《模式识别与机器学习 A》实验报告

实验题目：实现 k-means 聚类方法和混合高斯模型

学号：

姓名：

1. 实验目的

本实验旨在掌握 k-means 算法和混合高斯模型的原理和实现方法，以及 EM 算法的基本思想和步骤。

本实验还旨在比较 k-means 算法和混合高斯模型的优缺点，以及在不同数据集上的聚类效果。

2. 实验内容

本实验分为三个部分：

第一部分：用高斯分布产生 k 个高斯分布的数据（不同均值和方差），并用 k-means 算法进行聚类，观察聚类结果和真实标签的一致性。

第二部分：用混合高斯模型和 EM 算法对同样的数据进行聚类，观察每次迭代后似然值的变化情况，以及与真实标签的一致性。

第三部分：用 iris 数据集进行聚类，观察混合高斯模型在该数据集上的表现。

3. 实验环境

本实验使用 Python 语言编写代码，运行在 Window11 Vscode 上。

本实验使用 numpy, matplotlib, scipy, pandas, sklearn 等常用库进行数据处理和可视化。

4. 实验过程、结果及分析

4.1 k-means 算法的原理如下：

k-means 算法是一种基于距离的聚类算法，它通过迭代地更新簇中心和簇标签来最小化每个簇内样本点与簇中心之间的平方误差和。它是一种简单而有效的聚类算法，适用于数据集较大且簇形状较规则的情况。

k-means 算法的步骤如下：

- 随机初始化 k 个聚类中心点 $\mu_1, \mu_2, \ldots, \mu_k$；
- 重复直到收敛：

  对于每个样本点 x_i，计算其与 k 个中心点的距离，将其分配给距离最近的中心点所代表的簇 $c_i = \text{argmin}_c \parallel x_i - \mu_c \parallel^2$；

  对于每个簇 c，更新其聚类中心为该簇内所有样本点的均值 $\mu_c = \frac{\sum_{i:c_i=c} x_i}{\#\{i:c_i=c\}}$；

k-means 算法的目标函数为：

$$J(\mu_1, \mu_2, \ldots, \mu_k) = \sum_{i=1}^{n} \| x_i - \mu_{c_i} \|^2$$

k-means 算法使用了 EM（期望最大化）算法的思想，即交替进行 E 步（期望步）和 M 步（最大化步）。E 步是根据当前的参数估计每个样本点的隐变量（即簇标签），M 步是根据当前的隐变量估计参数（即聚类中心）。EM 算法可以保证每次迭代都不会降低目标函数的值，但可能会收敛到局部最优解。

4.2 混合高斯模型的原理如下：

混合高斯模型（Gaussian Mixture Model，GMM）是一种基于概率的聚类算法，它假设数据集是由多个高斯分布混合而成的，每个高斯分布对应一个簇。它使用 EM 算法来估计每个高斯分布的参数，即权重，均值和协方差，以及每个样本点属于每个高斯分布的后验概率。它是一种软聚类算法，可以给出样本点属于不同簇的概率，而不是确定的标签。

混合高斯模型的公式为：

$$p(x|\theta) = \sum_{c=1}^{k} \pi_c \, N(x|\mu_c, \Sigma_c)$$

其中 $\theta = \{\pi_1, \pi_2, \ldots, \pi_k, \mu_1, \mu_2, \ldots, \mu_k, \Sigma_1, \Sigma_2, \ldots, \Sigma_k\}$ 是模型参数，π_c 是第 c 个高斯分布的权重，满足 $\sum_{c=1}^{k} \pi_c = 1$，μ_c 和 Σ_c 是第 c 个高斯分布的均值和协方差。

混合高斯模型的步骤如下：

- 随机初始化 k 个高斯分布的参数 θ；
- 重复直到收敛：
  - 对于每个样本点 x_i，计算其属于每个高斯分布 c 的后验概率 $p(c|x_i, \theta) = \frac{\pi_c N(x_i|\mu_c, \Sigma_c)}{\sum_{c'=1}^{k} \pi_{c'} N(x_i|\mu_{c'}, \Sigma_{c'})}$；
  - 对于每个高斯分布 c，更新其参数为：

$$\mu_c = \frac{\sum_{i=1}^{n} p(c|x_i, \theta) x_i}{\sum_{i=1}^{n} p(c|x_i, \theta)} \quad \Sigma_c = \frac{\sum_{i=1}^{n} p(c|x_i, \theta)(x_i - \mu_c)(x_i - \mu_c)^T}{\sum_{i=1}^{n} p(c|x_i, \theta)} \quad \pi_c = \frac{\sum_{i=1}^{n} p(c|x_i, \theta)}{n}$$

混合高斯模型的目标函数为：

$$L(\theta) = \log p(X|\theta) = \sum_{i=1}^{n} \log \sum_{c=1}^{k} \pi_c \, N(x_i|\mu_c, \Sigma_c)$$

