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聚类和混合高斯模型

学号：1190202401

姓名：陈豪

一、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。

二、实验要求及实验环境

2.1 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。

2.2 实验环境

- Windows 10
- python3.9.2
- vscode

三、设计思想（本程序中的用到的主要算法及数据结构）

3.1 算法原理

3.1.1 Kmeans

给定样本集$D=\{\mathbf{x_1},\mathbf{x_2},\ldots,\mathbf{x_m}\}$，“k-means“算法针对聚类所得簇$C=\{\mathbf{C_1},\mathbf{C_2},\ldots,\mathbf{C_k}\}$，最小化平方误差

$$E=\sum_{i=1}^{k}\sum_{\mathbf{x}\in\mathbf{C_i}}||\mathbf{x}-\mu_{\mathbf{i}}||_2^2 \tag{1}$$

式(1)中，$\mu_{\mathbf{i}}=\dfrac{1}{|\mathbf{C_i}|}\sum_{\mathbf{x}\in\mathbf{C_i}}\mathbf{x}$是簇$\mathbf{C_i}$的均值向量。直观来看，式$(1)$在一定程度上刻画了簇内样本围绕均值向量的紧密程度，E值越小则簇内样本的相似度越高。

最小化式(1)并不容易，找到它的最优解需考察样本集D所有可能的簇划分，这是一个NP难问题。因此，kmeans算法采用了贪心策略，通过迭代化来近似求解式(1)。

迭代优化的策略如下：

1. 首先初始化一组均值向量
2. 根据初始化的均值向量给出样本集D的一个划分，样本距离那个簇的均值向量距离最近，则将该样本划归到哪个簇
3. 再根据这个划分来计算每个簇内真实的均值向量，如果真实的均值向量与假设的均值向量相同，假设正确；否则，将真实的均值向量作为新的假设均值向量，回到1.继续迭代求解。

3.1.2 Gmm

首先回顾多元高斯分布生成的 n 维随机变量 x 的密度函数为：

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

其中 μ 为 n 维的均值向量，Σ 为 $n\times n$ 的协方差矩阵。

因此定义高斯混合分布如下

$$p_{\mathcal{M}}(x)=\sum_{i=1}^{k}\alpha_i p(\mathbf{x}|\mu_{\mathbf{i}},\Sigma_i) \tag{3}$$

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。其中$\mu_{\mathbf{i}},\Sigma_i$是第$i$个高斯分布的均值和协方差矩阵，$\alpha_i>0$为相应的混合系数，满足$\sum_{i=1}^{k}\alpha_i=1$。

假设样本的生成过程由高斯混合分布给出：首先根据$\alpha_1,\alpha_2,\ldots,\alpha_k$定义的先验分布选择高斯分布混合成分，其中$\alpha_i$为选择第i个混合成分的概率；然后，根据被选择的高斯混合成分的概率密度函数进行采样，从而生成相应的样本。

样本集$D=\{x_1,x_2,\ldots,x_m\}$由上述过程生成：令随机变量$z_j\in\{1,2,\ldots,k\}$表示生成样本x_j的高斯混合成分，其取值未知。显然，z_j的先验概率$p(z_j=i)$对应于 $\alpha_i(i=1,2,\ldots,k)$。那么根据贝叶斯定理，z_j的后验分布对应于：

$$p_{\mathcal{M}}(z_j=i|\mathbf{x_j})=\frac{p(z_j=i)\cdot p_{\mathcal{M}}(\mathbf{x_j}|z_j=i)}{(\quad)}=\frac{\alpha_i\cdot p(\mathbf{x_j}|\mu_{\mathbf{i}},\Sigma_i)}{k} \tag{4}$$

$$p_{\mathcal{M}}(\mathbf{x_j}) \qquad \sum_{l=1}^{k} \alpha_l p(\mathbf{x_j}|\mu_l, \Sigma_l)$$

换言之，$p_{\mathcal{M}}(z_j = i|\mathbf{x_j})$给出了样本$\mathbf{x_j}$由第$i$个高斯混合分布生成的后验概率。

当式(3)已知时，混合高斯模型将样本集D划分成了k个簇$C = \{\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C_k}\}$，对于每一个样本$\mathbf{x_j}$，其簇标记为$\lambda_j$，如下确定:

$$\lambda_j = \mathbf{arg\ max}_i p_{\mathcal{M}}(z_j = i|\mathbf{x_j}) \tag{5}$$

关键在与参数$\{\alpha_i, \mu_i, \Sigma_i | i \in \{1, 2, \ldots, k\}\}$的求解，如果给定样本集$D$可以采用极大似然估计(最大化对数似然):

$$LL(D) = \ln\left(\prod_{j=1}^{m} p_{\mathcal{M}}(\mathbf{x_j})\right) = \sum_{j=1}^{m} \ln\left(\sum_{i=1}^{k} \alpha_i \cdot p(\mathbf{x_j}|\mu_i, \Sigma_i)\right) \tag{6}$$

