机器学习 实验三

【学号】22336259　　【姓名】谢宇桐　　【专业】计算机科学与技术

实验问题：

MNIST 数据集为例，探索 K-Means 和 GMM 这两种聚类算法的性能。

数据处理方法：

数据集加载与预处理：

```
# 加载数据
train_data = pd.read_csv('mnist_train.csv')
test_data = pd.read_csv('mnist_test.csv')

# 分离特征和标签
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

# 数据归一化
X_train = X_train / 255.0
X_test = X_test / 255.0
```

精度计算：

```
def calculate_accuracy(y_true, clusters):
    """计算聚类精度"""
    y_pred = np.zeros_like(y_true)
    for i in range(10):
        mask = (clusters == i)
        y_pred[mask] = np.bincount(y_true[mask]).argmax()
    return np.mean(y_pred == y_true)

# 计算聚类精度
acc_ = calculate_accuracy(y_train, clusters)
print(f'...聚类精度: {acc_:.4f}')
# GMM因为是列表，所以写为如下：
print(f'GMM_{cov}聚类精度: {np.mean(accuracies):.4f}')
```

时间计算：

```
import time
# 算法里
    start_time = time.time()  # 记录开始时间
    # ...
    end_time = time.time()  # 记录结束时间
    elapsed_time = end_time - start_time  # 计算运行时间
```

实验内容：

1. **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）

   K-Means：

   K-Means 算法是一种经典的聚类算法，用于将数据点划分为K个簇（cluster）。每个簇由一个聚类中心（centroid）代表，算法的目标是最小化簇内距离的平方和，即每个点到其聚类中心的距离的平方和。

   以下是K-Means算法的基本步骤：

   初始化：随机选择K个数据点作为初始聚类中心（质心）。

   分配：将每个数据点分配给最近的聚类中心，形成K个簇。

   更新：重新计算每个簇的聚类中心，通常是簇中所有点的均值。

   迭代：重复分配和更新步骤，直到满足停止条件，比如聚类中心的变化小于某个阈值，或者达到预设的最大迭代次数。

   输出：输出最终的聚类结果，包括每个数据点的簇分配和聚类中心。

   这些步骤将在下面的代码中体现。其中初始化部分将在第二项中详细展开。

   ```
   import numpy as np
   import pandas as pd

   def k_means(X, k, max_iter=100, init_method='k-means++'):
       # 初始化
       if init_method == 'random':
           centroids = X[np.random.choice(X.shape[0], k, replace=False)]
       elif init_method == 'k-means++':
           centroids = initialize_centroids_plusplus(X, k)

       for _ in range(max_iter):
           # 分配阶段
           clusters = np.array([np.argmin([np.inner(c - x, c - x) for c in
   centroids]) for x in X])
           # 更新阶段
           new_centroids = np.array([X[clusters == j].mean(axis=0) for j in
   range(k)])
           # 检查收敛
           if np.all(centroids == new_centroids):
               break
   ```

```
        centroids = new_centroids
    return clusters, centroids
```

GMM:

GMM全称高斯混合模型（Gaussian Mixture Model），是一种概率模型，用于表示多个高斯分布的混合。GMM假设数据是由多个不同的高斯分布生成的，每个高斯分布称为一个组件（component）。GMM常用于聚类分析，因为它可以提供每个数据点属于每个簇的概率。