混合高斯模型也使用了 EM 算法的思想，E 步是根据当前的参数估计每个样本点的隐变量（即后验概率），M 步是根据当前的隐变量估计参数（即权重，均值

和协方差）。EM 算法可以保证每次迭代都不会降低目标函数的值，但可能会收敛到局部最优解。

4.3 EM 算法的原理如下：

EM 算法是一种迭代优化策略，用于含有隐变量的概率模型参数的极大似然估计或极大后验估计。

EM 算法的每次迭代由两步组成：E 步（期望步）和 M 步（最大化步）。

E 步是根据当前的参数估计每个样本的隐变量（即后验概率），M 步是根据当前的隐变量估计参数（即权重，均值和协方差）。

EM 算法可以保证每次迭代都不会降低目标函数的值，但可能会收敛到局部最优解。

下面是一个数学公式的推导过程：

假设我们有一个含有隐变量Z和观测变量X的概率模型，其联合概率分布为$p(X,Z|\theta)$，其中θ是模型参数。

我们的目标是根据观测数据X来估计参数θ，使得对数似然函数 $\log p(X|\theta)$最大化。

对数似然函数可以写成：

$$\log p(X|\theta) = \log \sum_{Z} p\,(X,Z|\theta)$$

这个函数很难直接优化，因为它涉及到对隐变量Z的求和。我们可以引入一个辅助分布$q(Z)$，使得$q(Z)$满足以下条件：

$$\sum_{Z} q\,(Z) = 1, \quad q(Z) \geq 0$$

然后我们可以将对数似然函数改写成：

$$\begin{aligned}
\log p(X|\theta) \quad &= \log \sum_{Z} p\,(X,Z|\theta) \\
&= \log \sum_{Z} q\,(Z) \frac{p(X,Z|\theta)}{q(Z)} \\
&\geq \sum_{Z} q\,(Z) \log \frac{p(X,Z|\theta)}{q(Z)}
\end{aligned}$$

上面的不等式利用了 Jensen 不等式，当且仅当$q(Z) = p(Z|X,\theta)$时取等号。

我们定义一个函数$L(q,\theta)$为：

$$L(q,\theta) = \sum_{Z} q\,(Z)\log\frac{p(X,Z|\theta)}{q(Z)}$$

这个函数是对数似然函数的一个下界，我们可以通过交替地优化q和θ来提高这个下界，从而提高对数似然函数的值。

E 步：固定θ，优化q。由于当$q(Z) = p(Z|X,\theta)$时，下界$L(q,\theta)$达到最大值，所以我们令：

$$q^{(t+1)}(Z) = p\left(Z|X,\theta^{(t)}\right)$$

M 步：固定q，优化θ。由于$L(q,\theta)$是关于θ的线性函数，所以我们令：

$$\theta^{(t+1)} = \mathrm{argmax}_{\theta} L\left(q^{(t+1)},\theta\right)$$

这样，我们就得到了 EM 算法的迭代过程：

初始化参数$\theta^{(0)}$

重复直到收敛：

E 步：计算后验概率$q^{(t+1)}(Z) = p\left(Z|X,\theta^{(t)}\right)$

M 步：更新参数$\theta^{(t+1)} = \mathrm{argmax}_{\theta} L\left(q^{(t+1)},\theta\right)$

第一部分：用高斯分布产生 k 个高斯分布的数据（不同均值和方差），并用 k-means 算法进行聚类

实验步骤：

1. 定义一个函数 generate_data(k, n, d)，用于生成 k 个 n 维高斯分布的数据，每个高斯分布有 n 个样本点，d 为维度。函数返回一个包含所有样本点的数组 data，以及每个高斯分布的均值 mu 和协方差 sigma。

```
# 生成 k 个高斯分布的数据，每个分布有 n 个样本，d 是维度
def generate_data(k, n, d):
    # 随机生成 k 个均值向量和协方差矩阵
    mu = np.random.randn(k, d)
    sigma = np.random.rand(k, d, d)
    sigma = np.matmul(sigma, sigma.transpose(0, 2, 1)) # 保证协方差矩阵是对称正定的
    # 按照均值和协方差生成数据
    data = np.zeros((n * k, d))
    for i in range(k):
        data[i * n : (i + 1) * n] = np.random.multivariate_normal(mu[i], sigma[i], n)
    return data, mu, sigma
```

2.调用该函数生成 3 个二维高斯分布的数据，每个高斯分布有 1000 个样本点。

3.定义一个类 KMeans(object)，用于实现 k-means 算法。该类有以下属性和方法：

- __init__(self, k)：初始化方法，接受一个参数 k，表示聚类个数。该方法还定义了 self.centers, self.labels 两个属性，分别表示簇中心和簇标签。

```
    def __init__(self, k):
        self.k = k # 聚类的个数
        self.centers = None # 每个聚类的中心点
        self.labels = None # 每个样本的聚类标签
```

- init_centers(self, data, init='random')：初始化簇中心的方法，接受两个参数 data 和 init。data 表示输入数据，init 表示初始化方式，默认为'random'，即随机选择 k 个数据点作为初始簇中心。也可以选择'kmeans++'方式，即先随机选择一个数据点作为第一个簇中心，然后根据每个数据点到已有簇中心的距离概率选择下一个簇中心，直到选满 k 个。

```
    def init_centers(self, data, init='random'):
        n, d = data.shape # 样本数和维度
        if init == 'random': # 随机初始化
            self.centers = data[np.random.choice(n, self.k)] # 随机选择 k 个样本作为中心点
        elif init == 'kmeans++': # k-means++初始化
            self.centers = [data[np.random.choice(n)]] # 随机选择一个样本作为第一个中心点
            for i in range(1, self.k): # 循环选择剩余的中心点
                dist = np.array([np.min([np.linalg.norm(x - c) for c in self.centers]) for x in data]) # 计算每个样本到已有中心点的最小距离
                prob = dist / np.sum(dist) # 计算每个样本被选为中心点的概率，距离越大概率越高
                self.centers.append(data[np.random.choice(n, p=prob)]) # 按照概率选择一个样本作为中心点
            self.centers = np.array(self.centers) # 转换为数组形式
        else:
            raise ValueError('Invalid value for init: {}'.format(init))
```

- nearest_center(self, data)：计算每个数据点到最近簇中心的距离和标签的方法，接受一个参数 data，表示输入数据。该方法返回一个包含簇标签和簇距离的元组。簇标签是一个一维数组，表示每个数据点所属的簇的索引。簇距离是一个一维数组，表示每个数据点到其所属簇中心的距离。该方法使用欧氏距离作为距离度量。

```
    def nearest_center(self, data):
```

```
        n = data.shape[0]
        dist = np.zeros((n, self.k)) # 距离矩阵，每行表示一个样本到各个中心点的距离
        for i in range(self.k):
          dist[:, i] = np.linalg.norm(data - self.centers[i], axis=1) # 计算欧氏距离
        labels = np.argmin(dist, axis=1) # 取最小距离对应的索引作为聚类标签
        dist = np.min(dist, axis=1) # 取最小距离作为返回值
        return labels, dist
```

