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： GMM 模型

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

实验要求：

（1）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）。

（2）用 k-means 聚类，测试效果；

（3）用混合高斯模型和实现的 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（与设定的结果比较）。

（4）应用：在 UCI 上找一个简单问题的数据，用实现的 GMM 进行聚类。

实验环境：

硬件环境：x64CPU

软件环境：pycharm

库版本号：anaconda python==3.6, numpy==1.17.0

三、设计思想（本程序中的用到的主要算法及数据结构）

3.1 K-Means 算法

K-Means 算法是一种经典的聚类算法，通过无监督学习的方法对样本数据进行聚类。算法的流程为：

1) 初始化 k 个聚类中心 $\mu_1, \mu_2, ..., \mu_k$（可以从输入样本中随机选择）。

2) 对于样本中的其他点 $\mathbf{x}_i$，分别计算该点到 k 个中心的距离，将这个点分入距离它最小的中心所在类。若该中心为 μ_j，则将 $\mathbf{x}_i$ 标记为 j，分入集合 C_j。

3) 所有点都已分类后，对于每个类别，重新计算其聚类中心 $\mu_k = \frac{1}{C_k}\sum_{\mathbf{x}\in C_k}\mathbf{x}$，即为该类中所有样本的质心。

4) 重复(2)，(3)步骤直到每一类中的样本都不再发生变化。

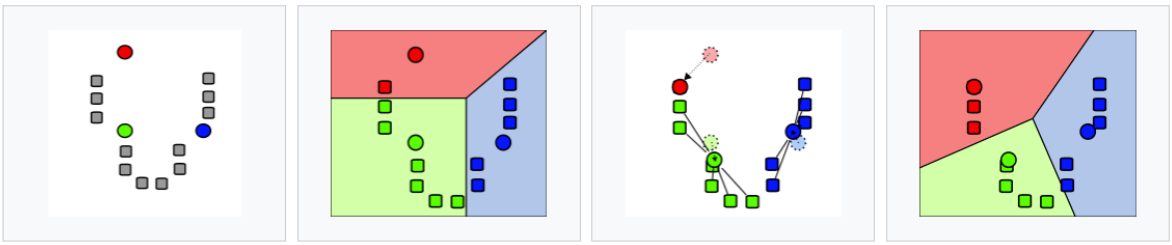

图 1 K-Means 迭代过程

3.2 高斯混合模型（GMM）

高斯混合模型是高斯分布通过线性组合进行叠加形成的。一些模型难以用一个独立的

分布来表示，但却能够很好地契合多个分布线性叠加的形式，这便是引入高斯混合模型的动机。混合高斯模型的数学形式为：

$$p(\mathbf{x})=\sum_{k=1}^{K}\pi_k N(\mathbf{x}\mid\mu_k,\Sigma_k)$$

其中，$N(\mathbf{x}\mid\mu_k,\Sigma_k)$ 是混合模型中每个高斯分布的概率密度，π_k 称为其混合系数，满足：

$$0\le\pi_k\le 1$$

$$\sum_{k=1}^{K}\pi_k=1$$

接下来我们从离散隐变量的角度来描述高斯混合模型。我们引入一个 K 维随机变量 z，其中有一个特定的元素 z_k 等于 1，其余元素等于 0。也就是说，根据哪个元素非零，z 有 K 个可能的状态。若根据高斯混合模型的混合比对 z 的边缘概率分布进行赋值，则有：

$$p(z_k=1)=\pi_k=\prod_{k=1}^{K}\pi_k^{z_k}$$

因此，π_k 可以被看作是 $z_k=1$ 的先验概率。给定 z 的取值，$\mathbf{x}$ 的条件分布即为对应的高斯分布：

$$p(\mathbf{x}\mid z_k=1)=N(\mathbf{x}\mid\mu_k,\Sigma_k)=\prod_{k=1}^{K}N(\mathbf{x}\mid\mu_k,\Sigma_k)^{z_k}$$

可见，给定一个 z，即可确定相应的 $\mathbf{x}$ 的分布，每一个样本点 $\mathbf{x}_i$ 都对应一个隐变量 z。隐变量潜在地影响着样本点的分布（在 GMM 中体现为不同高斯分布的混合比）。

由此，可以得到 $\mathbf{x}$ 和 z 的联合分布 $p(\mathbf{x},z)=p(z)p(\mathbf{x}\mid z)$，进而得到 $\mathbf{x}$ 的边缘分布：

$$\begin{aligned}p(\mathbf{x})&=\sum_{k=1}^{K}p(z_k=1)p(\mathbf{x}\mid z_k=1)\\&=\sum_{k=1}^{K}\pi_k N(\mathbf{x}\mid\mu_k,\Sigma_k)\end{aligned}$$

我们发现，$\mathbf{x}$ 的边缘分布正是混合高斯的形式。似乎引入隐变量 z 并没有太大的作用。但是事实上，在采用 EM 算法时，隐变量 z 的存在可以大大简化计算。

在 GMM 中，还有一个起重要作用的量，即为给定 $\mathbf{x}$ 下 z 的条件概率，用 $\gamma(z_k)$ 表示。$\gamma(z_k)$ 是观测到 $\mathbf{x}$ 后，$z_k=1$ 的后验概率，可以被看作是分量 k 对于观测值 $\mathbf{x}$ 的“责任”。其值可以利用贝叶斯定理求出：

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

3.3 EM 算法

EM 算法即为期望最大化算法，分为两个步骤，分别是期望步骤（expectation step）和最大化步骤（maximum step）。在期望步骤中，根据参数的假设值，计算未知变量的期望估计；在最大化步骤中，根据未知变量的估计值，给出当前参数的极大似然估计。

期望步骤：

$$compute:\ p(z\mid \mathbf{x},\theta^{old})$$

最大化步骤：

$$\theta^{new}=\arg\max_{\theta}Q(\theta,\theta^{old})$$

$$Q(\theta,\theta^{old})=\sum_{z}p(z\mid \mathbf{x},\theta^{old})\ln p(\mathbf{x},z\mid \theta)$$

下面对混合高斯模型中的 EM 算法进行推导。

首先求出对数似然函数，

$$p(\mathbf{x},z\mid \mu,\Sigma,\pi)=\sum_{n=1}^{N}\ln\sum_{k=1}^{K}\pi_k^{z_{nk}}N(\mathbf{x}_n\mid \mu_k,\Sigma_k)^{z_{nk}}$$

使其关于 μ_k 导数为 0 得，

$$\mu_k=\frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})\mathbf{x}_n$$

$$N_k=\sum_{n=1}^{N}\gamma(z_{nk})$$

使其关于 Σ_k 导数为 0 得，

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

对 π_k 进行优化，需要考虑到约束条件 $\sum_{k=1}^{K}\pi_k=1$，因此使用拉格朗日乘数法进行优化：

$$\ln p(\mathbf{x}\mid \mu,\Sigma,\pi)+\lambda(\sum_{k=1}^{K}\pi_k-1)$$

使其关于 π_k 导数为 0 得，

$$\pi_k=\frac{N_k}{N}$$

以上即为 GMM 的对数似然函数和最大似然估计的过程，将其描述为 EM 算法：

1) 初始化 μ_k, Σ_k, π_k，可以用 K-Means 的聚类结果对其进行优化。

2) E 步骤：使用当前的参数值计算责任 $\gamma(z_k) = \dfrac{\pi_k N(\mathbf{x}_n \mid \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(\mathbf{x}_n \mid \mu_j, \Sigma_j)}$

3) M 步骤：使用当前的责任重新估计参数

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n$$

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

$$\pi_k^{new} = \frac{N_k}{N}$$

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk})$$

4) 计算对数似然函数 $p(\mathbf{x}, z \mid \mu, \Sigma, \pi) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \pi_k^{z_{nk}} N(\mathbf{x}_n \mid \mu_k, \Sigma_k)^{z_{nk}}$

