在样本点中选取k个点作为暂时的聚类中心，
2. 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”，
3. 接着每一个类的期望，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算法实验原理

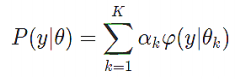
1. EM算法

EM算法是一种迭代优化策略，由于它的计算方法中每一次迭代都分两步，其中一个为期望步（E步），另一个为极大步（M步），所以算法被称为EM算法（Expectation-Maximization Algorithm）。EM算法受到缺失思想影响，最初是为了解决数据缺失情况下的参数估计问题，其算法基础和收敛有效性等问题在Dempster、Laird和Rubin三人于1977年所做的文章《Maximum likelihood from incomplete data via the EM algorithm》中给出了详细的阐述。

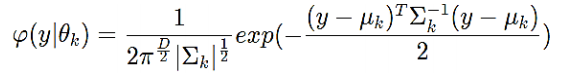
EM算法的基本思想是：

1. 首先根据己经给出的观测数据，估计出模型参数的值；
2. 然后再依据上一步估计出的参数值估计缺失数据的值，
3. 再根据估计出的缺失数据加上之前己经观测到的数据重新再对参数值进行估计，然后反复迭代，直至最后收敛，迭代结束。
4. GMM算法

混合高斯模型指具有如下形式的概率分布模型：



其中是样本中类k的数据所占比例，, 是第k类中高斯分布的概率分布函数。其中具体为

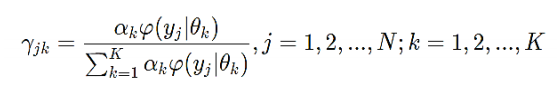


其中将记为,就是我们常说的隐变量，因为有隐变量，混合高斯 模型无法求出解析解，但是可以用 EM 算法迭代求解完成分类。

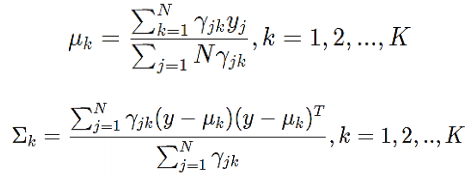
