《模式识别与机器学习 A》实验报告

实验题目：k-means 聚类方法和混合高斯模型

学　　号：2021112845

姓　　名：张智雄

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）。

a) 用 K-means 聚类，测试效果；

b) 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）。

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

Windows11; Anaconda+python3.11; VS Code

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）

4.1 算法理论支撑

4.1.1 K-means 聚类算法

K-means 聚类算法的核心思想为假定聚类内部点之间的距离应该小于数据点与聚类外部的点之间的距离。即使得**每个数据点和与它最近的中心之间距离的平方和最小**。

假设数据集为$X = \{x_1, \dots, x_N\}, x_i \in \mathbb{R}^D$，我们的目标是将数据集划分为$K$个类别$Y$。令$\mu_k \in \mathbb{R}^D, k = 1, \dots, K$表示各类别的中心。聚类问题等价于求概率分布：

$$P(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)}$$

K-means 聚类相当于假设$P(X|Y)$服从多元高斯分布（**特征之间相互独立，协方差矩阵**$\Sigma = \lambda \boldsymbol{I}$），且$P(Y)$为等概率均匀分布。而$P(X)$为已知数据分布，从似然的角度看，极大化$P(Y|X)$即等价于极大化$P(X|Y) \sim -\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)$，即最小化各数据点到其类别的均值。

多元正态分布：$\mathcal{N}(x|\mu,\Sigma) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \exp\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\}$

引入二值指示变量$r_{nk} \in \{0,1\}$表示数据点的分类情况，则可定义目标函数为

$$\min_{r,\mu} J = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|x_n - \mu_k\|^2$$

因此最优化过程可以划分为两步：

1) 固定μ，优化r_{nk}。由于J关于r_{nk}是线性的，因此可以对每个n分别进行最小化，即对r_{nk}根据与聚类中心的距离进行最优化：

$$r_{nk}=\begin{cases}1, & k=argmin_j\|x_n-\mu_j\|^2\\ 0, & \text{其他情况}\end{cases}$$

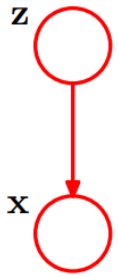
2) 固定r_{nk}，优化μ。由于J是μ的二次函数，对其求导等于零得

$$\frac{\partial J}{\partial \mu_k}=2\sum_{n=1}^{N}r_{nk}(x_n-\mu_k)=0\Rightarrow\mu_k=\frac{\sum_n r_{nk}x_n}{\sum_n r_{nk}}$$

即μ_k等于类别 k 的所有数据点的均值。

4.1.2 混合高斯模型

任意连续概率密度都能用多个高斯分布的线性组合叠加的高斯混合概率分布$p(x)=\sum_{k=1}^{K}\pi_k\mathcal{N}(x|\mu_k,\Sigma_k)$来描述。引入"1 of K"编码的二值随机变量z，满足$z_k\in\{0,1\}$且$\sum_{k=1}^{K}z_k=1$。



由右图模型定义联合概率分布$p(x,z)=p(z)\cdot p(x|z)$，z的边缘**先验**概率分布设为$p(z_k=1)=\pi_k(0\le\pi_k\le1\text{ 且 }\sum_{k=1}^{K}\pi_k=1)$，也可写作$p(z)=\prod_{k=1}^{K}\pi_k^{z_k}$。

那么，x的条件概率分布为：

$$p(x|z_k=1)=\mathcal{N}(x|\mu_k,\Sigma_k)\Leftrightarrow p(x|z)=\prod_{k=1}^{K}\mathcal{N}(x|\mu_k,\Sigma_k)^{z_k}$$

于是可以给出$p(x)=\sum_z p(z)\cdot p(x|z)=\sum_{k=1}^{K}\pi_k\mathcal{N}(x|\mu_k,\Sigma_k)$，同时，$z$的条件后验概率$\gamma(z_k)$由贝叶斯定理得（已知为$x$，类别为$z_k$的概率）：

$$\gamma(z_k)\equiv p(z_k=1|x)=\frac{p(z_k=1)p(x|z_k=1)}{\sum_{j=1}^{K}p(z_j=1)p(x\mid z_j=1)}=\frac{\pi_k\mathcal{N}(x\mid\mu_k,\Sigma_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(x\mid\mu_j,\Sigma_k)}$$