5) 若对数似然函数还未满足收敛准则，则重复(2), (3), (4)步骤。

四、实验结果与分析

4.1 生成数据

生成数据的方法是根据给定均值与协方差矩阵的高斯分布，随机生成 K 个类别的数据，代码如下所示：

```python
def generate_data(mu, sigma, num_class, num_sample):
    """
    generate data of k classes, and data in each class satisfy gauss distribution
    :param mu: mean
    :param sigma: variance
    :param num_class: number of classes (k)
    :param num_sample: number of samples in each class
    :return: generated data
    """
    assert len(mu) == len(sigma)
    assert len(mu[0]) == len(sigma[0])
    assert len(sigma) == num_class
    assert len(num_sample) == len(sigma)
    class_id = 0
    x_dim = len(mu[0])
    data = []
    for num in num_sample:
        for i in range(num):
            x = generate_gauss_data(mu[class_id], sigma[class_id], x_dim)
            x.append(class_id)
            data.append(x)
        class_id += 1
    return data
```

图 2 生成数据部分代码

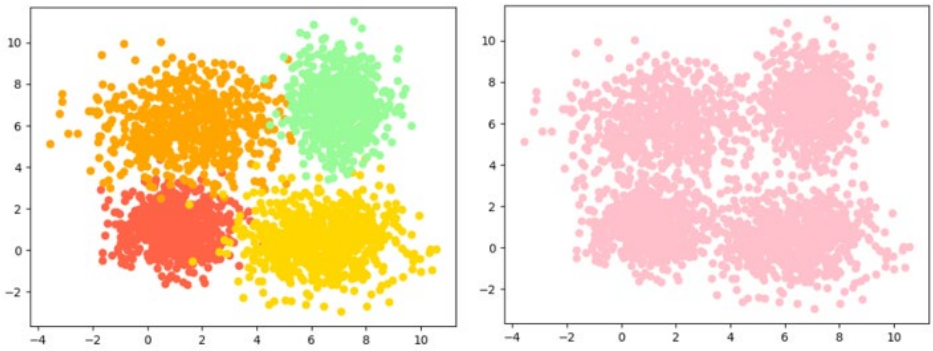

图 3 生成数据可视化

4.2 K-Means

4.2.1 数据分布对聚类结果的影响

选取多组分布不同的数据，测试 K-Means 的聚类结果。

1. 四类二维数据，每类的均值和方差分别如下所示：

$$\mu_1^1 = 1, \sigma_1^1 = 1; \mu_2^1 = 1, \sigma_2^1 = 1$$
$$\mu_1^2 = 1.5, \sigma_1^2 = 1.7; \mu_2^2 = 6, \sigma_2^2 = 1.4$$
$$\mu_1^3 = 6.5, \sigma_1^3 = 1.5; \mu_2^3 = 0.5, \sigma_2^3 = 1.2$$
$$\mu_1^4 = 7, \sigma_1^4 = 0.9; \mu_2^4 = 7, \sigma_2^4 = 1.3$$

可视化分布情况和聚类结果如下所示，“+”代表 K-Means 选取的聚类中心：

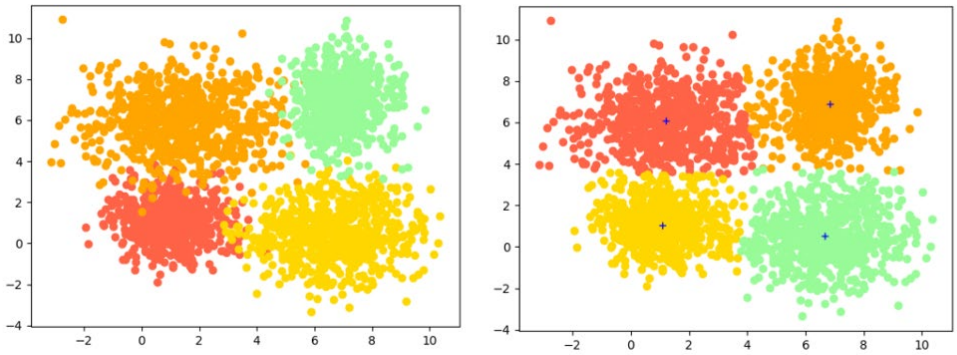

图 4 第一组数据及其聚类结果

2. 五类二维数据，每类的均值和方差分别如下所示：

$$\mu_1^1 = 0, \sigma_1^1 = 1; \mu_2^1 = 2, \sigma_2^1 = 1$$
$$\mu_1^2 = 1, \sigma_1^2 = 1; \mu_2^2 = 3, \sigma_2^2 = 1$$
$$\mu_1^3 = 3, \sigma_1^3 = 1; \mu_2^3 = 2, \sigma_2^3 = 1$$
$$\mu_1^4 = 3, \sigma_1^4 = 1; \mu_2^4 = 4, \sigma_2^4 = 1$$
$$\mu_1^5 = 4, \sigma_1^5 = 1; \mu_2^5 = 5, \sigma_2^5 = 1$$

可视化分布情况和聚类结果如下所示：

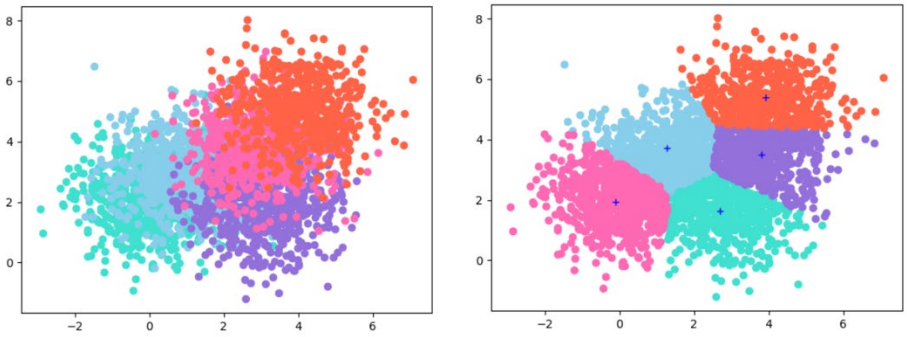

图 5 第二组数据及其聚类结果

3. 三类二维数据，每类的均值和方差分别如下所示：

$$\mu_1^1 = 1, \sigma_1^1 = 3; \mu_2^1 = 1, \sigma_2^1 = 3$$
$$\mu_1^2 = 1.5, \sigma_1^2 = 1.5; \mu_2^2 = 0.5, \sigma_2^2 = 0.5$$
$$\mu_1^3 = 6, \sigma_1^3 = 6; \mu_2^3 = 2, \sigma_2^3 = 2$$

可视化分布情况和聚类结果如下所示：

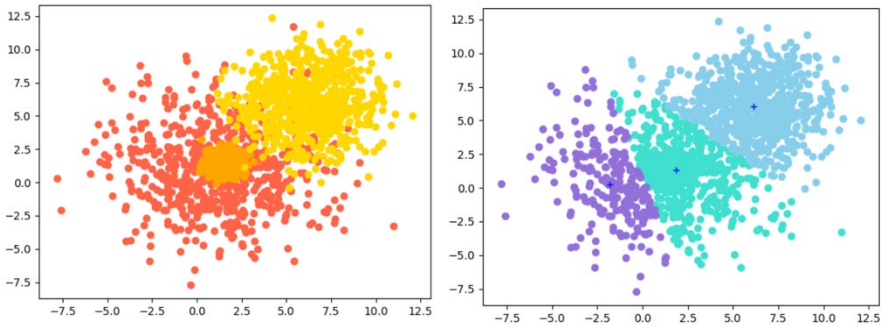

图 6 第三组数据及其聚类结果

4. 四类二维数据，每类的均值和方差分别如下所示：

$$\mu_1^1 = 1, \sigma_1^1 = 0.5; \mu_2^1 = 1, \sigma_2^1 = 0.5$$
$$\mu_1^2 = 2.5, \sigma_1^2 = 0.5; \mu_2^2 = 2.5, \sigma_2^2 = 0.5$$
$$\mu_1^3 = 6, \sigma_1^3 = 0.6; \mu_2^3 = 4, \sigma_2^3 = 0.6$$
$$\mu_1^4 = 5, \sigma_1^4 = 0.5; \mu_2^4 = 10, \sigma_2^4 = 0.5$$