使式(6)最大化，对μ_i求导令导数为0有:

$$\sum_{j=1}^{m} \frac{\alpha_i \cdot p(\mathbf{x_j}|\mu_i, \Sigma_i)}{\sum_{l=1}^{k} \alpha_l \cdot p(\mathbf{x_j}|\mu_l, \Sigma_l)} \Sigma_i^{-1}(\mathbf{x_j} - \mu_\mathbf{i}) = 0 \tag{7}$$

两边同乘Σ_i进行化简有:

$$\mu_i = \frac{\sum_{j=1}^{m} p_{\mathcal{M}}(z_j = i|\mathbf{x_j}) \cdot \mathbf{x_j}}{\sum_{j=1}^{m} p_{\mathcal{M}}(z_j = i|\mathbf{x_j})} \tag{8}$$

即各个混合成分的均值可以通过样本加权平均来估计，权重样本是每个样本属于该成分的后验概率。

同理式(6)对Σ_i求导令导数为0有:

$$\Sigma_i = \frac{\sum_{j=1}^{m} p_{\mathcal{M}}(z_j = i|\mathbf{x_j}) \cdot (\mathbf{x_j} - \mu_\mathbf{i})(\mathbf{x_j} - \mu_\mathbf{i})^T}{\sum_{j=1}^{m} p_{\mathcal{M}}(z_j = i|\mathbf{x_j})} \tag{9}$$

对于混合系数α_i，由于其还需要满足$\alpha_i \geq 0, \sum_i^k \alpha_i = 1$，所以在式$(6)$的基础上增加拉格朗日项:

$$LL(D) + \lambda\left(\sum_{i=1}^{k} \alpha_i - 1\right) \tag{10}$$

其中λ为拉格朗日乘子，由式(10)对α_i求导并令导数为0有:

$$\sum_{j=1}^{m} \frac{p(\mathbf{x_j}|\mu_i, \Sigma_i)}{\sum_{l=1}^{k} \alpha_l \cdot p(\mathbf{x_j}|\mu_l, \Sigma_l)} + \lambda = 0 \tag{11}$$

两边同乘以α_i，对所有样本求和有$\lambda = -m$，有:

$$\alpha_i = \frac{1}{m} \sum_{j=1}^{m} \frac{\alpha_i \cdot p(\mathbf{x_j}|\mu_i, \Sigma_i)}{\sum_{l=1}^{k} \alpha_l \cdot p(\mathbf{x_j}|\mu_l, \Sigma_l)} \tag{12}$$

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。

3.2 算法具体实现实现

3.2.1 Kmeans

1. 首先随机选择一个样本作为均值向量
2. 进行迭代，直到选择到k个均值向量：

   假设当前已经选择到i个均值向量$\{\mu_1, \mu_2, \ldots, \mu_i\}$，则在$D/\ \{\mu_1, \mu_2, \ldots, \mu_i\}$选择距离已选出的$i$个均值向量距离最远的样本

   将其加入初始均值向量，得到$\{\mu_1, \mu_2, \ldots, \mu_i, \mu_{i+1}\}$
3. 重复迭代直到算法收敛：
   1. 初始化$\mathbf{C_i} = \emptyset, i = 1, 2, \ldots, k$
   2. 对$\mathbf{x_j}, j = 1, 2, \ldots, m$标记为$\lambda_j$，使得$\lambda_j = \mathbf{arg\ min}_i ||\mathbf{x_j} - \mu_i||$，即使得每个$\mathbf{x_j}$都是属于距离其最近的均值向量所在的簇
   3. 将样本$\mathbf{x_j}$划分到相应的簇$\mathbf{C}_{\lambda_j} = \mathbf{C}_{\lambda_j} \cup \{\mathbf{x_j}\}$
   4. 重新计算每个簇的均值向量$\hat{\mu_i} = \frac{1}{|\mathbf{C_i}|} \sum_{\mathbf{x} \in \mathbf{C}_i} \mathbf{x}$
   5. 如果对于所有的$i \in 1, 2, \ldots, k$，均有$\hat{\mu_i} = \mu_i$，则终止迭代；否则将重新赋值$\mu_i = \hat{\mu_i}$进行迭代

3.1.2 Gmm

给定样本集D和高斯混合成分数目k。

1. 初始化参数$\{\alpha_i = \frac{1}{k}, \mu_i, \Sigma_i = I \in \{1, 2, \ldots, k\}\}$,其中$\mu_i$同上述Kmeans方法一样以及$\mathbf{C_i} = \emptyset$
2. 开始迭代至带到迭代次数或者是参数值不再发生变化：
   1. E步，根据式(4)计算每个样本由各个混合高斯成分生成的后验概率
   2. M步，根据式$(8)(9)(12)$更新参数$\{\alpha_i, \mu_i, \Sigma_i | i \in \{1, 2, \ldots, k\}\}$
3. 根据式(5)确定每个样本的簇标记λ_j，并将其加入相应的簇$\mathbf{C}_{\lambda_j} = \mathbf{C}_{\lambda_j} \cup \{\mathbf{x_j}\}$
4. 输出簇划分$C = \{\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C_k}\}$

