

# 机器学习 实验报告

|    |          |    |    |
|----|----------|----|----|
| 学号 | 22336180 | 姓名 | 马岱 |
|----|----------|----|----|

## 一、实验环境

- OS: Windows 11
- IDE: Visual Studio Code
- programming language: python

## 二、实验题目

以 MNIST 数据集为例，探索 K-Means 和 GMM 这两种聚类算法的性能。数据下载 FTP 地址：  
<ftp://172.18.167.164/Assignment3/material> (建议使用 FTP 客户端链接，用户名与密码均为 student)

要求：

- 1) 自己实现 K-Means 算法及用 EM 算法训练 GMM 模型的代码。可调用 numpy, scipy 等软件包中的基本运算，但不能直接调用机器学习包（如 sklearn）中上述算法的实现函数；
- 2) 在 K-Means 实验中，探索两种不同初始化方法对聚类性能的影响；
- 3) 在 GMM 实验中，探索使用不同结构的协方差矩阵（如：对角且元素值都相等、对角但对元素值不要求相等、普通矩阵等）对聚类性能的影响。同时，也观察不同初始化对最后结果的影响；
- 4) 在给定的训练集上训练模型，并在测试集上验证其性能。使用聚类精度(Clustering Accuracy, ACC)作为聚类性能的评价指标。由于 MNIST 数据集有 10 类，故在实验中固定簇类数为 10。

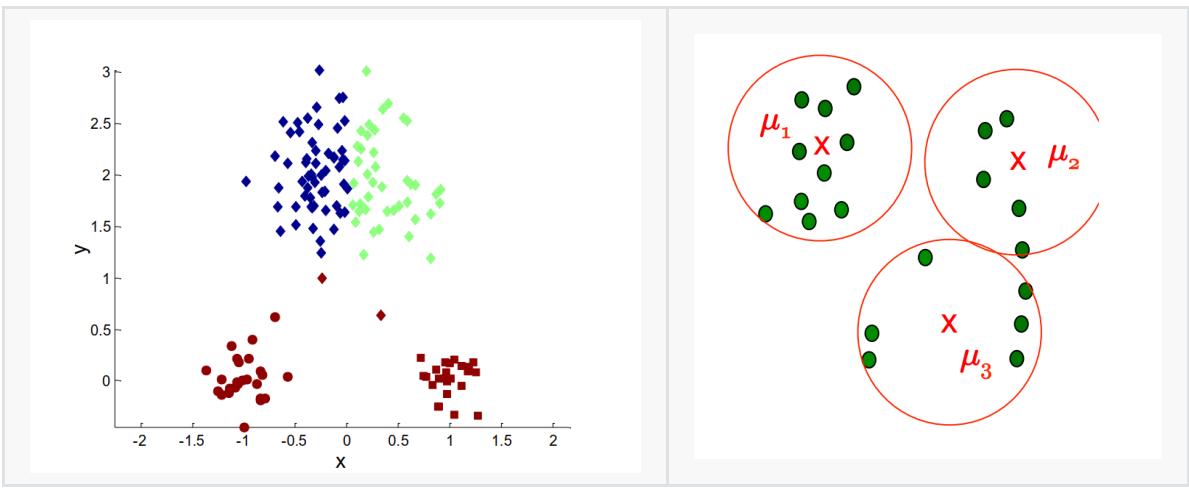
实验报告需包含（但不限于）：

- 1) 简要描述 K-Means 和 GMM 的算法流程；
- 2) 采用的训练方法，包括参数初始化方法、优化方法以及其他训练技巧等；
- 3) 通过观察实验结果，结合理论知识，比较 K-means 聚类方法和 EM 训练的 GMM 聚类方法之间的优劣；
- 4) 实验结果以及讨论，包括模型性能、训练时间、不同聚类算法的效果差异等。

## 三、实验内容

### 1. 模型理论知识&算法原理

**[K-Means]**



*k-means*的目的是：把n个点（可以是样本的一次观察或一个实例）划分到k个聚类中，使得每个点都属于离他最近的均值（此即聚类中心）对应的聚类，以之作为聚类的标准。已知观测集 $(x_1, x_2, \dots, x_n)$ ，其中每个观测都是一个d维实向量，k-均值聚类要把这n个观测划分到k个集合中( $k \leq n$ )，使得组内平方和(WCSS within-cluster sum of squares)最小。换句话说，它的目标是找到使得下式满足的聚类 $S_i$ ，

$$\arg \min_{\mathbf{S}} \sum_{i=1}^k \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2$$

其中 $\boldsymbol{\mu}_i$ 是 $S_i$ 中所有点的均值。

### 【K-Means 算法步骤】

#### 1. 问题定义：

- 数据矩阵  $A \in \mathbb{R}^{N \times P}$ ，每一行表示一个数据点，总共有  $N$  个点，每个点的维度是  $P$ 。
- 将  $A$  分成  $K$  个簇，通过最小化以下目标函数实现：
- $\Phi \in \mathbb{R}^{N \times K}$  是分配矩阵，每行只有一个分量为 1，其余为 0，表示每个点分配到一个簇。
- $H \in \mathbb{R}^{K \times P}$  是簇中心矩阵，每行对应一个簇中心。

$$\min \|A - \Phi H\|_F^2$$

#### 2. 目标函数的两部分：

- 固定  $\Phi$ ，更新  $H$ ：计算每个簇的中心。
- 固定  $H$ ，更新  $\Phi$ ：根据当前簇中心重新分配点。

### 初始化

随机初始化  $\Phi$  或  $H$ ：

- 初始化  $H$ ：随机选取  $K$  个点作为初始簇中心。
- 或初始化  $\Phi$ ：随机分配每个点到某个簇。

### 迭代过程

#### 1. 更新 $H$ （计算簇中心）

对于每个簇  $j$ ，计算簇内样本点的均值，更新簇中心：

$$h_j = \frac{\sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i}{|C_j|}$$

在矩阵形式下：

$$H = \arg \min_H \|A - \Phi H\|_F^2$$

这是因为每个簇中心是对应簇内点到中心距离平方和最小化的结果。

## 2. 更新 $\Phi$ (重新分配点)

对于每个点  $\mathbf{x}_i$ , 找到距离最近的簇中心  $h_j$ , 并将点重新分配到对应的簇:

$$\phi_{ij} = \begin{cases} 1 & \text{if } j = \arg \min_k \|\mathbf{x}_i - h_k\|^2 \\ 0 & \text{otherwise} \end{cases}$$

换句话说:

- 对于每个点, 计算它与所有簇中心的距离。
- 分配到使距离最小的那个簇。

### 收敛条件

重复更新  $\Phi$  和  $H$ , 直到以下条件之一满足:

- 簇中心  $H$  不再变化 (或变化小于某一阈值)。
- 最大迭代次数达到。

---

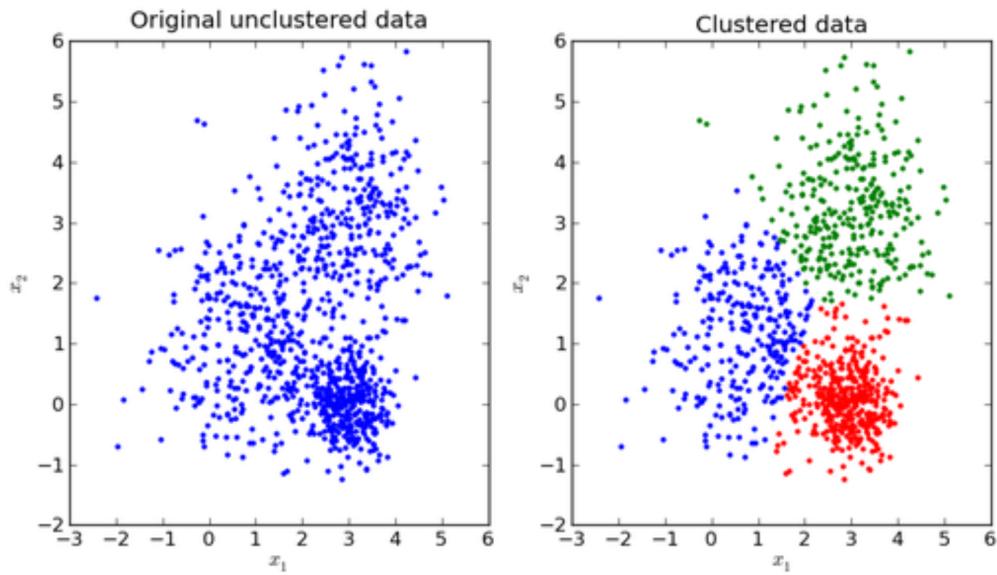
注意:

- **$\Phi$  的性质:** 每行只有一个分量为 1, 表示点的分配。
- **目标分解:**
  - 分解为  $\min_H$  和  $\min_\Phi$  两部分。
- **更新公式:**
  - 分配矩阵更新通过最小距离选择。
  - K-Means 的核心思想就是交替优化分配矩阵  $\Phi$  和簇中心矩阵  $H$ 。