可视化分布情况和聚类结果如下所示：

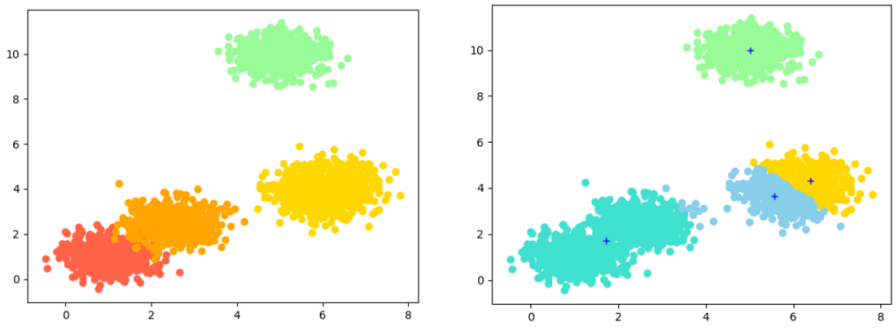

图 7 第四组数据及其聚类结果

从四组数据的聚类结果可以看出，当高斯分布比较分散时，K-Means 可以对数据较好地聚类；但如果多个高斯分布重叠在一起，K-Means 的聚类效果与原分布相差较大。这是因为 K-Means 基于每个样本点与聚类中心的欧氏距离进行聚类，二维情况下的决策面实际上是两个聚类中心的垂直平分线，仍然是一个线性决策面。但不同的高斯分布是非线性可分的，因此 K-Means 基于欧氏距离决策方案并不符合高斯分布的数据特点，当不同类数据耦合度较高时，K-Means 就很难对它们很好地进行聚类。

4.2.2 初始化对聚类结果的影响

初始点的选取对聚类结果有很大的影响，一个好的初始化（初始聚类中心相距较远或与原始分布的均值点距离较近）会使得最终聚类结果更加接近原始分布。采用下面这组四类数据进行实验：

$$\mu_1^1 = 1, \sigma_1^1 = 0.5; \mu_2^1 = 1, \sigma_2^1 = 0.5$$
$$\mu_1^2 = 2.5, \sigma_1^2 = 0.5; \mu_2^2 = 2.5, \sigma_2^2 = 0.5$$
$$\mu_1^3 = 6, \sigma_1^3 = 0.6; \mu_2^3 = 4, \sigma_2^3 = 0.6$$
$$\mu_1^4 = 5, \sigma_1^4 = 0.5; \mu_2^4 = 10, \sigma_2^4 = 0.5$$

数据分布可视化如下：

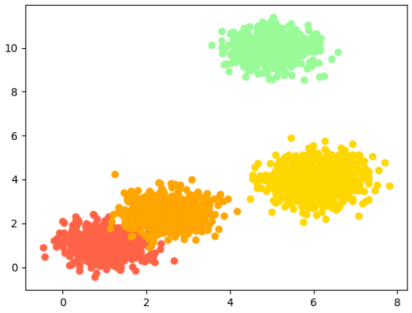

图 8 初始化实验的数据分布

这组数据的特点是有两类数据的分布相距较近，另外两类数据的分布相距较远，K-Means 很容易将相距较近的两类数据归为一类。我们随机选取四组不同的初始化在这组数据上运行 K-Means 算法：

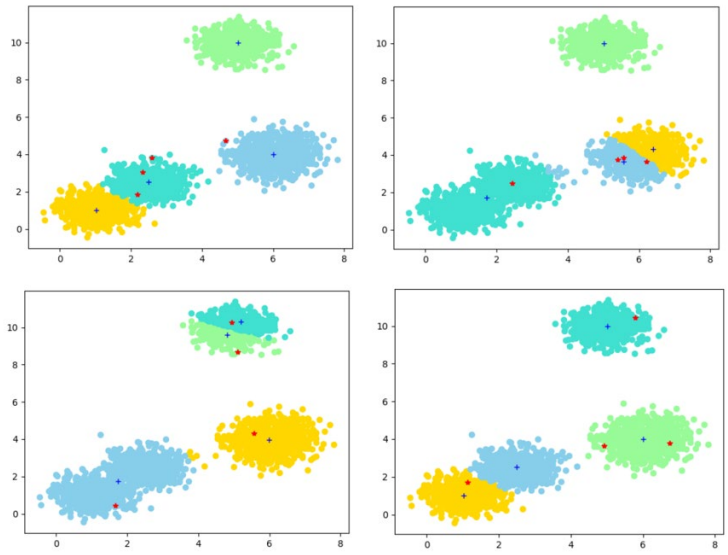

图 9 四组不同初始化下的聚类结果

红色“*”表示初始化的聚类中心，蓝色“+”表示最终的聚类中心。通过对这四组聚类结果的观察，可以推测在利用 K-Means 算法时，初始聚类中心相距较远、与原始分布的均值点距离较近时，往往可以迭代出比较接近原始分布的聚类结果。

4.2.3 数据量对聚类结果的影响

生成一组类别之间距离较近但重叠部分较小的三类二维数据：

$$\mu_1^1 = 1, \sigma_1^1 = 3; \mu_2^1 = 1, \sigma_2^1 = 3$$
$$\mu_1^2 = 6, \sigma_1^2 = 2; \mu_2^2 = 4, \sigma_2^2 = 2$$
$$\mu_1^3 = 10, \sigma_1^3 = 2.5; \mu_2^3 = 10, \sigma_2^3 = 2$$

分别在数据量为每类 10、50、100、500 个样本的条件下进行实验，结果如下所示：

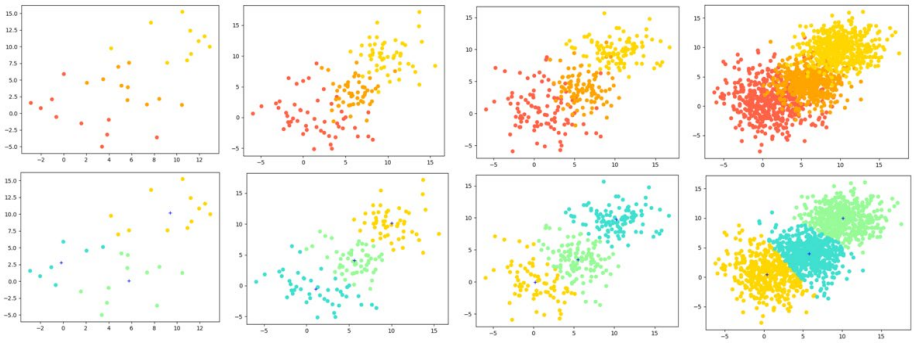

图 10 不同数据量下的聚类结果

上面是原始数据的分布，下面是聚类的结果，可以发现当数据量为每类 10 个样本时，K-Means 的聚类效果并不好，这是因为不同类别的数据间距不大，在数据量较小的情况下，不同类的数据并不能很好地通过距离来分辨；当数据量逐渐增大时，数据分布的特点逐渐显露，同类的样本点有更大的几率挨在一起，也因此更容易通过欧氏距离的大小区分数据类别，使得 K-Means 聚类效果逐渐变好。

总而言之，K-Means 聚类的实质就是通过欧几里得距离对样本点进行划分，各种因素对 K-Means 聚类效果的影响根本上就是样本点之间欧几里得距离大小关系的改变。

