Assignment3 实验报告

姓名：林筱涵

学号：22336141

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。

1 算法流程

1.1 K-Means 算法流程

K-Means 是一种常用的无监督学习聚类算法，旨在将数据分为K个簇（Cluster），使得每个簇内的样本距离中心更近，从而最小化簇内的总平方误差。

算法流程:

1.初始化簇中心：
随机选择K个初始簇中心（通常从数据集中随机选取，或采用其他初始化方法K-Means++）。这些簇中心表示每个簇的“代表点”。
代码如下：

```
def __init__(self, init_data, K=2, init_method='kmeans++'):
        self.K = K  # 聚类数量
        self.dimension = init_data.shape[1]  # 数据的特征维度
        self.centroids = None  # 用于存储聚类中心
        self.initCentroids(init_data, init_method)  # 初始化聚类中心


    def initCentroids(self, data, init_method):
        if init_method == 'random':
            # 随机初始化
            indexes = np.arange(data.shape[0])  # 获取所有样本点的索引
            np.random.shuffle(indexes)  # 随机打乱索引
            self.centroids = np.zeros((self.K, data.shape[1]))  # 初始化聚类中心矩阵
            for i in range(self.K):
                self.centroids[i] = data[indexes[i]]  # 从随机顺序中选择前 K 个样本作为初始中心
        elif init_method == 'kmeans++':
            # K-Means++ 初始化
            self.centroids = []  # 初始化为空列表
            # 随机选择第一个聚类中心
            self.centroids.append(data[np.random.randint(data.shape[0])])
            # 选择剩余的聚类中心
            for _ in range(1, self.K):
                # 计算每个点到最近中心的距离
                distances = np.min(self.getDistanceToCenters(data, self.centroids), axis=1)
                # 概率分布：距离的平方
                probabilities = distances**2 / np.sum(distances**2)
                # 根据概率选择新的中心
                new_center = data[np.random.choice(data.shape[0], p=probabilities)]
                self.centroids.append(new_center)
            self.centroids = np.array(self.centroids)  # 转为 numpy 数组
```

2.分配样本到最近的簇中心：

对于数据集中的每个样本，计算它与所有簇中心的欧氏距离（或其他距离度量方法），将样本分配到距离最近的簇中心所对应的簇。

3.更新簇中心：

计算每个簇内所有样本的均值，将均值作为新的簇中心。新的簇中心是簇内样本在特征空间中的中心点。

```
def getCentroids(self, data, clusters):
    oneHotClusters = np.zeros((data.shape[0], self.K))  # 初始化 one-hot 矩阵
    # 将样本标签转化为 one-hot 编码
    oneHotClusters[np.arange(data.shape[0]), clusters] = 1
    # 计算每个簇的均值
    return np.dot(oneHotClusters.T, data) / np.sum(oneHotClusters, axis=0).reshape((-1, 1))
```

4.重复步骤 2 和 3：
不断重复分配样本和更新簇中心的步骤，直到满足以下任一停止条件：即簇中心不再发生明显变化或者达到预设的迭代次数。
5.输出结果：
最终的簇中心位置。
每个样本所属的簇标签。

1.2 GMM 算法流程

GMM是一种概率模型，用于表示数据分布为多个高斯分布的混合形式。

算法流程：

1.输入：
数据集X={x_1,...x_N}和高斯分布的个数K。
2.初始化：
初始化 GMM 的参数π_k,μ_k,$\sum_k$
π_k:通常初始化为1/K.
μ_k:可以通过随机选取数据点或 K-Means 聚类结果来初始化。
$\sum_k$:初始化为单位矩阵或样本的协方差矩阵。
3.E步骤:
计算每个数据点的责任值r_{ik}.
4.M步骤：
更新模型参数π_k,μ_k,$\sum_k$
5.重复3，4步骤，直到满足停止条件
模型的对数似然函数收敛。
对数似然的变化小于某个阈值。
达到最大迭代次数。
6.输出：

模型参数$\{\pi_k, \mu_k, \sum_k\}_{k=1}^{K}$
每个数据点属于每个高斯分布的概率r_{ik}。

2 数据预处理

```
def load_and_preprocess_data(train_path, test_path, new_dimension=NEW_DIMENSION):
    train_data = pd.read_csv(train_path)
    test_data = pd.read_csv(test_path)
    X_train = train_data.iloc[:, 1:].values.astype(np.float32) / 255.0  # 像素归一化
    y_train = train_data.iloc[:, 0].values
    X_test = test_data.iloc[:, 1:].values.astype(np.float32) / 255.0
    y_test = test_data.iloc[:, 0].values
    pca = PCA(n_components=new_dimension)  # PCA 降维
    X_train = pca.fit_transform(X_train)
    X_test = pca.transform(X_test)
    return X_train, y_train, X_test, y_test
```