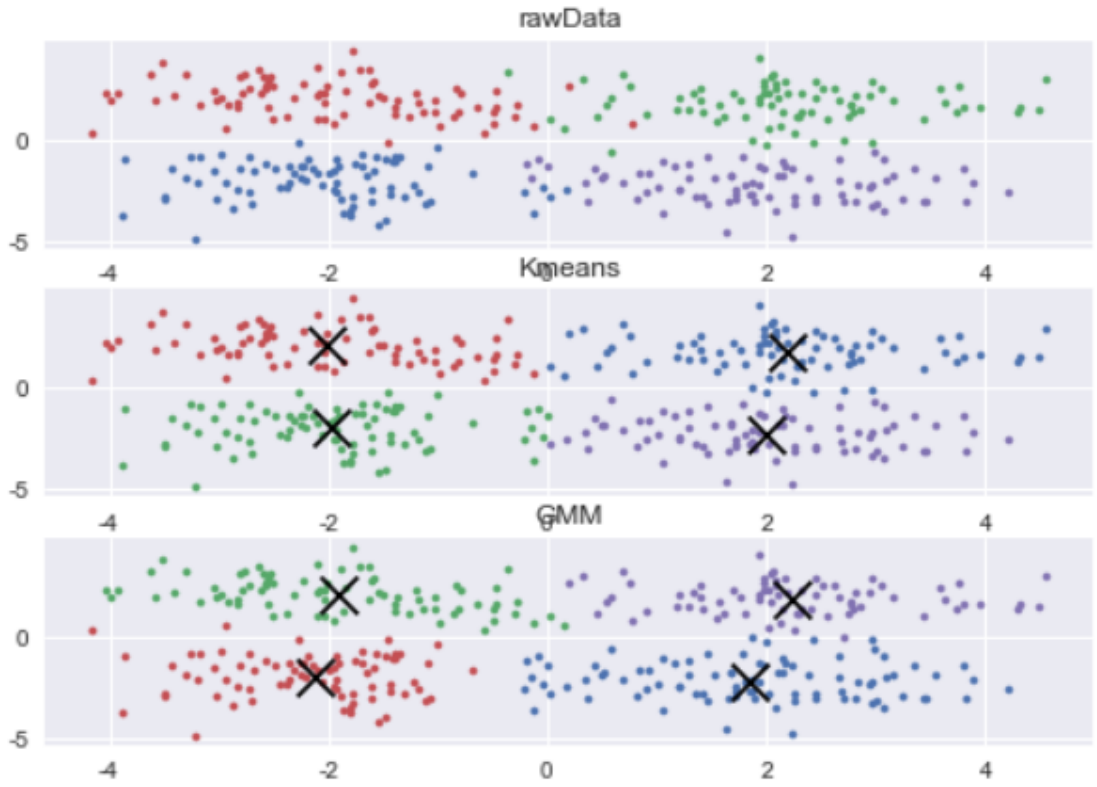
四、实验结果与分析

4.1生成数据上测试

4.1.1 K = 4 方差为1 协方差0

对于样本由二元高斯混合分布生成，方差为1，协方差0，均值分别为$(-2, -2), (-2, 2), (2, -2), (2, 2)$ ，数量都为80，据理论可知这样生成的样本是比较符合kmeans模型的。