于是此聚类过程可以看做将概率分布$p(x)$解耦成K个高斯分布，对应K个类别。对于数据集$X=\{x_1,\dots,x_N\},x_i\in\mathbb{R}^D,X\in\mathbb{R}^{N\times D}$，对应隐变量表示为$Z\in\mathbb{R}^{N\times K}$。

则对数似然函数为

$$\ln p(X\mid\pi,\mu,\Sigma)=\sum_{n=1}^{N}\ln\left\{\sum_{k=1}^{K}\pi_k\mathcal{N}(x_n\mid\mu_k,\Sigma_k)\right\}$$

将此似然函数关于μ_k求导（假设Σ_k非奇异），令$N_k=\sum_{n=1}^{N}\gamma(z_{nk}),\gamma(z_{nk})\equiv p(z_k=1|x_n)$为能被分配到聚类$k$的有效数量，可以得到：

$$\sum_{n=1}^{K}\underbrace{\frac{\pi_k\mathcal{N}(x_n\mid\mu_k,\Sigma_k)}{\sum_j\pi_j\mathcal{N}(x_n\mid\mu_j,\Sigma_j)}}_{\gamma(z_{nk})}\Sigma_k^{-1}(x_n-\mu_k)=0\Rightarrow\mu_k=\frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})x_n$$

由此式μ_k可视为当前所有点数据为第k类的概率加权平均。

同样地，将此函数关于Σ_k求导等于 0 可以得到：

$$\Sigma_k = \frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})(x_n-\mu_k)(x-\mu_k)^T$$

最后使用拉格朗日乘子法关于π_k优化 $\ln p(X \mid \pi,\mu,\Sigma)+\lambda(\sum_{k=1}^{K}\pi_k-1)$（$\pi_k$需要满足和为 1 的条件）得到：

$$\sum_{n=1}^{N}\frac{\mathcal{N}(x_n \mid \mu_k,\Sigma_k)}{\sum_j \pi_j \mathcal{N}(x_n \mid \mu_j,\Sigma_j)}+\lambda=0 \Rightarrow \pi_k=\frac{N_k}{N}$$

使用 EM 算法优化$\ln\ p(X \mid \pi,\mu,\Sigma)$即可总结为以下步骤：

ALGORITHM 1　EM for Gaussian Mixture Models

1：**input** X ←数据集，K ←类别数目，$iter$ ←迭代次数；

2：初始化均值μ_k、协方差Σ_k和混合系数π_k

3：计算对数似然$\ln\ p(X \mid \pi,\mu,\Sigma) \leftarrow \sum_{n=1}^{N}\ln\{\sum_{k=1}^{K}\pi_k\mathcal{N}(x_n \mid \mu_k,\Sigma_k)\}$

3：**while** $i < iter$ **do**

4：　　$\gamma(z_{nk}) \leftarrow \frac{\pi_k\mathcal{N}(x_n|\mu_k,\Sigma_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(x_n|\mu_j,\Sigma_j)}$；　　（E 步）

5：　　$\mu_k^{new} \leftarrow \frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})x_n$；

6：　　$\Sigma_k^{new} \leftarrow \frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})(x_n-\mu_k^{new})(x-\mu_k^{new})^T$；　　（M 步）

7：　　$\pi_k^{new} \leftarrow \frac{N_k}{N}$；

8：**end while**

9：**return** μ_k,Σ_k,π_k //返回最优参数；

4.2 实验设计

4.2.1 随机数据生成

```python
def generate_data(k, count):
    means = np.zeros((k, 2))                    # 各个高斯分布的均值
    data = np.zeros((count, 3))                  # 待分类的数据
    means = [[2, 2], [-2, -2], [2, -2], [-2, 2]]
    cov = np.array([[1, 0], [0, 1]])             # 协方差矩阵
    classes = []
    index = 0
    for i in range(k):
        for j in range(int(count / k)):
            data[index, 0:2] = np.random.multivariate_normal(means[i], cov)
            data[index, 2] = i
            index += 1
            if i not in classes:
                classes.append(i)
    return means, data, classes
```