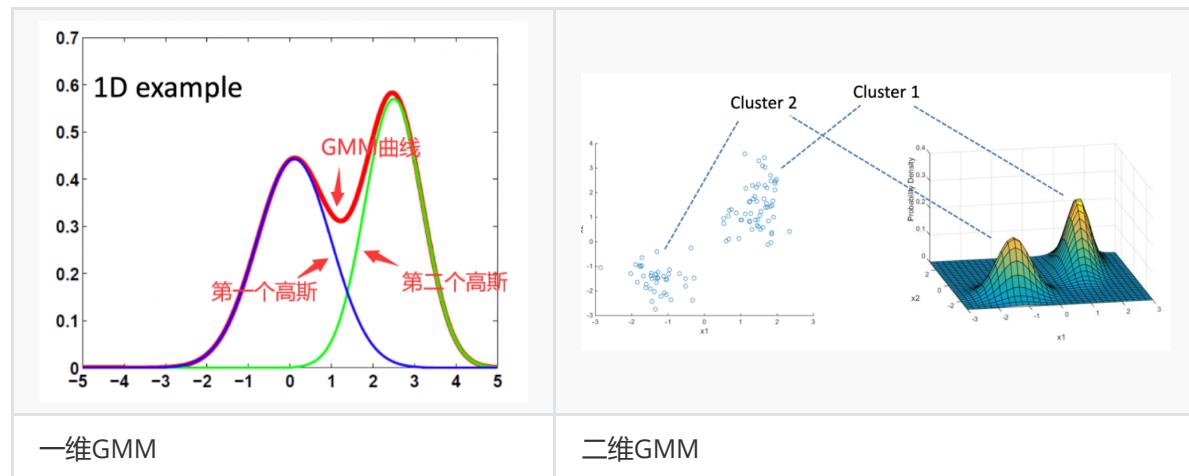
- 
- 1: Select  $K$  points as the initial centroids.
  - 2: **repeat**
  - 3:   Form  $K$  clusters by assigning all points to the closest centroid.
  - 4:   Recompute the centroid of each cluster.
  - 5: **until** The centroids don't change
- 

1. 选择初始化的  $k$  个样本作为初始聚类中心  $a = a_1, a_2, \dots, a_k$ ;
  2. 针对数据集中每个样本  $x_i$  计算它到  $k$  个聚类中心的距离并将其分到距离最小的聚类中心所对应的类中;
  3. 针对每个类别  $a_j$ , 重新计算它的聚类中心  $a_j = \frac{1}{|c_i|} \sum_{x \in c_i} x$  (即属于该类的所有样本的质心);
  4. 重复上面 2 3 两步操作, 直到达到某个中止条件 (迭代次数、最小误差变化等)。
- 

### 【GMM】



高斯分布，又叫正态分布（Normal Distribution），是最最重要的概率分布。原因就是因为它在生活和工程中都非常常见。它的概率密度函数（PDF, probability density function）中间高两边低，且关于均值对称。高斯混合模型（Gaussian Mixed Model）简称GMM，指多个单高斯分布函数的线性组合，理论上GMM可以拟合出任意类型的分布。高斯混合模型通常用于解决同一集合下的数据包含多个不同的分布的情况，具体应用有聚类、密度估计、生成新数据等。严格来说，GMM 不是一种聚类算法，而只是一种用来拟合数据分布的模型。然而，我们可以假设每个簇的样本都符合一个高斯分布。这样，我们可以使用训练集的样本数据来训练、拟合一个 GMM 模型。通过计算某个样本数据中各个高斯分布的贡献比例，我们就可以推断这个样本数据所属的簇。某个簇的高斯分布对这个样本数据的贡献越大，那么该样本就越可能属于这个簇。



$$\text{全概率公式: } P(D) = P(D \cap A) + P(D \cap B) + P(D \cap C)$$

$$= P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)$$

$$\begin{aligned} \text{贝叶斯公式: } P(\theta|D) &= \frac{P(\theta \cap D)}{P(D)} \\ &= \frac{P(D|\theta)P(\theta)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)} \end{aligned}$$

GMM是使用多个高斯分布来进行建模的。每一个高斯分布都未知（均值方差都未知），每个高斯对整体GMM的贡献比例也未知。那么采集一组数据，想要知道每一个数据出自哪一个高斯，就得采用迭代的方法来计算该数据出自某一个高斯的概率：比如第一个数据有可能出自第一个高斯，也有可能出自第二个高斯，也有可能出自第三个高斯……这时就得用迭代的算法。上面的全概率公式中， $P(A)$ 就表示第一个高斯的概率分布， $P(B)$ 表示第二个高斯的概率分布， $P(C)$ 表示第三个高斯的概率分布。 $P(D|A)$ 表示在第一个高斯中取得数据D的概率， $P(D|B)$ 表示在第二个高斯中取得数据D的概率， $P(D|C)$ 表示在第三个高斯中取得数据D的概率。 $P(D)$ 表示取得整体数据D的概率。而公式中的参数theta，在GMM中是一个向量，表示的是各个高斯的均值和方差，以及每个高斯对整体GMM的贡献比例lambda。由此，GMM就和全概率公式和贝叶斯公式扯上了关系。

先简单地回顾一下多元高斯分布的定义。对于n维样本空间X中的随机向量x，若其概率密度函数为：

$$p(x) = \frac{1}{(2\pi)^{n/2}(\det \Sigma)^{1/2}} \exp \left\{ -\frac{1}{2}(x - \mu)^\top \Sigma^{-1}(x - \mu) \right\}$$

则称x服从高斯分布。其中 $\mu$ 是n维均值向量， $\Sigma$ 是 $n \times n$ 的协方差矩阵， $\det \Sigma$ 是方阵 $\Sigma$ 的行列式。从上面式子可知，高斯分布完全由均值向量 $\mu$ 和协方差矩阵 $\Sigma$ 这两个参数确定。为了明确显示高斯分布与相应参数的依赖关系，常常将高斯分布的概率密度函数 $p(x)$ 记为 $p(x | \mu, \Sigma)$ 或者 $N(x | \mu, \Sigma)$ 。

MLE: 最大似然估计

$$\theta^* = \operatorname{argmax} P(D|\theta)$$

使得当前得到的这组数据的概率达到最大

MAP: 最大后验估计， Maximum a posterior probability estimation

$$\begin{aligned} \theta^* &= \operatorname{argmax} P(\theta|D) \\ &= \operatorname{argmax} \frac{P(\theta) \cdot P(D|\theta)}{P(D)} \\ &\approx \operatorname{argmax} P(\theta) \cdot P(D|\theta) \end{aligned}$$

使得这个参数theta的概率达到最大

MLE与MAP的区别：有没有加入先验知识 $P(\theta)$

高斯混合模型（GMM）的数学表达式：

$$GMM = \sum_k \lambda_k \mathcal{N}(\mu_k, \sigma_k)$$

其中各符号的含义如下：

- $\lambda_k$ : 混合系数，表示第k个高斯分布在总体中的权重，满足 $\lambda_k > 0$ 且 $\sum_k \lambda_k = 1$ 。
- $\mathcal{N}(\mu_k, \sigma_k)$ : 第k个高斯分布的概率密度函数，其均值为 $\mu_k$ ，标准差为 $\sigma_k$ （对于多维情况，则是均值向量和协方差矩阵）。
- $k$ : 表示第k个高斯分布，总共K个分布。

GMM用作聚类算法的思想很简单：假设样本数据服从混合高斯分布，根据样本数据集推出混合高斯分布的各个参数，以及各个样本最可能属于哪个高斯分布，这样，GMM的K（K值往往由用户提供）个成分对应于聚类任务中的K个簇，每个样本划到最可能的高斯分布所对应的簇中。由于GMM是一种生成模型，可以理解为以下随机过程生成数据：随机从K个高斯分布中选择一个k（根据混合系数 $\pi_k$ ）；从选定的第k个高斯分布 $\mathcal{N}(\mu_k, \Sigma_k)$ 中生成一个数据点。换句话说：**每个数据点的生成有“隐变量”z，表示数据属于哪个高斯分布**。z是一个离散变量，取值范围为 $\{1, 2, \dots, K\}$ 。因此隐变量的概率分布为： $p(z = k) = \pi_k$ ， $k \in \{1, 2, \dots, K\}$ ，数据点x的条件分布为： $p(x | z = k) = \mathcal{N}(x | \mu_k, \Sigma_k)$ ，因此，联合分布可以写为： $p(x, z) = p(z) \cdot p(x | z)$ ，对于观测数据x，隐变量z被积分掉后，得到边缘分布：

$$p(\mathbf{x}) = \sum_{k=1}^K p(z=k) \cdot p(\mathbf{x} | z=k) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

其中：

- $K$ : 高斯分布的数量 (簇的数量)。
- $\pi_k$ : 第  $k$  个高斯分布的混合系数, 满足  $\pi_k > 0$  且  $\sum_{k=1}^K \pi_k = 1$ 。
- $\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ : 第  $k$  个高斯分布的概率密度函数:

$$\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{(2\pi)^{d/2} |\boldsymbol{\Sigma}_k|^{1/2}} \exp \left( -\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) \right)$$

- $\boldsymbol{\mu}_k$ : 第  $k$  个高斯分布的均值 (簇中心)。
- $\boldsymbol{\Sigma}_k$ : 第  $k$  个高斯分布的协方差矩阵。

