哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称： 机器学习

课程类型： 选修

实验题目： 实现 k-means 聚类方法和混合高斯模型

学号：1190200208
姓名：李旻翀

一、实验目的

实现一个 k-means 算法和混合高斯模型，并用 EM 算法估计模型中的参数。

二、实验要求及实验环境

1. 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）。

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）。

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。

2. 实验环境

VS code 2021 + Python + numpy + matplotlib

三、设计思想（本程序中的用到的主要算法及数据结构）

1. EM 算法原理

本次实验中设计的两个算法 K-means 算法和 GMM 算法在本质上都是 EM 算法的具体应用，在 EM 算法的基础上加上了一些比较强的假设。因此，下面首先介绍 EM 算法的思想。

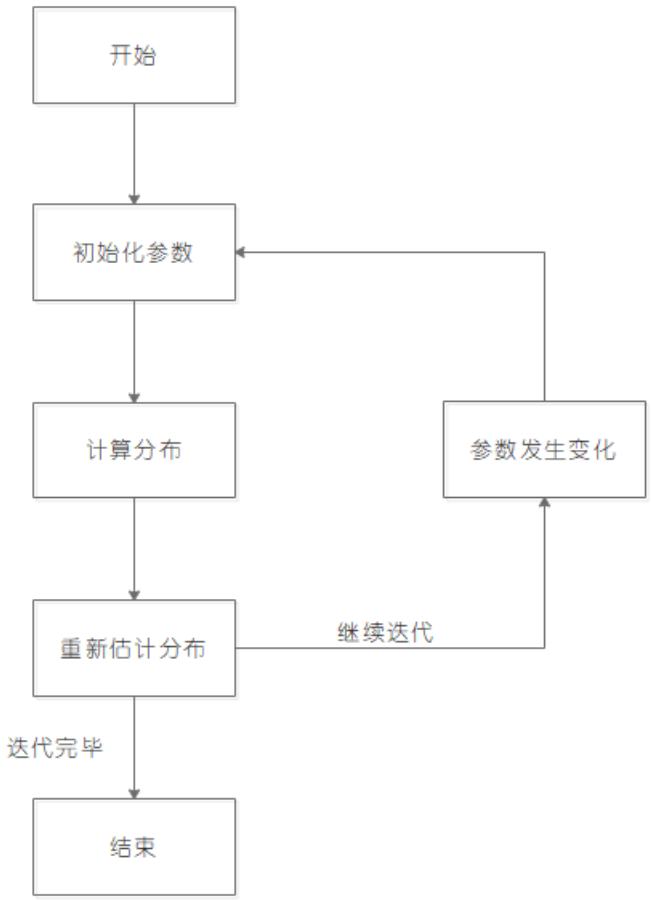
EM 算法是一种迭代算法，1977 年由 Dempster 等人总结提出，用于含有隐变量（Hidden variable）的概率模型参数的最大似然估计。

在 EM 算法中，每次迭代包含两个步骤：

1） E-step：求期望，调整分布
2） M-step：根据调整的分布，求使得目标函数最大化的参数，从而更新了参数

接着参数的更新又可以调整分布，不断循环，直到参数的变化收敛，这一过程目标函数是向更优的方向趋近的，但是在某些情况下会陷入局部最优而非全局最优的问题。

算法的流程图示意如下：



2. K-means 算法

K-means 算法采取迭代优化的方法进行近似求解，其具体步骤如下[1]：

1) 初始化 k 个聚类中心（k 为人为设定的常数）
2) 遍历所有样本，根据样本到 k 个聚类中心的距离度量，判断该样本的标签（类别）。样本距离哪个类的聚类中心最近， 则将该样本划归到哪个聚类中心。
3) 根据划分结果重新计算每个类内真实的聚类中心，如果 k 个新算出聚类中心与前一次聚类中心的距离小于误差值，则停止迭代；否则，将此步中算出的聚类中心作为新的假设聚类中心，回到 1)继续迭代求解。（即循环直到聚类中心收敛）

