《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

1. 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。

2. 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。

3. 实验环境

Windows10; python3.9;PyCharm 2021.2.2

4. 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）

(1) 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：

A. 先在样本点中选取 k 个点作为暂时的聚类中心，

B. 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”，

C. 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。

(2) 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：

A. 首先根据已经给出的观测数据，估计出模型参数的值；

B. 然后再依据上一步估计出的参数值估计缺失数据的值，

C. 再根据估计出的缺失数据加上之前已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。

2) GMM 算法

混合高斯模型指具有如下形式的概率分布模型：

$$P(y|\theta) = \sum_{k=1}^{K} \alpha_k \varphi(y|\theta_k)$$

其中α_k是样本中类 k 的数据所占比例，$\sum_{k=1}^{K} \alpha_k = 1$，$\varphi(y|\theta_k)$是第 k 类中高斯分布的概率分布函数。其中$\varphi(y|\theta_k)$具体为

$$\varphi(y|\theta_k) = \frac{1}{2\pi^{\frac{D}{2}} |\Sigma_k|^{\frac{1}{2}}} exp(-\frac{(y-\mu_k)^T \Sigma_k^{-1} (y-\mu_k)}{2})$$

其中将(α, Σ, μ)记为θ,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γ,协方差矩阵，均值和α

2. E 步：初始化响应度矩阵 γ ,其中γ_{jk} 表示第j 个样本属于第 k 类的概率，如下：

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

3. M 步：将响应度矩阵求解，更新均值，协方差矩阵和α

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

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

4. 重复 2 ，3 步，迭代求解，直至μ的改变收敛。

(3) 实验过程

首先生成数据：

```
# 生成数据（高斯分布）
def generate_data(num=1000, plot=True, k=2):
    mu = np.array([[1, 2], [-1, -2], [3, -2]])
    X = np.zeros((k * num, 2))
    # 协方差矩阵
    cov = np.array([[[1, 0], [0, 2]], [[2, 0], [0, 1]], [[1, 0], [0, 1]]])
    for i in range(k):
        X[i * num:(i+1)*num, :] = np.random.multivariate_normal(mu[i], cov[i, :, :], num)
    if plot:
        for i in range(k):
            plt.scatter(X[i * num:(i + 1) * num, 0], X[i * num:(i + 1) * num, 1], marker='.', label=i)
        plt.show()
    # print(X.T.shape, Y.T.shape)
    return X
```

接着进行 kmeans 聚类。
Kmeans 聚类主函数代码：

```
def kmeans(dataset, k):
    """
    kmeans 算法
    :param dataset: 数据集 shape(num, m)
    :param k: 中心点个数
    :return: 中心点，聚类结果
    """
    num = dataset.shape[0]
    # 随机化初始点
    init = random.sample(range(num), k)
    centroids = dataset[init]
    # print(centroids)
    flag = 1
    cluster = []
    indexs = np.zeros(num)
    while flag:
        dislist = eu_distance(dataset, centroids)
        newcentroids, cluster, indexs = cal_cens(dataset, dislist)
        change = newcentroids - centroids
        # 迭代结束条件
        if not np.any(change):
            flag = 0
        centroids = newcentroids
    return centroids, cluster, indexs
```

其中 eu_distance 函数用来计算欧拉距离，代码如下

```
def eu_distance(dataset, centroids):
    """
    计算数据集中每一个节点到每一个中心的欧拉距离
    :param dataset: shape(num, m)
    :param centroids: shape(k, m)
    :return: distance array
    """
    dislist = []
    for data in dataset:
        # print((data - centroids)**2)
        # print(np.sum((data - centroids)**2, axis=1))
        dis = np.sum((data - centroids) ** 2, axis=1) ** 0.5
        dislist.append(dis)
    dislist = np.array(dislist)
    return dislist
```

接着是分簇，更新中心点：

```python
def cal_cens(dataset, dislist):
    k = dislist.shape[1]
    num = dislist.shape[0]
    cluster = []
    # 初始化簇
    for i in range(k):
        cluster.append([])

    # 计算每一个点到中心的最小距离
    indexs = np.argmin(dislist, axis=1)
    # print(indexs)
    # 将每一个点加入相关的簇
    for i in range(num):
        j = indexs[i]
        cluster[j].append(dataset[i].tolist())
    # print(cluster)

    # 更新中心点
    newcentroids = []
    for i in range(k):
        clu = np.array(cluster[i])
        centroid = np.mean(clu, axis=0)
        newcentroids.append(centroid)
    # print(newcentroids)

    newcentroids = np.array(newcentroids)
    return newcentroids, cluster, indexs
```