对聚类数目为4，kmeans和gmm的分类效果和源数据比较如下，其中X为分布后的各个簇的中心



4.1.1.1 kmeans迭代次数

```
kmeans
迭代次数:  1
迭代次数:  2
迭代次数:  3
迭代次数:  4
迭代次数:  5
```

kmeans的迭代次数为5

4.1.1.2 gmm迭代次数和似然值变化

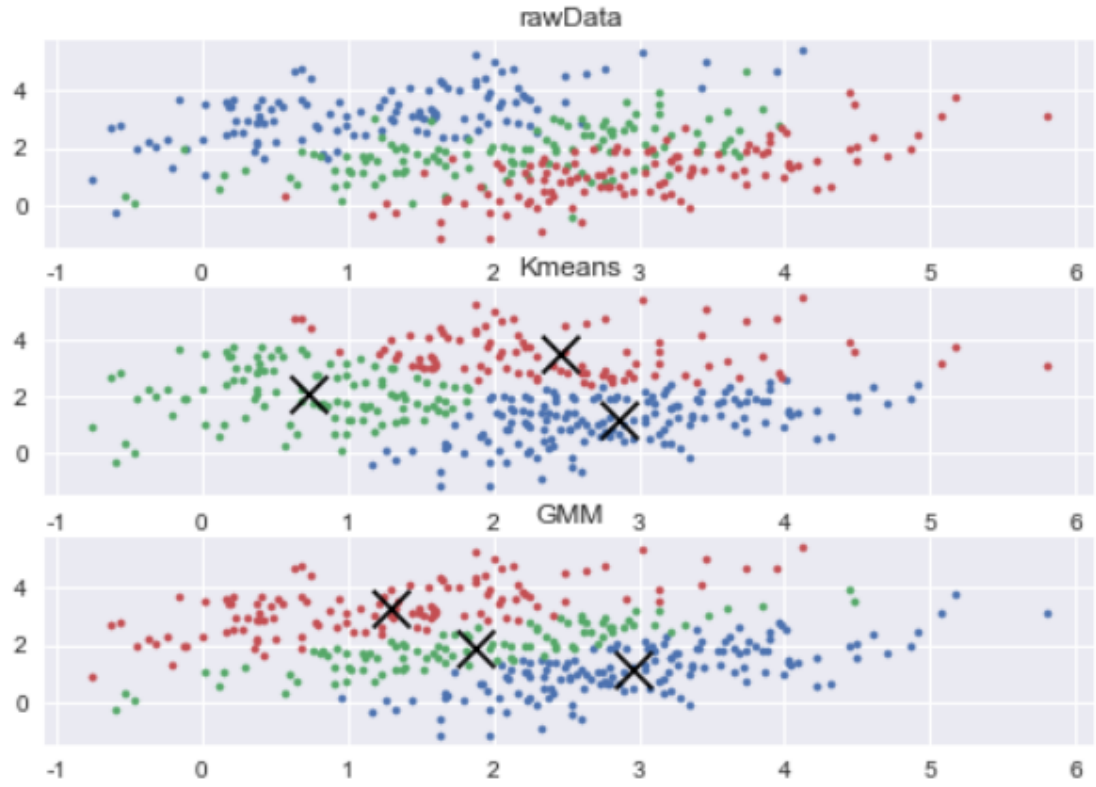
```
gmm
迭代次数：  1
当前的似然函数值：  -1295.09233465199
diff：  5.859177881032874
迭代次数：  2
当前的似然函数值：  -1291.340464293785
diff：  0.42630501840050816
迭代次数：  3
当前的似然函数值：  -1290.0772911816466
diff：  0.23310333757560267
迭代次数：  4
当前的似然函数值：  -1289.616500882246
diff：  0.12937775263746568
迭代次数：  5
当前的似然函数值：  -1289.4256092651565
diff：  0.07866497017111787
迭代次数：  6
当前的似然函数值：  -1289.337682352695
show more (open the raw output data in a text editor) ...
当前的似然函数值：  -1289.2479846123792
diff：  0.0010279938626215916
迭代次数：  20
当前的似然函数值：  -1289.2479684763036
diff：  0.0007728282182779518
```

gmm迭代次数为20，似然函数值一直增大

4.1.2 K = 3 方差为1 协方差0.5

对于样本由二元高斯混合分布生成，方差为1，协方差0.5，均值分别为$(1,3),(2,2),(3,1)$ ，数量都为120，根据理论可知这样生成的样本是比较符合gmm模型的。

对聚类数目为3，kmeans和gmm的分类效果和源数据比较如下，其中X为分布后的各个簇的中心



从上图可以看出GMM的结果更加符合样本的数据分布

4.1.2.1 kmeans迭代次数

```
kmeans
迭代次数:  1
迭代次数:  2
迭代次数:  3
迭代次数:  4
迭代次数:  5
迭代次数:  6
迭代次数:  7
迭代次数:  8
迭代次数:  9
迭代次数:  10
迭代次数:  11
迭代次数:  12
迭代次数:  13
迭代次数:  14
迭代次数:  15
迭代次数:  16
迭代次数:  17
迭代次数:  18
迭代次数:  19
迭代次数:  20
迭代次数:  21
迭代次数:  22
迭代次数:  23
```

kmans迭代次数为23

4.1.2.2 gmm迭代次数和似然值变化

```
当前的似然函数值:  -1153.5729714556148
diff:  0.0010146020253373736
迭代次数:  443
当前的似然函数值:  -1153.5728126163494
diff:  0.0009857692166648 45
```

gmm总迭代次数为443，最终似然值是-1153

4.2uci数据集上测试

这里查找的都是存在类别标签的数据，这样可以方便检验。然后由于分类后标签名字不同了，这里通过全排列找到使得精确率最高的类别。存在一些误差，但能够直观感受聚类结果。

4.2.1 irisData

这个数据集是经典的鸢尾花数据集。样本的维度是四维：

- 花萼长度
- 花萼宽度
- 花瓣长度
- 花瓣宽度

以此来预测鸢尾花属于(Setosa, Versicolour, Virginica)三类中的哪一类。

4.2.1.1 kmeans

```
kmeans
迭代次数：  1
迭代次数：  2
迭代次数：  3
迭代次数：  4
迭代次数：  5
迭代次数：  6
迭代次数：  7
Kmeans acurracy in iris：  0.8933333333333333
```