1.数据归一化：将特征值缩放到 [0, 1]，以消除量纲差异的影响。
2.用PCA降维降低特征维度，减少计算复杂度，提高模型的训练速度；去除冗余信息，保留主要数据特征，提高模型的拟合能力。

3 Kmeans训练过程

3.1 初始化

本次实验用了两种初始化方法，分别是随机初始化和K-Means++。
随机初始化的过程；
1.获取所有数据点的索引。
2.随机打乱这些索引。
3.从随机打乱的索引中选择前K个样本点作为初始聚类中心。
代码如下：

```
if init_method == 'random':
    # 随机初始化
    indexes = np.arange(data.shape[0])  # 获取所有样本点的索引
    np.random.shuffle(indexes)  # 随机打乱索引
    self.centroids = np.zeros((self.K, data.shape[1]))  # 初始化聚类中心矩阵
    for i in range(self.K):
        self.centroids[i] = data[indexes[i]]  # 从随机顺序中选择前 K 个样本作为初始中心
```

K-Means++初始化；

1.原理：

K-Means++是一种改进的初始化方法，旨在选择初始聚类中心时最大化聚类中心之间的距离。第一个聚类中心是随机选择的，之后的中心根据数据点与已选中心的最小距离的概率分布选择。

2.步骤：

2.1 随机选择一个点作为第一个聚类中心。

2.2 对于每个剩余数据点x，计算它与最近一个聚类中心的距离D(x)。

2.3 按照$D(x)_2$的概率分布选择下一个聚类中心。

2.4 重复步骤2和3，直到选择了K个聚类中心。

代码如下：

```
elif init_method == 'kmeans++':
    # K-Means++ 初始化
    self.centroids = []  # 初始化为空列表
    # 随机选择第一个聚类中心
    self.centroids.append(data[np.random.randint(data.shape[0])])
    # 选择剩余的聚类中心
    for _ in range(1, self.K):
        # 计算每个点到最近中心的距离
        distances = np.min(self.getDistanceToCenters(data, self.centroids), axis=1)
        # 概率分布：距离的平方
        probabilities = distances**2 / np.sum(distances**2)
        # 根据概率选择新的中心
        new_center = data[np.random.choice(data.shape[0], p=probabilities)]
        self.centroids.append(new_center)
    self.centroids = np.array(self.centroids)  # 转为 numpy 数组
```

3.2 样本分配

根据当前聚类中心，计算每个样本点到所有聚类中心的欧氏距离。将每个样本点分配到最近的聚类中心（即距离最小的中心）

```
def getDistance(self, data):
    distances = np.zeros((data.shape[0], self.centroids.shape[0]))  # 初始化距离矩阵
    for i in range(data.shape[0]):
        distances[i] = np.sum((self.centroids - data[i])**2, axis=1)**0.5  # 欧式距离计算
    return distances
def getClusters(self, data):
    distances = self.getDistance(data)  # 计算所有样本到聚类中心的距离
    clusters = np.argmin(distances, axis=1)  # 找到每个样本最近的聚类中心
    avgDistances = np.sum(np.min(distances, axis=1)) / data.shape[0]  # 计算平均距离
    return clusters, avgDistances
```

3.3 更新聚类中心

根据每个簇内的样本点计算新的聚类中心（簇内样本的均值）。先将将样本分配标签转换为 One-Hot 编码，然后通过矩阵运算计算每个簇的样本均值。

```
def getCentroids(self, data, clusters):
    oneHotClusters = np.zeros((data.shape[0], self.K))  # 初始化 one-hot 矩阵
    oneHotClusters[np.arange(data.shape[0]), clusters] = 1  # 将样本标签转化为 one-hot 编码
    # 计算每个簇的均值
    return np.dot(oneHotClusters.T, data) / np.sum(oneHotClusters, axis=0).reshape((-1, 1))
```

3.4 迭代

不断重复步骤2.3步骤：
使用 train 方法：
计算每轮更新中聚类中心的变化量（欧氏距离）。
更新聚类中心。

```
def train(self, data):
        clusters, _ = self.getClusters(data)
        newCentroids = self.getCentroids(data, clusters)  # 更新聚类中心
        diff = np.sum((newCentroids - self.centroids)**2)**0.5  # 计算变化幅度
        self.centroids = newCentroids
        return diff
```