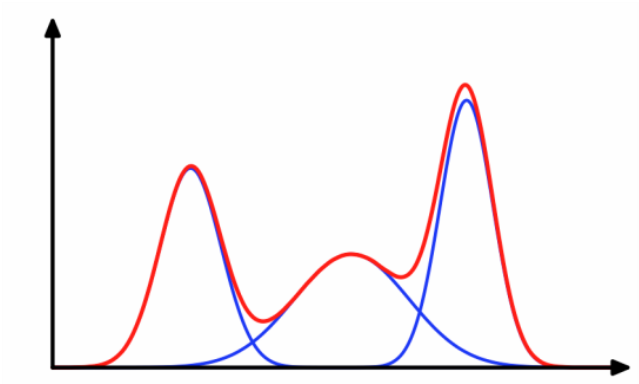
K-means 算法的核心代码如下：

```
# K-means算法核心
    while True:
        iter_count += 1
        distance = np.zeros(k)  # 保存某一次迭代中，点到所有聚类中心的距离

        for i in range(N):
            point = data[i, :]  # 此次选取，用来计算距离的样本点
            for j in range(k):
                t_center = center[j, :]  # 此次选取，用来计算距离的聚类中心
                distance[j] = np.linalg.norm(
                    point - t_center)  # 更新该样本点到聚类中心的距离
            min_index = np.argmin(distance)  # 找到该点对应的最近聚类中心
            category[i, dim] = min_index

        num = np.zeros(k)  # 保存每个类别的样本点数
        count = 0  # 计数有多少个聚类中心更新的距离小于误差值
        new_center_sum = np.zeros((k, dim))  # 临时变量
        new_center = np.zeros((k, dim))  # 保存此次迭代得到的新的聚类中心

        for i in range(N):
            label = int(category[i, dim])
            num[label] += 1  # 统计各类的样本点数
            new_center_sum[label, :] += category[i, :dim]

        for i in range(k):
            if num[i] != 0:
                new_center[i, :] = new_center_sum[i, :] / \
                    num[i]  # 计算本次样本点得出的聚类中心
            new_k_distance = np.linalg.norm(new_center[i, :] - center[i, :])
            if new_k_distance < kmeans_epsilon:  # 计数更新距离小于误差的聚类中心个数
                count += 1

        if count == k:  # 所有聚类中心的更新距离都小于误差值，结束循环
            return category, center, iter_count, num
        else:  # 否则更新聚类中心
            center = new_center
```

3.GMM(Gaussian Mixture Model)

混合模型（Mixture Model）是一个可以用来表示在总体分布（distribution）中含有 K 个子分布的概率模型，换句话说，混合模型表示了观测数据在总体中的概率分布，它是一个由 K 个子分布组成的混合分布。混合模型不要求观测数据提供关于子分布的信息，来计算观测数据在总体分布中的概率。

高斯混合模型可以看作是由 K 个单高斯模型组合而成的模型，这 K 个子模型是混合模型的隐变量（Hidden variable）。一般来说，一个混合模型可以使用任何概率分布，这里使用高斯混合模型是因为高斯分布具备很好的数学性质以及良好的计算性能。如下图，表示有 3 个子高斯分布所组成的混合分布[2]。



多元高斯分布生成的d维随机变量x的密度函数为：

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

其中μ为d维均值向量，$\sum$ 为$d\times d$维协方差矩阵，

定义如下信息：

x_j代表第j个观测数据，$j=1,2,...,N$

K是混合模型中子高斯模型的数量，$k=1,2,...,K$

α_k是观测数据属于第k个子模型的概率，有$\alpha_k\geq 0$，$\sum_{k=1}^{K}\alpha_k=1$

$\phi(x|\theta_k)$是第k个子模型的高斯分布密度函数，$\theta_k=(\mu_k,{\sigma_k}^2)$

γ_{jk}表示第j个观测数据属于第k个子模型的概率

高斯混合模型的概率分布为：

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

对于这个模型而言，参数$\theta=(\tilde{\mu}_k,\tilde{\sigma}_k,\tilde{\alpha}_k)$，也就是每个子模型的期望、方差（或协方差）、在混合模型中发生的概率。

GMM 算法的具体过程如下[3]：

1) 初始化参数

2) E-step：依据当前参数，计算每个数据j来自子模型k的可能性

$$\gamma_{jk}=\frac{\alpha_k\phi(x_j|\theta_k)}{\sum_{k=1}^{K}\alpha_k\phi(x_j|\theta_k)},j=1,2,\ldots,N;k=1,2,\ldots,K$$

3) M-step：计算新一轮迭代的模型参数

$$\mu_k = \frac{\sum_j^N (\gamma_{jk} x_j)}{\sum_j^N \gamma_{jk}}, k = 1, 2, \ldots, K$$

$$\Sigma_k = \frac{\sum_j^N \gamma_{jk}(x_j - \mu_k)(x_j - \mu_k)^T}{\sum_j^N \gamma_{jk}}, k = 1, 2, \ldots, K \quad \text{（用这一轮更新后的 } \mu_k \text{ ）}$$

$$\alpha_k = \frac{\sum_{j=1}^N \gamma_{jk}}{N}, k = 1, 2, \ldots, K$$

4) 重复计算 E-step 和 M-step 直至收敛 （$|\theta_{i+1} - \theta_i| < \varepsilon$, ε 是一个很小的正数，表示经过一次迭代之后参数变化非常小）