图 1　随机数据生成代码截图

如上图代码，使用 np.random.multivariate_normal 方法按给定的协方差矩阵和均值按多元高斯分布初始化k个类别的数据点。

4.2.2 K-means 聚类

首先初始化k个类的中心，这里采取的是从数据集中随机选取k个样本作为初始的k类中心。

```
# 初始化中心点
def init_center(self):
    self.center = np.zeros((self.k, self.data_x.shape[1]))
    for i in range(self.k):
        random = np.random.randint(0, self.data_x.shape[0])
        #random = np.random.randint(i*(self.data_x.shape[0]/self.k), (i+1)*(self.data_x.shape[0]/self.k))
        self.center[i] = (self.data_x[random])
```

图 2 K-means 初始化代码截图

而后是更新中心的算法，主要是分为两步：

1) 通过计算每个样本到k个中心的距离（欧式距离），然后选取最小的距离对应的那个聚类中心作为样本标签，将该样本划分到这个类中，

2) 根据更新后的类别计算类内均值，作为新的中心。

```
def update_center(self):
    self.clusters = [[] for i in range(self.k)]
    for i in range(self.data_x.shape[0]):
        min_dis = 100000
        min_index = 0
        for j in range(self.k):
            dis = self.distance(self.data_x[i], self.center[j])
            if dis < min_dis:
                min_dis = dis
                min_index = j
        self.clusters[min_index].append(self.data_x[i])
    for i in range(self.k):
        self.center_old = self.center
        self.center[i] = np.mean(np.array(self.clusters[i]), axis=0)
```

图 3 更新中心代码截图

重复上述过程，直至中心更新距离较之上次变化较小时退出迭代。

```
def train(self, limit=100000):
    self.init_center()
    self.update_center()
    for i in range(limit):
        self.update_center()
        if np.sum(np.abs(self.center - self.center_old)) < 0.1:
            break
        self.center_old = self.center
    return self.clusters, self.center
```

图 4 迭代代码截图

4.2.3 混合高斯模型 GMM

首先初始化k个类的均值、协方差和混合系数，可以有随机生成和采用 K-means 聚类的结果两种方式进行初始化。

```
def init_gauss(self):
    self.mean = []
    self.cov = []
    self.alpha = []
    # 使用k-means的结果初始化
    if self.center is not None:
        for i in range(self.k):
            self.mean.append(self.center[i])
            self.cov.append(np.eye(self.data_x.shape[1]))
            self.alpha.append(1 / self.k)
    # 随机初始化
    else:
        for i in range(self.k):
            random = np.random.randint(0, self.data_x.shape[0])
            #random = np.random.randint(i*(self.data_x.shape[0]/self.k), (i+1)*(self.data_x.shape[0]/self.k))
            self.mean.append(self.data_x[random])
            self.cov.append(np.eye(self.data_x.shape[1]))
            self.alpha.append(1 / self.k)
```

图 5 GMM 初始化代码截图

根据对数似然计算公式$\ln\ p(X\mid\pi,\mu,\Sigma)\leftarrow\sum_{n=1}^{N}\ln\{\sum_{k=1}^{K}\pi_k\mathcal{N}(x_n\mid\mu_k,\Sigma_k)\}$以及 关于$\mu_k$、$\Sigma_k$和$\pi_k$的导数进行 EM 更新：

1) E 步：计算$\gamma(z_{nk})\leftarrow\frac{\pi_k\mathcal{N}(x_n|\mu_k,\Sigma_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(x_n|\mu_j,\Sigma_j)}$，即各数据点的类别概率；

2) M 步：计算新的均值$\mu_k^{new}=\frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})x_n$，混合系数$\pi_k^{new}\leftarrow\frac{N_k}{N}$以及协方差$\Sigma_k^{new}=\frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})(x_n-\mu_k^{new})(x-\mu_k^{new})^T$

```
def E_step(self):
    self.gamma = np.zeros((self.data_x.shape[0], self.k))
    for i in range(self.data_x.shape[0]):
        for j in range(self.k):
            self.gamma[i, j] = self.alpha[j] * self.gaussian(self.data_x[i], self.mean[j], self.cov[j])
        self.gamma[i] /= np.sum(self.gamma[i])

def M_step(self):
    for j in range(self.k):
        Nk = np.sum(self.gamma[:, j])
        self.mean[j] = np.sum(self.gamma[:, j].reshape(-1, 1) * self.data_x, axis=0) / Nk
        self.cov[j] = np.dot((self.data_x - self.mean[j]).T,
                             (self.data_x - self.mean[j]) * self.gamma[:, j].reshape(-1, 1)) / Nk
        self.alpha[j] = Nk / self.data_x.shape[0]
```