4.2.4 聚类过程可视化

为了更直观地理解 K-Means 算法的聚类过程，我们对以下数据进行聚类，每轮聚类后都对数据进行一次可视化，观察 K-means 的迭代更新过程。

使用的四类二维数据的均值和方差如下所示：

$$\mu_1^1 = 1, \sigma_1^1 = 1; \mu_2^1 = 1, \sigma_2^1 = 1$$
$$\mu_1^2 = 0.5, \sigma_1^2 = 1.4; \mu_2^2 = 5, \sigma_2^2 = 1.4$$
$$\mu_1^3 = 6, \sigma_1^3 = 1.5; \mu_2^3 = 1.5, \sigma_2^3 = 1.2$$
$$\mu_1^4 = 5, \sigma_1^4 = 1.3; \mu_2^4 = 5.5, \sigma_2^4 = 1.4$$

数据原始分布如下所示：

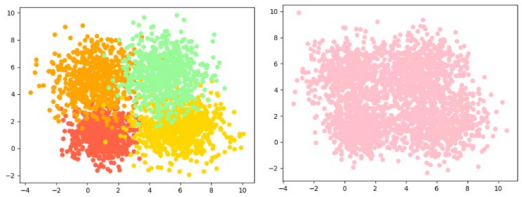

图 11 数据原始分布

聚类过程如下所示：

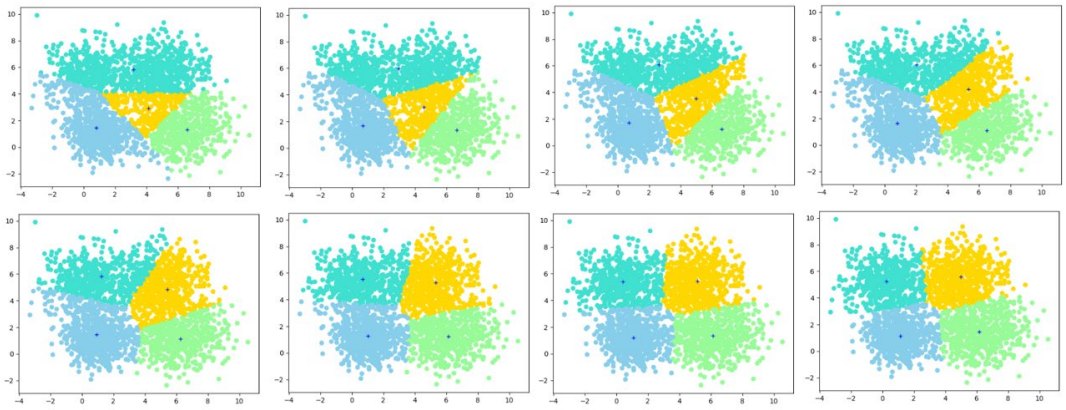

图 12 聚类过程可视化

从图示可以看出，K-Means 就是在一轮一轮的迭代中更新每一类的聚类中心，计算欧氏距离，逐步改善聚类结果。

4.3 GMM

4.3.1 数据分布对聚类结果的影响

选取几组数据之间重叠性比较高的数据分布，对 K-Means 和 GMM 的聚类结果进行比较。其中 GMM 用 K-Means 的聚类结果进行初始化。具体而言，均值被初始化为 K-Means 得到的对应类别的聚类中心，协方差矩阵被初始化为聚类后对应类别样本的协方差，混合系数被初始化为对应类别中数据点所占比例。

```
def init_gmm_with_kmeans(center, cluster, num_class, num_sample):
    mu_init = center
    sigma_init = []
    pi_init = np.zeros(num_class)
    classify = []
    for i in range(num_class):
        classify.append([])
    for r in cluster:
        classify[int(r[-1])].append(r[:-1])
    for k in range(num_class):
        if len(classify[k]) == 1:
            dim = len(classify[k][0])
            sigma_init.append(np.zeros((dim, dim)))
        else:
            sigma_init.append(np.cov(classify[k], rowvar=False))
        pi_init[k] = len(classify[k]) / num_sample
    sigma_init = np.array(sigma_init)
    return mu_init, sigma_init, pi_init
```

图 13 GMM 初始化部分代码

下面对每组数据的聚类结果进行分析：

1. 两类二维数据，均值和方差如下所示：

$$\mu_1^1 = 1, \sigma_1^1 = 3; \mu_2^1 = 1, \sigma_2^1 = 3$$

$$\mu_1^2 = 1.5, \sigma_1^2 = 0.5; \mu_2^2 = 1.5, \sigma_2^2 = 0.5$$

数据原始分布、K-Means、GMM 聚类结果分别

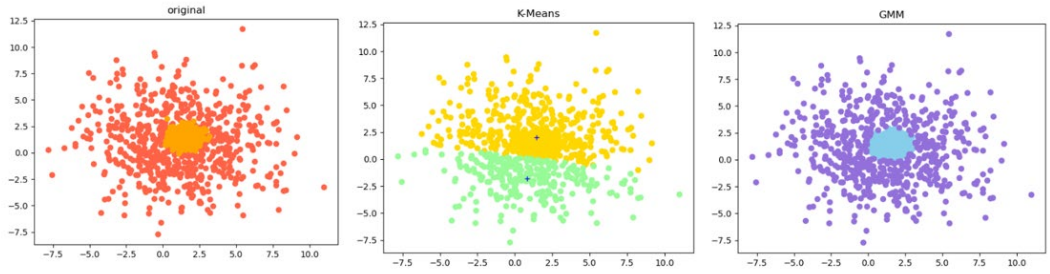

图 14 第一组数据分布与 K-Means, GMM 聚类结果

GMM 预测分布结果如下所示：

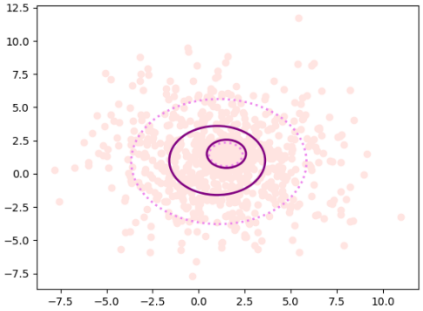

图 15 第一组数据 GMM 预测分布

2. 三类二维数据，均值和方差如下所示：

$$\mu_1^1 = 1, \sigma_1^1 = 3; \mu_2^1 = 1, \sigma_2^1 = 3$$

$$\mu_1^2 = 3, \sigma_1^2 = 2; \mu_2^2 = 3, \sigma_2^2 = 2$$

$$\mu_1^3 = 1.1, \sigma_1^3 = 0.1; \mu_2^3 = 1.1, \sigma_2^3 = 0.1$$

数据原始分布、K-Means、GMM 聚类结果分别如下所示：

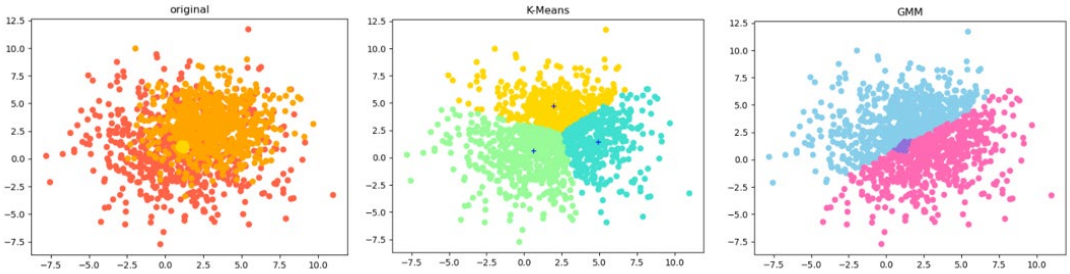

图 16 第二组数据分布与 K-Means, GMM 聚类结果

GMM 预测分布结果如下所示，深色实线代表原高斯分布，浅色虚线代表 GMM 学习到的分布：

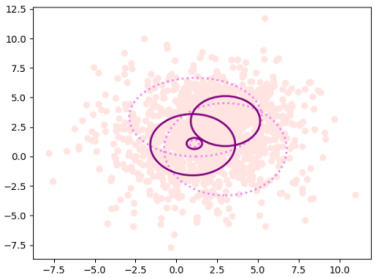

图 17 第二组数据 GMM 预测分布

3. 四类二维数据，均值和方差如下所示：

$$\mu_1^1 = 1, \sigma_1^1 = 1; \mu_2^1 = 1, \sigma_2^1 = 1$$
$$\mu_1^2 = 1.5, \sigma_1^2 = 1; \mu_2^2 = 3, \sigma_2^2 = 1$$
$$\mu_1^3 = 4, \sigma_1^3 = 1; \mu_2^3 = 1.5, \sigma_2^3 = 1$$
$$\mu_1^4 = 3.75, \sigma_1^4 = 1; \mu_2^4 = 4, \sigma_2^4 = 1$$