GMM 算法的核心代码如下：

```
# GMM算法核心步骤
    for i in range(GMM_epoch):
        count += 1
        old_loss = GMM_loss(data, k, N, new_mu, new_sigma, new_alpha)
        gamma = GMM_estep(data, k, N, new_mu, new_sigma, new_alpha)
        new_mu, new_sigma, new_alpha = GMM_mstep(
            data, gamma, k, N, dim, new_mu)
        new_loss = GMM_loss(data, k, N, new_mu, new_sigma, new_alpha)

        if i % 1 == 0:
            print(str(i) + "\t\t\t" + str(new_loss))
            argmax = np.argmax(gamma, axis=1)
            for j in range(N):
                category[j] = argmax[j]  # 更新类标签

        if (abs(new_loss - old_loss) < GMM_epsilon):  # 小于误差值，结束循环
            break

    argmax = np.argmax(gamma, axis=1)
    num = np.zeros(k)

    for j in range(N):
        label = argmax[j]
        category[j] = label  # 更新类标签
        num[label] += 1
    return category, num, count
```

4. 自行生成数据

在设计的实验中，需要用到多种聚类分布对算法实现进行测试，若每次都逐个修改聚类数，方差，均值等数据则非常麻烦，因此，直接将具体数据封装到 config 中。如图所示，在 config 中封装了两个不同的聚类配置：

```python
# 初始设置
config_0 = {
    'k': 4,  # 聚类数
    'n': 300,  # 每类的样本点数
    'dim': 2,  # 样本点维度
    'mu': np.array([
        [-5, 4], [5, 4], [3, -4], [-5, -5]
    ]),  # 均值
    'sigma': np.array([
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0, 2]]
    ])  # 方差
}

config_1 = {
    'k': 8,  # 聚类数
    'n': 300,  # 每类的样本点数
    'dim': 2,  # 样本点维度
    'mu': np.array([
        [-4, 3], [4, 2], [1, -4], [-5, -3], [0, 0], [6, -1], [-1, 8], [7, -4]
    ]),  # 均值
    'sigma': np.array([
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0, 2]],
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0], [0, 2]]
    ])  # 方差
}

configs = [config_0, config_1]
```

需要调用时，代码则变得十分简洁清晰：

```python
config = configs[0]  # 选取配置
k, n, dim, mu, sigma = config['k'], config['n'], config['dim'], config['mu'], config['sigma']
```

5. 计算准确率

考虑到经过算法更新后的 label 与初始生成时的 label 数值可能有所不同（例如，data 中的 label 是根据 0,1,2,3 的顺序分配的，而经过算法分出的 category 的标签可能第一组为 0 以外的其他数），因此，有必要采取其他方法计算分类准确率。

可以考虑在算法运行时统计出每个类别的样本点数，保存在 num 列表中，因为初始生成样本点时每组的样本点数量相同，因此我们可以考虑，用每组样本点数量减去均值，求其绝对值之和再除以二，便是分类错误的样本点的数目，由此便能很简单地计算出分类的准确率。

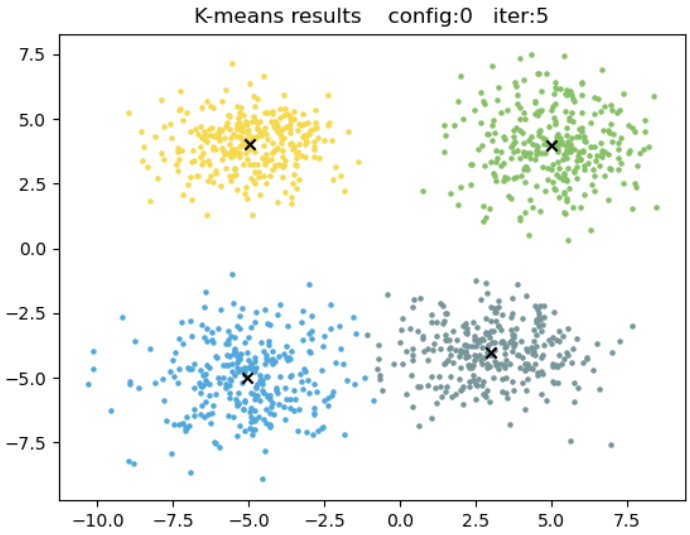
具体代码实现如下：

```python
def calc_acc(num, n, N):
    "计算K-means算法的准确率"
    # 思想：将各类点数统一减去n（最开始分类时每一类的数目），求其中元素绝对值之和，除以二，即为分类错误点的数量
    temp = np.abs(num - n)
    wrong = np.sum(temp) / 2
    return 1 - wrong / N
```