接着通过 test 函数来测试 kmeans

```python
def test_kmeans(centroids, cluster, k):
    for i in range(k):
        temp = np.array(cluster[i])
        plt.scatter(temp[:, 0], temp[:, 1], alpha=0.5, label=i)
        plt.scatter(centroids[i, 0], centroids[i, 1], c='r')
```

然后是高斯混合模型的 EM 算法：
主函数如下：

```python
# EM算法
def em_algorithm(dataset, k, iteration=1000, threshold=1e-10):
    # alpha shape(k, 1)
    # mu shape(k, m)
    # sigma shape(k, m)
    # gama shape(num, k)
    alpha, mu, sigma = init_GMM(dataset, k)
    num = dataset.shape[0]
    gama = np.zeros((num, k))

    for iter in range(iteration):
        prev_mu = mu
        gama = e_gama(dataset, mu, alpha, sigma)
        alpha, mu, sigma = m_theta(dataset, gama, mu)
        if np.sum(abs(prev_mu - mu)) < threshold:  # 均值基本不变，结束迭代
            print(f'iter: {iter},\nmu={mu},\nsigma={sigma},\nalpha={alpha}')
            break
        if iter % (int(iteration/10)) == 0:
            print(f'iter={iter},\nmu={mu},\nsigma={sigma},\nalpha={alpha}')

    cluster, max_index = classify(dataset, mu, gama)
    return cluster, max_index
```

其中计算 gama 的函数如下：

```python
# 更新gama
def e_gama(dataset, mu, alpha, sigma):
    num = dataset.shape[0]
    k = mu.shape[0]
    gama = np.zeros((num, k))

    for i in range(num):
        phi = gauss_density_probability(dataset[i], mu, sigma)
        sum = np.dot(phi, alpha)
        for j in range(k):
            gama[i, j] = alpha[j, 0] * phi[0, j] / sum
    return gama
```

接着更新参数：

```python
# 更新参数
def m_theta(dataset, gama, mu):
    m = dataset.shape[1]
    k = gama.shape[1]
    num = dataset.shape[0]
    cal_alpha = np.zeros((k, 1))
    cal_mu = np.zeros((k, m))
    cal_sigma = np.zeros((k, m))
    sum_gama = np.sum(gama, axis=0)

    for i in range(k):
        sum_mu = np.zeros((1, m))
        sum_sigma = np.zeros((1, m))
        for j in range(num):
            data = dataset[j]
            sum_mu += gama[j, i] * data
            sum_sigma += gama[j, i] * (data - mu[i]) ** 2
        cal_mu[i] = sum_mu / sum_gama[i]
        cal_sigma[i] = sum_sigma / sum_gama[i]
        cal_alpha[i] = sum_gama[i] / num
    return cal_alpha, cal_mu, cal_sigma
```

然后进行分簇：

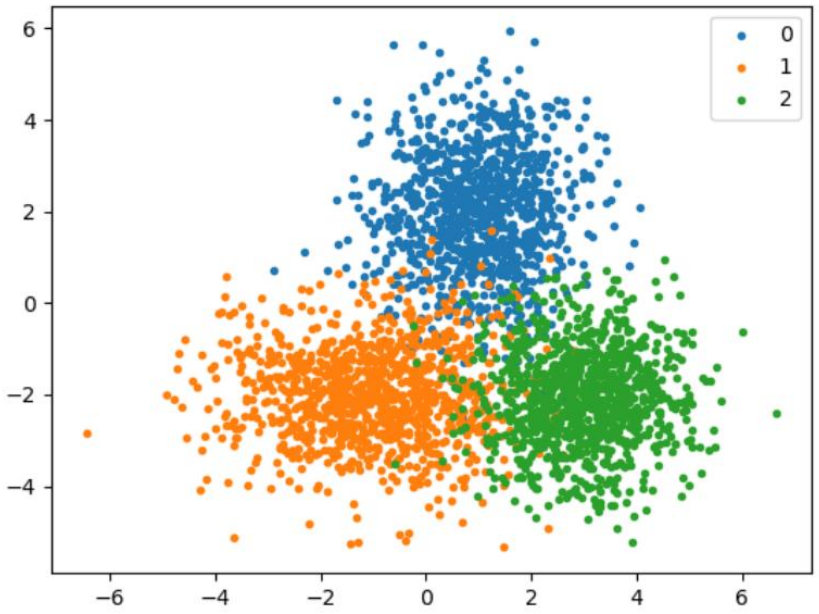
```python
# 分簇，返回簇和索引
def classify(dataset, mu, gama):
    num = dataset.shape[0]
    k = mu.shape[0]
    cluster = []
    for i in range(k):
        cluster.append([])
    max_index = np.argmax(gama, axis=1)
    for i in range(num):
        cluster[max_index[i]].append(dataset[i])
    return cluster, max_index
```