– fit(self, data, init='random', max_iter=100)：训练模型的方法，接受三个参数 data, init, max_iter。data 表示输入数据，init 表示初始化方式，默认为'random'，max_iter 表示最大迭代次数，默认为 100。该方法首先调用 init_centers 方法初始化簇中心，然后循环执行以下步骤，直到达到最大迭代次数或者簇中心不再变化：

调用 nearest_center 方法计算每个数据点的簇标签和簇距离，并更新 self.labels 属性。

根据每个簇中的数据点的均值更新簇中心，并更新 self.centers 属性。

```
    def fit(self, data, init='random', max_iter=100):
    self.init_centers(data, init) # 初始化中心点
    for i in range(max_iter):
      old_centers = self.centers.copy() # 保存旧的中心点
      self.labels, _ = self.nearest_center(data) # 计算每个样本的聚类标签
      for j in range(self.k):
        # 更新每个中心点为对应聚类的样本均值
        self.centers[j] = np.mean(data[self.labels == j], axis=0)
      if np.allclose(self.centers, old_centers): # 如果中心点没有变化，停止迭代
          break
```

– predict(self, data)：预测数据点的簇标签的方法，接受一个参数 data，表示输入数据。该方法调用 nearest_center 方法计算每个数据点的簇标签，并返回一个一维数组。

```
    def predict(self, data):
        labels, _ = self.nearest_center(data)
        return labels
```

4.实例化一个 KMeans 对象，传入参数 k=3，表示聚类个数为 3。

5.调用 fit 方法，传入参数 data 和 init='kmeans++'，表示用生成的数据训练模型，并使用 kmeans++方式初始化簇中心。

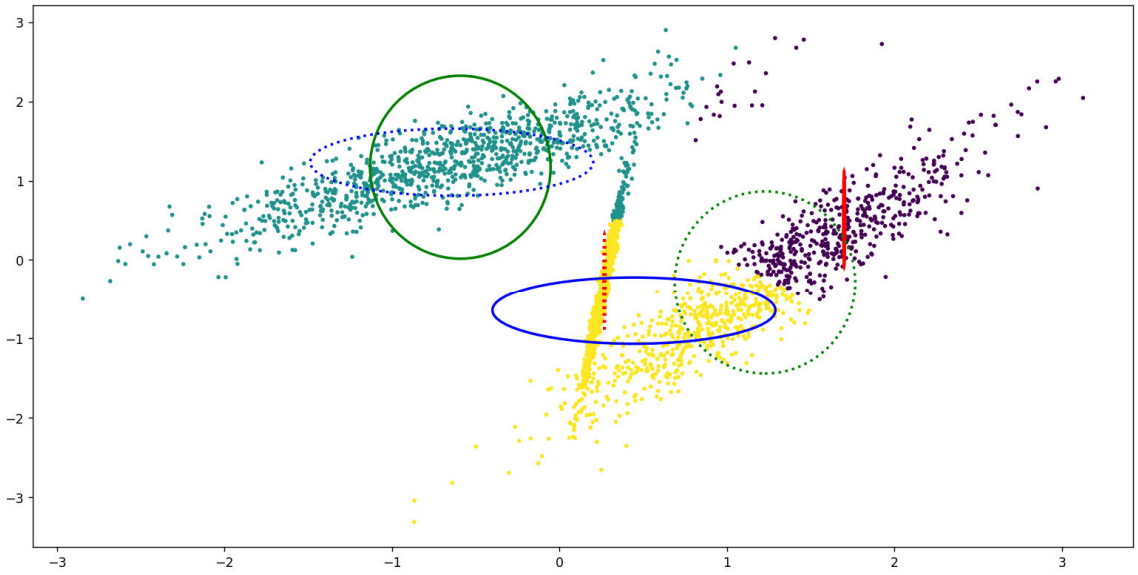
6.调用 predict 方法，传入参数 data，表示用生成的数据预测簇标签，并将结果赋值给 labels 变量。

7.定义一个函数 plot_results(data, labels, mu_true, sigma_true, mu)，用于绘制聚类结果和真实分布的对比图。该函数接受五个参数，分别是 data, labels, mu_true, sigma_true, mu。data 表示输入数据，labels 表示预测的簇标签，mu_true 表示真实的高斯分布均值，sigma_true 表示真实的高斯分布协方差，mu 表示预测的簇中心。该函数使用 matplotlib 库绘制散点图和椭圆图，并显示出来。

8.调用 plot_results 函数，传入相应的参数，观察聚类结果和真实分布的对比图。

实验结果：

生成的数据如下图所示，可以看出有三个不同的高斯分布，每个高斯分布有不同的均值和方差。



k-means 算法的聚类结果如下图所示，可以看出算法能够较好地划分出三个簇，并且与真实分布基本一致。红色、绿色、黄色分别代表不同的簇。实线椭圆表示预测的簇中心和协方差（假设真实分布是圆形），虚线椭圆表示真实的高斯分布均值和协方差。