4.2.1.2 gmm

```
gmm
迭代次数：  1
当前的似然函数值：  -252.3845057641663
diff：  5.106437127059129
迭代次数：  2
当前的似然函数值：  -217.7986140177301
diff：  0.6391923263940014
迭代次数：  3
当前的似然函数值：  -198.9769383129687
diff：  0.26842322900153437
迭代次数：  4
当前的似然函数值：  -196.74889628861936
diff：  0.10717492108269845
迭代次数：  5
当前的似然函数值：  -195.3499995531327

show more (open the raw output data in a text editor) ...

diff：  0.0017885606536541259
迭代次数：  26
当前的似然函数值：  -191.0241376387101
diff：  0.0005631538586390207
GMM acurracy in iris：  0.74
```

4.2.2 Data_User_Modeling_Dataset_Hamdi Tolga KAHRAMAN

4.2.2.1 kmeans

```
kmeans
迭代次数： 1
迭代次数： 2
迭代次数： 3
迭代次数： 4
迭代次数： 5
迭代次数： 6
迭代次数： 7
迭代次数： 8
迭代次数： 9
迭代次数： 10
迭代次数： 11
迭代次数： 12
Kmeans acurracy in usermodelingdata： 0.5930232558139535
```

4.2.2.2 gmm

```
gmm
迭代次数： 1
当前的似然函数值： 79.10197218248557
diff： 5.734753300220757
迭代次数： 2
当前的似然函数值： 81.198427241871
diff： 0.028303724673218913
迭代次数： 3
当前的似然函数值： 85.74435825662725
```

show more (open the raw output data in a text editor) ..

```
diff： 0.0010072686652894734
迭代次数： 86
当前的似然函数值： 194.55572283177685
diff： 0.0008934083648892504
GMM acurracy in usermodelingdata： 0.4069767441860465
```

4.2.3 seeds_dataset

4.2.3.1 kmeans

```
kmeans
迭代次数:  1
迭代次数:  2
迭代次数:  3
迭代次数:  4
迭代次数:  5
迭代次数:  6
迭代次数:  7
迭代次数:  8
Kmeans acurracy in iris:  0.5550239234449761
```

4.2.3.2 gmm

```
show more (open the raw output data in a text editor) ...
diff:  0.0012094708917499727
迭代次数:  43
当前的似然函数值:  1275.2080056481127
diff:  0.0009518113612395562
GMM acurracy in seedsData:  0.4784688995215311
```

五、结论

- 通过实验结果可以看出gmm的随着迭代次数增加似然值一直在增大，这说明gmm算法的em算法是合理的，可以得到正确的结果
- 通过比较kmeans和gmm的迭代次数可以看出，kmeans相比gmm方法速度更快
- 在uci数据集上gmm算法的精确率不如kmeans好，这可能是因为这些数据集中存在的噪声比较大，而gmm算法在噪声比较大的环境下聚类的效果要差一些。也可能是因为选择的uci数据集更加符合kmeans假设的球状分布。
- 同时可以注意到在我选择的后两者的uci数据集上虽然似然函数的值一直在增大，但是可以看到它的值为整数。这是没问题的，因为它的似然对数是对于概率密度函数求得的。
- kmeans算法和gmm算法都比较容易受到初始的均值向量选择问题的影响，因此如果选择不好初始的簇中心值容易使之陷入局部最优解
- GMM使用EM算法进行迭代优化，因为其涉及到隐变量的问题，没有之前的完全数据，而是在不完全数据上进行。