假设样本的生成过程由高斯混合分布给出。高斯混合分布的抽样过程如下：首先，随机地在  $K$  个高斯混合成分之中选一个，每个成分被选中的概率是它的系数  $\pi_k$ ；然后，再单独考虑这个被选择的混合成分，根据它的概率密度函数进行采样，从而生成相应的样本。若样本集  $D = x_1, x_2, \dots, x_N$  由上述过程生成，如何通过样本集估计出模型参数  $(\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) | 1 \leq k \leq K$  呢？显然，对于给定的样本集  $D$ ，**如果知道每个样本点是由哪个高斯混合成分生成，则直接采用“极大似然估计”即可求解参数**，即最大化对数似然：**目标**：通过  $K$  个高斯分布拟合数据点  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$ 。GMM 的目标是最大化对数似然函数：

$$\mathcal{L} = \log \prod_{i=1}^N p(\mathbf{x}_i) = \sum_{i=1}^N \log \left( \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

问题在于，对于这个数据集  $D$  我们并不知道每个样本点是由哪个高斯混合成分生成（这称为“不完全数据集”），从而无法直接使用极大似然估计来求解参数。这种情况下，我们常用 EM 算法来迭代优化求解。因此引入隐变量  $z$  的后验概率  $\gamma_{ik}$  来辅助计算。

**(E步)** 我们引入一个  $K$  维随机变量  $z$ （通常称之为隐含变量，Latent Variable）来描述“样本点是由哪个高斯混合成分生成”这个未知事情，它满足条件： $z_k (1 \leq k \leq K)$  只能取 0 或者 1；且  $\sum_{k=1}^K z_k = 1$ 。例如，样本集共有 3 个类别，某一样本点如果它由第 1 个高斯混合成分生成，则相应随机变量  $z = (1, 0, 0)$ ，如果它由第 2 个高斯混合成分生成，则相应随机变量  $z = (0, 1, 0)$ ，依此类推。这样， $P(z_k = 1)$  可表示当前样本点由第  $k$  个高斯混合成分生成的概率。显然有， $P(z_k = 1) = \pi_k$ 。给定样本点  $x_i$ ，它由第  $k$  个高斯混合成分生成的概率（后验概率）为  $pM(z_k = 1 | x_i)$ ，为方便起见把它简记为  $\gamma_{ik}$ ，由贝叶斯公式和全概率公式知：计算隐变量的后验概率（即数据点属于第  $k$  个高斯分布的概率），其中  $\gamma_{ik}$  是软分配概率，表示  $\mathbf{x}_i$  属于第  $k$  个簇的概率。但实际上我们更多将  $\gamma_{ik}$  定义成当第  $x_i$  来自第  $k$  个高斯分量时  $\gamma_{ik} = 1$ ，否则  $\gamma_{ik} = 0$ 。

$$\begin{aligned} \gamma_{ik} &\triangleq p_M(z_k = 1 | \mathbf{x}_i) \\ &= \frac{P(z_k = 1) \cdot p_M(\mathbf{x}_i | z_k = 1)}{p_M(\mathbf{x}_i)} \\ &= \frac{P(z_k = 1) \cdot p_M(\mathbf{x}_i | z_k = 1)}{\sum_{j=1}^K P(z_j = 1) \cdot p_M(\mathbf{x}_i | z_j = 1)} \\ &= \frac{\pi_k \cdot p_M(\mathbf{x}_i | z_k = 1)}{\sum_{j=1}^K \pi_j \cdot p_M(\mathbf{x}_i | z_j = 1)} \\ &= \frac{\pi_k \cdot \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^K \pi_j \cdot \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \end{aligned} \quad (\text{EM算法中“E step”})$$

**(M步)** 利用 E 步的后验概率  $\gamma_{ik}$ ，更新模型参数  $\pi_k$ 、 $\boldsymbol{\mu}_k$ 、 $\boldsymbol{\Sigma}_k$ ：

$$\pi_k = \frac{\sum_{i=1}^N \gamma_{ik}}{N}$$

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^N \gamma_{ik} \mathbf{x}_i}{\sum_{i=1}^N \gamma_{ik}}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^N \gamma_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top}{\sum_{i=1}^N \gamma_{ik}}$$

(**重复**) 作为隐变量,  $\gamma_{ik}$  是不可知的。虽然不可知, 但我们可以根据当前的样本数据和 *GMM* 参数 “猜测”  $\gamma_{ik}$  的值, 再基于 “猜测值” 反过来估计 *GMM* 的参数。这就是 EM 算法的思想。在 E 步与 M 步之间交替迭代, 直到参数收敛 (对数似然函数不再显著增加)。

**当迭代结束后, 样本点  $x_i$  所属的类别  $\lambda_i$  由下式确定**

$$\lambda_i = \arg \max_{k \in \{1, 2, \dots, K\}} \gamma_{ik}$$


---

### 【EM】

EM 算法用来解决含有隐变量 (latent variable(s)) 的估计问题。EM 算法就是构造一个期望(数学上, 期望就是平均值), 然后在此基础上进行操作, 使得概率变得最大。主要用于解决含有隐变量的概率估计问题。在高斯混合模型 (GMM, Gaussian mixture module), 隐马尔可夫模型 (HMM, Hidden Markov Model) 等数学模型中都有涉及。EM 算法是一个迭代算法, 每一次迭代都会分为 2 个阶段: E 步 (求期望) 和 M 步 (求最大)。

给定数据集  $X = \{x_1, x_2, \dots, x_N\}$ , 我们假设它来自一个概率分布, 但这个分布依赖于一些无法观测的隐变量  $Z = \{z_1, z_2, \dots, z_N\}$ 。目标是最大化观测数据的对数似然函数:

$$\log p(X|\theta) = \log \int_Z p(X, Z|\theta) dZ$$

其中:

- $\theta$  是模型的参数。
- $Z$  是隐变量 (例如在 GMM 中, 隐变量是样本属于哪个高斯分布的类别标记)。

但是直接优化  $\log p(X|\theta)$  通常很难, 因为积分或求和计算复杂。因此, EM 通过引入**隐变量的分布**  $q(Z)$  来转化为更容易处理的形式。

### EM 算法的核心思想: Lower Bound 与 KL 散度

**Jensen 不等式** 是 EM 的核心: 通过引入分布  $q(Z)$ , 我们将  $\log p(X|\theta)$  的优化转化为两项的求和:

$$\log p(X|\theta) \geq \mathcal{L}(q, \theta) = \int q(Z) \log \frac{p(X, Z|\theta)}{q(Z)} dZ$$

其中:

- $\mathcal{L}(q, \theta)$ : 称为 Lower Bound, 表示  $\log p(X|\theta)$  的下界。
- $KL(q||p) = \int q(Z) \log \frac{q(Z)}{p(Z|X, \theta)} dZ$ : 是 KL 散度, 度量  $q(Z)$  与真实后验分布  $p(Z|X, \theta)$  的差异。

由于  $\log p(X|\theta) = \mathcal{L}(q, \theta) + KL(q||p)$ , 最大化  $\mathcal{L}(q, \theta)$  等价于最大化  $\log p(X|\theta)$ 。

- **E 步 (Expectation) : 固定  $\theta$ , 优化  $q(Z)$** , 选择使 KL 散度最小的  $q(Z)$ , 即让  $q(Z) = p(Z|X, \theta)$ 。因此:

$$q(Z) = p(Z|X, \theta^{(t)})$$

这一步是计算隐变量的后验分布。

- **M 步 (Maximization) : 固定  $q(Z)$ , 优化  $\theta$**  将  $q(Z)$  代入 Lower Bound, 并最大化:

$$\theta^{(t+1)} = \arg \max_{\theta} \mathcal{L}(q, \theta)$$

---

**输入:** 样本集  $D = \{x_1, x_2, \dots, x_N\}$ ;  
高斯混合成分个数  $K$

**过程:**

- 1: 初始化高斯混合分布的模型参数  $\{(\pi_k, \mu_k, \Sigma_k) \mid 1 \leq k \leq K\}$
- 2: **repeat**
- EM 算法的 E 步. 3: **for**  $i = 1, 2, \dots, N$  **do**
- 4:     计算  $x_i$  由各混合成分生成的后验概率, 即
- $$\gamma_{ik} = \frac{\pi_k \cdot \mathcal{N}(x_i \mid \mu_k, \Sigma_k)}{\sum_{j=1}^K \pi_j \cdot \mathcal{N}(x_i \mid \mu_j, \Sigma_j)}$$
- 5: **end for**
- EM 算法的 M 步. 6: **for**  $k = 1, 2, \dots, K$  **do**
- 7:     计算新均值向量:  $\mu'_k = \frac{\sum_{i=1}^N \gamma_{ik} x_i}{\sum_{i=1}^N \gamma_{ik}}$
- 8:     计算新协方差矩阵:  $\Sigma'_k = \frac{\sum_{i=1}^N \gamma_{ik} (x_i - \mu'_k)(x_i - \mu'_k)^T}{\sum_{i=1}^N \gamma_{ik}}$
- 9:     计算新混合系数:  $\pi'_k = \frac{1}{N} \sum_{i=1}^N \gamma_{ik}$
- 10: **end for**
- 11: 将模型参数  $\{(\pi_k, \mu_k, \Sigma_k) \mid 1 \leq k \leq K\}$  更新为  $\{(\pi'_k, \mu'_k, \Sigma'_k) \mid 1 \leq k \leq K\}$
- 12: **until** 满足停止条件 (例如已达到最大迭代轮数, 或似然函数  $LL(D)$  增长很少)
- 13:  $C_k = \emptyset$  ( $1 \leq k \leq K$ )
- 14: **for**  $i = 1, 2, \dots, N$  **do**
- 15:     根据下式确定  $x_i$  的簇标记  $\lambda_i$ ;
- $$\lambda_i = \arg \max_{k \in \{1, 2, \dots, K\}} \gamma_{ik}$$
- 16:     将  $x_i$  划入相应的簇:  $C_{\lambda_i} = C_{\lambda_i} \cup \{x_i\}$
- 17: **end for**