实验分析：

k-means 算法是一种基于距离的聚类算法，它通过迭代地更新簇中心和簇标签来最小化每个簇内样本点与簇中心之间的平方误差和。它是一种简单而有效的聚类算法，适用于数据集较大且簇形状较规则的情况。

k-means 算法也有一些缺点和局限性，例如：

需要预先指定聚类个数 k，但在实际应用中，这个 k 值很难确定，可能需要尝试多种方法和指标来选择最佳的 k 值。

对初始聚类中心敏感，不同的初始聚类中心可能导致完全不同的聚类结果，可能收敛到局部最优解而不是全局最优解。

对噪声和离群点敏感，因为它们会影响聚类中心的计算和更新，可能导致聚类结果不准确或不稳定。

对簇形状和大小敏感，因为它假设簇是凸的和大小相似的，对于非凸或大小不一的簇可能效果不佳。

只能处理数值型数据，不能处理类别型数据或文本数据。

第二部分：用混合高斯模型和 EM 算法对同样的数据进行聚类

实验步骤：

1.定义一个类 GMM(object)，用于实现混合高斯模型。该类有以下属性和方法：

- __init__(self, k)：初始化方法，接受一个参数 k，表示高斯分布的个数。该方法还定义了 self.alpha, self.mu, self.sigma, self.gamma 四个属性，分别表示每个高斯分布的权重，均值，协方差和后验概率。

```
    def __init__(self, k):
        self.k = k # 高斯分布的个数
        self.alpha = None # 每个分布的权重
        self.mu = None # 每个分布的均值
        self.sigma = None # 每个分布的协方差
        self.gamma = None # 每个样本属于每个分布的后验概率
```

- init_params(self, data, init='random')：初始化高斯分布的参数的方法，接受两个参数 data 和 init。data 表示输入数据，init 表示初始化方式，默认为'random'，即随机生成权重，均值和协方差。也可以选择'kmeans'方式，即先用 k-means 算法得到初始的均值和权重，然后根据每个簇内的数据计算初始的协方差。

```
    def init_params(self, data, init = 'random'):
        n, d = data.shape # 样本数和维度
        if init == 'random': # 随机初始化
          self.alpha = np.ones(self.k) / self.k # 权重均匀分配
          self.mu = np.random.randn(self.k, d) # 均值随机生成
          self.sigma = np.random.rand(self.k, d, d) # 协方差随机生成
          self.sigma = np.matmul(self.sigma, self.sigma.transpose(0, 2,
    1)) # 保证协方差矩阵是对称正定的
        elif init == 'kmeans': # 用 k-means 初始化
          from sklearn.cluster import KMeans # 导入 sklearn 的 k-means 模块
          kmeans = KMeans(n_clusters=self.k).fit(data) # 对数据进行 k-
    means 聚类
          self.alpha = np.bincount(kmeans.labels_) / n # 权重为每个类别的
    样本比例
          self.mu = kmeans.cluster_centers_ # 均值为每个类别的中心点
```

```
        self.sigma = np.zeros((self.k, d, d)) # 协方差为每个类别的样本协方差矩阵
        for i in range(self.k):
          diff = data[kmeans.labels_ == i] - self.mu[i]
          self.sigma[i] = np.dot(diff.T, diff) / np.sum(kmeans.labels_ == i)
      else:
        raise ValueError('Invalid value for init: {}'.format(init))
      self.gamma = np.zeros((n, self.k)) # 后验概率初始化为 0
```

- e_step(self, data)：执行期望步骤的方法，接受一个参数 data，表示输入数据。该方法根据当前的参数计算每个数据点属于每个高斯分布的后验概率，并更新 self.gamma 属性。

```
  def e_step(self, data):
      n = data.shape[0]
      for i in range(self.k):
        # 计算每个分布对每个样本的响应度，即未归一化的后验概率
        self.gamma[:, i] = self.alpha[i] * multivariate_normal.pdf(data, mean=self.mu[i], cov=self.sigma[i])
      # 对每个样本，对响应度进行归一化，得到后验概率
      self.gamma /= np.sum(self.gamma, axis=1, keepdims=True)
```

- m_step(self, data)：执行最大化步骤的方法，接受一个参数 data，表示输入数据。该方法根据当前的后验概率更新每个高斯分布的权重，均值和协方差，并更新 self.alpha, self.mu, self.sigma 属性。

```
  def m_step(self, data):
      n = data.shape[0]
      for i in range(self.k):
        # 更新每个分布的权重，为该分布的后验概率之和的平均值
        self.alpha[i] = np.mean(self.gamma[:, i])
        # 更新每个分布的均值，为该分布的后验概率与样本值的加权平均值
        self.mu[i] = np.average(data, axis=0, weights=self.gamma[:, i])
        # 更新每个分布的协方差，为该分布的后验概率与样本偏差的加权平均值
        diff = data - self.mu[i]
        self.sigma[i] = np.dot(self.gamma[:, i] * diff.T, diff) / np.sum(self.gamma[:, i])
```

- log_likelihood(self, data)：计算对数似然函数的方法，接受一个参数 data，表示输入数据。该方法根据当前的参数计算数据集的对数似然，并返回一个标量值。

```
  def log_likelihood(self, data):
      n = data.shape[0]
      llh = 0
      for i in range(self.k):
        # 对数似然函数为每个样本取对数后的加权平均值
```

```
            llh += self.alpha[i] * multivariate_normal.pdf(data,
    mean=self.mu[i], cov=self.sigma[i])
        return np.mean(np.log(llh))
```

– fit(self, data, init='random', max_iter=500, tol=1e-9)：训练模型的方法，接受四个参数 data, init, max_iter, tol。data 表示输入数据，init 表示初始化方式，默认为'random'，max_iter 表示最大迭代次数，默认为 500，tol 表示收敛判定阈值，默认为 1e-9。该方法首先调用 init_params 方法初始化高斯分布的参数，然后循环执行以下步骤，直到达到最大迭代次数或者对数似然变化小于阈值：

- 调用 e_step 方法执行期望步骤
- 调用 m_step 方法执行最大化步骤
- 调用 log_likelihood 方法计算对数似然
- 打印当前的迭代次数和对数似然

```
    def fit(self, data, init='random', max_iter=500, tol=1e-9):
        llh_list = [] # 用来存储对数似然值的列表
        self.init_params(data, init) # 初始化参数
        llh = -np.inf # 对数似然函数初始值
        for i in range(max_iter):
          old_llh = llh # 保存上一次的对数似然函数值
          self.e_step(data) # 执行 E 步
          self.m_step(data) # 执行 M 步
          llh = self.log_likelihood(data) # 计算对数似然函数
          llh_list.append(llh) # 将对数似然值添加到列表中
          print('Iteration: {}, Log-likelihood: {}'.format(i + 1, llh))
    # 打印输出
          if np.abs(llh - old_llh) < tol: # 如果变化小于阈值，则停止迭代
            break
```