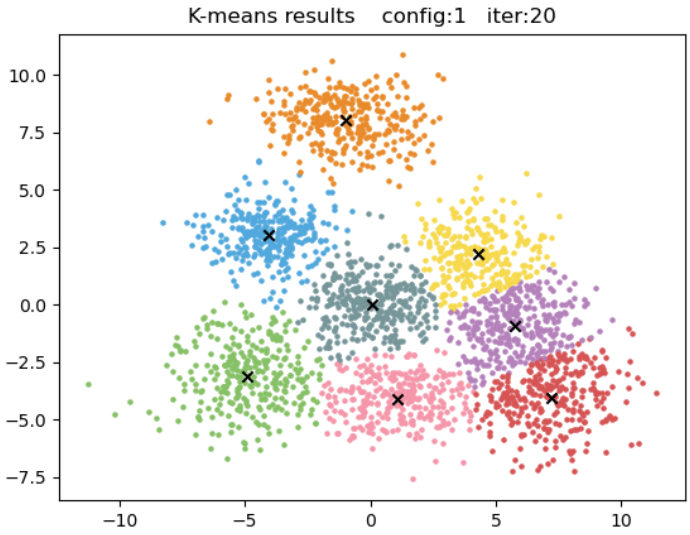
四、实验结果与分析

1. K-means 算法，自行生成数据

我们设置 K-means 算法的迭代误差 υ 为 1e-8，观察其在 config_0 与 config_1 下的运行效果：




```
acc:0.9991666666666666
```

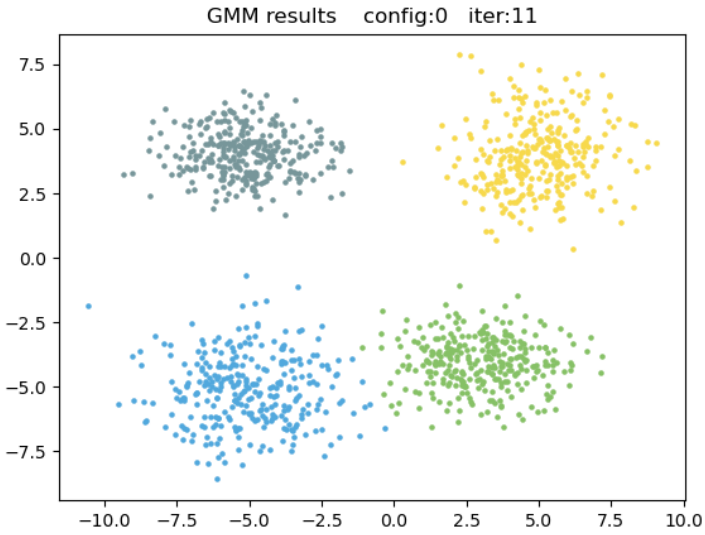



```
acc:0.9841666666666666
```

可以看出，在两种配置下，我们的 K-means 聚类模型都取得了很不错的效果，并且在实际运行时，运行速度很快。

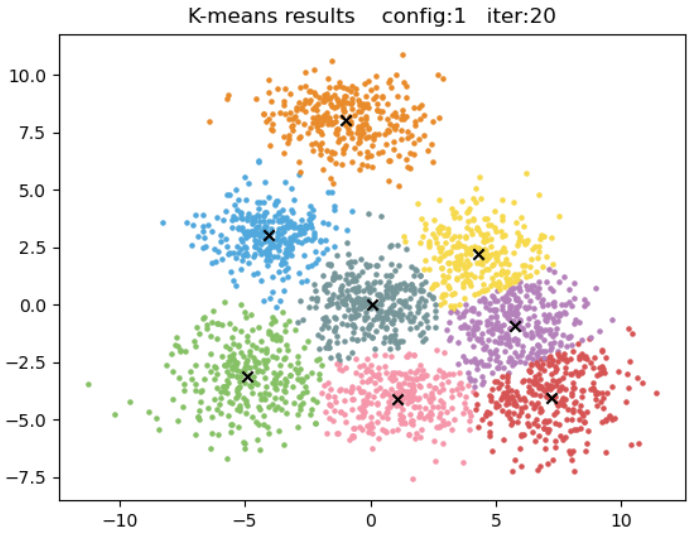
2. GMM 算法，自行生成数据

设置 GMM 算法的迭代误差 υ 为 1e-8，最大迭代次数为 40，观察其在 config_0 与 config_1 下的运行效果：



Iteration:	log likelihood
0	-5722.342657674656
1	-5722.282651646422
2	-5722.277636211191
3	-5722.276772958291
4	-5722.276620812076
5	-5722.2765939628825
6	-5722.276589223785
7	-5722.276588387221
8	-5722.276588239544
9	-5722.276588213475
10	-5722.276588208887