**输出:** 簇划分  $C = \{C_1, C_2, \dots, C_K\}$

---

## 2. 关键代码展示

### 【数据预处理】

```

.....
读取mnist_test和mnist_train数据集
mnist_train.csv文件中包含了60000个训练样本 10个类别0~9。
mnist_test.csv文件包含10000个测试样本 10个类别。文件每一行有785个值 第一列是类别标签0~9 其
余784列是手写字体的像素值。
.....

def read_data(train_path, test_path):
    train_data = pd.read_csv(train_path)
    test_data = pd.read_csv(test_path)

    X_train = train_data.iloc[:, 1:].values
    Y_train = train_data.iloc[:, 0].values
    X_test = test_data.iloc[:, 1:].values
    Y_test = test_data.iloc[:, 0].values

    return X_train, Y_train, X_test, Y_test

# 数据预处理
def preprocess(X_train, Y_train, X_test, Y_test):

```

```

X_train = X_train.astype(np.float32) / 255.0
X_test = X_test.astype(np.float32) / 255.0
Y_train = Y_train.astype(np.int32)
Y_test = Y_test.astype(np.int32)

return X_train, Y_train, X_test, Y_test

```

## 【K-Means】

```

# K-means算法
class KMeans:
    def __init__(self, K, epoch = 50, init_method = "random"):
        self.K = K
        self.epoch = epoch
        self.init_method = init_method
        self.train_acc = []
        self.test_acc = []
        self.center = []

    def init_centers(self, X): # 不同的初始化方法，随机选择K个样本点作为初始中心或者使用kmeans++方法
        if self.init_method == "random": # 随机选择K个样本点作为初始中心
            return X[np.random.choice(X.shape[0], self.K, replace=False)]
        elif self.init_method == "kmeans++": # 使用kmeans++方法，具体做法是先随机选择一个样本点作为第一个中心，然后计算每个样本点到最近中心的距离的平方，以此为权重随机选择下一个中心
            center1 = X[np.random.choice(X.shape[0])]
            for k in range(1, self.K):
                dist = np.min(np.linalg.norm(X[:, np.newaxis] - center1[:k], axis=2)**2, axis=1) # 计算每个样本点到最近中心的距离的平方，np.linalg.norm()计算范数
                prob = dist / np.sum(dist) # 以此为权重随机选择下一个中心
                new_center = np.random.choice(X.shape[0], p=prob)
                center1 = np.vstack([center1, X[new_center]])
            return center1
    def fit(self, X_train, Y_train, X_test, Y_test):
        self.centers = self.init_centers(X_train)
        for _ in range(self.epoch):
            dist = np.linalg.norm(X_train[:, np.newaxis] - self.centers, axis=2) # 计算每个样本点到中心的距离
            labels = np.argmin(dist, axis=1) # np.argmin()返回最小值的索引
            new_centers = np.array([X_train[labels == k].mean(axis=0) for k in range(self.K)])
            if np.all(new_centers == self.centers):
                break;
            self.centers = new_centers

            train_acc = eva_acc(labels, Y_train, self.K)
            self.train_acc.append(train_acc)
            test_labels = test(X_test, self.centers)
            test_acc = eva_acc(test_labels, Y_test, self.K)
            self.test_acc.append(test_acc)
            self.center.append(np.linalg.norm(new_centers - self.centers))
            print(f"Iteration {_ + 1}, Train accuracy: {train_acc:.2%}, Test accuracy: {test_acc:.2%}")
        return labels, new_centers

```

## 【EM->GMM】

```

# GMM 类定义
class GMM:
    def __init__(self, K, epoch=10, init_method="random", cov_type = "full",
pca_dim=None):
        self.K = K # 高斯分布的数目
        self.epoch = epoch # 最大迭代次数
        self.init_method = init_method # 初始化方法
        self.pca_dim = pca_dim
        self.mu = None # 均值
        self.sigma = None # 协方差矩阵
        self.pi = None # 混合权重
        self.cov_type = cov_type # 协方差类型
        self.train_acc = []
        self.test_acc = []
        self.epoch_times = [] # 用于存储每个epoch的训练时间
    # 对角且元素值都相等: self.sigma = np.array([np.diag(np.ones(n_features) *
sigma_value)] * self.K)
    # 对角但对元素值不要求相等: self.sigma =
np.array([np.diag(np.random.rand(n_features) * sigma_max_value)] * self.K)
    # 普通: self.sigma = np.array([np.cov(X_train.T)] * self.K)
    def fit(self, X_train, Y_train, X_test, Y_test):
        # 初始化参数
        n_samples, n_features = X_train.shape
        if self.init_method == "random":
            # 随机初始化均值、协方差和权重
            self.mu = X_train[np.random.choice(n_samples, self.K,
replace=False)]
            if self.cov_type == "full":
                self.sigma = np.array([np.cov(X_train.T)] * self.K)
            elif self.cov_type == "diag":
                self.sigma = np.array([np.diag(np.diag(np.cov(X_train.T))) for _
in range(self.K)])
            elif self.cov_type == "diag_same":
                self.sigma =
np.array([np.diag([np.diag(np.cov(X_train.T)).mean()]] * n_features) for _ in
range(self.K))
                self.pi = np.ones(self.K) / self.K
        elif self.init_method == "kmeans++":
            # 使用kmeans++初始化参数
            center1 = X_train[np.random.choice(n_samples)]
            for k in range(1, self.K):
                dist = np.min(np.linalg.norm(X_train[:, np.newaxis] -
center1[:k], axis=2)**2, axis=1)
                prob = dist / np.sum(dist)
                new_center = X_train[np.random.choice(n_samples, p=prob)]
                center1 = np.vstack([center1, new_center])
            self.mu = center1
            if self.cov_type == "full":
                self.sigma = np.array([np.cov(X_train.T)] * self.K)
            elif self.cov_type == "diag":
                self.sigma = np.array([np.diag(np.diag(np.cov(X_train.T))) for _
in range(self.K)])
            elif self.cov_type == "diag_same":
                self.sigma =
np.array([np.diag([np.diag(np.cov(X_train.T)).mean()]] * n_features) for _ in
range(self.K))
                self.pi = np.ones(self.K) / self.K

```

```

# EM算法
for _ in range(self.epoch):
    start_time = time.time() # 记录每个epoch开始时间
    # E步骤: 计算每个点属于每个簇的概率
    gamma = self.e_step(X_train)

    # M步骤: 更新参数
    self.mu, self.sigma, self.pi = self.m_step(X_train, gamma)
    print(f"Epoch {_ + 1} completed")

    # 记录每个epoch的训练时间
    epoch_time = time.time() - start_time
    self.epoch_times.append(epoch_time)

    # 保存每轮的准确率
    train_pred = self.predict(X_train)
    test_pred = self.predict(X_test)
    self.train_acc.append(eva_acc(train_pred, Y_train, self.K))
    self.test_acc.append(eva_acc(test_pred, Y_test, self.K))

def print_epoch_times(self):
    total_time = sum(self.epoch_times)
    print(f"Total training time: {total_time:.2f} seconds")
    for epoch, epoch_time in enumerate(self.epoch_times, 1):
        print(f"Epoch {epoch} time: {epoch_time:.2f} seconds")

def guassian(self, data, mean, cov):
    """
    计算高维高斯分布的概率密度
    :param data: 用于采样的数据
    :param mean: 均值
    :param cov: 协方差
    """
    N = multivariate_normal(mean=mean, cov=cov)
    return N.pdf(data)

# def guassian(self, x, mu, sigma):
#     return np.exp(-0.5 * np.sum(np.dot(x - mu, np.linalg.inv(sigma)) * (x - mu), axis=1)) / (np.sqrt(np.linalg.det(sigma)) * np.power(2 * np.pi, x.shape[1] / 2))

def e_step(self, X):
    n_samples, _ = X.shape
    gamma = np.zeros((n_samples, self.K))
    for i in range(self.K):
        gamma[:, i] = self.pi[i] * self.gaussian(X, self.mu[i], self.sigma[i])
    # 归一化
    gamma /= np.sum(gamma, axis=1, keepdims=True)

    return gamma

def m_step(self, X_train, gamma): # 更新模型参数
    n_samples, _ = X_train.shape
    N_k = np.sum(gamma, axis=0) # 分母

    for k in range(self.K):

```

```

        # 更新均值
        self.mu[k] = np.dot(gamma[:, k], X_train) / N_k[k]
        # 更新协方差
        diff = X_train - self.mu[k]
        self.sigma[k] = np.dot(gamma[:, k] * diff.T, diff) / N_k[k]
        # 更新权重
        self.pi[k] = N_k[k] / n_samples
    return self.mu, self.sigma, self.pi

def predict(self, X):
    # 预测每个点属于哪个簇
    gamma = self.e_step(X)
    return np.argmax(gamma, axis=1)