最后进行 EM 算法的测试：

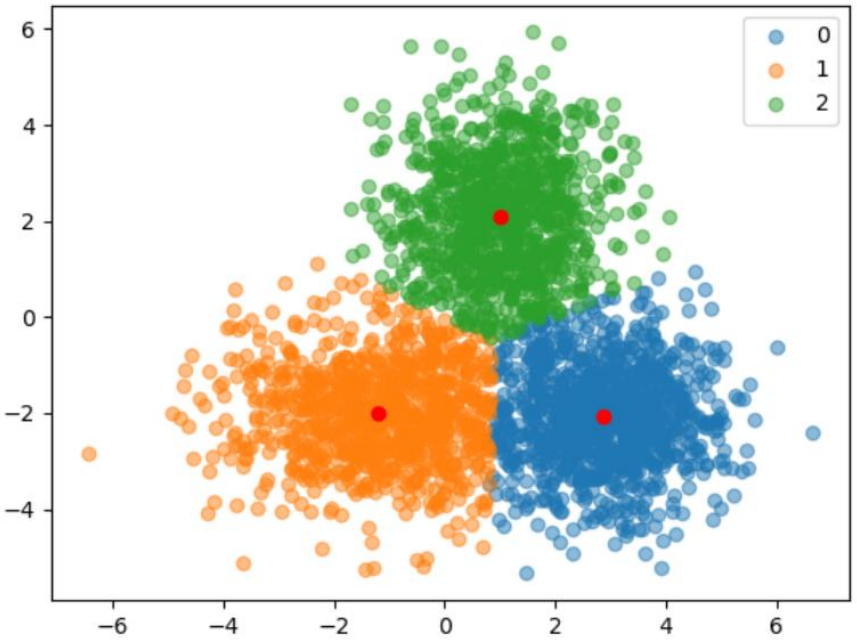
```python
# 测试GMM，画图
def test_GMM(cluster, k):
    for i in range(k):
        temp = np.array(cluster[i])
        plt.scatter(temp[:, 0], temp[:, 1], alpha=0.5, label=i)
```

(4) 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：



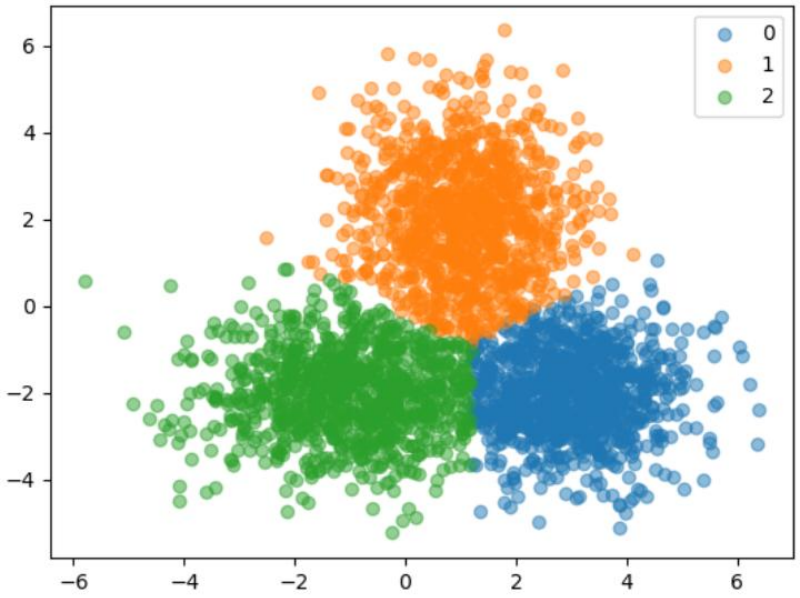
接着测试 kmeans 算法的效果：

```
D:\Anaconda\python.exe C:\Users\86151\Desktop\模式识别与机器学习\实验\实验3\kmeans.py
准确率为0.929
```