acc:0.9983333333333333



```
Iteration:              log likelihood
0                       -12607.530394513506
1                       -12606.621280167856
2                       -12606.238407328066
3                       -12605.996838992938
4                       -12605.823474771005
5                       -12605.689816135016
6                       -12605.581543009832
7                       -12605.4907733547
8                       -12605.412934366273
9                       -12605.345231099289
10                      -12605.285842737772
11                      -12605.233491094317
12                      -12605.187208611456
13                      -12605.146214825289
14                      -12605.109851500298
15                      -12605.07754885645
16                      -12605.048807690693
17                      -12605.023189191108
18                      -12605.000308190152
19                      -12604.979827795532
20                      -12604.96145450899
21                      -12604.944933535657
22                      -12604.93004425462
23                      -12604.91659591765
24                      -12604.904423657836
```

可以看出，在两种配置下，GMM 聚类模型同样也能取得很不错的效果。在迭代过程中，GMM 的 log likelihood 始终保持着降低的趋势（可能由于设置的各类均值与方差不好，降低的幅度较小）。GMM 算法的不足是在实际运行时，其运行速度比 K-means 要慢许多。

3. 采用 UCI 数据集

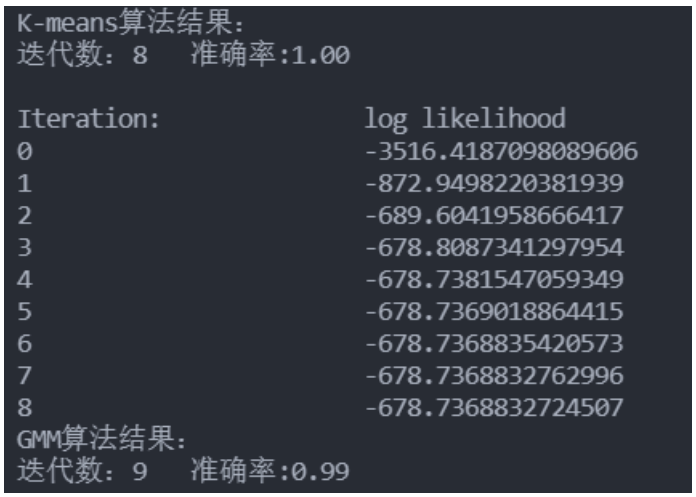
我们选取 UCI 的 Iris 数据集对我们的算法进行测试[4]。从数学角度来讲，Iris 数据集包含 150 个样本数据，每个样本数据由四个维度构成（花萼长度，花萼宽度，花瓣长度，花瓣宽度）。我们需要根据这四个维度的数据，来预测鸢尾花属于(Setosa, Versicolour, Virginica)三类中的哪一类。在样本中，每一类鸢尾花有 50 个样本。

我们读取 Iris 数据集并进行测试，测试结果如下：

```
K-means算法结果:
迭代数: 7   准确率:1.00

Iteration:              log likelihood
0                       -3516.4187098089606
1                       -872.9498220381939
2                       -689.6041958666417
3                       -678.8087341297954
4                       -678.7381547059349
5                       -678.7369018864415
6                       -678.7368835420573
7                       -678.7368832762996
8                       -678.7368832724507
GMM算法结果:
迭代数: 9   准确率:0.99
```

```
K-means算法结果：
迭代数：12   准确率:0.99

Iteration:              log likelihood
0                       -3632.0735093520175
1                       -1689.2045898809035
2                       -2091.5096241696006
3                       -1735.0677055515166
4                       -2339.1209238407255
5                       -1570.3927953557006
6                       -3323.6230474443496
7                       -1810.4366071199888
8                       -4494.730281259483
9                       -1943.6336168846049
10                      -5817.071923874155
11                      -2610.053877249066
12                      -2147.801936882232
13                      -1134.8351748341938
14                      -684.9276212810939
15                      -678.7341550135379
16                      -678.7368391014161
17                      -678.7368826329777
18                      -678.7368832631332
19                      -678.7368832722597
GMM算法结果：
迭代数：20   准确率:0.99
```

```
K-means算法结果：
迭代数：12   准确率:0.69

Iteration:              log likelihood
0                       -1623.912803013421
1                       -1395.6937369308785
2                       -1348.4286876925858
3                       -1489.8190707462866
4                       -1508.8106646952456
5                       -1507.461057236561
6                       -703.5944468018088
7                       -679.1865654681885
8                       -678.7437850284012
9                       -678.7369887123396
10                      -678.7368848015591
11                      -678.736883294542
12                      -678.736883272715
13                      -678.7368832723988
GMM算法结果：
迭代数：14   准确率:0.99
```