```

## 【train】

```

# # 评估聚类精度
# def clustering_accuracy(y_true, y_pred):
#     # 找到标签之间的最佳映射 构造混淆矩阵
#     contingency = np.zeros((10, 10))
#     for i in range(len(y_true)):
#         contingency[y_true[i], y_pred[i]] += 1
#     # 通过匈牙利算法找到最佳匹配
#     rows, cols = linear_sum_assignment(-contingency)
#     return contingency[rows, cols].sum() / len(y_true)
def hungarian_algorithm(labels, Y, K):
    """
    使用匈牙利算法对聚类标签进行对齐
    """

    # 构造代价矩阵
    cost_matrix = np.zeros((K, K))
    # 填充代价矩阵
    for i in range(K):
        for j in range(K):
            # 对每个聚类标签 i 和真实标签 j, 计算误差 (例如不匹配数量)
            cost_matrix[i, j] = np.sum((labels == i) & (Y == j))

    # 使用匈牙利算法 (线性求解最优化) 来找到最小化代价的标签映射
    row_ind, col_ind = linear_sum_assignment(-cost_matrix) # 使用负的代价矩阵来最大化匹配
    # 根据匈牙利算法找到的映射, 重新映射聚类标签
    new_labels = np.zeros_like(labels)
    for i in range(K):
        new_labels[labels == i] = col_ind[i]

    return new_labels

def eva_acc(labels, Y, K):
    """
    计算聚类准确率
    """

    labels = np.array(labels)
    Y = np.array(Y)
    # 使用匈牙利算法对齐标签
    aligned_labels = hungarian_algorithm(labels, Y, K)
    # 计算对齐后的准确率
    accuracy = np.mean(aligned_labels == Y)
    return accuracy

```

### 【test】

```
def test(X_test, centers):  
    dist = np.linalg.norm(X_test[:, np.newaxis] - centers, axis=2)  
    labels = np.argmin(dist, axis=1)  
    return labels
```

### 【plot】

```
def plot(train_acc, test_acc):  
    plt.plot(train_acc, label="Train Clustering Accuracy")  
    plt.plot(test_acc, label="Test Clustering Accuracy")  
    plt.xlabel("Epoch")  
    plt.ylabel("Clustering Accuracy")  
    plt.legend()  
    plt.grid(True)  
    plt.show()  
  
# 绘制数据集的分类图  
def plot_classification(X, Y, K):  
    plt.figure(figsize=(8, 6))  
    # 使用PCA降维到二维  
    pca = PCA(n_components=2)  
    X_pca = pca.fit_transform(X)  
  
    # 绘制散点图, 不同的类别用不同的颜色表示  
    plt.scatter(X_pca[:, 0], X_pca[:, 1], c=Y, cmap='tab10', s=20)  
    plt.colorbar() # 显示颜色条  
    plt.title(f"Classification Plot (K={K})")  
    plt.xlabel("PCA 1")  
    plt.ylabel("PCA 2")  
    plt.show()
```

## 四、实验结果及分析

### K-Means

#### 【训练过程】——初始化方法、超参数选择、用到的训练技巧

【初始化方法】K-Means 算法的性能受初始化方法影响很大。不同的初始化方法决定了聚类中心的起始位置，进而影响算法的收敛速度和结果的质量。本次实验中我采用的初始化方法有：

1. 随机初始化 (Random Initialization) 随机选择 K 个数据点作为初始聚类中心。但可能导致算法陷入局部最优，尤其是在数据集中不同簇的分布不均匀时。但由于随机初始化可能导致聚类中心分布不均匀，容易陷入局部最优，因此可以考虑增加n\_init (初始化次数) 来多次运行K-Means，每次使用不同的初始中心，最终选择最优的聚类结果。
2. K-means++ 初始化：通过对数据点间的距离进行加权选择初始聚类中心。首先随机选择一个点作为第一个中心，然后根据当前聚类中心的距离平方来加权选择下一个中心，直到选出 K 个初始中心。能够有效避免随机初始化带来的坏结果，通常比随机初始化的性能更好，能够加速收敛。但是计算量相对较大，因为每次选择中心时需要计算每个数据点到最近中心的距离。

#### 【超参数选择】

1. K (簇的数量) : K 是 K-Means 的主要超参数, 选择 K 的值通常较为困难。常见的选择方法有:

- 肘部法则 (Elbow Method) : 绘制不同 K 值下的误差平方和 (SSE) 与 K 值的关系图。当误差平方和的下降趋于平缓时, 选择该 K 值作为合适的簇数。
- 轮廓系数 (Silhouette Score) : 通过计算每个数据点与其簇内其他点的相似度以及与最近簇的相似度, 来评估簇的紧密度与分离度, 轮廓系数越大, 说明聚类效果越好。
- Gap Statistic: 通过比较不同 K 值下的聚类效果与一个参考数据集 (如随机数据集) 来选择最佳的 K 值。

本实验由于已经给出K的数量, 因此我们不需要讨论K的数量。K即为10

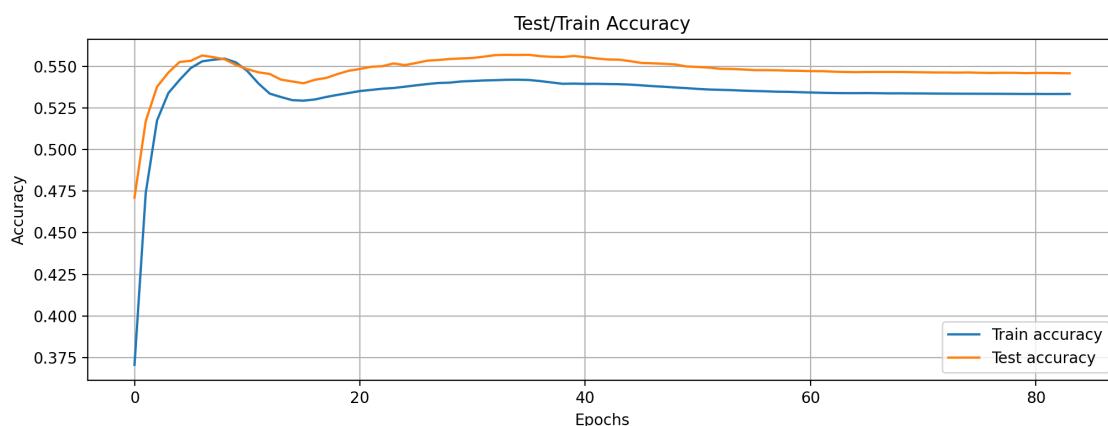
2. 最大迭代次数 (max\_iter) : 最大迭代次数限制了算法运行的时间。在一些非常大的数据集上, 算法可能需要更多的迭代才能收敛, 因此选择合理的最大迭代次数非常重要。

本次实验设置为 50 次, 具体可根据数据量和聚类效果来调整。

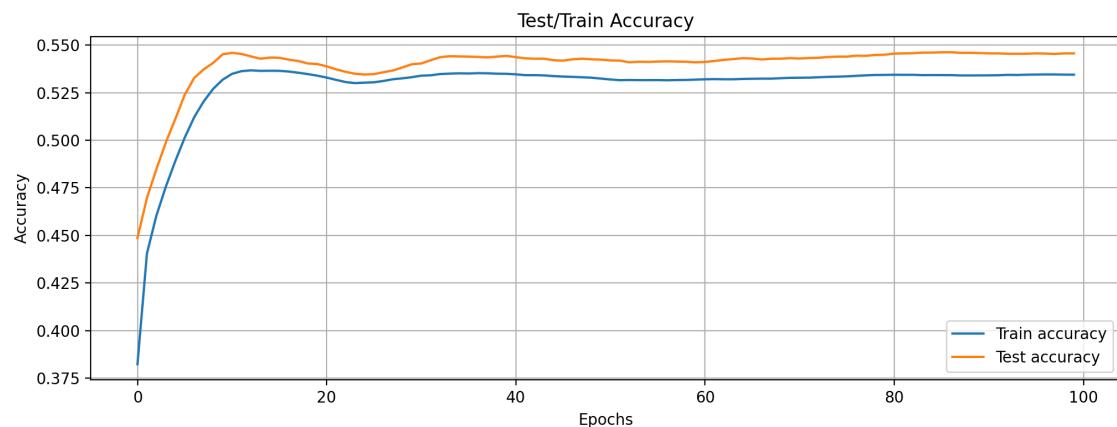
【用到的训练技巧】匈牙利算法通常用于最优匹配问题, 特别是在K-Means聚类中, 它可以解决聚类结果标签与真实标签之间的最优匹配。由于K-Means算法产生的簇标签并不与真实标签顺序一致, 匈牙利算法可以将簇标签与实际类别标签进行最优匹配, 从而提高评估准确性。再训练的过程中我使用了该算法来评估准确性。

| 评测\初始化方法 | 随机初始化          | k-means++初始化方法 |
|----------|----------------|----------------|
| 训练时间     | 172.29 seconds | 153.66 seconds |
| 训练集准确率   | 53.23%         | 53.45%         |
| 测试集准确率   | 54.58%         | 54.57%         |