具体的 EM 算法为：

1. 初始化响应度矩阵,协方差矩阵，均值和

2. E 步：初始化响应度矩阵γ,其中 表示第个样本属于第类的概 率，如下：



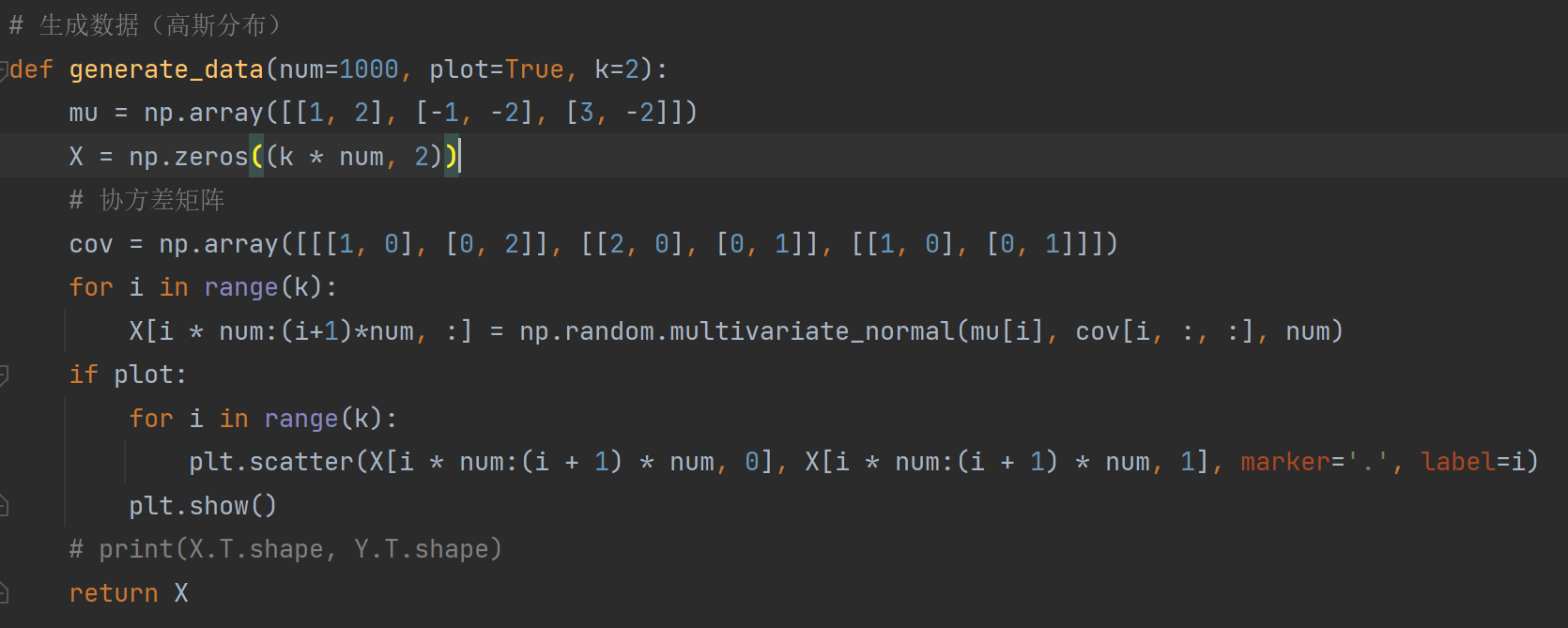
3. M 步：将响应度矩阵求解，更新均值，协方差矩阵和



4. 重复 2 ，3 步，迭代求解，直至的改变收敛。

### 实验过程

首先生成数据：

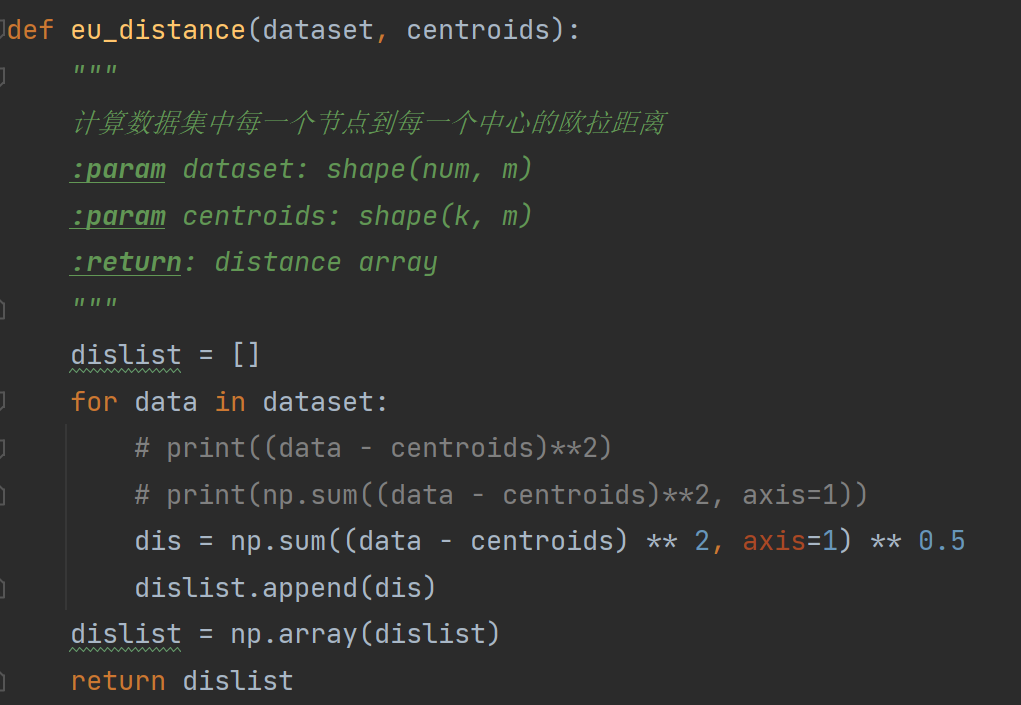


接着进行kmeans聚类。

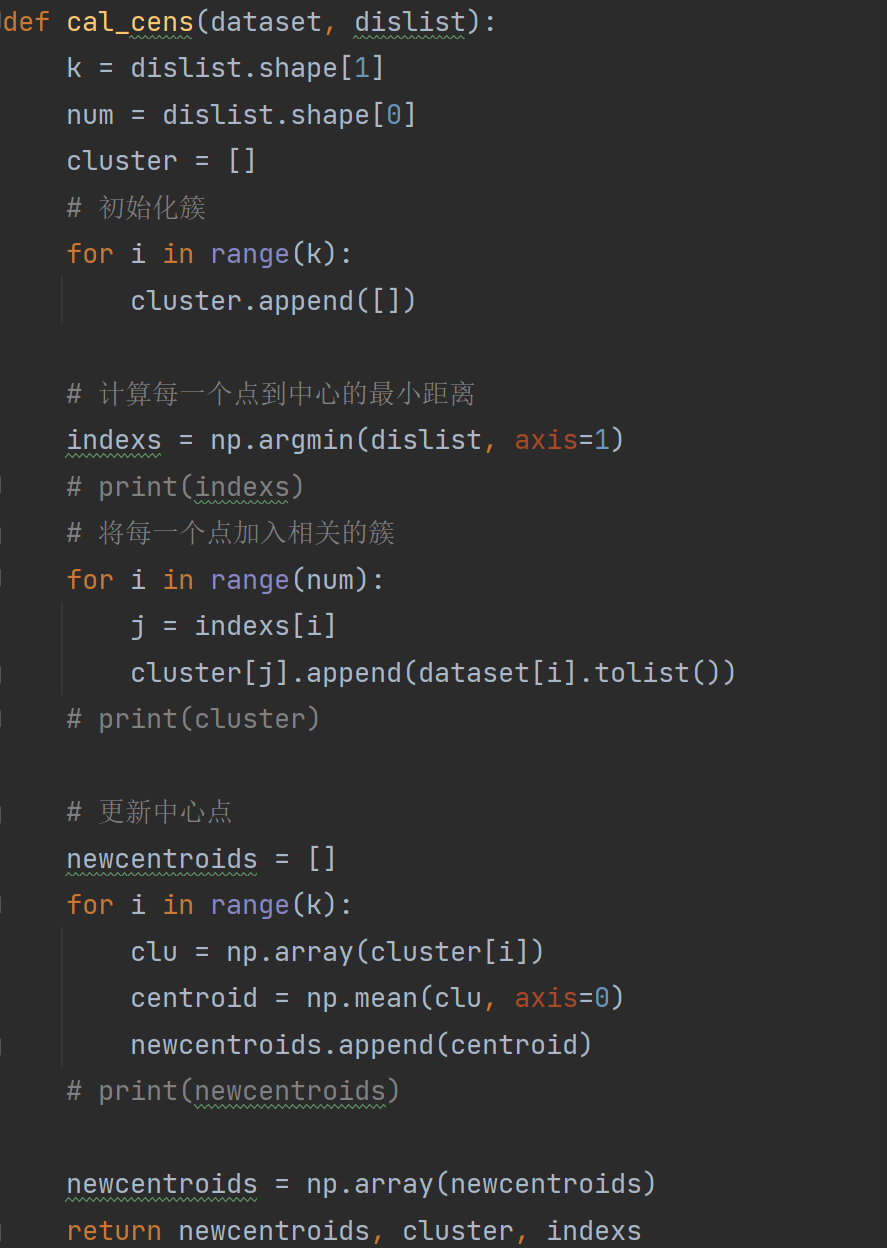
Kmeans聚类主函数代码：



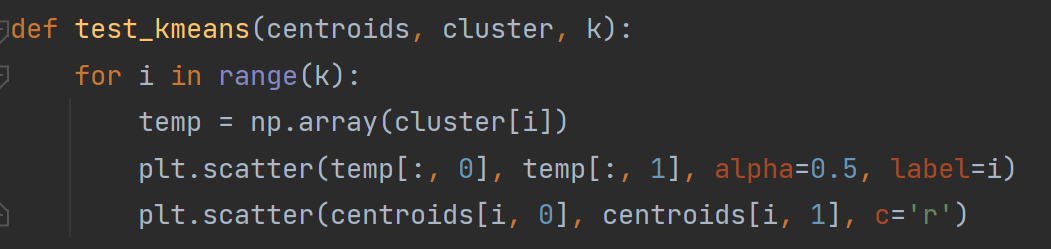
其中eu\_distance 函数用来计算欧拉距离，代码如下



接着是分簇，更新中心点：



接着通过test函数来测试kmeans