图 6 EM 更新代码截图

重复上述过程，直至似然函数较之上次变化较小时退出迭代。

```
def train(self, limit=100000):
    self.init_gauss()
    likelihood = []
    likelihood.append(self.likelihood())
    for i in range(limit):
        self.E_step()
        self.M_step()
        likelihood.append(self.likelihood())
        if np.abs(likelihood[i + 1] - likelihood[i]) < 0.1:
            break
    plt.figure()
    plt.plot(likelihood)
    plt.show()
    return self.mean, self.cov, self.alpha
```

图 7 迭代代码截图

4.3 实验结果及分析

4.3.1 自己生成数据结果比较

设置均值为$(-2,-2),(2,-2),(2,2),(-2,2)$，协方差矩阵均为$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$，生成1000组数据，而后先后使用K-means和GMM进行聚类，聚类结果如下：

K-means正确率为96.7%，GMM正确率为96.9%（在使用K-means的结果作为GMM初始化的条件下）。

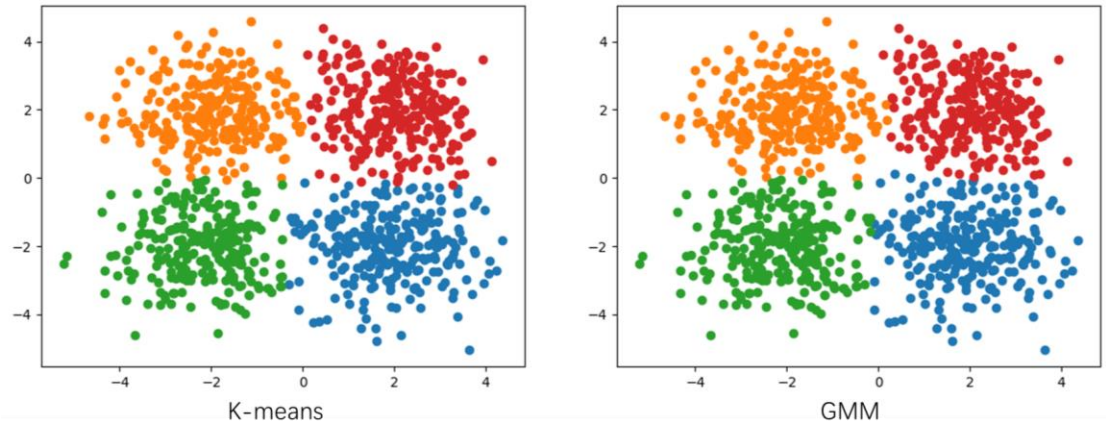


图8 聚类结果可视化（$k=4$）

同时，经过反复实验发现，K-means和GMM的聚类结果严重依赖于初始化的结果，当初始化中心（或均值）选取区分度不高时，结果往往会比较差。

而GMM以K-means的结果初始化，能够一定程度上优化K-means的聚类结果（特别是当分类结果不好时）。

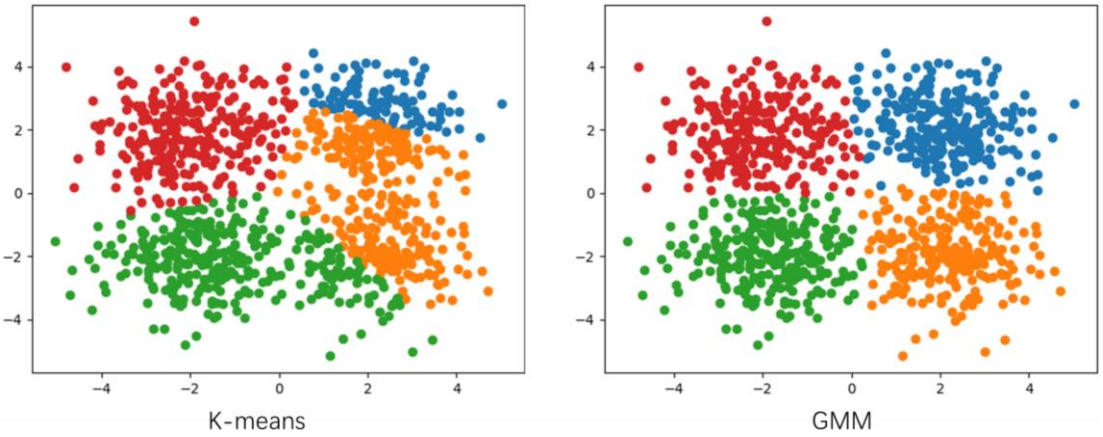


图 9 GMM在K-means基础上的优化（$k=4$）

而特征之间不相互独立时，GMM聚类结果明显优于K-means。这是由于K-means假设数据呈球状分布，不能够区分开具有不相互独立的特征。