### 【Random】



### 【Kmeans++】



### 【分析】

我们可以看到当你使用**random初始化**时，聚类中心选择的随机性导致聚类性能不稳定，因此准确率较低，大概只有53%左右。因为随机初始化方法直接从数据集中随机选择K个点作为初始聚类中心。这种方式没有考虑样本之间的距离分布，因此可能导致某些簇的初始化选择不理想，尤其是当数据分布不均匀时。但反观我们再看**k-means++**，克服random初始化方法的不足，通过一种概率方法来选择初始聚类中心，优先选择数据集中的那些离其他点较远的样本作为聚类中心。具体来说，第一个聚类中心随机选择，接下来的每个聚类中心是根据与已有聚类中心的距离的平方反比的概率分布选择的。**k-means++**能够选取初始聚类中心，使得簇之间的距离尽量远离其他簇，从而提高聚类性能。相比random初始化，它可以有效避免初始聚类中心选择不均匀的问题，减少收敛时的迭代次数。其实可以看到这两个准确率差不多，但实际这个点的选择是随机的，因此准确率接近也有可能，但是一般来说，**k-means++**的初始化方法下的准确率应该更高一点。

在训练k-means++的时候有一次准确率先达到了60%附近，但在之后又降到了55%附近，通过查阅网上资料，了解到可能是以下原因：开始时准确率达到60%，可能是因为初始中心选择较好，收敛速度快，聚类效果较好。而随着迭代进行，聚类中心逐渐调整并且适应数据集中的局部结构，最终可能会在一些“边界”或“噪声”数据上产生过拟合或者不太合适的划分，导致准确率下降。另外有可能是K-Means算法本身具有局部最优解的特点，即它可能在收敛后找到的解不是全局最优解。尽管k-means+能够帮助初始聚类中心更合理地分布，避免了随机初始化时的低质量结果，但在训练过程中，随着聚类中心的更新，算法仍可能陷入局部最优解。特别是在数据存在较强的噪声、离群点，或者簇形状不规则时，K-Means容易“收敛”到一个不理想的解，从而导致准确率出现波动。

## GMM

### 【训练过程】——初始化方法、超参数选择、用到的训练技巧

#### 【初始化方法】

1. 随机初始化随机选择 K 个数据点作为初始聚类中心。在这种方法下，模型的参数（例如聚类中心）被随机选择。虽然这种方法简单，但随机性可能导致初始点落在不利的区域，造成模型从一开始就陷入较差的局部最优解，收敛速度可能较慢，甚至可能完全错过全局最优解。
2. K-means++ 初始化：通过对数据点间的距离进行加权选择初始聚类中心。首先随机选择一个点作为第一个中心，然后根据当前聚类中心的距离平方来加权选择下一个中心，直到选出 K 个初始中心。能够有效避免随机初始化带来的坏结果，通常比随机初始化的性能更好，能够加速收敛。但是计算量相对较大，因为每次选择中心时需要计算每个数据点到最近中心的距离。

#### 【超参数选择】

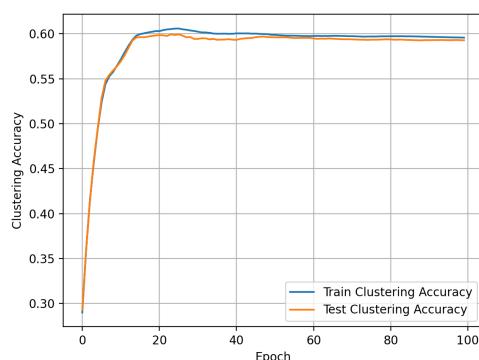
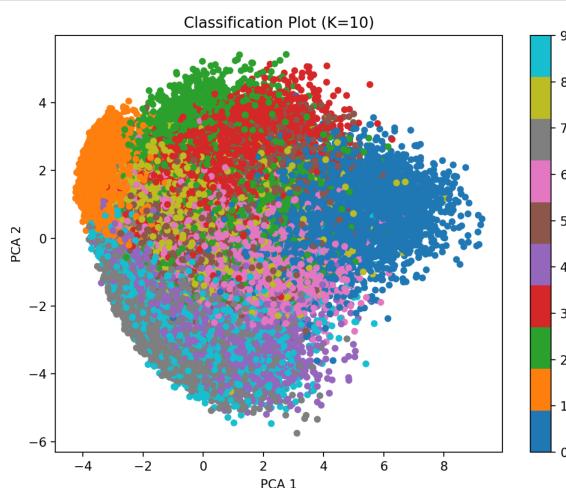
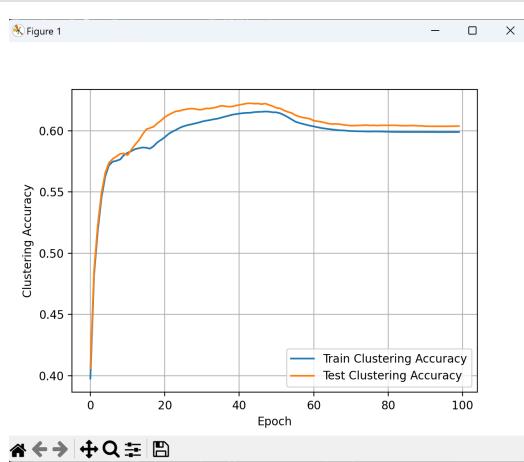
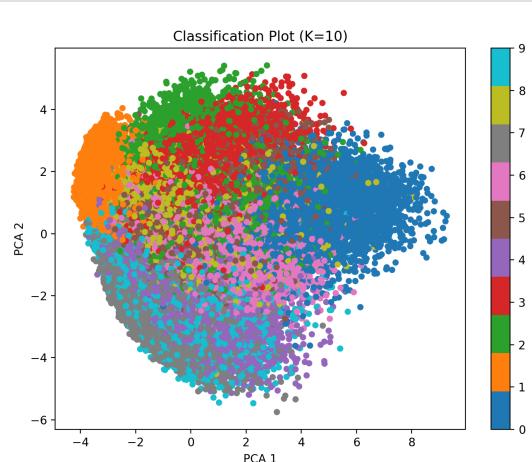
1. 学习率：根据模型训练的收敛情况，选择了适当的学习率范围。
2. 正则化参数：通过实验选定最优的正则化强度，平衡模型的拟合能力。
3. 初始化方式：对比随机初始化与k-means++初始化，以获得更高性能。
4. 协方差矩阵形式：设置三种形式（普通、对角不等值、对角等值）以评估不同假设下的模型性能。
5. PCA：使用不同的PCA维数

#### 【用到的训练技巧】

匈牙利算法通常用于最优匹配问题，特别是在聚类中，它可以解决聚类结果标签与真实标签之间的最优匹配。由于聚类算法产生的簇标签并不与真实标签顺序一致，匈牙利算法可以将簇标签与实际类别标签进行最优匹配，从而提高评估准确性。再训练的过程中我使用了该算法来评估准确性。

#### 初始化的影响

| 评测\初始化方法 | 随机初始化          | k-means++初始化方法 |
|----------|----------------|----------------|
| 训练时间     | 108.33 seconds | 80.92 seconds  |
| 训练集准确率   | 0.59575        | 0.59903        |
| 测试集准确率   | 0.5928         | 0.6039         |

**随机初始化训练曲线****聚类图****k-means初始化训练曲线****聚类图**

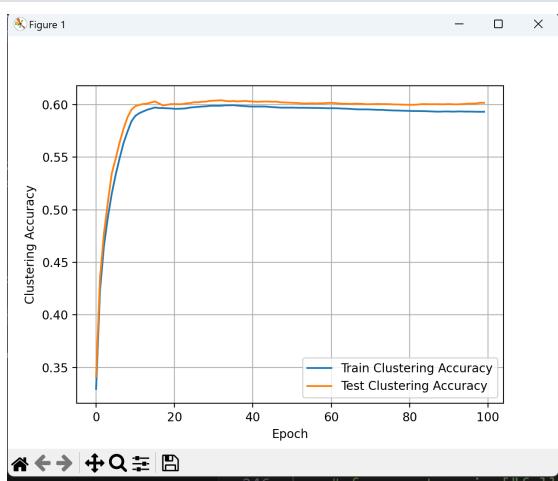
## 【分析】

- 收敛速度：** k-means++初始化通过合理选择初始聚类中心，避免了随机初始化可能产生的初始模型不良，从而减少了训练中的波动，提高了训练效率。训练时间上，k-means++初始化显著低于随机初始化（80.92秒 vs 108.33秒）。因此，k-means++初始化有助于加速模型的收敛。
- 测试集准确率：** 由于k-means++初始化能够有效选择较为合理的起始点，避免了模型从不好的初始点开始，因此能够在训练过程中保持较好的准确性，最终在测试集上的准确率也相对较高（0.6039 vs 0.5928）。这说明更好的初始化方法能够帮助模型获得更好的泛化能力，避免过拟合。

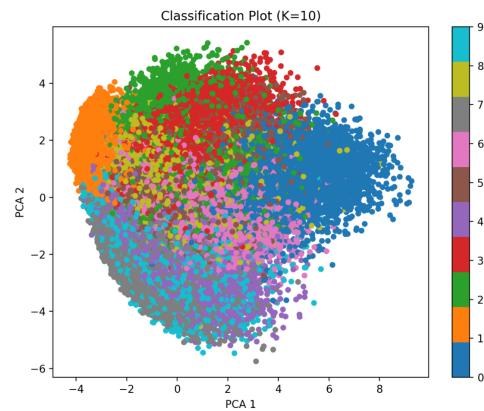
## 降维的影响

| 评测\PCA | 200           | 100            | 50            |
|--------|---------------|----------------|---------------|
| 训练时间   | 94.48 seconds | 119.15 seconds | 98.53 seconds |
| 训练集准确率 | 0.59328       | 0.6279         | 0.61525       |
| 测试集准确率 | 0.6018        | 0.6279         | 0.6227        |