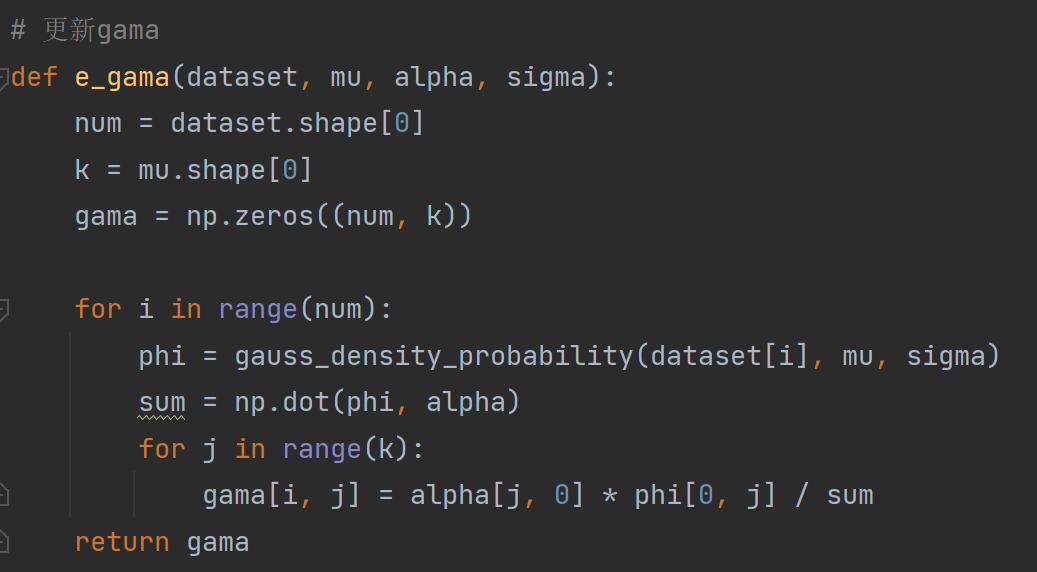


然后是高斯混合模型的EM算法：

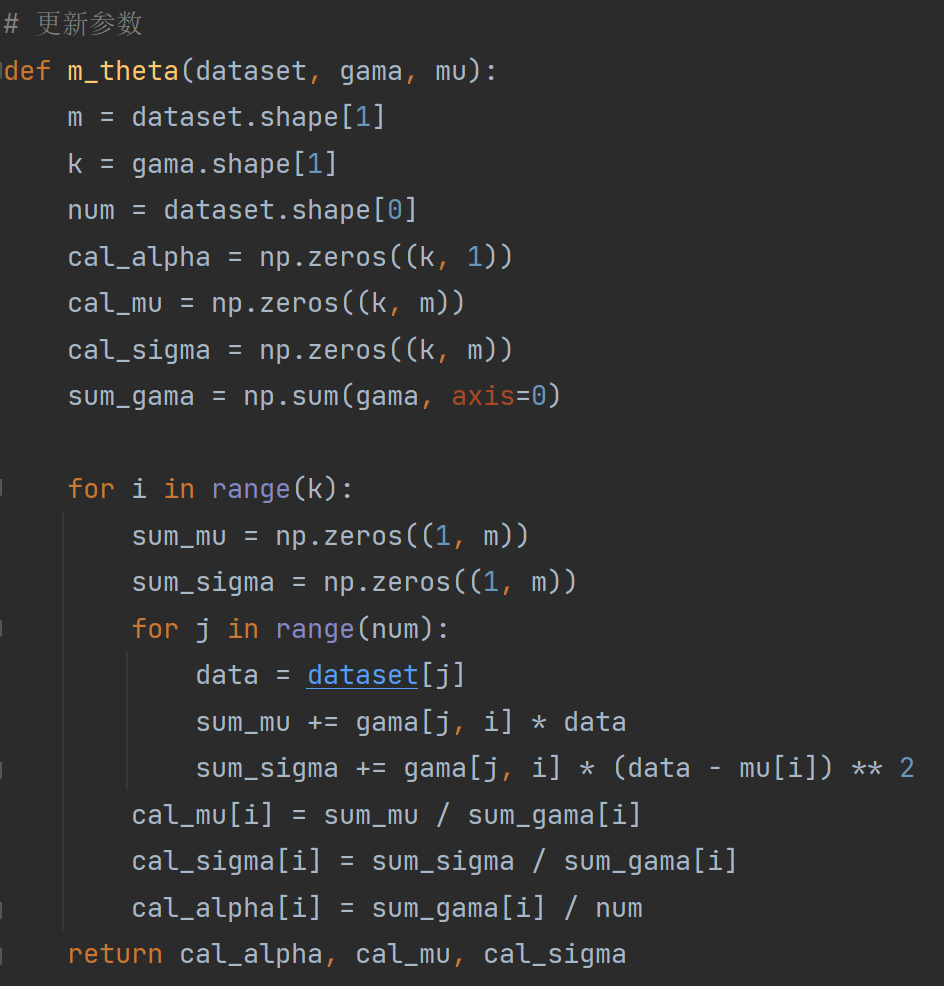
主函数如下：



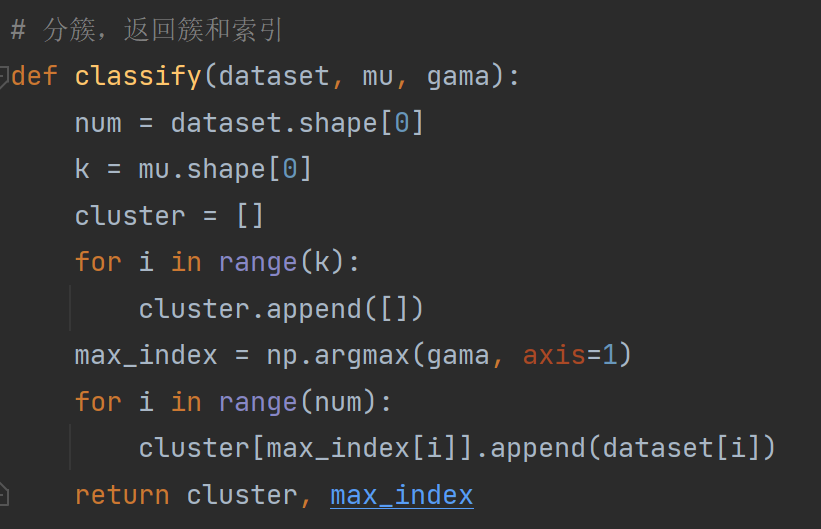
其中计算gama的函数如下：



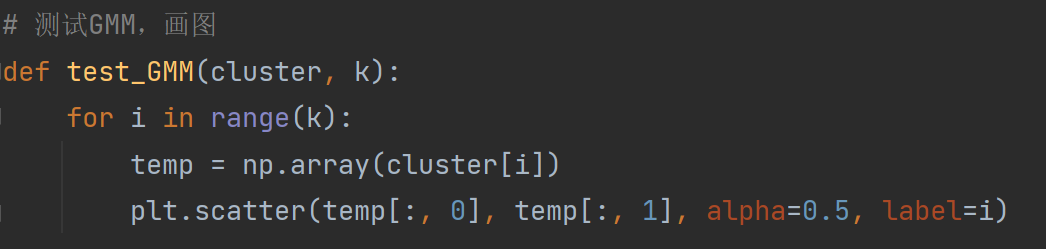
接着更新参数：



然后进行分簇：

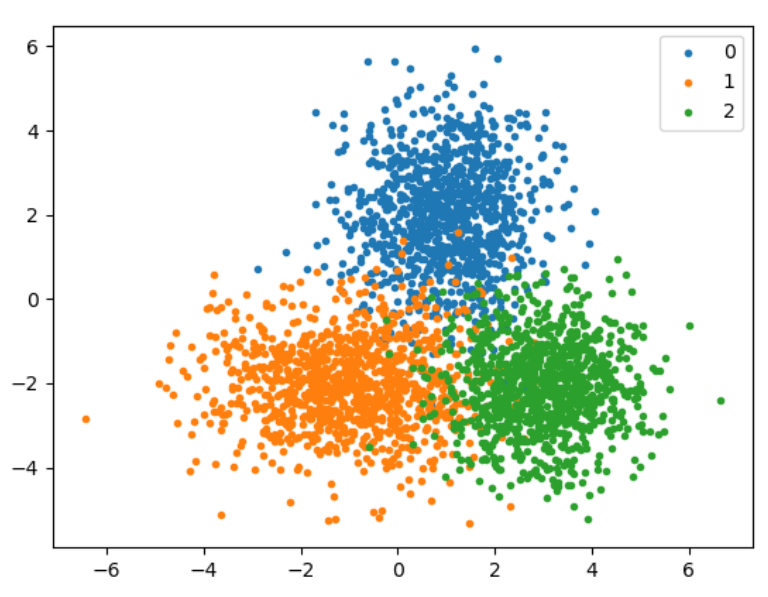


最后进行EM算法的测试：

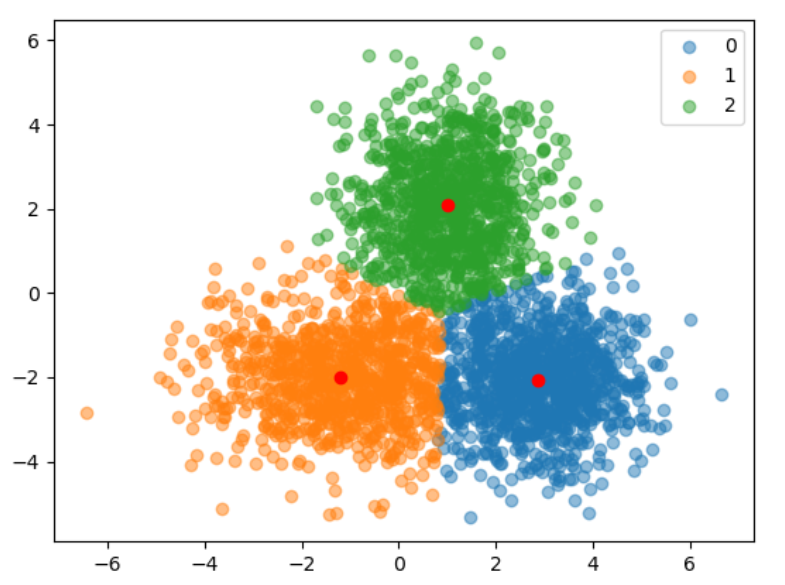


### 实验测试

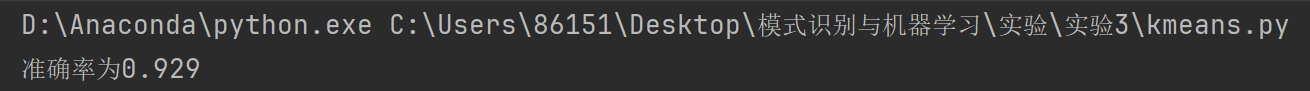
首先用高斯分布产生3个高斯分布的数据（不同均值和方差），生成的数据如下图所示：