可以注意到，GMM 算法的准确率始终稳定在 0.99 左右，log likelihood 也保持着降低的趋势，总体分类效果良好；而四次实验中，K-means 算法总体准确率较高，但有一次的准确率低至 0.69，这可能是因为 K-means 算法的聚类结果严重依赖于初始化时的聚类中心，当初始化的聚类中心“不好”的时候，会导致整个的聚类结果表现较差，因此，我们可以多次实验来选择最好的分类效果。

五、结论

1. K-means 和 GMM 都是 EM 算法的体现。两者都遵循 EM 算法的 E 步和 M 步的迭代优化模式。其中，K-means 算法的假设更强，相对来说也更好理解，而 GMM 算法则用混合高斯模型来描述聚类结果。假设多个高斯模型对总模型的贡献是有不一定相同的权重，且样本也是按概率从属于某一类。
2. 根据实验结果，总体上来讲两者都能较好的解决简单的分类问题，其中 K-means 算法速度相对来说更快，但 GMM 算法的准确率比较能反应模型的真实效果。
3. 两个算法都存在着可能只取到局部最优的问题。尤其对于 K-means 算法来说，初值的选取对其结果影响较大。可以考虑多次实验取比较平均的结果作为最终的分类结果。

六、参考文献

[1] 阿泽.【机器学习】K-means（非常详细）[EB/OL].2020[2021-10-23].https://zhuanlan.zhihu.com/p/78798251.
[2] 戴文亮.高斯混合模型（GMM）[EB/OL].2020[2021-10-23].https://zhuanlan.zhihu.com/p/30483076.
[3] cruft.如何简单易懂的解释高斯混合（GMM）模型？[EB/OL].2020[2021-10-23].https://zhuanlan.zhihu.com/p/101162925.
[4] UCI Machine Learning Repository.Iris Data Set[EB/OL].1988[2021-10-24].https://archive.ics.uci.edu/ml/datasets/Iris.

七、附录：源代码（带注释）

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
import pandas as pd

kmeans_epsilon = 1e-8  # K-means 算法的迭代误差
GMM_epsilon = 1e-8  # GMM 算法的迭代误差
GMM_epoch = 40  # GMM 算法的迭代次数
con = 1  # 选择的设置

colors = [
    '#77969A', '#F7D94C', '#86C166', '#51A8DD', '#E98B2A', '#B481BB',
'#F596AA', '#D75455'
]  # 颜色库

# 初始设置
config_0 = {
    'k': 4,  # 聚类数
    'n': 300,  # 每类的样本点数
    'dim': 2,  # 样本点维度
    'mu': np.array([
        [-5, 4], [5, 4], [3, -4], [-5, -5]
    ]),  # 均值
    'sigma': np.array([
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0],
[0, 2]]
    ])  # 方差
}

config_1 = {
    'k': 8,  # 聚类数
    'n': 300,  # 每类的样本点数
    'dim': 2,  # 样本点维度
    'mu': np.array([
        [-4, 3], [4, 2], [1, -4], [-5, -3], [0, 0], [6, -1], [-1, 8],
[7, -4]
    ]),  # 均值
    'sigma': np.array([
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0],
[0, 2]],
        [[2, 0], [0, 1]], [[2, 0], [0, 2]], [[3, 0], [0, 1]], [[3, 0],
[0, 2]]
```

```
    ])  # 方差
}

configs = [config_0, config_1]


def gen_data(k, n, dim, mu, sigma):
    "生成数据"
    raw_data = np.zeros((k, n, dim))
    # true_label = np.zeros(k * n) # 每个点的真实标签
    for i in range(k):
        raw_data[i] = np.random.multivariate_normal(mu[i], sigma[i], n)
    data = np.zeros((k * n, dim))
    for i in range(k):
        data[i * n:(i + 1) * n] = raw_data[i]
        # for j in range(i * n, (i + 1) * n):
        #     true_label[j] = i # 给真实标签赋值
    return data


def kmeans(data, k, N, dim):
    "K-means 算法实现"
    category = np.zeros((N, dim+1))
    category[:, 0:dim] = data[:, :]  # 多出的一维用于保存类别标签
    center = np.zeros((k, dim))  # k 行 dim 列，保存各类中心
    for i in range(k):  # 随机以某些样本点作为初始点
        center[i, :] = data[np.random.randint(0, N), :]

    iter_count = 0  # 迭代次数

    # K-means 算法核心
    while True:
        iter_count += 1
        distance = np.zeros(k)  # 保存某一次迭代中，点到所有聚类中心的距离

        for i in range(N):
            point = data[i, :]  # 此次选取，用来计算距离的样本点
            for j in range(k):
                t_center = center[j, :]  # 此次选取，用来计算距离的聚类中
心
                distance[j] = np.linalg.norm(
                    point - t_center)  # 更新该样本点到聚类中心的距离
            min_index = np.argmin(distance)  # 找到该点对应的最近聚类中心
            category[i, dim] = min_index
```