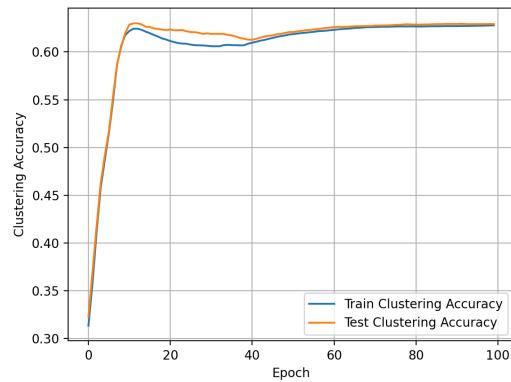
### 200维训练曲线



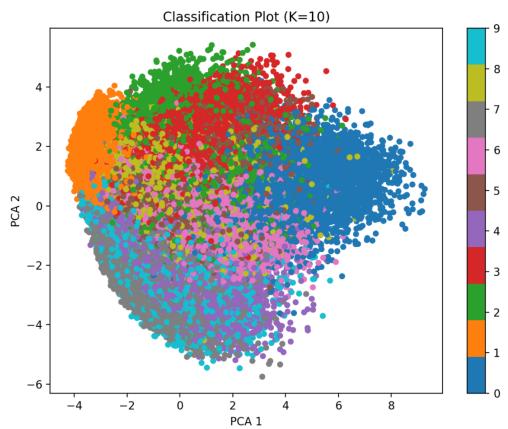
### 聚类图



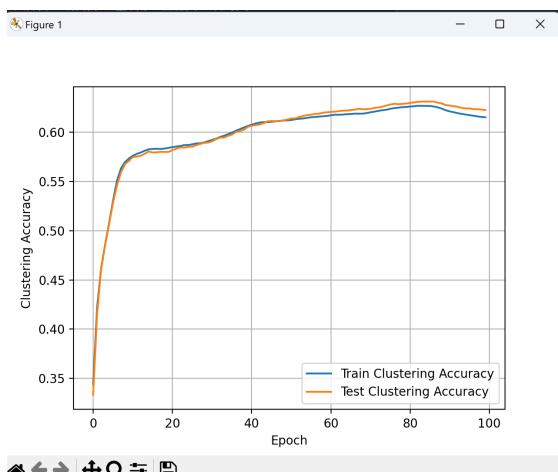
### 100维训练曲线



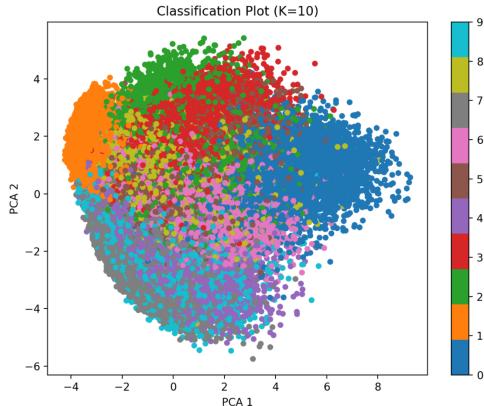
### 聚类图



### 50维训练曲线



### 聚类图



## 【分析】

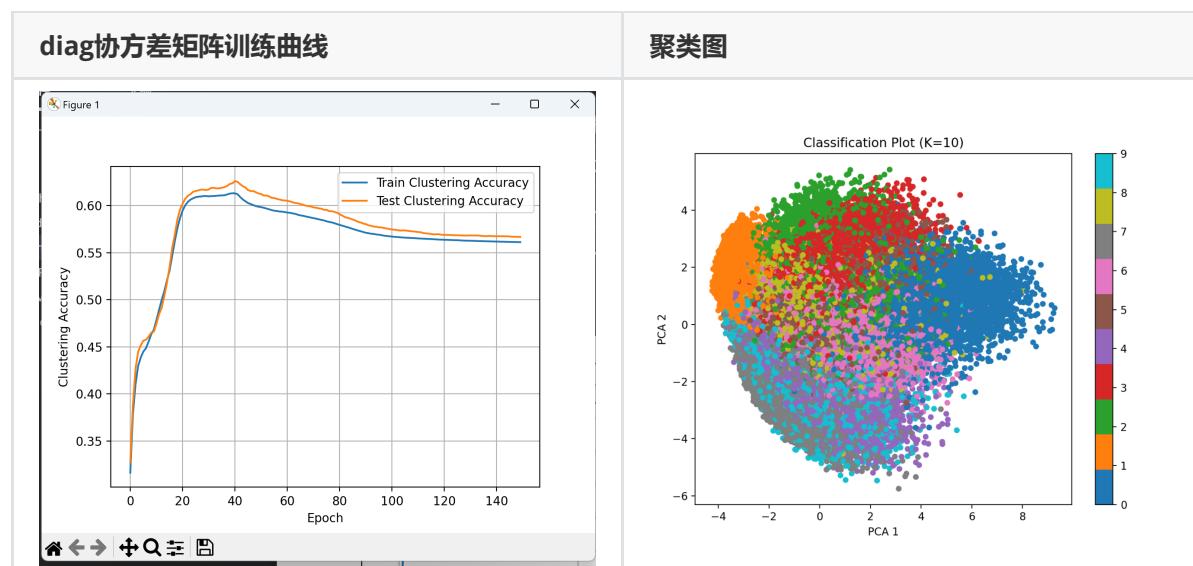
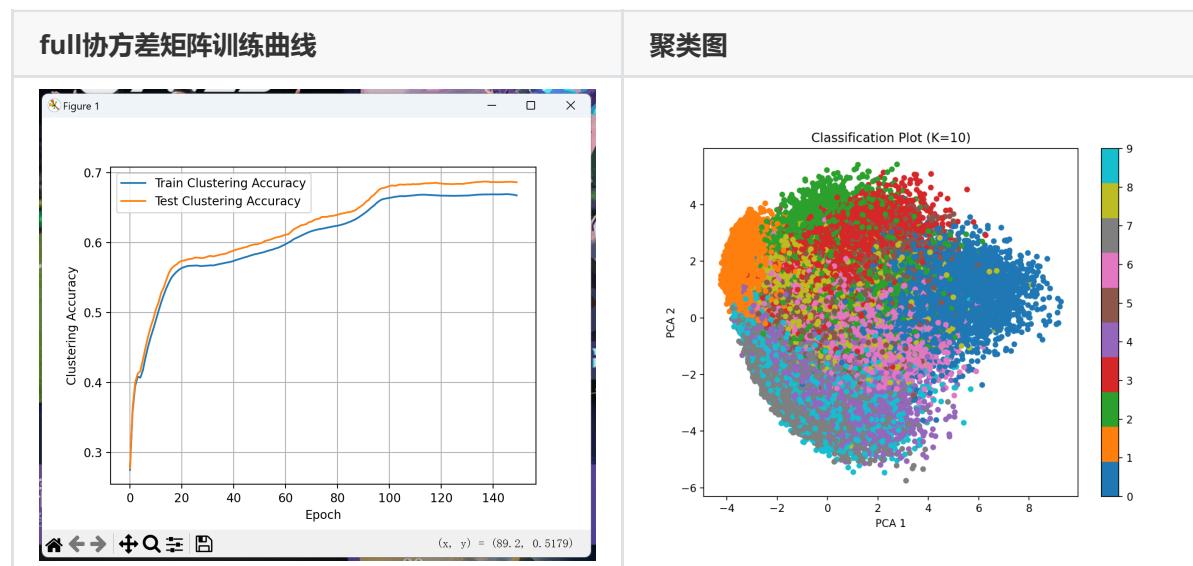
- 准确率：在100维的情况下，训练集准确率最高，达到了0.6279。
- 50维和200维的训练集准确率较低，分别为0.61525和0.59328。可以看出，**较低维度（50维）** 和 **较高维度（200维）** 相较于100维存在一定的精度损失。其中200维的测试集准确率明显低于100维和50维，可能是由于在高维空间中，模型过拟合的风险较高，导致测试集的泛化能力下降。
- 较高维度（200维）：** 高维度下的特征空间虽然包含了更多的信息，但可能导致了噪声和冗余信息的增多。虽然计算量没有显著增加，但过多的无关特征可能会影响模型的精度，导致测试集的准确率较低。