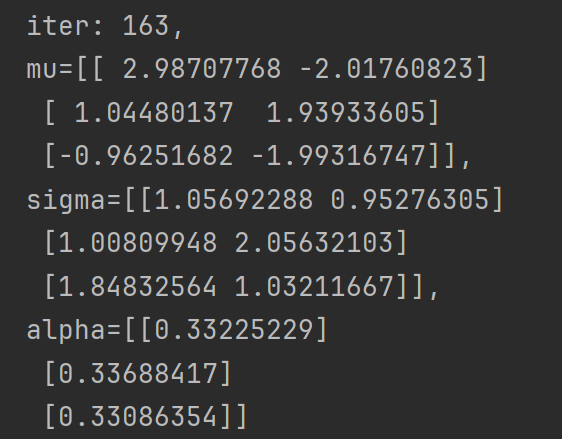
接着用kmeans算法来进行聚类，聚类结果如下图所示：



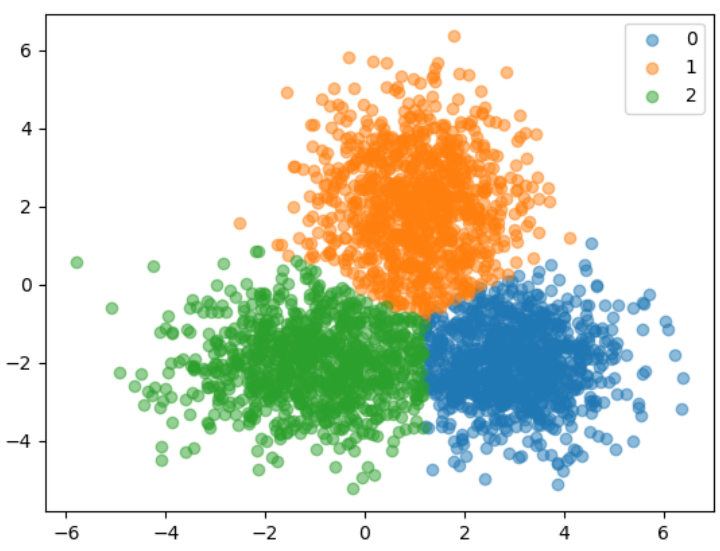
接着测试kmeans算法的效果：



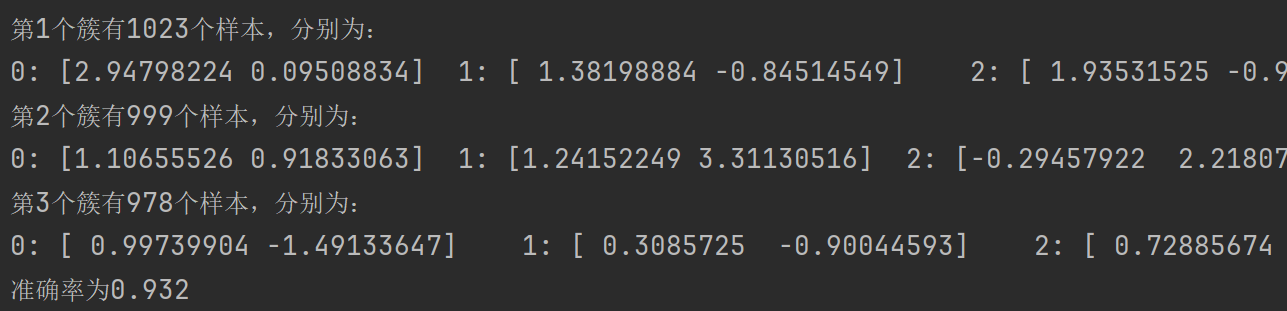
接着用混合高斯模型和你实现的EM算法估计参数，看看每次迭代后似然值变化情况，最后拟合的参数如下图所示：



聚类结果如下图所示：



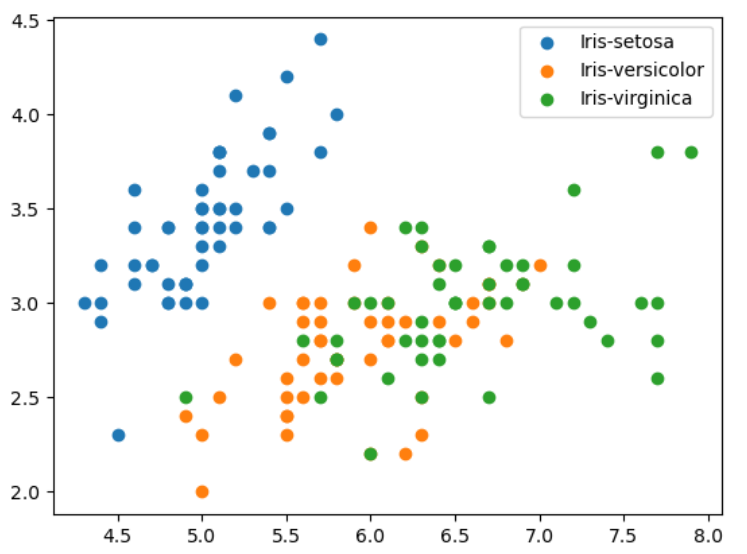
准确率如下图所示：



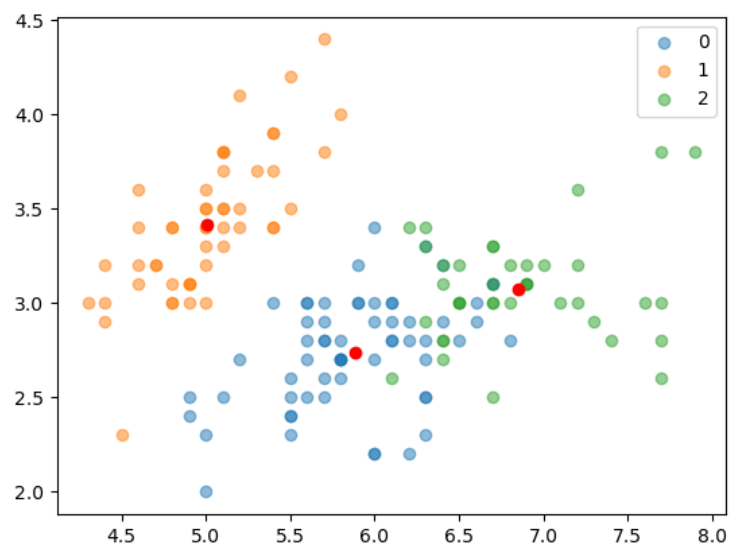
可以看出本次聚类结果混合高斯模型的EM算法要略胜于kmeans算法，但由于两种算法初始点的随机性，导致算法的收敛结果和初始点有很大关系，所以二者算法的效果相差不大。