下图为均值为$(-2,-2),(2,-2),(2,2),(-2,2)$，协方差矩阵均为$\begin{bmatrix}1 & 0.36\\ 0.36 & 1\end{bmatrix}$，生成1000组数据，先后使用K-means和GMM进行聚类的聚类结果如下：

此时K-means正确率为91.8%，GMM正确率为97.6%。

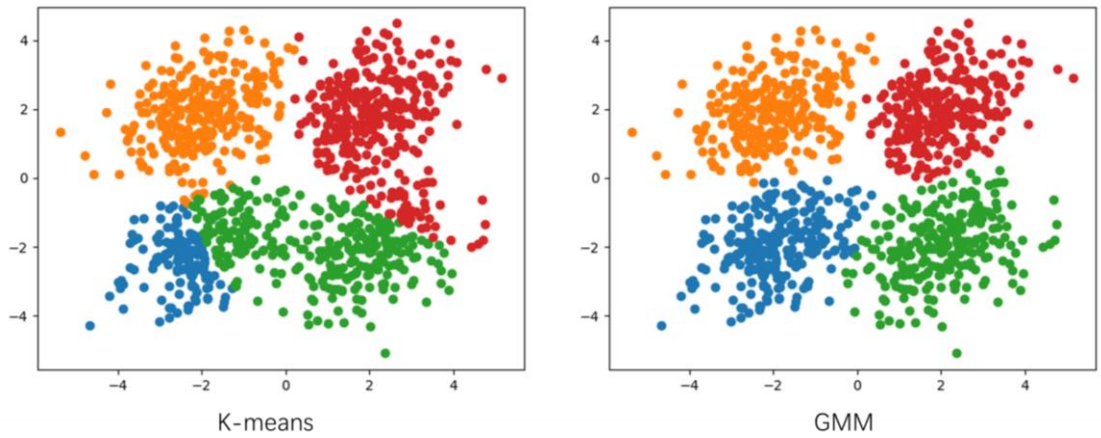


图 10 特征不独立时的聚类结果（$k=4$）

4.3.2 UCI-Iris 数据集结果比较

而后使用 Iris 数据集进行聚类，K-means 分类正确率为 88%，数据如下：

表格 1 K-means 分类结果

	类别 1	类别 2	类别 3
均值	[6.04 2.81 4.60 1.52]	[5.01 3.34 1.60 0.30]	[7.01 3.10 5.92 2.16]

GMM 的分类正确率为 96.7%，各聚类的矩阵、协方差、混合系数数据如下：

表格 2 GMM 分类结果

	类别 1	类别 2	类别 3
均值	[5.92 2.78 4.20 1.30]	[5.01 3.42 1.46 0.24]	[6.54 2.94 5.48 1.98]
协方差矩阵	$\begin{bmatrix}0.28 & 0.10 & 0.18 & 0.05\\0.10 & 0.09 & 0.09 & 0.04\\0.18 & 0.09 & 0.20 & 0.06\\0.05 & 0.04 & 0.06 & 0.03\end{bmatrix}$	$\begin{bmatrix}0.12 & 0.10 & 0.02 & 0.01\\0.10 & 0.14 & 0.01 & 0.01\\0.02 & 0.01 & 0.02 & 0.01\\0.01 & 0.01 & 0.01 & 0.01\end{bmatrix}$	$\begin{bmatrix}0.38 & 0.09 & 0.30 & 0.06\\0.09 & 0.11 & 0.08 & 0.06\\0.30 & 0.08 & 0.33 & 0.07\\0.06 & 0.06 & 0.07 & 0.09\end{bmatrix}$
混合系数	0.30	0.33	0.37