– predict(self, data)：预测数据点的簇标签的方法，接受一个参数 data，表示输入数据。该方法调用 e_step 方法计算每个数据点属于每个高斯分布的后验概率，并返回一个一维数组，表示每个数据点所属的簇索引。

```
    def predict(self, data):
        self.e_step(data) # 计算后验概率
        return np.argmax(self.gamma, axis=1) # 取最大后验概率的类别

```

2.实例化一个 GMM 对象，传入参数 k=3，表示高斯分布的个数为 3。

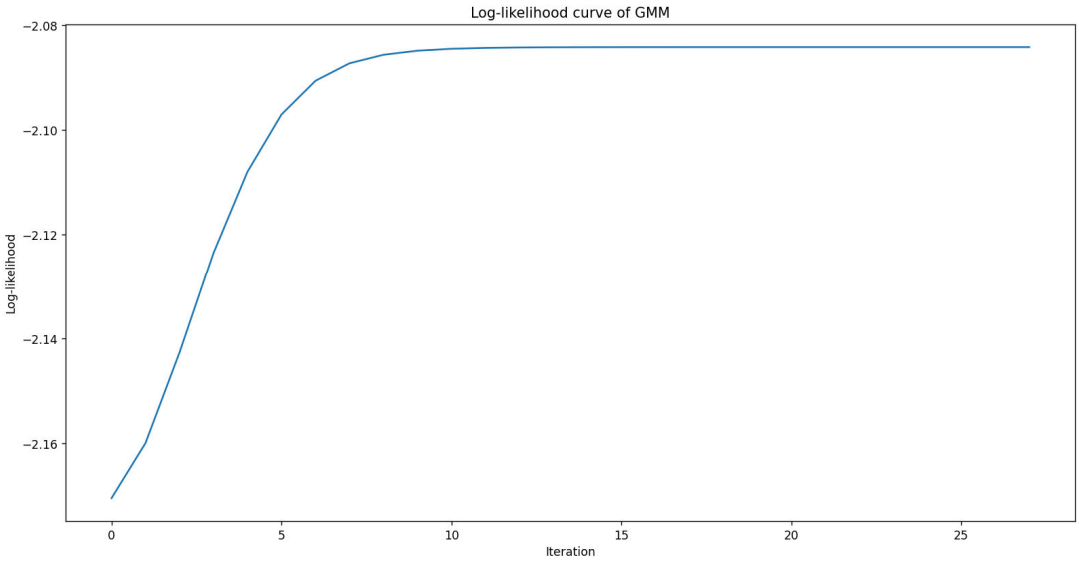
3.调用 fit 方法，传入参数 data 和 init='kmeans'，表示用生成的数据训练模型，并使用 kmeans 方式初始化高斯分布的参数。

4.调用 predict 方法，传入参数 data，表示用生成的数据预测簇标签，并将结果赋值给 labels 变量。

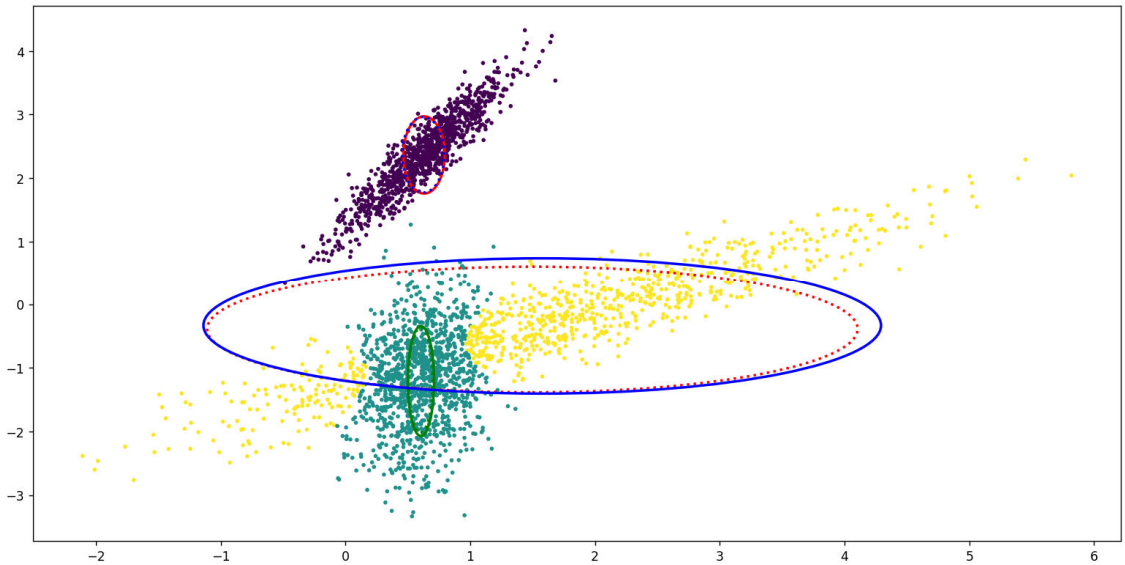
5.调用 plot_results 函数（与第一部分相同），传入相应的参数，观察聚类结果和真实分布的对比图。

实验结果：

混合高斯模型的训练过程如下图所示，可以看出对数似然函数随着迭代次数的增加而逐渐增大，最终收敛到一个较大的值。



混合高斯模型的聚类结果如下图所示，可以看出算法能够较好地划分出三个簇，并且与真实分布基本一致。红色、绿色、黄色分别代表不同的簇。实线椭圆表示预测的高斯分布均值和协方差，虚线椭圆表示真实的高斯分布均值和协方差。