```
        num = np.zeros(k)  # 保存每个类别的样本点数
        count = 0  # 计数有多少个聚类中心更新的距离小于误差值
        new_center_sum = np.zeros((k, dim))  # 临时变量
        new_center = np.zeros((k, dim))  # 保存此次迭代得到的新的聚类中心

        for i in range(N):
            label = int(category[i, dim])
            num[label] += 1  # 统计各类的样本点数
            new_center_sum[label, :] += category[i, :dim]

        for i in range(k):
            if num[i] != 0:
                new_center[i, :] = new_center_sum[i, :] / \
                    num[i]  # 计算本次样本点得出的聚类中心
            new_k_distance = np.linalg.norm(new_center[i, :] -
center[i, :])
            if new_k_distance < kmeans_epsilon:  # 计数更新距离小于误差的
聚类中心个数
                count += 1

        if count == k:  # 所有聚类中心的更新距离都小于误差值，结束循环
            return category, center, iter_count, num
        else:  # 否则更新聚类中心
            center = new_center


def calc_acc(num, n, N):
    "计算 K-means 算法的准确率"
    # 思想：将各类点数统一减去 n（最开始分类时每一类的数目），求其中元素绝对值
之和，除以二，即为分类错误点的数量
    temp = np.abs(num - n)
    wrong = np.sum(temp) / 2
    return 1 - wrong / N


def kmeans_show(k, n, dim, mu, sigma):
    "K-means 算法的结果展示"

    data = gen_data(k, n, dim, mu, sigma)
    # print(data.shape)  # (1200,2)
    N = data.shape[0]  # N 为样本点总数

    category, center, iter_count, num = kmeans(data, k, N, dim)
```

```
    acc = calc_acc(num, n, N)

    for i in range(N):  # 绘制分为各类的样本点
        color_num = int(category[i, dim] % len(colors))
        plt.scatter(category[i, 0], category[i, 1],
                    c=colors[color_num], s=5)

    for i in range(k):
        plt.scatter(center[i, 0], center[i, 1],
                    c='black', marker='x')  # 绘制各个聚类中心

    print("acc:" + str(acc))

    plt.title("K-means results " + "   config:" +
              str(con) + "   iter:" + str(iter_count))
    plt.show()
    return


def GMM_loss(data, k, N, mu, sigma, alpha):
    "计算 GMM 算法的 loss"
    loss = 0
    for i in range(N):
        temp = 0
        for j in range(k):  # 根据公式计算 loss
            temp += alpha[j] * \
                multivariate_normal.pdf(data[i], mean=mu[j],
cov=sigma[j])
        loss += np.log(temp)
    return loss


def GMM_estep(data, k, N, mu, sigma, alpha):
    "GMM 算法的 E-step"
    gamma = np.zeros((N, k))  # 初始化隐变量 γ
    for i in range(N):
        marginal_prob = 0
        for j in range(k):
            marginal_prob += alpha[j] * multivariate_normal.pdf(data[i],
                                                                mean=mu
[j], cov=sigma[j])
        for j in range(k):
            gamma[i, j] = alpha[j] * multivariate_normal.pdf(data[i],
```

```
                                                        mean=mu[j],
cov=sigma[j]) / marginal_prob
    return gamma


def GMM_mstep(data, gamma, k, N, dim, mu):
    "GMM 算法的 M-step"
    new_mu = np.zeros((k, dim))
    new_sigma = np.zeros((k, dim, dim))
    new_alpha = np.zeros(k)

    for i in range(k):
        sum_gamma = 0
        for j in range(N):
            sum_gamma += gamma[j, i]

        # 计算新的 μ
        mu_temp = np.zeros(dim)
        for j in range(N):
            mu_temp += gamma[j, i] * data[j]
        new_mu[i] = mu_temp / sum_gamma

        # 计算新的 σ
        sigma_temp = np.zeros(dim)
        for j in range(N):
            dis = data[j] - mu[i]
            sigma_temp += dis**2 * gamma[j, i]

        new_sigma_temp = np.eye(dim)
        new_sigma_temp[0, 0] = sigma_temp[0]
        new_sigma_temp[1, 1] = sigma_temp[1]
        new_sigma[i] = new_sigma_temp / sum_gamma