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：

```
iter: 163,
mu=[[ 2.98707768 -2.01760823]
 [ 1.04480137  1.93933605]
 [-0.96251682 -1.99316747]],
sigma=[[1.05692288 0.95276305]
 [1.00809948 2.05632103]
 [1.84832564 1.03211667]],
alpha=[[0.33225229]
 [0.33688417]
 [0.33086354]]
```

聚类结果如下图所示：

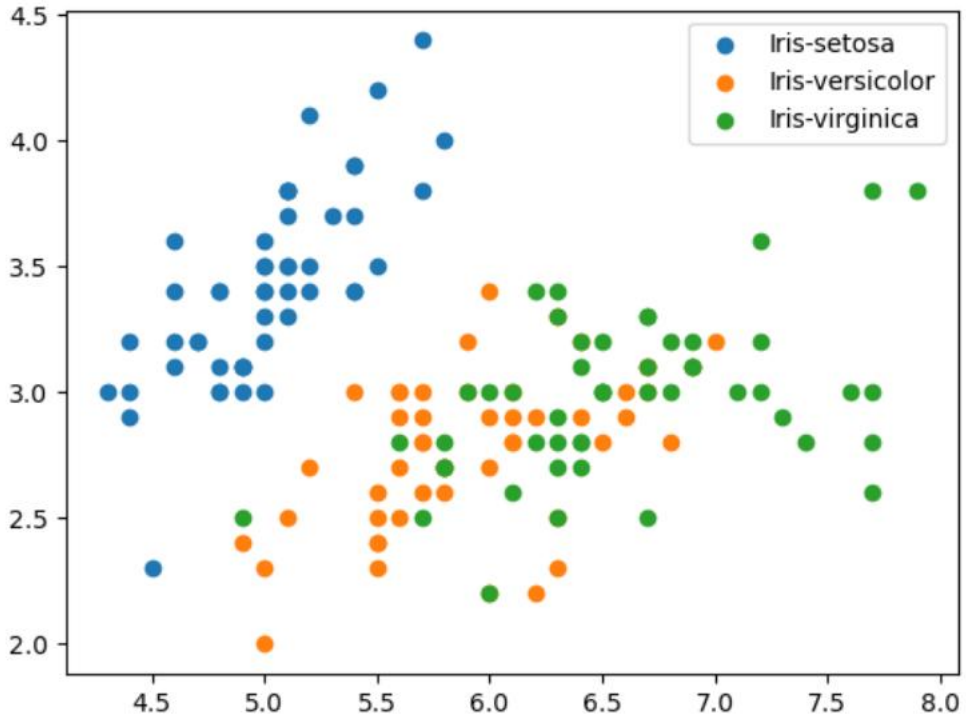


准确率如下图所示：

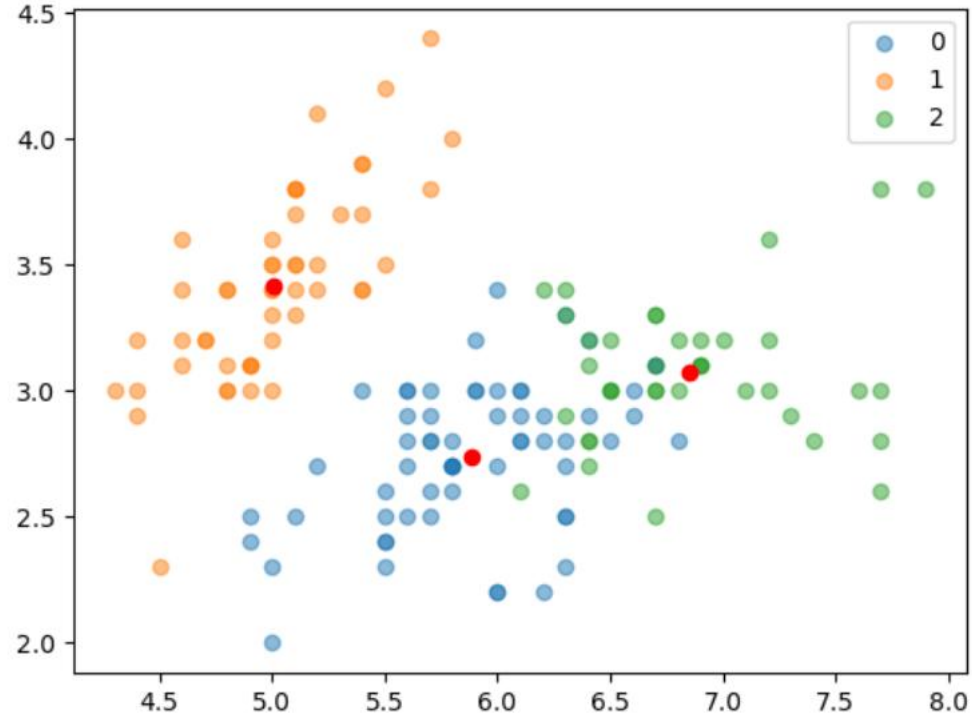
```
第1个簇有1023个样本，分别为：
0: [2.94798224 0.09508834]  1: [ 1.38198884 -0.84514549]    2: [ 1.93531525 -0.9
第2个簇有999个样本，分别为：
0: [1.10655526 0.91833063]  1: [1.24152249 3.31130516]  2: [-0.29457922  2.21807
第3个簇有978个样本，分别为：
0: [ 0.99739904 -1.49133647]    1: [ 0.3085725  -0.90044593]    2: [ 0.72885674
准确率为0.932
```