六、参考文献

《机器学习》周志华

《统计学习方法》李航

七、附录：源代码（带注释）

lab3.py

```
from itertools import permutations
import numpy as np
```

```
from display import displayCompareResult,displayCompareRaw, displayRawData
from getdata import generateData
from gmm import GMM
from kmeans import KMeans
import pandas as pd

def readIrisData(fileName):
    df = pd.DataFrame(pd.read_csv(fileName))
    data = df.iloc[:,0:4].to_numpy()
    tag = []
    for i in range(len(df)):
        if df.iloc[i,4] == 'Iris-setosa':
            tag.append(0)
        elif df.iloc[i,4] == 'Iris-versicolor':
            tag.append(1)
        elif df.iloc[i,4] == 'Iris-virginica':
            tag.append(2)
    return data,tag

def readSeedsData(fileName):
    df = pd.DataFrame(pd.read_csv(fileName,delimiter='\t'))
    data = df.iloc[:,0:7].to_numpy()
    tag = df.iloc[:,7].to_numpy()
    return data,tag

def readUserModelingData(fileName):
    df = pd.DataFrame(pd.read_csv(fileName,delimiter='\t'))
    data = df.iloc[:,0:5].to_numpy()
    tag = []
    for i in range(len(df)):
        if df.iloc[i,5] == 'High':
            tag.append(0)
        elif df.iloc[i,5] == 'Low':
            tag.append(1)
        elif df.iloc[i,5] == 'very_low':
            tag.append(2)
        elif df.iloc[i,5] == 'Middle':
            tag.append(3)
    return data,tag
def accuracy(realLabel,predictLabel,k):
    """
    使用全排列的方式计算聚类准确率
    """
    classes = list(permutations(range(k), k))
    counts = np.zeros(len(classes))
    for i in range(len(classes)):
        for j in range(realLabel.shape[0]):
            if int(realLabel[j]) == classes[i][int(predictLabel[j])]:
                counts[i] += 1
    return np.max(counts) / realLabel.shape[0]

def myTest(data,tag,k):
    model = KMeans(data,k)
    c1,clusterCentroids1,tag1 = model.initializeRemoteK()
    # displayCompareRaw(data,c1,clusterCentroids1,title='Kmeans,accuracy=
{}'.format(accuracy(np.array(tag),tag1,len(mean))))

    model = GMM(data,k)
```

```
    c2,clusterCentroids2,tag2 = model.train()
    # displayCompareRaw(data,c2,clusterCentroids2,title='GMM,accuracy=
{}'.format(accuracy(np.array(tag),tag2,len(mean))))
    displayCompareResult(data,k,c1,c2,clusterCentroids1,clusterCentroids2,title1
='Kmeans',title2 = 'GMM')

if __name__ == '__main__':
    # 符合kmeans模型的样本
    x = 2
    mean = [np.array((-x,-x)),np.array((x,x)),np.array((-x,x)),np.array((x,-x))]
    size = [80,80,80,80]
    data = generateData(mean,0.6,2,size,len(mean))
    tag = [int(i/80) for i in range(sum(size))]
    myTest(data,tag,len(mean))

    # 符合gmm模型的样本
    mean = [np.array((1,3)),np.array((2,2)),np.array((3,1))]
    size = [120,120,120]
    data = generateData(mean,0.6,1,size,len(mean))
    tag = [int(i/80) for i in range(sum(size))]
    myTest(data,tag,3)

    # #鸢尾花数据集聚类
    # irisData,irisTag = readIrisData('iris.csv')
    # model = KMeans(irisData,3)
    # c1,clusterCentroids1,tag1 = model.initializeRemoteK()
    # print('Kmeans acurracy in iris: ',accuracy(np.array(irisTag),tag1,3))
    # model = GMM(irisData,3)
    # c2,clusterCentroids2,tag2 = model.train()
    # print('GMM acurracy in iris: ',accuracy(np.array(irisTag),tag2,3))

    # #种子数据集
    # data,tag = readSeedsData('seeds_dataset.txt')
    # print(data,tag)
    # model = KMeans(data,3)
    # c1,clusterCentroids1,tag1 = model.initializeRemoteK()
    # print('Kmeans acurracy in  seedsData: ',accuracy(np.array(tag),tag1,3))
    # model = GMM(data,3)
    # c2,clusterCentroids2,tag2 = model.train()
    # print('GMM acurracy in seedsData: ',accuracy(np.array(tag),tag2,3))

    # #DataUserModelingData
    # data,tag = readUserModelingData('Data_User_Modeling_Dataset_Hamdi Tolga
KAHRAMAN.csv')
    # print(np.array(data).shape,np.array(tag).shape)
    # model = KMeans(data,4)
    # c1,clusterCentroids1,tag1 = model.initializeRemoteK()
    # print('Kmeans acurracy in
usermodelingdata: ',accuracy(np.array(tag),tag1,4))
    # model = GMM(data,4)
    # c2,clusterCentroids2,tag2 = model.train()
    # print('GMM acurracy in usermodelingdata: ',accuracy(np.array(tag),tag2,4))
```