        # 计算新的 π
        new_alpha[i] = sum_gamma / N

    return new_mu, new_sigma, new_alpha


def GMM(data, k, N, dim, mu, sigma):
    "GMM 算法"
    category = np.zeros(N)  # 初始化每个样本点的类标签
    alpha = np.array([1 / k] * k)  # 初始化第 k 个高斯模型的权重
```

```
    new_mu, new_sigma, new_alpha = mu, sigma, alpha

    count = 0  # 记录真实的迭代次数

    print("Iteration:\t\tlog likelihood")

    # GMM 算法核心步骤
    for i in range(GMM_epoch):
        count += 1
        old_loss = GMM_loss(data, k, N, new_mu, new_sigma, new_alpha)
        gamma = GMM_estep(data, k, N, new_mu, new_sigma, new_alpha)
        new_mu, new_sigma, new_alpha = GMM_mstep(
            data, gamma, k, N, dim, new_mu)
        new_loss = GMM_loss(data, k, N, new_mu, new_sigma, new_alpha)

        if i % 1 == 0:
            print(str(i) + "\t\t\t" + str(new_loss))
            argmax = np.argmax(gamma, axis=1)
            for j in range(N):
                category[j] = argmax[j]  # 更新类标签

        if (abs(new_loss - old_loss) < GMM_epsilon):  # 小于误差值，结束
循环
            break

    argmax = np.argmax(gamma, axis=1)
    num = np.zeros(k)

    for j in range(N):
        label = argmax[j]
        category[j] = label  # 更新类标签
        num[label] += 1
    return category, num, count


def GMM_show(k, n, dim, mu, sigma):
    "GMM 算法的结果展示"

    data = gen_data(k, n, dim, mu, sigma)
    # print(data.shape)  # (1200,2)
    N = data.shape[0]  # N 为样本点总数

    category, num, count = GMM(data, k, N, dim, mu, sigma)
```

```
    acc = calc_acc(num, n, N)
    print("acc:" + str(acc))

    for i in range(N):  # 绘制分为各类的样本点
        color_num = int(category[i] % len(colors))
        plt.scatter(data[i, 0], data[i, 1],
                    c=colors[color_num], s=5)

    plt.title("GMM results " + "   config:" + str(con) + "   iter:" +
str(count))
    plt.show()
    return


def UCI_read():
    "读入 UCI 数据并切分"
    raw_data = pd.read_csv("./iris.csv")
    data = raw_data.values
    label = data[:, -1]
    np.delete(data, -1, axis=1)
    # print(data.shape) # (150, 5)
    return data, label


def UCI_show():
    "使用 UCI 数据集测试算法"
    data, label = UCI_read()

    k = 3  # 聚类数
    N = data.shape[0]  # 样本数量
    n = N / k  # 每一类样本数量
    dim = data.shape[1]  # 样本维度

    # K-means 算法计算结果
    category, center, iter_count, num = kmeans(data, k, N, dim)
    kmeans_acc = calc_acc(num, n, N)
    print("K-means 算法结果: ")
    print("迭代数: %d   准确率:%.2f\n" % (iter_count, kmeans_acc))

    # GMM 算法计算结果
    GMM_mu = np.array([[0] * dim] * k)
    GMM_sigma = np.array([np.eye(dim)] * k)
    temp_counts = np.array([0] * k)
    temp = np.array([[0] * dim] * k)
```

```
    for i in range(N):
        c = int(category[i, -1])
        temp_counts[c] += 1
        for j in range(dim):
            temp[c, j] += category[i, j]

    for i in range(k):
        for j in range(dim):
            GMM_mu[i, j] = temp[i, j] / temp_counts[i]

    for i in range(k):
        for j in range(dim):
            for t in range(N):
                if category[i, -1] == i:
                    GMM_sigma[i, j,
                              j] += pow((category[i, j] - GMM_mu[i, j]),
2)
            GMM_sigma[i, j, j] /= temp_counts[i]

    category, num, count = GMM(data, k, N, dim, GMM_mu, GMM_sigma)
    GMM_acc = calc_acc(num, n, N)
    print("GMM 算法结果：")
    print("迭代数：%d   准确率:%.2f\n" % (count, GMM_acc))
    return


# 主函数
method = 2

config = configs[con]  # 选取配置
k, n, dim, mu, sigma = config['k'], config['n'], config['dim'],
config['mu'], config['sigma']

if method == 0:
    kmeans_show(k, n, dim, mu, sigma)
elif method == 1:
    GMM_show(k, n, dim, mu, sigma)
elif method == 2:
    UCI_show()
```