将上述聚类结果按第一个特征和第二个特征进行可视化结果如下：

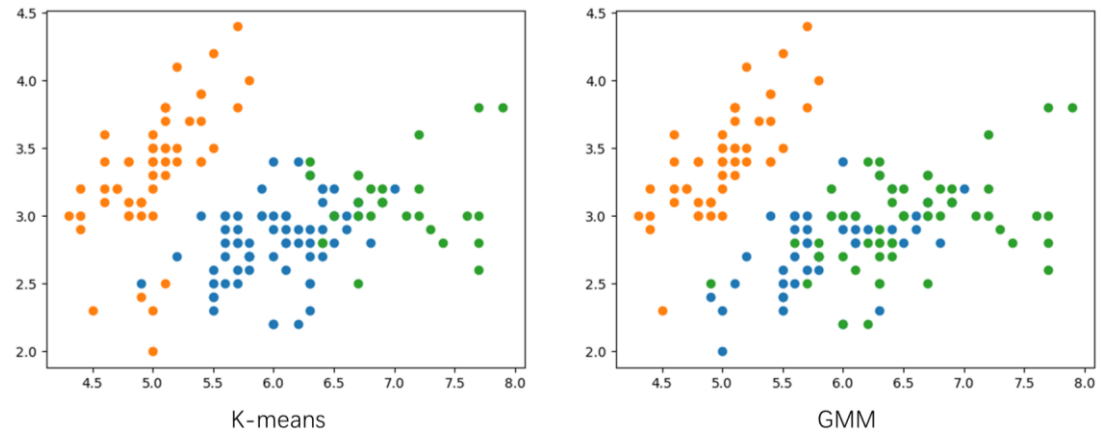


图 11 Iris 聚类结果

5、 实验结论

K-means 和 GMM 都是 EM 算法的体现。两者共同之处都有隐变量，遵循 EM 算法的 E 步和 M 步的迭代优化。

K-means 其实就是一种特殊的高斯混合模型，提出了强假设，假设每一类在样本数据中出现的概率相等均为$\frac{1}{k}$，每个特征间相互独立，且满足变量间的协方差矩阵为单位对角阵。在此条件下，可以直接用欧氏距离作为 K-means 的协方差去衡量相似性。

而 GMM 用混合高斯模型来描述聚类结果，且假设多个高斯模型对总模型的贡献是有权重的，且样本属于某一类也是由概率的，从而相似度也由离散的0,1变成了需要通过全概率公式计算的值，分类效果一般好于 K-means。

实验发现，初值的选取对 K-means 和 GMM 的效果影响较大，初始化较差可能会导致陷入局部最优解而得不到较好的聚类结果。可以使用 GMM 对 K-means 的分类结果进行进一步优化。

6、 完整实验代码

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import itertools
import random

def random_seed(seed):
    np.random.seed(seed)
    random.seed(seed)

def iris():
    data = pd.read_csv('./Iris.data')
    data["Species"] = data["Species"].map({"Iris-setosa": 1, "Iris-versicolor": 0, "Iris-virginica": 2})
    data = np.array(data[:].values)

    return data

def generate_data(k, count):
    means = np.zeros((k, 2))                    # 各个高斯分布的均值
    data = np.zeros((count, 3))                 # 待分类的数据
    means = [[2, 2], [-2, -2], [2, -2], [-2, 2]]
    cov = np.array([[1, 0.36], [0.36, 1]])   # 协方差矩阵
    classes = []
```