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：

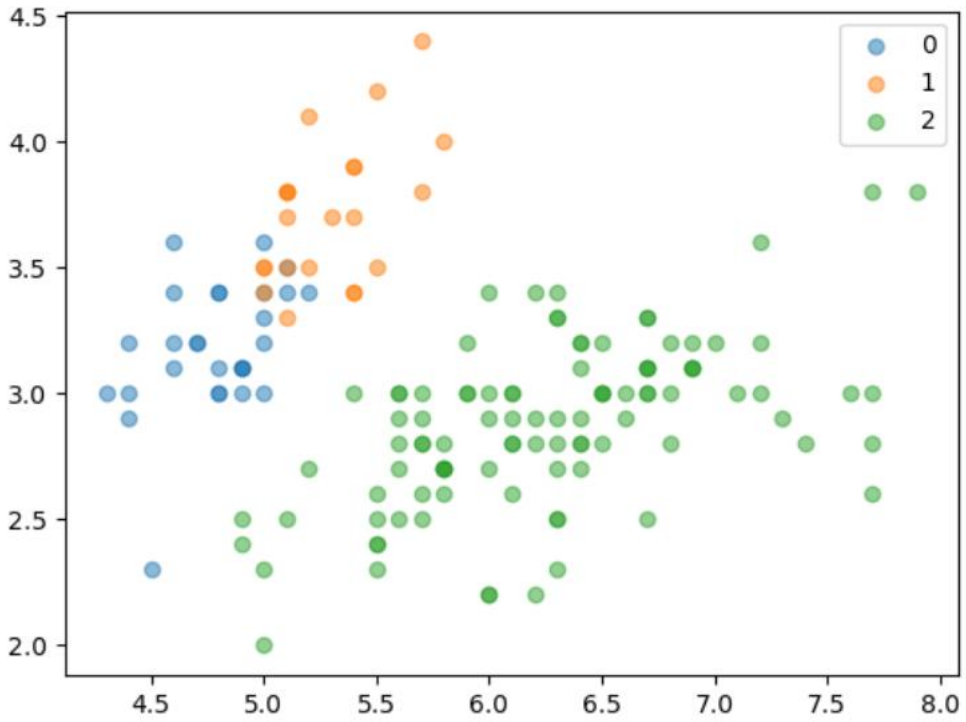


Kmeans 聚类结果如下图所示：




```
准确率为0.887
```

GMM 算法聚类结果如下图所示：




```
iter: 78,
mu=[[4.77537643 3.17940064 1.41943794 0.19485089]
 [5.28257484 3.7041398  1.51744092 0.30294192]
 [6.26199995 2.87200002 4.90599986 1.67599995]],
sigma=[[0.05798302 0.06761989 0.02859574 0.00309577]
 [0.05797491 0.08165844 0.02535586 0.01468862]
 [0.43495605 0.10961601 0.67476441 0.17862405]],
alpha=[[0.18176635]
 [0.15156695]
 [0.66666669]]
```

```
第1个簇有28个样本，分别为：
0: [5.1 3.5 1.4 0.2]    1: [4.9 3.  1.4 0.2]    2: [4.7 3.2 1.3
第2个簇有22个样本，分别为：
0: [5.4 3.9 1.7 0.4]    1: [5.4 3.7 1.5 0.2]    2: [5.8 4.  1.2
第3个簇有100个样本，分别为：
0: [7.  3.2 4.7 1.4]    1: [6.4 3.2 4.5 1.5]    2: [6.9 3.1 4.9
准确率为0.853
```