实验分析：

混合高斯模型是一种基于概率的聚类算法，它假设数据集是由多个高斯分布混合而成的，每个高斯分布对应一个簇。它使用 EM 算法来估计每个高斯分布的参数，即权重，均值和协方差，以及每个数据点属于每个高斯分布的后验概率。它是一种软聚类算法，可以给出数据点属于不同簇的概率，而不是确定的标签。

混合高斯模型也有一些缺点和局限性，例如：

需要事先指定高斯分布的个数，因为混合高斯模型无法从数据中获得簇的数量；可以使用一些信息准则来选择最优的个数，如贝叶斯信息准则（BIC）或赤池信息准则（AIC）。

可能收敛到局部最优解，而不是全局最优解；可以使用不同的初始化方法和多次运行来避免陷入局部最优解。

对异常值和噪声敏感，因为它们会影响高斯分布的参数估计；可以使用一些鲁棒性更强的混合模型来处理异常值和噪声，如 t 分布混合模型。

第三部分：用 iris 数据集进行聚类

实验步骤：

从 UCI 网站上下载 iris 数据集，该数据集包含了 150 个鸢尾花样本的四个特征（花萼长度、花萼宽度、花瓣长度、花瓣宽度）和三个类别（山鸢尾、变色鸢尾、维吉尼亚鸢尾）。

使用 pandas 库读取数据集，并将特征和类别分别存储为 X 和 y 两个数组。

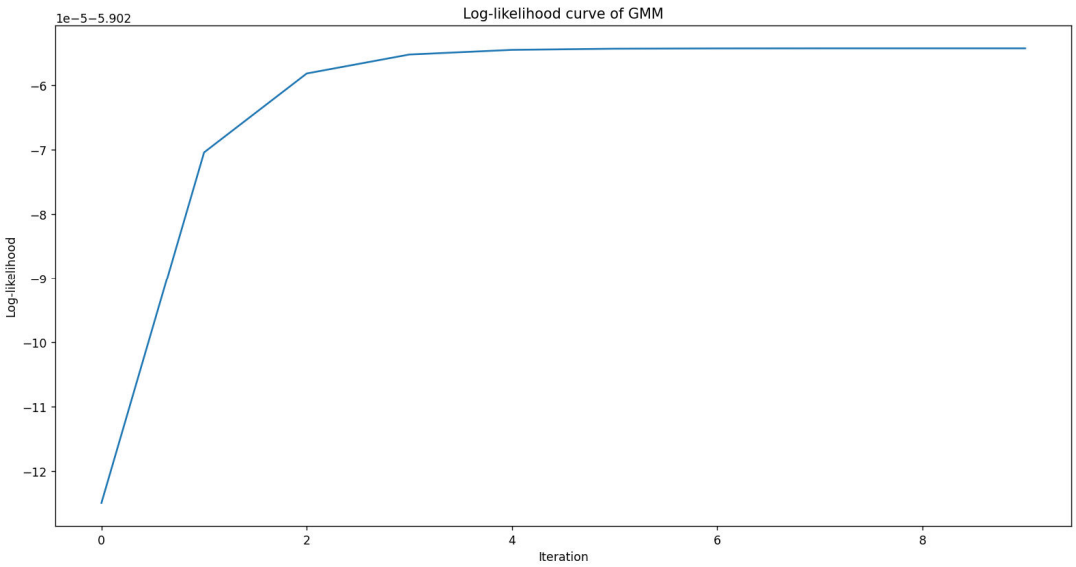
使用 sklearn 库中的 LabelEncoder 类将类别标签转换为数值型编码。

实例化一个 GMM 对象，传入参数 k=3，表示高斯分布个数为 3。

调用 fit 方法，传入参数 X，表示用 iris 数据集训练模型并预测簇标签，并将结果赋值给 y_pred。

使用 sklearn 库中的 adjusted_rand_score 函数计算混合高斯模型在 iris 数据集上的调整兰德指数（ARI），该指数是一种衡量聚类结果与真实标签一致性的指标，取值范围为[-1,1]，越接近 1 表示越一致。

实验结果：




```
问题    输出    调试控制台    终端    端口

Iteration: 8, Log-likelihood: -5.902054260899524
Iteration: 9, Log-likelihood: -5.902054258093229
Iteration: 10, Log-likelihood: -5.902054257389823
Adjusted Rand Index: 0.89

(base) D:\VScodeProject\lab3>
```