```
24.         index = 0
25.         for i in range(k):
26.             for j in range(int(count / k)):
27.                 data[index, 0:2] = np.random.multivariate_normal(me
    ans[i], cov)
28.                 data[index, 2] = i
29.                 index += 1
30.                 if i not in classes:
31.                     classes.append(i)
32.         return means, data, classes
33.
34.     class K_means:
35.         def __init__(self, data, k):
36.             self.k = k
37.             self.data_x = data[:, 0:-1]
38.             self.data_y = data[:, -1]
39.
40.         # 初始化中心点
41.         def init_center(self):
42.             self.center = np.zeros((self.k, self.data_x.shape[1]))
43.             for i in range(self.k):
44.                 random = np.random.randint(0, self.data_x.shape[0])
45.                 #random = np.random.randint(i*(self.data_x.shape[0]
    /self.k), (i+1)*(self.data_x.shape[0]/self.k))
46.                 self.center[i] = (self.data_x[random])
47.
48.         def distance(self, x, y):
49.             return np.sqrt(np.sum(np.square(x - y)))
50.
51.         def update_center(self):
52.             self.clusters = [[] for i in range(self.k)]
53.             for i in range(self.data_x.shape[0]):
54.                 min_dis = 100000
55.                 min_index = 0
56.                 for j in range(self.k):
57.                     dis = self.distance(self.data_x[i], self.center
    [j])
58.                     if dis < min_dis:
59.                         min_dis = dis
60.                         min_index = j
61.                 self.clusters[min_index].append(self.data_x[i])
62.             for i in range(self.k):
63.                 self.center_old = self.center
```

```
64.                 self.center[i] = np.mean(np.array(self.clusters[i])
    , axis=0)
65.
66.         def train(self, limit=100000):
67.             self.init_center()
68.             self.update_center()
69.             for i in range(limit):
70.                 self.update_center()
71.                 if np.sum(np.abs(self.center - self.center_old)) <
    0.1:
72.                     break
73.                 self.center_old = self.center
74.             return self.clusters, self.center
75.
76.         def accuary(self, classes):
77.             classes = list(itertools.permutations(classes, len(clas
    ses)))
78.             acc = []
79.             for cls in classes:
80.                 count = 0
81.                 for i in range(self.data_x.shape[0]):
82.                     min_dis = 100000
83.                     min_index = 0
84.                     for j in range(self.k):
85.                         dis = self.distance(self.data_x[i], self.ce
    nter[j])
86.                         if dis < min_dis:
87.                             min_dis = dis
88.                             min_index = j
89.                     if cls[int(self.data_y[i])] == min_index:
90.                         count += 1
91.                 acc.append(count / self.data_x.shape[0])
92.             #print(acc)
93.             return max(acc)
94.
95.         def draw(self):
96.             plt.figure()
97.             for i in range(self.k):
98.                 plt.scatter(np.array(self.clusters[i])[:, 0], np.ar
    ray(self.clusters[i])[:, 1])
99.             plt.show()
100.
101.    class GMM:
102.        def __init__(self, data, k, center=None):
```

```
103.             self.k = k
104.             self.data_x = data[:, 0:-1]
105.             self.data_y = data[:, -1]
106.             self.center = center
107.
108.         def init_gauss(self):
109.             self.mean = []
110.             self.cov = []
111.             self.alpha = []
112.             # 使用k-means 的结果初始化
113.             if self.center is not None:
114.                 for i in range(self.k):
115.                     self.mean.append(self.center[i])
116.                     self.cov.append(np.eye(self.data_x.shape[1]))
117.                     self.alpha.append(1 / self.k)
118.             # 随机初始化
119.             else:
120.                 for i in range(self.k):
121.                     random = np.random.randint(0, self.data_x.shape
    [0])
122.                     #random = np.random.randint(i*(self.data_x.shap
    e[0]/self.k), (i+1)*(self.data_x.shape[0]/self.k))
123.                     self.mean.append(self.data_x[random])
124.                     self.cov.append(np.eye(self.data_x.shape[1]))
125.                     self.alpha.append(1 / self.k)
126.
127.         def gaussian(self, x, mean, cov):
128.             return np.exp(-
    0.5 * np.dot(np.dot((x - mean), np.linalg.inv(cov)), (x - mean)
    .T)) / (np.sqrt(np.linalg.det(cov)) * np.power(2 * np.pi, self.
    data_x.shape[1] / 2))
129.
130.         def likelihood(self):
131.             sum = 0.0
132.             for k in range(self.k):
133.                 sum += self.alpha[k] * self.gaussian(self.data_x, s
    elf.mean[k], self.cov[k])
134.             return np.sum(np.log(sum))
135.             #return np.sum(np.log(np.sum(self.alpha[k] * self.gauss
    ian(self.data_x, self.mean[k], self.cov[k]) for k in range(self
    .k))))
136.
137.         def E_step(self):
138.             self.gamma = np.zeros((self.data_x.shape[0], self.k))
```