以下是GMM算法的基本步骤：

初始化：随机选择初始参数，包括每个高斯分布的均值（mean）、协方差（covariance）和混合权重（mixture weights）。

E步骤（期望步骤）：对于每个数据点，计算其属于每个高斯分布的概率，这个概率称为责任度（responsibility）。

M步骤（最大化步骤）：根据E步骤计算的责任度，更新每个高斯分布的参数。

迭代：重复E步骤和M步骤，直到模型收敛或达到最大迭代次数。

输出：输出最终的聚类结果，包括每个数据点的责任度和高斯分布的参数。

```
def initialize_gmm_params(X, k, cov_type='diagonal_equal'):
    """初始化GMM参数"""
    n_features = X.shape[1]
    weights = np.ones(k) / k
    means = X[np.random.choice(X.shape[0], k, replace=False)]
    covariances = []

    for i in range(k):
        if cov_type == 'diagonal_equal':
            # 对角且元素值都相等
            cov = np.eye(n_features) * np.var(X)
        elif cov_type == 'diagonal':
            # 对角但对元素值不要求相等
            cov = np.diag(np.var(X, axis=0))
        elif cov_type == 'full':
            # 普通矩阵
            diff = X - means[i]
            cov = np.dot(diff.T, diff) / (X.shape[0] - 1)
        else:
            raise ValueError("Invalid covariance matrix type")

        # 添加正则化确保协方差矩阵是正定的
        cov = cov + 1e-6 * np.eye(n_features)
        covariances.append(cov)

    return weights, means, covariances


def e_step(X, weights, means, covariances):
    """E步骤"""
    n_samples = X.shape[0]
    k = len(weights)
    responsibilities = np.zeros((n_samples, k))
```

```
    for i in range(k):
        try:
            responsibilities[:, i] = weights[i] * multivariate_normal.pdf(X,
mean=means[i], cov=covariances[i])
        except np.linalg.LinAlgError:
            responsibilities[:, i] = 0  # 如果协方差矩阵是奇异的，设置责任度为0
    responsibilities /= (responsibilities.sum(axis=1, keepdims=True) + 1e-10)
 # 添加小的正数避免除以零
    return responsibilities

def m_step(X, responsibilities):
    """M步骤"""
    k = responsibilities.shape[1]
    n_features = X.shape[1]
    weights = responsibilities.mean(axis=0)
    means = np.dot(responsibilities.T, X) / (responsibilities.sum(axis=0)[:,
np.newaxis] + 1e-10)
    covariances = []
    for i in range(k):
        diff = X - means[i]
        cov = np.dot(responsibilities[:, i] * diff.T, diff) /
(responsibilities[:, i].sum() + 1e-10)
        # 添加正则化确保协方差矩阵是正定的
        cov = cov + 1e-6 * np.eye(n_features)
        covariances.append(cov)
    return weights, means, covariances

def gmm(X, k, cov_type='diagonal_equal', max_iter=100):
    """GMM算法"""
    start_time = time.time()  # 记录开始时间
    weights, means, covariances = initialize_gmm_params(X, k, cov_type)
    for iteration in range(max_iter):
        # E步骤
        responsibilities = e_step(X, weights, means, covariances)
        # M步骤
        weights, means, covariances = m_step(X, responsibilities)
        # 检查是否所有责任度都有效
        if np.any(np.isnan(responsibilities)) or
np.any(np.isinf(responsibilities)):
            break

    end_time = time.time()  # 记录结束时间
    elapsed_time = end_time - start_time  # 计算运行时间

    return responsibilities, means, elapsed_time

def run_gmm_experiments(X, y, k, n_experiments=10):
    for cov in {'full', 'diagonal_equal', 'diagonal'}:
        accuracies = []
        times = []
        for seed in range(n_experiments): # 进行多次实验，每次实验使用不同的随机种子
            np.random.seed(seed)
            responsibilities, gmm_means, time_gmm = gmm(X, k, cov_type=cov)
            gmm_clusters = np.argmax(responsibilities, axis=1) # 将责任度最高的
簇分配给每个数据点，得到聚类结果
            acc_gmm = calculate_accuracy(y, gmm_clusters)
```

```
            accuracies.append(acc_gmm)
            times.append(time_gmm)
        print(f'GMM_{cov}聚类精度: {np.mean(accuracies):.4f}, 运行时间:
{np.mean(times):.4f}秒')

run_gmm_experiments(X_train, y_train, 10)
```

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；**

· 随机初始化

随机初始化是最简单的初始化方法，它从数据集中随机选择k个不重复的样本作为初始质心。这种方法的优点是实现简单，但它可能不会总是产生好的聚类结果，特别是在数据集的聚类结构复杂或者初始质心选择不佳的情况下。

```
if init_method == 'random':
        centroids = X[np.random.choice(X.shape[0], k, replace=False)]
```

·K-Means++初始化

K-Means++是一种更复杂的初始化方法，它旨在通过减少初始质心之间的距离来提高聚类性能。K-Means++的步骤如下：

·从数据集中随机选择第一个质心。

·对于每个数据点，计算它到已选择的最近质心的距离。

·根据距离的平方，为每个数据点分配一个概率，然后使用这些概率来选择下一个质心。

·重复上述过程，直到选择了k个质心。

```
def initialize_centroids_plusplus(X, k):
    """使用K-Means++初始化方法"""
    centroids = [X[np.random.randint(X.shape[0])]] # 随机选择第一个质心
    for _ in range(1, k):
        distances = np.min(np.sum((X - np.array(centroids)[:,
np.newaxis])**2, axis=2), axis=0) # 计算每个数据点到最近质心的最小距离
        probabilities = distances / distances.sum() # 根据距离计算选择每个数据点作
为新质心的概率
        cumulative_probabilities = probabilities.cumsum() # 计算累积概率
        r = np.random.rand()
        index = np.searchsorted(cumulative_probabilities, r) # 根据累积概率选择
新的质心
        centroids.append(X[index]) # 将新质心添加到质心列表中
    return np.array(centroids)
```