kmeans.py

```
import numpy as np
import random
```

```
import collections

"""
kmeans方法划分聚类，如果选择聚类数过多而样本难以分出这么多类就会出现错误，因为这里没有考虑丢弃k
"""
class KMeans(object):
    def __init__(self,data,k,delta = 1e-7) -> None:
        """
        data：数据
        k：聚类数
        """
        self.data = data
        self.k = k
        self.delta = delta
        self.tag = np.zeros(self.data.shape[0])
        self.dataSize = len(data)

    def euclideanDistance(self,x,y):
        """
        计算欧式距离
        """
        return np.linalg.norm(x-y)

    def train(self):
        """
        训练
        """
        print('kmeans')
        times = 0
        c = collections.defaultdict(list)
        while True:
            c = collections.defaultdict(list)
            for i in range(self.dataSize):
                self.tag[i] =
np.argmin([self.euclideanDistance(self.data[i],self.clusterCentroids[j]) for j
in range(self.k)])
                c[self.tag[i]].append(self.data[i])
            newClusterCentroids = [np.mean(c[i],axis=0) for i in range(self.k)]
            times += 1
            print('迭代次数：',times)
            if
self.euclideanDistance(np.array(self.clusterCentroids),np.array(newClusterCentro
ids)) < self.delta:
                break
            else:
                self.clusterCentroids = newClusterCentroids
        return c,self.clusterCentroids,self.tag

    def initializeRemoteK(self):
        """
        从数据集中首先随机选择一个样本点作为初始均值向量，
        然后总是选择与当前样本最远的样本点作为下一个均值向量
        """
        self.clusterCentroids = []

 self.clusterCentroids.append(self.data[np.random.randint(0,self.dataSize)])
        for i in range(1,self.k):
```

```
            maxDistance =
np.sum([self.euclideanDistance(self.data[0],self.clusterCentroids[k]) for k in
range(i)])
            temp = 0
            for j in range(self.dataSize):
                newMaxDistance =
np.sum([self.euclideanDistance(self.data[j],self.clusterCentroids[k]) for k in
range(i)])
                if maxDistance < newMaxDistance:
                    maxDistance = newMaxDistance
                    temp = j
            self.clusterCentroids.append(self.data[temp])
        return self.train()

    def randomInitializeK(self):
        """
        从数据集中随机选择k个样本作为初始均值向量
        这个总是出现nan问题不好用，不用这个了
        """
        self.clusterCentroids =
np.array(random.sample(self.data.tolist(),self.k))
        return self.train()
```

gmm.py

```
import collections
import numpy as np
from scipy.stats import multivariate_normal

class GMM(object):

    def __init__(self,data,k,delta=1e-3) -> None:
        self.data = data
        self.k = k
        self.delta = delta
        self.dataSize = data.shape[0]
        self.dim = data.shape[1]
        self.initializeParameter()

    def initializeParameter(self):
        self.alpha = [1/self.k for i in range(self.k)]
        self.sigma = [np.eye(self.dim) for i in range(self.k)]
        self.initializeMu()
        self.__lastAlpha = np.array(self.alpha) #保存上一次的混合系数
        self.__lastMu = np.array(self.mu) #保存上一次的均值
        self.__lastSigma = np.array(self.sigma) #保存上一次的协方差矩阵
        print('initial:',self.mu)

    def __likelihood(self):
        """
        计算似然值
        """
        total = 0
        for j in range(self.dataSize):
            total +=
np.log(np.sum([self.alpha[i]*multivariate_normal.pdf(self.data[j],self.mu[i],sel
f.sigma[i]) for i in range(self.k)]))
```

```
        return total

    def __eStep(self):
        """
        EM算法E步
        计算样本中的数据由各混合成分生成的后验概率
        使用multivariate_normal.pdf函数计算多元正态分布的值
        """
        self.gamma = np.zeros((self.dataSize,self.k))
        for j in range(self.dataSize):
            total =
np.sum([self.alpha[i]*multivariate_normal.pdf(self.data[j],self.mu[i],self.sigma
[i]) for i in range(self.k)])
            for i in range(self.k):
                self.gamma[j][i] =
self.alpha[i]*multivariate_normal.pdf(self.data[j],self.mu[i],self.sigma[i])\
                    /total

    def __mStep(self):
        """
        EM算法M步
        """
        for i in range(self.k):
            gamma = np.expand_dims(self.gamma[:, i], axis=1)
            self.mu[i] = np.sum(gamma*self.data,axis=0)/gamma.sum()
            self.sigma[i] =  (self.data - self.mu[i]).T.dot((self.data -
self.mu[i]) * gamma)/ gamma.sum()
            self.alpha[i] = gamma.sum() / self.dataSize

    def __converged(self):
        """
        用来判断是否收敛
        """
        difference = self.euclideanDistance(np.array(self.mu),self.__lastMu)\
            +self.euclideanDistance(np.array(self.sigma),self.__lastSigma)\
            +self.euclideanDistance(np.array(self.alpha),self.__lastAlpha)
        print('diff: ',difference)
        if difference > self.delta :
            self.__lastAlpha = np.array(self.alpha) #保存上一次的混合系数
            self.__lastMu = np.array(self.mu) #保存上一次的均值
            self.__lastSigma = np.array(self.sigma) #保存上一次的协方差矩阵
            return False
        else:
            return True

    def train(self):
        print('gmm')
        times = 0
        while True:
            self.__eStep()
            self.__mStep()
            times+=1
            print('迭代次数: ',times)
            print('当前的似然函数值: ',self.__likelihood())
            if self.__converged():
                break
        c = collections.defaultdict(list)
        for j in range(self.dataSize):
```

```
            c[np.argmax(self.gamma[j,:])].append(self.data[j])
        self.tag = np.zeros(self.dataSize)
        for j in range(self.dataSize):
            self.tag[j] = np.argmax(self.gamma[j,:])
        return c,self.mu,self.tag

    def euclideanDistance(self,x,y):
        """
        计算欧式距离
        """
        return np.linalg.norm(x-y)

    def initializeMu(self):
        """
        从数据集中首先随机选择一个样本点作为初始均值向量，
        然后总是选择与当前样本最远的样本点作为下一个均值向量
        """
        self.mu = []
        self.mu.append(self.data[np.random.randint(0,self.dataSize)])
        for i in range(1,self.k):
            maxDistance =
np.sum([self.euclideanDistance(self.data[0],self.mu[k]) for k in range(i)])
            temp = 0
            for j in range(self.dataSize):
                newMaxDistance =
np.sum([self.euclideanDistance(self.data[j],self.mu[k]) for k in range(i)])
                if maxDistance < newMaxDistance:
                    maxDistance = newMaxDistance
                    temp = j
            self.mu.append(self.data[temp])
```