```
139.          for i in range(self.data_x.shape[0]):
140.              for j in range(self.k):
141.                  self.gamma[i, j] = self.alpha[j] * self.gaussia
      n(self.data_x[i], self.mean[j], self.cov[j])
142.              self.gamma[i] /= np.sum(self.gamma[i])
143.
144.      def M_step(self):
145.          for j in range(self.k):
146.              Nk = np.sum(self.gamma[:, j])
147.              self.mean[j] = np.sum(self.gamma[:, j].reshape(-
      1, 1) * self.data_x, axis=0) / Nk
148.              self.cov[j] = np.dot((self.data_x - self.mean[j]).T
      ,
149.                                   (self.data_x - self.mean[j]) *
       self.gamma[:, j].reshape(-1, 1)) / Nk
150.              self.alpha[j] = Nk / self.data_x.shape[0]
151.
152.      def train(self, limit=100000):
153.          self.init_gauss()
154.          likelihood = []
155.          likelihood.append(self.likelihood())
156.          for i in range(limit):
157.              self.E_step()
158.              self.M_step()
159.              likelihood.append(self.likelihood())
160.              if np.abs(likelihood[i + 1] - likelihood[i]) < 0.1:
161.                  break
162.          plt.figure()
163.          plt.plot(likelihood)
164.          plt.show()
165.          return self.mean, self.cov, self.alpha
166.
167.      def accuary(self, classes):
168.          z = np.argmax(self.gamma, axis=1)
169.          classes = list(itertools.permutations(classes, len(clas
      ses)))
170.          acc = []
171.          for cls in classes:
172.              count = 0
173.              for i in range(self.data_x.shape[0]):
174.                  if z[i] == cls[int(self.data_y[i])]:
175.                      count += 1
176.              acc.append(count / self.data_x.shape[0])
177.          #print(acc)
```

```
178.             return max(acc)
179.
180.         def draw(self):
181.             z = np.argmax(self.gamma, axis=1)
182.             plt.figure()
183.             for i in range(self.k):
184.                 plt.scatter(self.data_x[z == i, 0], self.data_x[z =
    = i, 1])
185.             plt.show()
186.
187.
188.     if __name__ == '__main__':
189.
190.         # random_seed(177)
191.         # k = 4
192.         # count = 1000
193.         # means, data, classes = generate_data(k, count)
194.         # print(classes)
195.
196.         # k_means = K_means(data, k)
197.         # clusters, center = k_means.train()
198.         # k_means.draw()
199.         # print("K-means 正确率：  ", k_means.accuary(classes))
200.         # print()
201.         # gmm = GMM(data, k, center)
202.         # mean, cov, alpha = gmm.train()
203.         # print(mean, cov, alpha)
204.         # gmm.draw()
205.         # print("GMM 正确率：  ", gmm.accuary(classes))
206.
207.         random_seed(6)
208.         data = iris()
209.         k_means = K_means(data, 3)
210.         clusters, center = k_means.train()
211.         k_means.draw()
212.         print("K-means 正确率：  ", k_means.accuary([0, 1, 2]))
213.         print(center)
214.         gmm = GMM(data, 3, center)
215.         mean, cov, alpha = gmm.train()
216.         print(mean, cov, alpha)
217.         gmm.draw()
218.         print("GMM 正确率：  ", gmm.accuary([0, 1, 2]))
219.
220.
```

7、 参考文献

[1]李航. 统计学习方法[M]. 清华大学出版社, 2012.

[2]周志华. 机器学习[M]. 清华大学出版社, 2016.

[3]Bishop C .Pattern Recognition and Machine Learning[M]. 2006.