5. 实验总体结论

(1) 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。

6. 完整实验代码

kmeans.py

```
# --*-- coding:utf-8 --*--
"""
作者：徐柯炎
日期：2023 年 10 月 12 日
"""
import numpy as np
import random
```

```
8. from matplotlib import pyplot as plt
9. from ucimlrepo import fetch_ucirepo
10. from functions import *
11.
12.
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
28.
29.
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([])
38.
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)
47.
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)
55.
56.     newcentroids = np.array(newcentroids)
57.     return newcentroids, cluster, indexs
58.
59.
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, d
    islist)
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
84.
85.
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, labe
    l=i)
90.         plt.scatter(centroids[i, 0], centroids[i, 1], c='r')
91.
```

```
92.
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()
105.
106.
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()
116.
117.
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 *
12.
13.
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
25.
26.
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
51.
52.
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))
58.
```

```
    for i in range(num):
        phi = gauss_density_probability(dataset[i], mu, sig
ma)
        sum = np.dot(phi, alpha)
        for j in range(k):
            gama[i, j] = alpha[j, 0] * phi[0, j] / sum
    return gama


# 更新参数
def m_theta(dataset, gama, mu):
    m = dataset.shape[1]
    k = gama.shape[1]
    num = dataset.shape[0]
    cal_alpha = np.zeros((k, 1))
    cal_mu = np.zeros((k, m))
    cal_sigma = np.zeros((k, m))
    sum_gama = np.sum(gama, axis=0)

    for i in range(k):
        sum_mu = np.zeros((1, m))
        sum_sigma = np.zeros((1, m))
        for j in range(num):
            data = dataset[j]
            sum_mu += gama[j, i] * data
            sum_sigma += gama[j, i] * (data - mu[i]) ** 2
        cal_mu[i] = sum_mu / sum_gama[i]
        cal_sigma[i] = sum_sigma / sum_gama[i]
        cal_alpha[i] = sum_gama[i] / num
    return cal_alpha, cal_mu, cal_sigma


# 分簇，返回簇和索引
def classify(dataset, mu, gama):
    num = dataset.shape[0]
    k = mu.shape[0]
    cluster = []
    for i in range(k):
        cluster.append([])
    max_index = np.argmax(gama, axis=1)
    for i in range(num):
        cluster[max_index[i]].append(dataset[i])
     return cluster, max_index

```

```
102.
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))
112.
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}')
122.
123.     cluster, max_index = classify(dataset, mu, gama)
124.     return cluster, max_index
125.
126.
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, lab
   el=i)
132.
133.
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])
```

```
            print(f'第{i + 1}个簇有{clu.shape[0]}个样本，分别为：')
            for j in range(clu.shape[0]):
                print(f'{j}: {clu[j]}', end='\t')
            print()
        acc = test_acc(max_index, k)
        print(f'准确率为{round(acc, 3)}')
        test_GMM(cluster, k)
        plt.legend()
        plt.show()


def test_iris():
        k = 3
        dataset = load_iris()
        cluster, max_index = em_algorithm(dataset, k)
        for i in range(k):
            clu = np.array(cluster[i])
             print(f'第{i+1}个簇有{clu.shape[0]}个样本，分别为：')
            for j in range(clu.shape[0]):
                print(f'{j}: {clu[j]}', end='\t')
            print()
        acc = test_acc(max_index, k)
        print(f'准确率为{round(acc, 3)}')
        test_GMM(cluster, k)
        plt.legend()
        plt.show()


if __name__ == '__main__':
        test_main()
        test_iris()
```

functions.py

```
# --*-- coding:utf-8 --*--
"""
作者：徐柯炎
日期：2023年10月13日
"""
import numpy as np
import random
from matplotlib import pyplot as plt
from ucimlrepo import fetch_ucirepo

```

```
11.
12. def load_iris():
13.     # 从 uci 获取 iris 数据集
14.     iris = fetch_ucirepo(id=53)
15.
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
38.
39.
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_no
    rmal(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
55. 
56. 
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)
67. 
68. 
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) * n
    um, 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)
```

7. 参考文献

[最佳聚类实践：高斯混合模型（GMM）]