数据原始分布、K-Means、GMM 聚类结果分别如下所示：

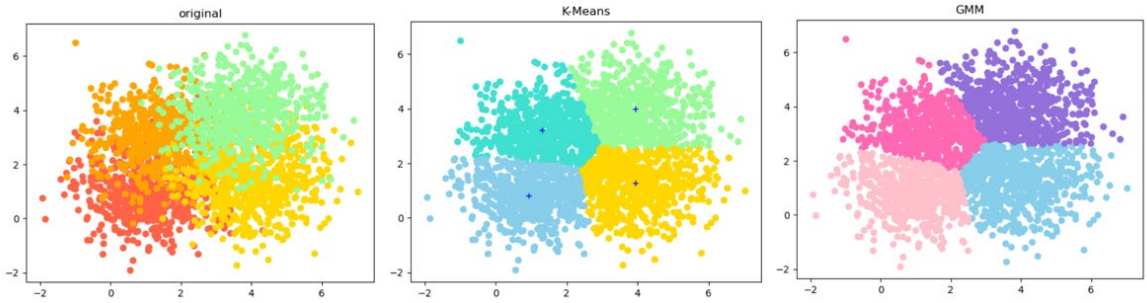

图 18 第三组数据分布与 K-Means, GMM 聚类结果

GMM 预测分布结果如下所示：

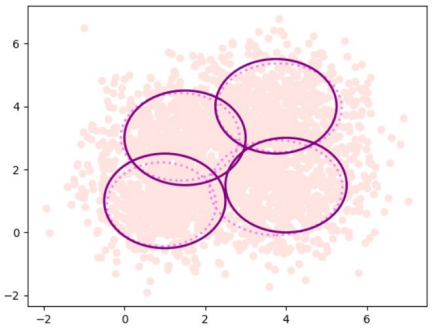

图 19 第三组数据 GMM 预测分布

4. 三类二维分布数据，均值和方差如下所示：

$$\mu_1^1 = 1, \sigma_1^1 = 3; \mu_2^1 = 1, \sigma_2^1 = 3$$
$$\mu_1^2 = 2, \sigma_1^2 = 3; \mu_2^2 = 2, \sigma_2^2 = 3$$
$$\mu_1^3 = 6, \sigma_1^3 = 0.5; \mu_2^3 = 6, \sigma_2^3 = 0.5$$

数据原始分布、K-Means、GMM 聚类结果分别如下所示：

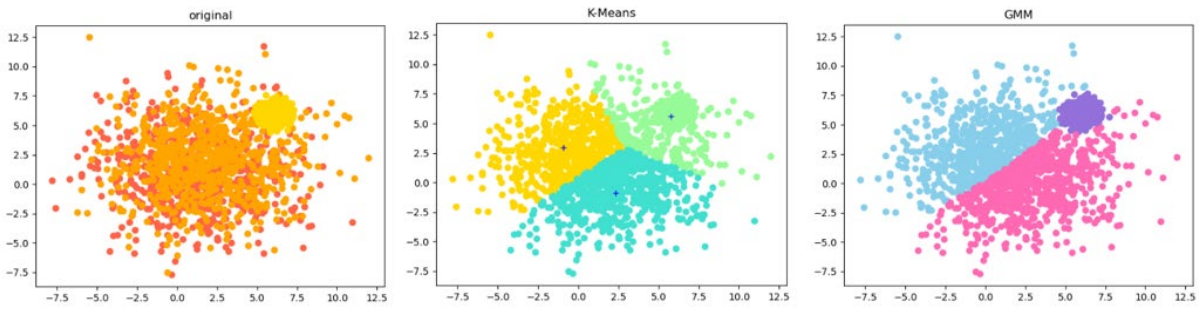

图 20 第四组数据分布与 K-Means, GMM 聚类结果

GMM 预测分布结果如下所示：

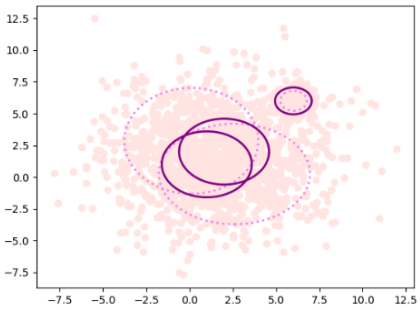

图 21 第四组数据 GMM 预测分布

通过对上述聚类结果的观察可以看出，相比于 K-Means 线性决策面的“硬边界”，GMM 能够用更加“软化”的边界对数据进行聚类，从而提升聚类效果，更好地处理分布重叠的情况。同时，通过对 GMM 习得的分布和原始分布的比较也可以看出，GMM 能够比较好地捕获高斯分布的特征（第一组、第三组）。但是在重合度非常高的情况下（第二组、第四组），GMM 的分布预测也会存在较大的偏差。

4.3.2 迭代次数对聚类结果的影响

在以下数据上进行测试：

$$\mu_1^1 = 1, \sigma_1^1 = 3; \mu_2^1 = 1, \sigma_2^1 = 3$$
$$\mu_1^2 = 1.5, \sigma_1^2 = 1; \mu_2^2 = 1.5, \sigma_2^2 = 1$$
$$\mu_1^3 = 10, \sigma_1^3 = 2; \mu_2^3 = 10, \sigma_2^3 = 2$$