接着是UCI上找一个简单问题数据，用kmeans和GMM进行聚类。

这里采用iris数据集，画图如下图所示：

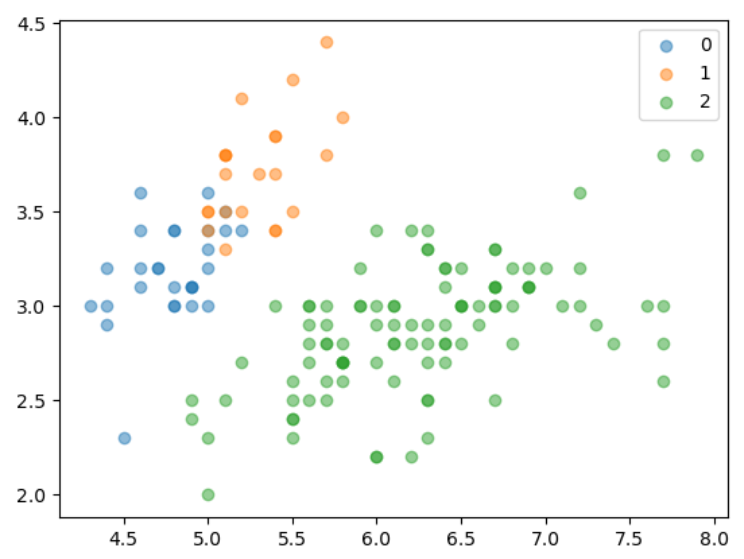


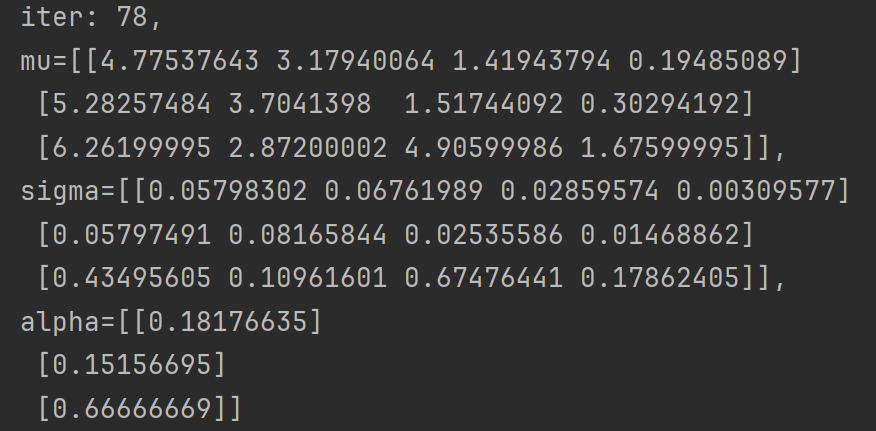
Kmeans聚类结果如下图所示：

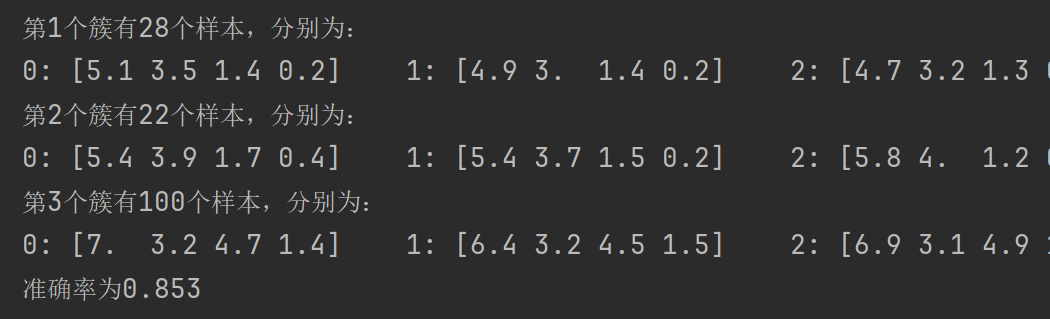




GMM算法聚类结果如下图所示：







## 实验总体结论

### Kmeans算法和GMM算法各自的优劣

K-means算法和高斯混合模型（GMM）算法都是聚类算法，但它们有不同的优劣势，适用于不同类型的数据和问题。

K-means算法：

优势：

1. 简单易懂：K-means是一种直观的算法，容易理解和实现。
2. 计算效率高：K-means通常在大型数据集上运行迅速，因为它的时间复杂度相对较低。
3. 适用于高维数据：K-means通常对高维数据集有较好的性能。

劣势：

1. 对初始中心敏感：初始聚类中心的选择会影响最终的聚类结果，可能导致陷入局部最小值。
2. 需要预先指定簇数K：需要提前知道要分成多少个簇，这对于一些问题可能不太现实。
3. 对异常值敏感：K-means容易受到异常值的干扰。

GMM算法：

优势：

1. 软聚类：GMM是一种软聚类方法，它可以为每个数据点提供属于每个簇的概率，而不是硬分配。
2. 适用于复杂分布：GMM能够建模更复杂的数据分布，因为它使用了高斯分布来描述每个簇。

劣势：

1. 复杂度高：相对于K-means，GMM的计算复杂度更高，因为它涉及参数估计，通常需要更多的计算资源。
2. 初始条件敏感：GMM对于初始条件也很敏感，不同的初始条件可能导致不同的聚类结果。
3. 需要足够的数据：GMM需要更多的数据来估计分布的参数，对于小型数据集可能不适用。

综合来说，K-means适合简单的硬聚类问题，当你有先验知识可以确定簇数时，计算效率较高。而GMM适合更复杂的数据分布和软聚类问题，但需要更多的计算资源和数据来估计参数。

## 完整实验代码

kmeans.py

1. # --\*-- coding:utf-8 --\*--
2. """
3. 作者：徐柯炎
4. 日期：2023年10月12日
5. """
6. **import** numpy as np
7. **import** random
8. **from** matplotlib **import** pyplot as plt
9. **from** ucimlrepo **import** fetch\_ucirepo
10. **from** functions **import** \*

13. **def** eu\_distance(dataset, centroids):
14. """
15. 计算数据集中每一个节点到每一个中心的欧拉距离
16. :param dataset: shape(num, m)
17. :param centroids: shape(k, m)
18. :return: distance array
19. """
20. dislist = []
21. **for** data **in** dataset:
22. # print((data - centroids)\*\*2)
23. # print(np.sum((data - centroids)\*\*2, axis=1))
24. dis = np.sum((data - centroids) \*\* 2, axis=1) \*\* 0.5
25. dislist.append(dis)
26. dislist = np.array(dislist)
27. **return** dislist