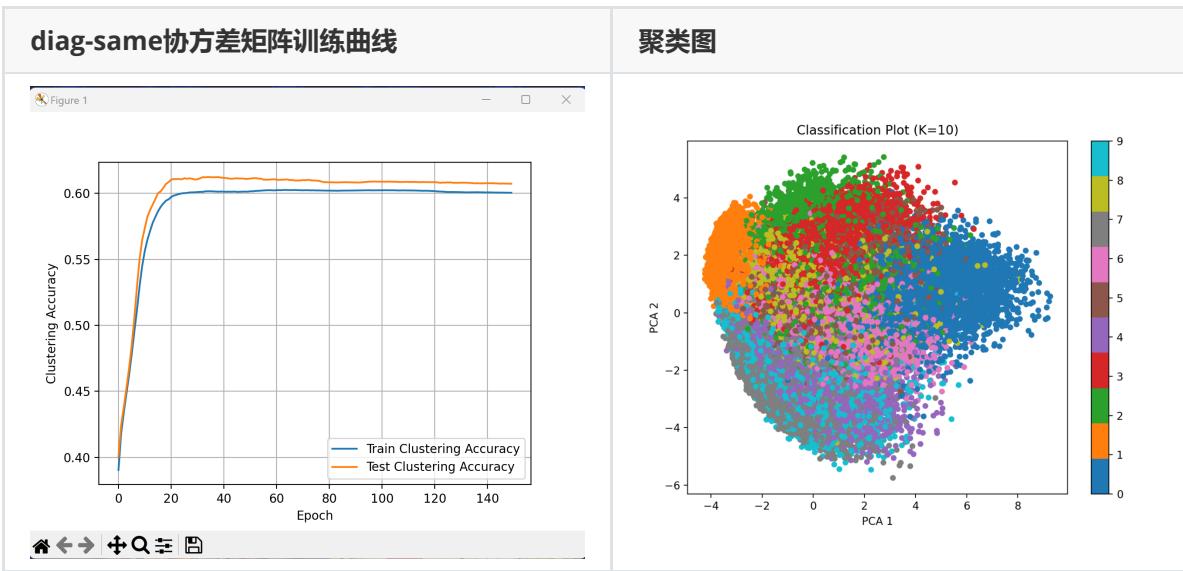
**较低维度（50维）**：低维度下的数据压缩损失可能导致丢失了一些重要的信息，虽然测试集的准确率略有提高，但与100维的表现接近。这表明，50维的降维设置在某些情况下仍能保持较好的效果，但无法完全保留原始数据的丰富特征。

**适中维度（100维）**：通过PCA选择合适的维度进行降维，使得训练和测试的效果最为均衡。100维下的训练和测试准确率较高，且计算开销适中。显然，PCA的降维过程在这个维度下最有效，既减少了计算量，又保留了大部分有效特征。

### 不同协方差矩阵的影响

| 评测\协方差矩阵形式 | 普通矩阵           | 对角不等值矩阵        | 对角等值矩阵         |
|------------|----------------|----------------|----------------|
| 训练时间       | 168.45 seconds | 162.45 seconds | 142.03 seconds |
| 训练集准确率     | 0.66775        | 0.5616         | 0.600483       |
| 测试集准确率     | 0.6866         | 0.5668         | 0.6161         |





## 【分析】

在训练过程中，协方差矩阵的不同形式也对模型的表现产生了影响。普通矩阵：在这种形式下，协方差矩阵包含了所有特征之间的协方差信息。这意味着每两个特征之间的相关性都会被考虑。对于特征之间关系复杂的数据，普通矩阵能够捕捉到这些关系，但计算开销较大，训练时间也较长。对角不等值矩阵：在这种形式下，协方差矩阵的非对角元素被设为零，只保留了各个特征的方差。这意味着假设不同特征之间存在不同的独立性。与普通矩阵相比，它能大幅降低计算复杂度，同时也能够捕捉到各个特征的单独变化。对角等值矩阵：这种形式进一步简化了模型，假设所有特征的方差相等，且特征之间完全独立。尽管它极大地减少了模型的复杂度，可能无法完全捕捉到数据的特征间复杂关系，但对于一些数据分布相对均匀的任务，它能提供不错的性能，且训练速度较快。

- 训练时间：普通矩阵考虑了特征间所有的协方差，训练时间较长，约为168.45 seconds。而对角矩阵形式（无论是否对角不等值）简化了计算，训练时间显著减少。对角等值矩阵的训练时间为162.45 seconds，仍然高于对角不等值矩阵142.03 seconds，这表明对角等值矩阵的简化假设可能使得模型泛化能力更好，但训练过程中略微增加了计算负担。
- 准确率：普通矩阵的准确率最高（训练集：0.66775，测试集：0.6866）。这是因为在这种形式下，协方差矩阵包含了所有特征之间的协方差信息。这意味着每两个特征之间的相关性都会被考虑。对于特征之间关系复杂的数据，普通矩阵能够捕捉到这些关系相比之下。对角不等值矩阵（训练集：0.5616，测试集：0.5668）简化了模型，减少了特征间的相关性，可以看到训练结果准确率很低。对角等值矩阵具有最简单的假设，假设特征之间完全独立且方差相同，相较于对角不等值训练结果较好，但相对于普通矩阵训练结果不好。训练集准确率为0.600483，测试集准确率为0.6161，原因可能是该数据集的特征间关系较为独立，且对角等值矩阵的假设较好地匹配了数据的内在结构。

## K-Means和GMM的对比

其实K-means和GMM主要有四个区别，一个是k-means算法是非概率模型，而GMM是概率模型。具体来讲就是，k-means算法基于欧氏距离的度量方式来将样本划分到与它距离最小的簇类，而GMM则是计算由各个高斯分布生成样本的概率，将样本划分到取得最大概率的高斯分布中。一个是两者需要计算的参数不同，k-means计算的是簇类的均值，GMM计算的是高斯分布的参数（即均值、方差和高斯混合系数）。还有k-means是硬聚类，要么属于这一类要么属于那一类；而GMM算法是软聚类，给出的是属于某些类别的概率。另外GMM每一步迭代的计算量比k-means要大。当然他们也有一定的联系，都是聚类算法，都需要指定K值，且都受初始值的影响。k-means初始化k个聚类中心，GMM初始化k个高斯分布，都是通过迭代的方式求解，而且都是局部最优解。k-means的求解过程其实也可以用EM算法的E步和M步来理解。

| 评测标准     | K-means                  | GMM                            |
|----------|--------------------------|--------------------------------|
| 聚类效果     | 对球形簇效果好，非球形簇效果差，容易受离群点影响 | 适应性强，可以处理复杂的、非球形簇，且对离群点不敏感     |
| 聚类方法     | 硬聚类（每个数据点仅属于一个簇）         | 软聚类（每个数据点可以属于多个簇）              |
| 计算复杂度    | 较低，适合大规模数据集              | 较高，计算量大，尤其在高维数据中               |
| 对初始值的敏感性 | 对初始聚类中心敏感，可能陷入局部最优       | 对初始参数（高斯分布参数）敏感                |
| 模型假设     | 假设各个簇的形状为球形且大小相似         | 允许每个簇有不同的形状、方向和大小（高斯分布的协方差矩阵）  |
| 适用场景     | 聚类结构简单、球形且较为均匀的数据        | 适用于复杂的、多模态数据，尤其是存在重叠簇或不规则形状的数据 |
| 稳定性      | 相对较差，可能受离群点和局部最优解影响      | 稳定性较好，能处理复杂模式的数据               |

- **K-means**适用于结构简单、球形的聚类问题，特别是在数据较为均匀且分布明确时，K-means具有较好的计算效率和较低的复杂度。
- **GMM**适用于更为复杂的聚类问题，能够处理非球形簇、重叠簇和不同密度的簇。它通过软聚类为每个数据点提供更多的可能性，并且能够拟合复杂的数据分布。然而，GMM的计算复杂度较高，训练时对参数选择和初始化比较敏感。

最终选择使用K-means还是GMM取决于问题的复杂性以及数据的分布特点。在简单的聚类问题中，K-means往往能提供快速且足够好的结果；而在需要捕捉复杂模式、分布不规则的数据时，GMM无疑是更好的选择。

## 五、实验总结 (remark)

本次实验分为两个部分，一个是K-means，另一个是使用EM实现的GMM，对于K-means,分别使用了两种不同的初始化方法初始化中心点的选择，由于是无监督学习，因此我们在训练模型的时候不需要给出正确的标签值，只需要给出x的数据值训练即可，最后为了探寻训练的准确率，使用了匈牙利算法来进行最优匹配，确定准确率。但是我们可以看到最终的准确率并不是很高，使用kmeans++的初始化方法，最高也才到达60%左右，使用随机初始化方法甚至仅有53%左右的准确率，由此可见无监督学习在聚类问题上还是存在一定的误差的。接着我实现了GMM算法，GMM其实也是一种无监督的聚类学习，但是其相对于K-means来说具有更强的聚类效果，他不同于kmeans的硬分类方法——每一个类都仅属于一个聚类中心，GMM使用多种高斯分布的混合模型来模拟每个点属于多个聚类中心的概率，最后我们通过EM来更新GMM的各种参数，可以看到GMM在聚类的准确率方法明显高于Kmeans方法。

在实现GMM的时候遇到了很多问题，首先是运行代码后cpu利用率达到100%，无法训练，解决办法是使用PCA将数据降维。其次是在训练的时候发现训练集的准确率在60%，但是测试集的准确率仅有15%左右，这是因为测试集和训练集的PCA不一致，导致测试集无法使用训练集训练得到的参数值，改变PCA的设置后（即需要测试集与训练集的数据一起降维）准确率就恢复正常了。另外就是在画图的时候，有遇到了维数不一致的原因（也是PCA导致的原因），这里改变PCA的一些参数就正常了。

总的来说，本次实验在代码上还是遇到了很多的Bug，修改GMM的代码就修改了接近一天，主要是这次的所有函数都是自己实现的，没有借助机器学习的库，这也导致可能在很多细节方面没有考虑到，特别是维数的处理QAQ。经过本次实验，我对python的数据处理语言和机器学习语言都有了一个很大的提升，同时也理解和掌握了老师上课讲的知识，自己梳理和推导一遍公式对聚类算法有了更深入的理解，

总之，本次实验还是掌握了很多很多，希望自己能够在之后的学习上更进一步！

---

## 六、参考文献

[k-means](#)

[GMM](#)

[EM](#)