多次运行，我们可以得到以下结果：

```
K-Means++聚类精度: 0.5831, 运行时间: 514.2166秒
随机初始化聚类精度: 0.5887, 运行时间: 366.8164秒

进程已结束，退出代码为 0
```

```
K-Means++聚类精度: 0.6058, 运行时间: 182.6605秒
随机初始化聚类精度: 0.5829, 运行时间: 255.9140秒

进程已结束，退出代码为 0
```

```
K-Means++聚类精度: 0.6173, 运行时间: 359.5367秒
随机初始化聚类精度: 0.5771, 运行时间: 255.4615秒

进程已结束，退出代码为 0
```

```
K-Means++聚类精度: 0.5658, 运行时间: 185.2374秒
随机初始化聚类精度: 0.5887, 运行时间: 258.5829秒

进程已结束，退出代码为 0
```

```
K-Means++聚类精度: 0.5937, 运行时间: 182.4555秒
随机初始化聚类精度: 0.5771, 运行时间: 358.1129秒

进程已结束，退出代码为 0
```

从理论上来说，K-Means++相比于随机初始化通常具有以下特点：

收敛速度更快：K-Means++通常可以更快地收敛到一个较好的局部最优解，因为它通过优化初始质心的选择来减少质心之间的距离，从而减少了算法迭代的次数。

聚类质量更高：K-Means++往往能够得到更高质量的聚类结果，因为它减少了初始质心选择的随机性，从而减少了聚类结果对初始质心选择的敏感性。精度也就更高。

但可以看到，从实际上跑出来的结果来看K-Means++的精度比随机初始化通常更高，但运行时间也出现了比随机初始化更长的现象。这体现了聚类算法的随机性与概率性，是正常的。

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；**

对角且元素值都相等（diagonal_equal）：

在这种初始化方法中，每个高斯分布的协方差矩阵是对角矩阵，且所有对角元素（即方差）都相等。这意味着模型假设所有特征的方差相同，但不同特征之间不相关。

```
if cov_type == 'diagonal_equal':
    # 对角且元素值都相等
    cov = np.eye(n_features) * np.var(X)
```

np.var(X)计算整个数据集的方差，然后这个方差被用来初始化所有特征的方差。这种方法简化了模型，因为它假设所有特征具有相同的不确定性。

对角但对元素值不要求相等（diagonal）：

在这种初始化方法中，每个高斯分布的协方差矩阵也是对角矩阵，但不同特征的方差可以不同。这意味着模型允许不同特征具有不同的方差，但仍然假设特征之间相互独立。

```
elif cov_type == 'diagonal':
    # 对角但对元素值不要求相等
    cov = np.diag(np.var(X, axis=0))
```

np.var(X, axis=0)计算每个特征的方差，然后这些方差被用来初始化对应特征的方差。这种方法提供了更多的灵活性，因为它允许模型捕捉每个特征的不同变异性。

普通矩阵（full）：

在这种初始化方法中，每个高斯分布的协方差矩阵是一个普通的对称矩阵，这意味着模型允许特征之间存在相关性。这种结构最灵活，因为它可以捕捉特征之间的复杂关系。

```
elif cov_type == 'full':
    diff = X - means[i]
    cov = np.dot(diff.T, diff) / (X.shape[0] - 1)
```

np.dot(diff.T, diff)计算样本协方差矩阵，然后通过除以(X.shape[0] - 1)进行无偏估计。这种方法初始化的协方差矩阵可以捕捉特征之间的线性关系，但计算和存储成本更高，且可能导致数值不稳定。

多次运行，得到以下结果：

```
GMM_diagonal_equal聚类精度: 0.2491, 运行时间: 38.6564秒
GMM_diagonal聚类精度: 0.1551, 运行时间: 12.5434秒
GMM_full聚类精度: 0.1182, 运行时间: 13.2157秒

进程已结束，退出代码为 0
```

```
GMM_diagonal_equal聚类精度: 0.2491, 运行时间: 74.7961秒
GMM_diagonal聚类精度: 0.1551, 运行时间: 12.5315秒
GMM_full聚类精度: 0.1182, 运行时间: 13.2451秒

进程已结束，退出代码为 0
```

从聚类精度来说，理论上full协方差矩阵应该提供最高的聚类精度，因为它最灵活，能够捕捉数据中的所有方差和协方差。然而，这也可能带来过拟合的风险。diagonal提供了适中的精度和灵活性。diagonal_equal在特征方差相似时可能表现良好，但在方差差异大时可能不够准确。

从运行时间来看，理论上diagonal_equal和diagonal通常比full快，因为它们需要估计的参数更少。full由于需要估计整个协方差矩阵，所以运行时间最长。

但可以看到我们运行出来的结果是精度：diagonal_equal > diagonal > full；时间：diagonal < full < diagonal_equal

diagonal_equal的运行时间异常地高，这可能是由于在初始化过程中对整个数据集计算方差导致的。其聚类精度最高，这可能意味着在这种特定情况下，假设所有特征具有相同的方差是一个合理的假设，从而使得模型能够更好地拟合数据。full精度最低，可能是出现了过拟合现象。