观察迭代 1-12 次的过程中 GMM 预测的分布变化情况：

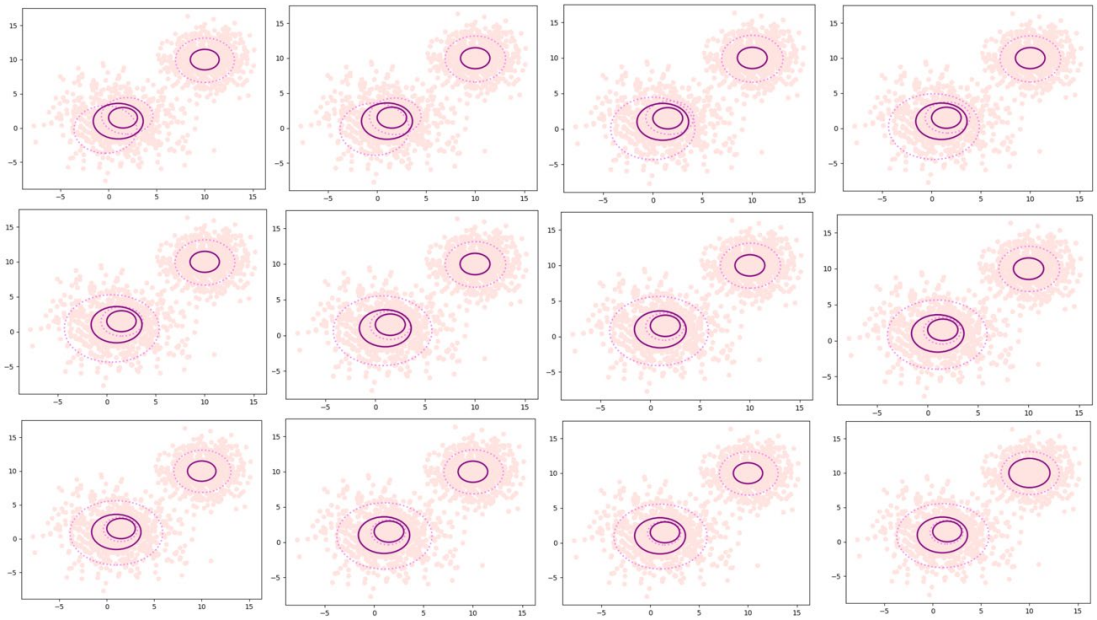

图 22 1-12 轮迭代下 GMM 预测分布结果变化情况

可以看出，在迭代的过程中，GMM 的预测分布逐渐向真实分布靠近。

对数似然的变化情况如下所示：

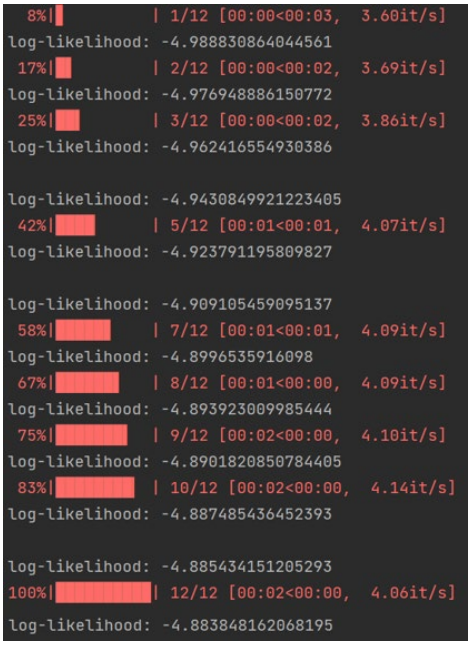

图 23 1-12 轮迭代下对数似然变化情况

可见对数似然随着迭代次数增加逐渐增大，这说明 GMM 模型正在逐渐收敛。

4.3.3 初始化对聚类结果的影响

在以下数据上进行实验：

$$\mu_1^1=1,\sigma_1^1=3;\mu_2^1=1,\sigma_2^1=3$$
$$\mu_1^2=1.5,\sigma_1^2=1;\mu_2^2=1.5,\sigma_2^2=1$$
$$\mu_1^3=10,\sigma_1^3=2;\mu_2^3=10,\sigma_2^3=2$$

分别用三组随机初始化和 K-Means 结果初始化进行实验，结果如下所示：

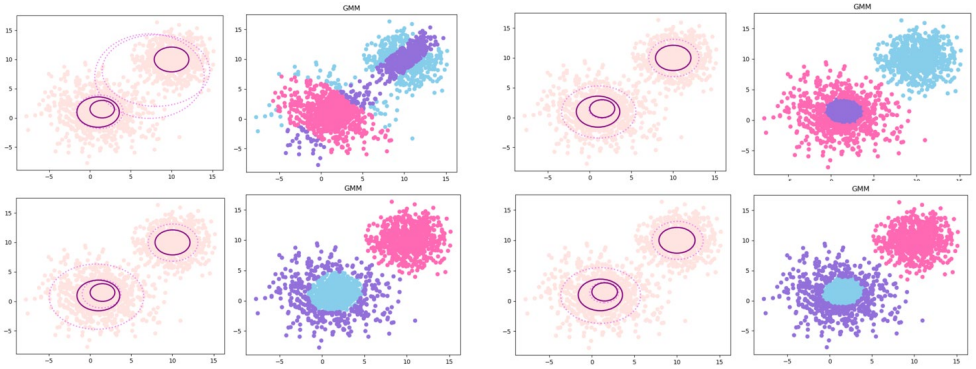

图 24 不同初始化下的聚类情况

可以发现，一个好的初始化点对 GMM 最终的聚类结果很重要。通常情况下，使用 K-Means 的聚类结果对 GMM 进行初始化效果较好。

以上数据具有一定的重叠性，初始化较差时会导致结果不理想。事实上，即使是分类难度较低的数据，在没有好的初始化点的情况下，也会出现很差的聚类结果，如下图所示：

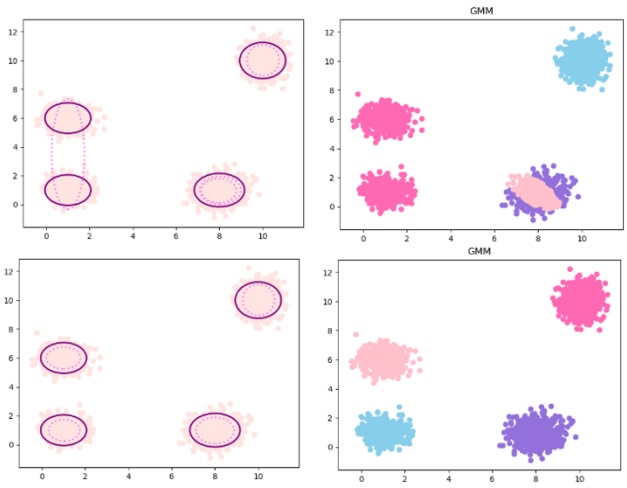

图 25 不理想初始化的聚类结果

4.3.4 数据量对聚类结果的影响

分别用每类 10 个、每类 50 个、每类 100 个、每类 500 个数据进行实验，数据原始分布如下：

$$\mu_1^1=1,\sigma_1^1=1;\mu_2^1=2,\sigma_2^1=1$$
$$\mu_1^2=1,\sigma_1^2=1;\mu_2^2=3,\sigma_2^2=1$$
$$\mu_1^3=3,\sigma_1^3=1;\mu_2^3=2,\sigma_2^3=1$$
$$\mu_1^4=3,\sigma_1^4=1;\mu_2^4=4,\sigma_2^4=1$$

聚类结果分别如下所示：

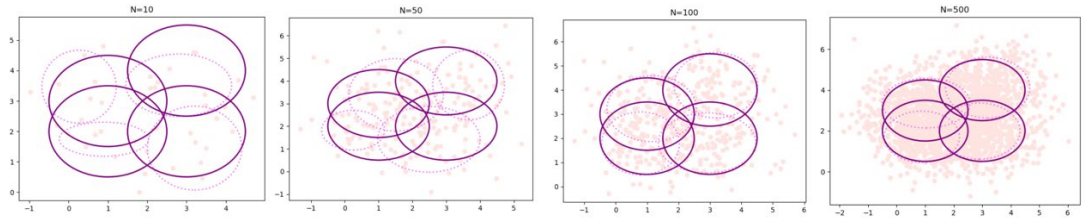

图 26 不同数据量的聚类结果

对比三个图可以看出，当数据量较少的时可能会发生预测的偏差，出现欠拟合现象；当数据量较大的时预测结果会有所改善。

同时如果不同类别样本的数据不平衡，分类会向数据较多的一边倾斜：

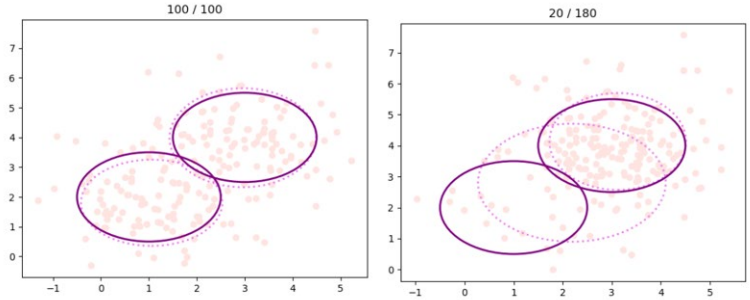

图 27 不平衡样本数据的聚类结果

4.3.5 协方差矩阵为奇异矩阵的处理方法

对协方差矩阵求逆时，有时程序会出现奇异矩阵singular matrix的报错。这是因为当一

组数据方差为0时，协方差矩阵为奇异矩阵，无法对其求逆。造成方差为0的原因有很多，当维度过高、样本量过小、初始化不理想使得聚类后某一类样本方差为0时，都可能出现上述情况。但这实际上是一个正常现象，并不是程序出错导致的。解决方法包括重新选择一组初始化，或者对协方差矩阵添加一个很小的正则项。