getdata.py

```
import numpy as np
import matplotlib.pyplot as plt

def generateData(mean,cov_xy,var,size,K=4):
    """
    生成指定的二元正态分布的数据
    mean：均值点
    cov_xy：协方差
    var：独立同分布的每个随机变量的方差
    K：生成的样本的类数
    size：数据量，是一个int类型数据
    """
    cov = [[var,cov_xy],[cov_xy,var]]
    sampledata = []
    for i in range(len(mean)):
        sampledata.append(np.random.multivariate_normal(mean[i], cov, size[i]))
    totalSize = np.sum(size)
    return np.array(sampledata).reshape(totalSize ,2)

if __name__ == '__main__':
    mean = [np.array((1,1)),np.array((4,4)),np.array((1,4)),np.array((4,1))]
    data = generateData(mean,0,1,len(mean))
    plt.scatter(data[:,0],data[:,1],'black')
    plt.show()
```

display.py

```
import matplotlib.pyplot as plt
import numpy as np



plt.rcParams['axes.facecolor']='snow'
plt.rcParams['font.sans-serif'] = ['SimHei']  #显示中文
plt.rcParams['axes.unicode_minus']=False #用来正常显示负号

def displayCompareRaw(rawData,c,clusterCentroids,title=''):
    """
    函数功能：展示聚类结果和原始数据对比
    """
    plt.style.use("seaborn")
    plt.subplot(211)
    plt.title('raw')
    displayRawData(rawData)
    plt.subplot(212)
    for i in range(len(c)):
        x = np.array(c[i])[:,0]
        y = np.array(c[i])[:,1]
        plt.scatter(x,y,marker=".", s=40)
        plt.scatter(clusterCentroids[i][0],clusterCentroids[i]
[1],marker='x',color='black')
    plt.title(title)
    plt.show()

def
displayCompareResult(rawData,k,c1,c2,clusterCentroids1,clusterCentroids2,title1,
title2):
    """
    函数功能:展现kmeans和gmm的对比结果
    """
    plt.style.use("seaborn")
    plt.subplot(311)
    plt.title('rawData')
    temp = 0
    sliceTemp = int(len(rawData)/k)
    postTemp = temp+sliceTemp
    for i in range(k):
        plt.scatter(rawData[temp:postTemp,:][:,0],rawData[temp:postTemp,:]
[:,1],marker='.',s=40)
        temp += sliceTemp
        postTemp = postTemp + sliceTemp
    plt.subplot(312)
    plt.title(title1)
    for i in range(len(c1)):
        x = np.array(c1[i])[:,0]
        y = np.array(c1[i])[:,1]
        plt.scatter(x,y,marker=".", s=40)
        plt.scatter(clusterCentroids1[i][0],clusterCentroids1[i]
[1],marker='x',s=250,color='black')
    plt.subplot(313)
    plt.title(title2)
```

```
    for i in range(len(c2)):
        x = np.array(c2[i])[:,0]
        y = np.array(c2[i])[:,1]
        plt.scatter(x,y,marker=".", s=40)
        plt.scatter(clusterCentroids2[i][0],clusterCentroids2[i]
[1],marker='x',s=250,color='black')
    plt.show()

def displayRawData(rawData,k):
    """
    函数功能：展示原始数据
    """
    plt.title('原始数据')
    temp = 0
    sliceTemp = int(len(rawData)/k)
    postTemp = temp+sliceTemp
    for i in range(k):
        plt.scatter(rawData[temp:postTemp,:][:,0],rawData[temp:postTemp,:]
[:,1],marker='.',s=40)
        temp += sliceTemp
        postTemp = postTemp + sliceTemp
    plt.show()
```