30. # 进行聚类，并计算新的中心
31. **def** cal\_cens(dataset, dislist):
32. k = dislist.shape[1]
33. num = dislist.shape[0]
34. cluster = []
35. # 初始化簇
36. **for** i **in** range(k):
37. cluster.append([])
39. # 计算每一个点到中心的最小距离
40. indexs = np.argmin(dislist, axis=1)
41. # print(indexs)
42. # 将每一个点加入相关的簇
43. **for** i **in** range(num):
44. j = indexs[i]
45. cluster[j].append(dataset[i].tolist())
46. # print(cluster)
48. # 更新中心点
49. newcentroids = []
50. **for** i **in** range(k):
51. clu = np.array(cluster[i])
52. centroid = np.mean(clu, axis=0)
53. newcentroids.append(centroid)
54. # print(newcentroids)
56. newcentroids = np.array(newcentroids)
57. **return** newcentroids, cluster, indexs

60. **def** kmeans(dataset, k):
61. """
62. kmeans 算法
63. :param dataset: 数据集 shape(num, m)
64. :param k: 中心点个数
65. :return: 中心点，聚类结果
66. """
67. num = dataset.shape[0]
68. # 随机化初始点
69. init = random.sample(range(num), k)
70. centroids = dataset[init]
71. # print(centroids)
72. flag = 1
73. cluster = []
74. indexs = np.zeros(num)
75. **while** flag:
76. dislist = eu\_distance(dataset, centroids)
77. newcentroids, cluster, indexs = cal\_cens(dataset, dislist)
78. change = newcentroids - centroids
79. # 迭代结束条件
80. **if** **not** np.any(change):
81. flag = 0
82. centroids = newcentroids
83. **return** centroids, cluster, indexs

86. **def** test\_kmeans(centroids, cluster, k):
87. **for** i **in** range(k):
88. temp = np.array(cluster[i])
89. plt.scatter(temp[:, 0], temp[:, 1], alpha=0.5, label=i)
90. plt.scatter(centroids[i, 0], centroids[i, 1], c='r')

93. **def** test\_main():
94. k = 3
95. num = 1000
96. dataset = generate\_data(num=num, k=k)
97. centroids, cluster, indexs = kmeans(dataset, k)
98. test\_kmeans(centroids, cluster, k)
99. # print(centroids)
100. # print(cluster)
101. acc = test\_acc(indexs, k)
102. **print**(f'准确率为{round(acc, 3)}')
103. plt.legend()
104. plt.show()

107. **def** test\_iris():
108. dataset = load\_iris()
109. k = 3
110. centroids, cluster, indexs = kmeans(dataset, k)
111. test\_kmeans(centroids, cluster, k)
112. acc = test\_acc(indexs, k)
113. **print**(f'准确率为{round(acc, 3)}')
114. plt.legend()
115. plt.show()

118. **if** \_\_name\_\_ == '\_\_main\_\_':
119. test\_main()
120. test\_iris()

GMM.py

1. # --\*-- coding:utf-8 --\*--
2. """
3. 作者：徐柯炎
4. 日期：2023年10月12日
5. """
6. **import** numpy as np
7. **import** random
8. **import** math
9. **from** matplotlib **import** pyplot as plt
10. **from** ucimlrepo **import** fetch\_ucirepo
11. **from** functions **import** \*

14. # 初始化参数
15. **def** init\_GMM(dataset, k):
16. num = dataset.shape[0]
17. mu = dataset[random.sample(range(num), k)]
18. var = np.var(dataset, axis=0)
19. sigma = np.tile(var, (k, 1))
20. alpha = np.ones((k, 1)) / k
21. # print(f'init mu{mu}')
22. # print(f'init sigma{sigma}')
23. # print(f'init alpha{alpha}')
24. **return** alpha, mu, sigma

27. # 定义高斯密度计算函数
28. **def** gauss\_density\_probability(data, mu, sigma):
29. """
30. 计算高斯概率密度。
31. :param data: shape(1, m)
32. :param mu: shape(k, m)
33. :param sigma: shape(k, m)
34. :return: p: shape(1, k)
35. """
36. # 高斯混合函数
37. k = mu.shape[0]
38. m = mu.shape[1]
39. p = np.zeros((1, k))
40. **for** i **in** range(k):
41. B\_det = 1
42. x\_aB = 0
43. **for** j **in** range(m):
44. B\_det \*= sigma[i, j] \* 2 \* np.pi
45. x\_aB += (data[j]-mu[i, j]) \*\* 2 / sigma[i, j]
46. part1 = math.sqrt(B\_det)
47. part2 = -0.5 \* x\_aB
48. fx\_k = 1 / part1 \* np.exp(part2)
49. p[0][i] = fx\_k
50. **return** p

53. # 更新gama
54. **def** e\_gama(dataset, mu, alpha, sigma):
55. num = dataset.shape[0]
56. k = mu.shape[0]
57. gama = np.zeros((num, k))
59. **for** i **in** range(num):
60. phi = gauss\_density\_probability(dataset[i], mu, sigma)
61. sum = np.dot(phi, alpha)
62. **for** j **in** range(k):
63. gama[i, j] = alpha[j, 0] \* phi[0, j] / sum
64. **return** gama

67. # 更新参数
68. **def** m\_theta(dataset, gama, mu):
69. m = dataset.shape[1]
70. k = gama.shape[1]
71. num = dataset.shape[0]
72. cal\_alpha = np.zeros((k, 1))
73. cal\_mu = np.zeros((k, m))
74. cal\_sigma = np.zeros((k, m))
75. sum\_gama = np.sum(gama, axis=0)
77. **for** i **in** range(k):
78. sum\_mu = np.zeros((1, m))
79. sum\_sigma = np.zeros((1, m))
80. **for** j **in** range(num):
81. data = dataset[j]
82. sum\_mu += gama[j, i] \* data
83. sum\_sigma += gama[j, i] \* (data - mu[i]) \*\* 2
84. cal\_mu[i] = sum\_mu / sum\_gama[i]
85. cal\_sigma[i] = sum\_sigma / sum\_gama[i]
86. cal\_alpha[i] = sum\_gama[i] / num
87. **return** cal\_alpha, cal\_mu, cal\_sigma