4.5.6 UCI数据集上的应用

选择鸢尾花数据集（Iris）进行实验。该数据集样本特征维度为4，类别数为3。聚类结果如下所示：

```
truth: [0 1 1 2 1 0 1 2 2 0 0 2 0 2 0 2 2 1 0 2 2 2 2 0 1 0 1 1 0 0 1 0 0 1 1 2 2
 2 1 1 0 2 2 2 0 0 1 1 0 2 0 0 0 0 2 0 1 0 0 2 1 2 1 1 1 1 0 2 2 0 1 2 1 1
 2 0 2 2 0 0 1 1 2 2 2 1 1 2 0 0 2 2 0 2 0 1 0 0 1 2 0 1 1 2 1 1 1 0 0 2 2
 0 0 2 2 1 1 1 0 2 1 1 0 2 0 0 1 2 1 0 0 1 0 1 0 2 1 0 2 1 1 2 1 2 2 2 1 0
 2 1]
prediction: [2 1 1 0 1 2 0 0 0 2 2 0 2 0 2 0 0 1 2 0 0 0 0 2 1 2 1 1 2 2 0 2 2 1 1 0 0
 0 1 1 2 0 0 0 2 2 1 1 2 0 2 2 2 2 0 2 1 2 2 0 1 0 1 1 1 1 2 0 0 2 1 0 1 1
 0 2 0 0 2 2 1 1 0 0 0 1 1 0 2 2 0 0 2 0 2 1 2 2 1 0 2 1 1 0 1 1 1 2 2 0 0
 2 2 0 0 1 1 0 2 0 1 1 2 0 2 2 1 0 1 2 2 1 2 1 2 0 0 2 0 1 1 0 1 0 0 0 1 2
 0 1]

Accuracy on Iris: 0.9733333333333334
```

图28 鸢尾花数据集聚类结果

由于使用K-Means进行初始化，且鸢尾花数据集的数据分布本身就是线性可分的，聚类难度较低，GMM的准确率可以高达97.3%。

五、结论

K-Means 倾向于得到线性决策面的“硬边界”，而 GMM 能够用更加“软化”的边界对数据进行聚类，从而提升聚类效果，更好地处理分布重叠的情况。因此，GMM 能够更好地捕获高斯分布的特征。

初始化对于 K-Means 和 GMM 都影响巨大，初始化能够直接影响聚类结果。对于 K-means，从已知样本点中选择相距较远的点作为初始聚类中心更有可能得到更好的结果；对于 GMM，可以直接用 K-Means 的结果进行初始化，效果通常比较理想。

除此之外，数据本身的分布特点也直接决定了 K-Means 和 GMM 能否对其正确分类，虽然通常情况下，GMM 能够在 K-Means 的结果上进一步优化，但是，在不同类别的数据重合度非常高的情况下，无论是 K-Means 还是 GMM 的分布预测都会存在较大的偏差。

迭代次数和样本数据量也对 K-Means 和 GMM 的聚类结果有一定影响。通常情况下，如果初始点选择恰当，随着迭代次数和样本数据量的增加，聚类效果会逐渐变好。但需要注意的是，如果样本数据存在类别间不平衡的现象，聚类结果会存在偏差。同时，迭代次数也不能过多，否则模型也很有可能发生过拟合而陷入局部最优解。

六、参考文献

[1] Christopher M. Bishop. 2006. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg.

[2] 周志华. 2016. 机器学习.

[3] Strang, G. 2006. Linear algebra and its applications. Belmont, CA: Thomson, Brooks/Cole.

[4] https://github.com/zkywsg/Daily-DeepLearning

七、附录：源代码（带注释）

utils.py:

```
1. import random
2. import numpy as np
3. import matplotlib.pyplot as plt
4.
5.
6. random.seed(116)
7. random.seed(116)
8.
9.
10.def generate_gauss_data(mu, sigma, dim):
11.    """
12.    generate data that satisfy gauss distribution
13.    :param mu: mean
14.    :param sigma: variance
15.    :param dim: dimension of parameter x
16.    :return: generated data
17.    """
18.    x = []
19.    for j in range(dim):
20.        x.append(random.gauss(mu[j], sigma[j]))
21.    return x
22.
23.
24.def generate_data(mu, sigma, num_class, num_sample):
25.    """
26.    generate data of k classes, and data in each class sati
   sfy gauss distribution
27.    :param mu: mean
28.    :param sigma: variance
29.    :param num_class: number of classes (k)
30.    :param num_sample: number of samples in each class
31.    :return: generated data
32.    """
33.    assert len(mu) == len(sigma)
34.    assert len(mu[0]) == len(sigma[0])
35.    assert len(sigma) == num_class
36.    assert len(num_sample) == len(sigma)
37.    class_id = 0
38.    x_dim = len(mu[0])
39.    data = []
40.    for num in num_sample:
```

```
        for i in range(num):
            x = generate_gauss_data(mu[class_id], sigma[class_id], x_dim)
            x.append(class_id)
            data.append(x)
        class_id += 1
    return data


def visualize_2d_data(data, num_class, color=True, color_list=None):
    if color_list is None:
        color_list = ['tomato', 'orange', 'gold',  'palegreen', 'turquoise', 'skyblue', 'mediumpurple', 'hotpink', 'pink', ]
    classify = []
    for i in range(num_class):
        classify.append([])
    for r in data:
        classify[int(r[-1])].append(r[:-1])

    class_id = 0
    for c in classify:
        c = np.array(c)
        if color:
            plt.plot(c[:, 0], c[:, 1], 'ro', color=color_list[class_id])
        else:
            plt.plot(c[:, 0], c[:, 1], 'ro', color=color_list[-1])
        class_id += 1
    plt.show()


def process_data(mu, sigma, num_sample, is_shuffle=True):
    num_class = len(num_sample)
    data = generate_data(mu, sigma, num_class, num_sample)
    plt.title('original')
    visualize_2d_data(data, num_class)
    random.shuffle(data)
    data = np.array(data)[:, :-1]
    return data


```

```
def evaluate(labels, logits):
    """
    evaluate the accuracy
    :param labels: true
    :param logits: prediction
    :return: accuracy
    """
    total = len(labels)
    correct = 0
    for i in range(total):
        if labels[i] == 0 and logits[i] == 2:
            correct += 1
        elif labels[i] == 1 and logits[i] == 1:
            correct += 1
        elif labels[i] == 2 and logits[i] == 0:
            correct += 1
    acc = correct / total
    return acc


if __name__ == '__main__':
    mu = [[1, 1], [0.5, 2], [4, 1.5], [4.5, 4]]
    sigma = [[1, 1], [1, 1], [1, 1], [1, 1]]
    num_sample = [650, 625, 675, 650]
    num_class = len(num_sample)
    data = generate_data(mu, sigma, num_class, num_sample
)
    visualize_2d_data(data, num_class)
    plt.show()
```

kmeans.py:

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

np.random.seed(0)


class KMeans:
    def __init__(self, num_class, data):
        self.data = data
        self.k = num_class
        self.init = self.initialize_center()
        self.center = self.init
```

```
14.         self.cluster = np.zeros(self.data.shape[0])
15.
16.     @property
17.     def _center(self):
18.         return self.center
19.
20.     @property
21.     def _cluster(self):
22.         return np.c_[self.data, self.cluster]
23.
24.     def initialize_center(self):
25.         random_id = np.random.choice(self.data.shape[0], si
    ze=self.k, replace=False)
26.
27.         return self.data[random_id]
28.
29.     def update_center(self):
30.         num_sample = self.data.shape[0]
31.         x_dim = self.data.shape[1]
32.         center = np.repeat(self.center, num_sample, axis=0)
    .reshape((self.k, num_sample, -1))
33.         distance = self.compute_distance(self.data, center)
34.         prev_cluster = self.cluster.view()
35.         self.cluster = np.argmin(distance, axis=0)
36.
37.         self.center = np.zeros((self.k, x_dim))
38.         total = np.zeros(self.k)
39.         for i in range(num_sample):
40.             self.center[self.cluster[i]] += self.data[i]
41.             total[self.cluster[i]] += 1
42.         total = np.repeat(total, x_dim).reshape((self.k, x_
    dim))
43.         self.center /= total
44.
45.         return (prev_cluster == self.cluster).all()
46.
47.     def forward(self, epoch):
48.         for e in range(epoch):
49.             flag = self.update_center()
50.             # self.visualize()
51.             if flag:
52.                 break
53.
54.     def visualize(self):
```