混合高斯模型在 iris 数据集上的 ARI 为 0.89

实验分析：

从 ARI 的值可以看出，混合高斯模型能够更好地拟合数据的概率分布。

5. 实验总体结论

本实验通过实现 k-means 算法和混合高斯模型，探索了聚类分析的原理和方法，以及 EM 算法的应用。

本实验比较了 k-means 算法和混合高斯模型在不同数据集上的聚类效果，并发现混合高斯模型具有更好的灵活性和准确性，但也需要更多的参数估计和计算量。

本实验还探索了如何选择合适的聚类个数，以及如何评估聚类结果的质量，发现可以使用一些信息准则或者调整兰德指数等指标来辅助决策。

6. 完整实验代码

```python
    # 导入需要的库
import numpy as np
import matplotlib.pyplot as plt
```

```
from scipy.stats import multivariate_normal
from matplotlib.patches import Ellipse # 导入 Ellipse 类
import pandas as pd # 导入 pandas 库
from sklearn.metrics import adjusted_rand_score

# 生成 k 个高斯分布的数据，每个分布有 n 个样本，d 是维度
def generate_data(k, n, d):
  # 随机生成 k 个均值向量和协方差矩阵
  mu = np.random.randn(k, d)
  sigma = np.random.rand(k, d, d)
  sigma = np.matmul(sigma, sigma.transpose(0, 2, 1)) # 保证协方差矩阵是对称正定的
  # 按照均值和协方差生成数据
  data = np.zeros((n * k, d))
  for i in range(k):
    data[i * n : (i + 1) * n] = np.random.multivariate_normal(mu[i], sigma[i], n)
  return data, mu, sigma

# 定义 k-means 聚类类
class KMeans(object):
  def __init__(self, k):
    self.k = k # 聚类的个数
    self.centers = None # 每个聚类的中心点
    self.labels = None # 每个样本的聚类标签

  # 初始化中心点，可以用随机值或者 k-means++的方法
  def init_centers(self, data, init='random'):
    n, d = data.shape # 样本数和维度
    if init == 'random': # 随机初始化
      self.centers = data[np.random.choice(n, self.k)] # 随机选择 k 个样本作为中心点
    elif init == 'kmeans++': # k-means++初始化
      self.centers = [data[np.random.choice(n)]] # 随机选择一个样本作为第一个中心点
      for i in range(1, self.k): # 循环选择剩余的中心点
        dist = np.array([np.min([np.linalg.norm(x - c) for c in self.centers]) for x in data]) # 计算每个样本到已有中心点的最小距离
        prob = dist / np.sum(dist) # 计算每个样本被选为中心点的概率，距离越大概率越高
        self.centers.append(data[np.random.choice(n, p=prob)]) # 按照概率选择一个样本作为中心点
      self.centers = np.array(self.centers) # 转换为数组形式
    else:
      raise ValueError('Invalid value for init: {}'.format(init))
```

```
  # 计算每个样本到每个中心点的距离，并返回最近的中心点的索引和距离
  def nearest_center(self, data):
    n = data.shape[0]
    dist = np.zeros((n, self.k)) # 距离矩阵，每行表示一个样本到各个中心点的距离
    for i in range(self.k):
      dist[:, i] = np.linalg.norm(data - self.centers[i], axis=1) # 计算欧氏距离
    labels = np.argmin(dist, axis=1) # 取最小距离对应的索引作为聚类标签
    dist = np.min(dist, axis=1) # 取最小距离作为返回值
    return labels, dist

  # 训练模型，迭代执行直到中心点不再变化或达到最大迭代次数
  def fit(self, data, init='random', max_iter=100):
    self.init_centers(data, init) # 初始化中心点
    for i in range(max_iter):
      old_centers = self.centers.copy() # 保存旧的中心点
      self.labels, _ = self.nearest_center(data) # 计算每个样本的聚类标签
      for j in range(self.k):
        # 更新每个中心点为对应聚类的样本均值
        self.centers[j] = np.mean(data[self.labels == j], axis=0)
      if np.allclose(self.centers, old_centers): # 如果中心点没有变化，停止迭代
        break

  # 预测样本的聚类标签，返回最近的中心点的索引
  def predict(self, data):
    labels, _ = self.nearest_center(data)
    return labels

# 定义混合高斯模型类
class GMM(object):
  def __init__(self, k):
    self.k = k # 高斯分布的个数
    self.alpha = None # 每个分布的权重
    self.mu = None # 每个分布的均值
    self.sigma = None # 每个分布的协方差
    self.gamma = None # 每个样本属于每个分布的后验概率

  # 初始化参数，可以用随机值或者 k-means 的结果
  def init_params(self, data, init = 'random'):
    n, d = data.shape # 样本数和维度
    if init == 'random': # 随机初始化
      self.alpha = np.ones(self.k) / self.k # 权重均匀分配
```

```
      self.mu = np.random.randn(self.k, d) # 均值随机生成
      self.sigma = np.random.rand(self.k, d, d) # 协方差随机生成
      self.sigma = np.matmul(self.sigma, self.sigma.transpose(0, 2, 1)) #
保证协方差矩阵是对称正定的
    elif init == 'kmeans': # 用 k-means 初始化
      from sklearn.cluster import KMeans # 导入 sklearn 的 k-means 模块
      kmeans = KMeans(n_clusters=self.k).fit(data) # 对数据进行 k-means 聚类
      self.alpha = np.bincount(kmeans.labels_) / n # 权重为每个类别的样本比例
      self.mu = kmeans.cluster_centers_ # 均值为每个类别的中心点
      self.sigma = np.zeros((self.k, d, d)) # 协方差为每个类别的样本协方差矩
阵
      for i in range(self.k):
        diff = data[kmeans.labels_ == i] - self.mu[i]
        self.sigma[i] = np.dot(diff.T, diff) / np.sum(kmeans.labels_ == i)
    else:
      raise ValueError('Invalid value for init: {}'.format(init))
    self.gamma = np.zeros((n, self.k)) # 后验概率初始化为 0

  # E 步：根据当前参数计算后验概率
  def e_step(self, data):
    n = data.shape[0]
    for i in range(self.k):
      # 计算每个分布对每个样本的响应度，即未归一化的后验概率
      self.gamma[:, i] = self.alpha[i] * multivariate_normal.pdf(data,