90. # 分簇，返回簇和索引
91. **def** classify(dataset, mu, gama):
92. num = dataset.shape[0]
93. k = mu.shape[0]
94. cluster = []
95. **for** i **in** range(k):
96. cluster.append([])
97. max\_index = np.argmax(gama, axis=1)
98. **for** i **in** range(num):
99. cluster[max\_index[i]].append(dataset[i])
100. **return** cluster, max\_index

103. # EM算法
104. **def** em\_algorithm(dataset, k, iteration=1000, threshold=1e-10):
105. # alpha shape(k, 1)
106. # mu shape(k, m)
107. # sigma shape(k, m)
108. # gama shape(num, k)
109. alpha, mu, sigma = init\_GMM(dataset, k)
110. num = dataset.shape[0]
111. gama = np.zeros((num, k))
113. **for** iter **in** range(iteration):
114. prev\_mu = mu
115. gama = e\_gama(dataset, mu, alpha, sigma)
116. alpha, mu, sigma = m\_theta(dataset, gama, mu)
117. **if** np.sum(abs(prev\_mu - mu)) < threshold:  # 均值基本不变，结束迭代
118. **print**(f'iter: {iter},\nmu={mu},\nsigma={sigma},\nalpha={alpha}')
119. **break**
120. **if** iter % (int(iteration/10)) == 0:
121. **print**(f'iter={iter},\nmu={mu},\nsigma={sigma},\nalpha={alpha}')
123. cluster, max\_index = classify(dataset, mu, gama)
124. **return** cluster, max\_index

127. # 测试GMM，画图
128. **def** test\_GMM(cluster, k):
129. **for** i **in** range(k):
130. temp = np.array(cluster[i])
131. plt.scatter(temp[:, 0], temp[:, 1], alpha=0.5, label=i)

134. **def** test\_main():
135. k = 3
136. dataset = generate\_data(plot=True, k=k)
137. cluster, max\_index = em\_algorithm(dataset, k)
138. **for** i **in** range(k):
139. clu = np.array(cluster[i])
140. **print**(f'第{i + 1}个簇有{clu.shape[0]}个样本，分别为：')
141. **for** j **in** range(clu.shape[0]):
142. **print**(f'{j}: {clu[j]}', end='\t')
143. **print**()
144. acc = test\_acc(max\_index, k)
145. **print**(f'准确率为{round(acc, 3)}')
146. test\_GMM(cluster, k)
147. plt.legend()
148. plt.show()

151. **def** test\_iris():
152. k = 3
153. dataset = load\_iris()
154. cluster, max\_index = em\_algorithm(dataset, k)
155. **for** i **in** range(k):
156. clu = np.array(cluster[i])
157. **print**(f'第{i+1}个簇有{clu.shape[0]}个样本，分别为：')
158. **for** j **in** range(clu.shape[0]):
159. **print**(f'{j}: {clu[j]}', end='\t')
160. **print**()
161. acc = test\_acc(max\_index, k)
162. **print**(f'准确率为{round(acc, 3)}')
163. test\_GMM(cluster, k)
164. plt.legend()
165. plt.show()

168. **if** \_\_name\_\_ == '\_\_main\_\_':
169. test\_main()
170. test\_iris()

functions.py

1. # --\*-- coding:utf-8 --\*--
2. """
3. 作者：徐柯炎
4. 日期：2023年10月13日
5. """
6. **import** numpy as np
7. **import** random
8. **from** matplotlib **import** pyplot as plt
9. **from** ucimlrepo **import** fetch\_ucirepo

12. **def** load\_iris():
13. # 从uci获取iris数据集
14. iris = fetch\_ucirepo(id=53)
16. # 数据（pd.dataframe格式）
17. X = iris.data.features
18. y = iris.data.targets
19. num = y.shape[0]
20. labels = []
21. # 数据标注
22. **for** i **in** range(num):
23. label = y.loc[i, 'class']
24. **if** label **in** labels:
25. **continue**
26. **else**:
27. labels.append(label)
28. X = np.array(X)
29. # y = np.array(y)
30. **for** i **in** range(len(labels)):
31. plt.scatter(X[50 \* i:50 \* (i + 1), 0], X[50 \* i:50 \* (i + 1), 1], label=labels[i])
32. plt.legend()
33. plt.show()
34. # print(X)
35. # print(y)
36. # print(labels)
37. **return** X

40. # 生成数据（高斯分布）
41. **def** generate\_data(num=1000, plot=True, k=2):
42. mu = np.array([[1, 2], [-1, -2], [3, -2]])
43. X = np.zeros((k \* num, 2))
44. # 协方差矩阵
45. cov = np.array([[[1, 0], [0, 2]], [[2, 0], [0, 1]], [[1, 0], [0, 1]]])
46. **for** i **in** range(k):
47. X[i \* num:(i+1)\*num, :] = np.random.multivariate\_normal(mu[i], cov[i, :, :], num)
48. **if** plot:
49. **for** i **in** range(k):
50. plt.scatter(X[i \* num:(i + 1) \* num, 0], X[i \* num:(i + 1) \* num, 1], marker='.', label=i)
51. plt.legend()
52. plt.show()
53. # print(X.T.shape, Y.T.shape)
54. **return** X

57. **def** find\_label(max\_index, left, right, k):
58. cnt = np.zeros(k)
59. **for** i **in** range(left, right):
60. **if** max\_index[i] == 0:
61. cnt[0] += 1
62. **elif** max\_index[i] == 1:
63. cnt[1] += 1
64. **elif** max\_index[i] == 2:
65. cnt[2] += 1
66. **return** np.argmax(cnt)

69. **def** test\_acc(max\_index, k):
70. num = int(len(max\_index) / k)
71. # print(num)
72. index = np.zeros(k)
73. cnt = 0
74. **for** i **in** range(k):
75. index[i] = find\_label(max\_index, i \* num, (i+1) \* num, k)
76. **for** j **in** range(num):
77. **if** index[i] == max\_index[i \* num + j]:
78. cnt += 1
79. **return** cnt / len(max\_index)

## 参考文献

[最佳聚类实践：高斯混合模型（GMM）](https://zhuanlan.zhihu.com/p/81255623)