```
        plt.title('K-Means')
        color_list = ['gold', 'palegreen', 'turquoise', 'skyblue', 'mediumpurple', 'hotpink', 'tomato', 'orange', ]
        x, y = [], []
        for i in range(self.k):
            x.append([])
            y.append([])
        for i in range(self.data.shape[0]):
            x[self.cluster[i]].append(self.data[i][0])
            y[self.cluster[i]].append(self.data[i][1])
        for i in range(self.k):
            plt.plot(x[i], y[i], 'ro', color=color_list[i])
        for center in self.center:
            plt.plot(center[0], center[1], marker='+', color='blue')
        # for center_init in self.init:
        #     plt.plot(center_init[0], center_init[1], marker='*', color='red')

        plt.show()

    @staticmethod
    def compute_distance(a, b):
        return np.sqrt(np.sum(np.square(a - b), axis=-1))


if __name__ == '__main__':
    pass
```

gmm.py:

```
import numpy as np
from matplotlib.patches import Ellipse
from scipy.stats import multivariate_normal
from tqdm import trange
import matplotlib.pyplot as plt
from utils import visualize_2d_data


class GaussianMixtureModel:
    def __init__(self, k, mu, sigma, pi, data):
        self.k = k
        self.mu = mu
        self.sigma = sigma
        self.pi = pi
```

```
        self.data = data

    def expectation_step(self):
        prob = []
        for i in range(self.k):
            prob.append(self.probability(i))
        prob = np.transpose(prob)
        numerator = self.pi * prob
        denominator = np.sum(numerator, axis=-1)
        denominator = np.repeat(denominator, self.k, axis=0).reshape((-1, self.k))
        responsible_matrix = numerator / denominator
        return responsible_matrix

    def maximization_step(self, responsible_matrix):
        valid_num = np.sum(responsible_matrix, axis=0)

        # update mu
        for i in range(self.k):
            responsibility = np.repeat(responsible_matrix[:, i], self.data.shape[1], axis=0).reshape(self.data.shape[0],

                                                                    -1)
            self.mu[i] = np.sum(self.data * responsibility, axis=0) / valid_num[i]

        # update sigma
        for i in range(self.k):
            self.sigma[i] = np.zeros((self.data.shape[1], self.data.shape[1]))
            bias = np.matrix(self.data - self.mu[i])
            for j in range(self.data.shape[0]):
                self.sigma[i] += np.dot(bias[j].T, bias[j]) * responsible_matrix[j][i]
            self.sigma[i] /= valid_num[i]

        # update pi
        self.pi = valid_num / self.data.shape[0]

    def log_likelihood(self):
        prob = []
        for i in range(self.k):
            prob.append(self.probability(i))
```

```
52.         prob = np.sum(np.transpose(prob) * self.pi, axis=-1
    )
53.         likelihood = np.mean(np.log(prob))
54.         return likelihood
55.
56.     def forward(self, epoch, log=True):
57.         for e in trange(epoch):
58.             response = self.expectation_step()
59.             self.maximization_step(response)
60.             if log:
61.                 print('\nlog-likelihood: ' + str(self.log_l
    ikelihood()))
62.
63.     def probability(self, class_id):
64.         x_dim = self.data.shape[1]
65.         denominator = ((2 * np.pi) ** (x_dim / 2)) * ((np.l
    inalg.det(self.sigma[class_id])) ** (1 / 2))
66.         bias = self.data - self.mu[class_id]
67.         numerator = np.exp(np.diagonal(- 1 / 2 * np.dot(np.
    dot(bias, np.linalg.inv(self.sigma[class_id])), bias.T)))
68.         prob = numerator / denominator
69.         # prob = np.diagonal(prob)
70.         return prob
71.
72.     def predict(self):
73.         prob = []
74.         for i in range(self.k):
75.             prob.append(self.probability(i))
76.         prob = np.transpose(prob)
77.         pred = np.argmax(prob, axis=-1)
78.         return pred
79.
80.     def visualize_distribution(self, mu_true=None, sigma_tr
    ue=None):
81.         plt.title('20 / 180')
82.         plt.scatter(self.data[:, 0], self.data[:, 1], c='mi
    styrose')
83.         ax = plt.gca()
84.         for i in range(self.k):
85.             plot_args = {'fc': 'None', 'lw': 2, 'edgecolor'
    : 'violet', 'ls': ':'}
86.             ellipse = Ellipse(self.mu[i], 3 * np.sqrt(self.
    sigma[i][0][0]), 3 * np.sqrt(self.sigma[i][1][1]), **plot_a
    rgs)
```

```
            ax.add_patch(ellipse)

        if (mu_true is not None) and (sigma_true is not None):
            for i in range(self.k):
                plot_args = {'fc': 'None', 'lw': 2, 'edgecolor': 'purple'}
                ellipse = Ellipse(mu_true[i], 3 * np.sqrt(sigma_true[i][0]), 3 * np.sqrt(sigma_true[i][1]), **plot_args)
                ax.add_patch(ellipse)
        plt.show()

    def visualize(self):
        pred = self.predict()
        cluster = np.c_[self.data, pred]
        plt.title('N=500')
           color_list = ['skyblue', 'mediumpurple', 'hotpink', 'pink', 'tomato', 'orange', 'gold', 'palegreen', 'turquoise',]
        visualize_2d_data(cluster, self.k, color_list=color_list)

if __name__ == '__main__':
    data = np.array([[1, 2], [0, 0], [1, 1]])
    mu = np.array([[0, 1], [0, 1]])
    sigma = np.array([[[1, 0], [0, 1]], [[2, 0], [0, 2]]])
    a = np.array([[1, 3], [2, 3], [4, 5]])
    b = np.array([1, 2, 3])
    de = np.sum(a, axis=-1)
    a[0] = np.array([1, 2])
    print(a)
```

load.py:

```
import numpy as np
import random


random.seed(0)


def load_iris():
    data = []
```

```
    with open('iris.data') as f:
        for line in f:
            l = line.strip('\n').split(',')
            x = [float(scalar) for scalar in l[:-1]]
            if l[-1] == 'Iris-setosa':
                x.append(0)
            elif l[-1] == 'Iris-versicolor':
                x.append(1)
            elif l[-1] == 'Iris-virginica':
                x.append(2)
            data.append(x)
    f.close()
    return data


def process_train(train):
    random.shuffle(train)
    X = []
    Y = []
    for d in train:
        X.append(d[:-1])
        Y.append(d[-1])
    X = normalize(np.array(X))
    return X, np.array(Y)


def normalize(data):
    mean = np.mean(data, axis=0)
    std = np.std(data, axis=0)
    data = (data - mean) / std
    return data


if __name__ == '__main__':
    data = load_iris()
    print(data)
```

main.py:

```
import numpy as np
import random
random.seed(0)
def load_iris():
    data = []
    with open('iris.data') as f:
```

```
        for line in f:
            l = line.strip('\n').split(',')
            x = [float(scalar) for scalar in l[:-1]]
            if l[-1] == 'Iris-setosa':
                x.append(0)
            elif l[-1] == 'Iris-versicolor':
                x.append(1)
            elif l[-1] == 'Iris-virginica':
                x.append(2)
            data.append(x)
    f.close()
    return data
def process_train(train):
    random.shuffle(train)
    X = []
    Y = []
    for d in train:
        X.append(d[:-1])
        Y.append(d[-1])
    X = normalize(np.array(X))
    return X, np.array(Y)
def normalize(data):
    mean = np.mean(data, axis=0)
    std = np.std(data, axis=0)
    data = (data - mean) / std
    return data
if __name__ == '__main__':
    data = load_iris()
    print(data)
```