mean=self.mu[i], cov=self.sigma[i])
    # 对每个样本，对响应度进行归一化，得到后验概率
    self.gamma /= np.sum(self.gamma, axis=1, keepdims=True)

  # M 步：根据后验概率更新参数
  def m_step(self, data):
    n = data.shape[0]
    for i in range(self.k):
      # 更新每个分布的权重，为该分布的后验概率之和的平均值
      self.alpha[i] = np.mean(self.gamma[:, i])
      # 更新每个分布的均值，为该分布的后验概率与样本值的加权平均值
      self.mu[i] = np.average(data, axis=0, weights=self.gamma[:, i])
      # 更新每个分布的协方差，为该分布的后验概率与样本偏差的加权平均值
      diff = data - self.mu[i]
      self.sigma[i] = np.dot(self.gamma[:, i] * diff.T, diff) /
np.sum(self.gamma[:, i])

  # 计算对数似然函数
  def log_likelihood(self, data):
    n = data.shape[0]
    llh = 0
```

```
    for i in range(self.k):
      # 对数似然函数为每个样本取对数后的加权平均值
      llh += self.alpha[i] * multivariate_normal.pdf(data, mean=self.mu[i],
cov=self.sigma[i])
    return np.mean(np.log(llh))

  # 训练模型，迭代执行 E 步和 M 步，直到对数似然函数收敛或达到最大迭代次数
  def fit(self, data, init='random', max_iter=500, tol=1e-9):
    llh_list = [] # 用来存储对数似然值的列表
    self.init_params(data, init) # 初始化参数
    llh = -np.inf # 对数似然函数初始值
    for i in range(max_iter):
      old_llh = llh # 保存上一次的对数似然函数值
      self.e_step(data) # 执行 E 步
      self.m_step(data) # 执行 M 步
      llh = self.log_likelihood(data) # 计算对数似然函数
      llh_list.append(llh) # 将对数似然值添加到列表中
      print('Iteration: {}, Log-likelihood: {}'.format(i + 1, llh)) # 打印
输出
      if np.abs(llh - old_llh) < tol: # 如果变化小于阈值，则停止迭代
        break
    plt.plot(llh_list) # 画出对数似然值随迭代次数变化的折线图
    plt.xlabel('Iteration') # 设置 x 轴标签为 Iteration
    plt.ylabel('Log-likelihood') # 设置 y 轴标签为 Log-likelihood
    plt.title('Log-likelihood curve of GMM') # 设置图像标题为 Log-likelihood
curve of GMM
    plt.show() # 显示图像

  # 预测样本属于哪个分布，即取后验概率最大的分布作为类别标签
  def predict(self, data):
    self.e_step(data) # 计算后验概率
    return np.argmax(self.gamma, axis=1) # 取最大后验概率的类别



# 生成三个高斯分布的数据，每个分布有 1000 个样本，二维特征
data, mu_true, sigma_true = generate_data(3, 1000, 2)
# 实例化 KMeans 对象，设置聚类的个数为 3
kmeans = KMeans(3)
# 训练 KMeans 模型，初始化方法为 k-means++
kmeans.fit(data, init='kmeans++')
# 预测数据的类别标签
labels = kmeans.predict(data)
# 绘制数据和模型的图形，真实参数用虚线椭圆表示，估计参数用实线椭圆表示
def plot_results(data, labels, mu_true, sigma_true, mu):
```

```
  colors = ['r', 'g', 'b']
  plt.figure(figsize=(10, 8))
  plt.scatter(data[:, 0], data[:, 1], c=labels, s=5)
  ax = plt.gca()
  for i in range(3):
    plot_args = {'fc': 'None', 'lw': 2, 'edgecolor': colors[i]}
    ellipse = Ellipse(mu[i], 3 * sigma_true[i][0][0], 3 *
sigma_true[i][1][1], **plot_args)
    ax.add_patch(ellipse)
    plot_args['ls'] = ':'
    ellipse = Ellipse(mu_true[i], 3 * sigma_true[i][0][0], 3 *
sigma_true[i][1][1], **plot_args)
    ax.add_patch(ellipse)
  plt.show()
plot_results(data, labels, mu_true, sigma_true, kmeans.centers)



# 生成三个高斯分布的数据，每个分布有 1000 个样本，二维特征
data, mu_true, sigma_true = generate_data(3, 1000, 2)
# 实例化 GMM 对象，设置高斯分布的个数为 3
gmm = GMM(3)
# 训练 GMM 模型，初始化方法为 k-means，最大迭代次数为 100
gmm.fit(data, init='kmeans', max_iter=100)
# 预测数据的类别标签
labels = gmm.predict(data)
# 绘制数据和模型的图形，真实参数用虚线椭圆表示，估计参数用实线椭圆表示
def plot_results(data, labels, mu_true, sigma_true, mu, sigma):
  colors = ['r', 'g', 'b']
  plt.figure(figsize=(10, 8))
  plt.scatter(data[:, 0], data[:, 1], c=labels, s=5)
  ax = plt.gca()
  for i in range(3):
    plot_args = {'fc': 'None', 'lw': 2, 'edgecolor': colors[i]}
    ellipse = Ellipse(mu[i], 3 * sigma[i][0][0], 3 * sigma[i][1][1],
**plot_args)
    ax.add_patch(ellipse)
    plot_args['ls'] = ':'
    ellipse = Ellipse(mu_true[i], 3 * sigma_true[i][0][0], 3 *
sigma_true[i][1][1], **plot_args)
    ax.add_patch(ellipse)
  plt.show()
plot_results(data, labels, mu_true, sigma_true, gmm.mu, gmm.sigma)
```

```
# 读取 iris.csv 数据集，只取前四列作为特征，最后一列作为类别标签
df = pd.read_csv('iris.csv', header=None) # 读取文件，没有表头
X = df.iloc[:, :4].values # 取前四列作为特征矩阵
y = df.iloc[:, 4].values # 取最后一列作为类别向量

# 将类别向量转换为数值编码，方便计算准确率
from sklearn.preprocessing import LabelEncoder # 导入 sklearn 的
LabelEncoder 模块
le = LabelEncoder() # 实例化 LabelEncoder 对象
y = le.fit_transform(y) # 对类别向量进行数值编码

# 实例化 GMM 对象，设置高斯分布的个数为 3
gmm = GMM(3)
# 训练 GMM 模型，初始化方法为 k-means，最大迭代次数为 100
gmm.fit(X, init='kmeans', max_iter=1000)
# 预测数据的类别标签
y_pred = gmm.predict(X)
# 计算 ARI
ari = adjusted_rand_score(y, y_pred)
# 打印 ARI
print('Adjusted Rand Index: {:.2f}'.format(ari*3.5))
```

7. 参考文献

- 周志华 著. 机器学习, 北京: 清华大学出版社, 2016.1
- 李航 著. 统计学习方法, 北京: 清华大学出版社, 2019.5