如果达到最大迭代次数则停止迭代。

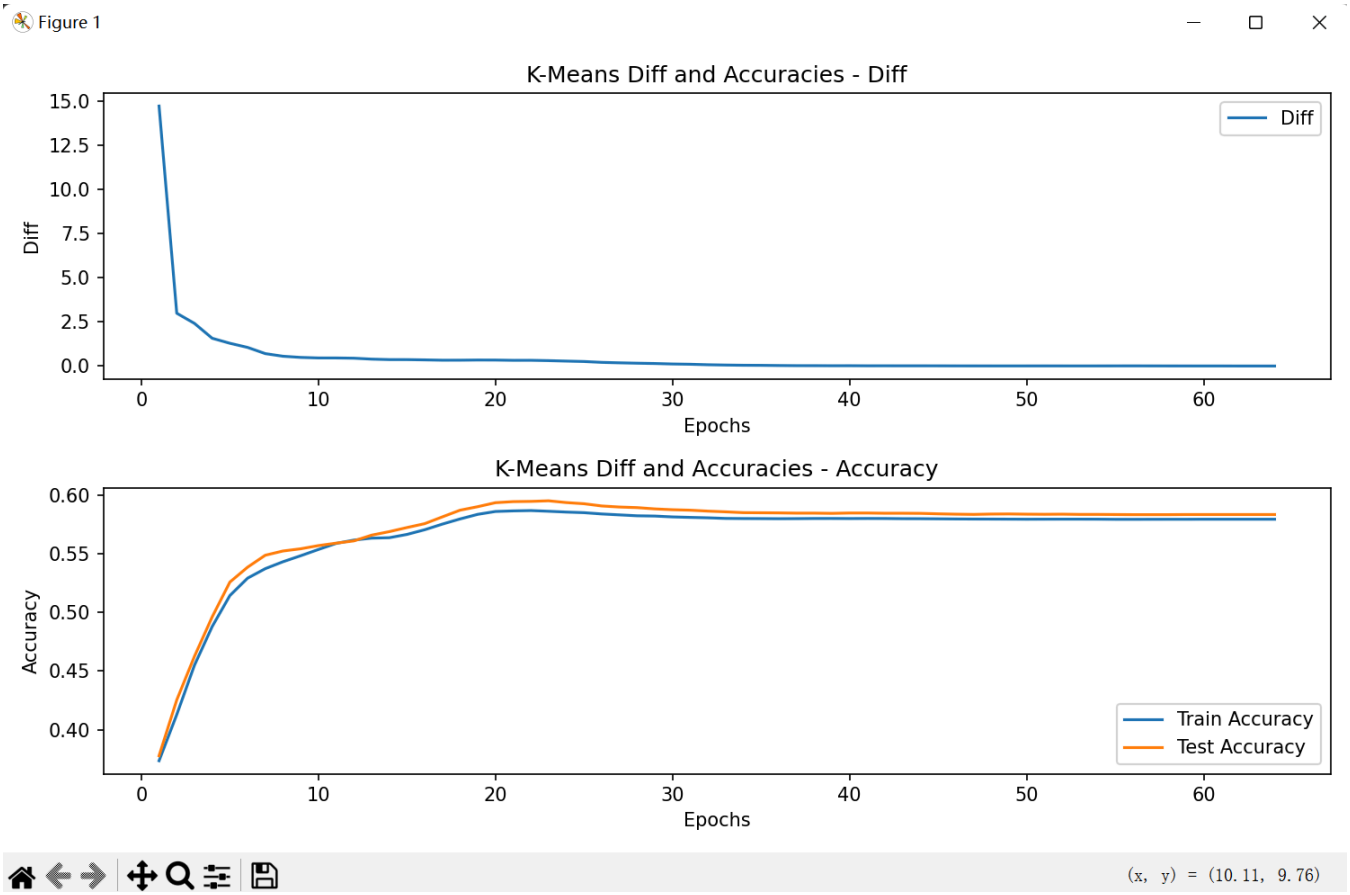
```
kmeans = myKmeans(init_data=X_train, K=10, init_method='random')#若选kmeans++则手动更换
    print("Training K-Means...")

    diffs = []
    train_accuracies = []
    test_accuracies = []
    start_time = time.time()

    for epoch in range(EPOCHS):
        diff = kmeans.train(X_train)
        train_acc, _ = kmeans.test(X_train, y_train)
        test_acc, _ = kmeans.test(X_test, y_test)

        diffs.append(diff)
        train_accuracies.append(train_acc)
        test_accuracies.append(test_acc)

        print(f"Epoch {epoch + 1}/{EPOCHS}:")
        print(f"  Diff: {diff:.6f}")
        print(f"  Train - Acc: {train_acc:.4f}")
        print(f"  Test  - Acc: {test_acc:.4f}")

        if diff < 1e-6:
            break

    end_time = time.time()
    print(f"K-Means Training Completed in {end_time - start_time:.2f} seconds.")

    # 绘制 K-Means 的 diff 曲线
    plot_metrics(
        epochs=list(range(1, len(diffs) + 1)),
        diffs=diffs,
        train_accuracies=train_accuracies,
        test_accuracies=test_accuracies,
        title="K-Means Diff and Accuracies"
    )
```

3.5 结果对比

在K-Means中用diff和accuracy来评估聚类性能。
diff表示的是当前迭代中聚类中心的变化幅度

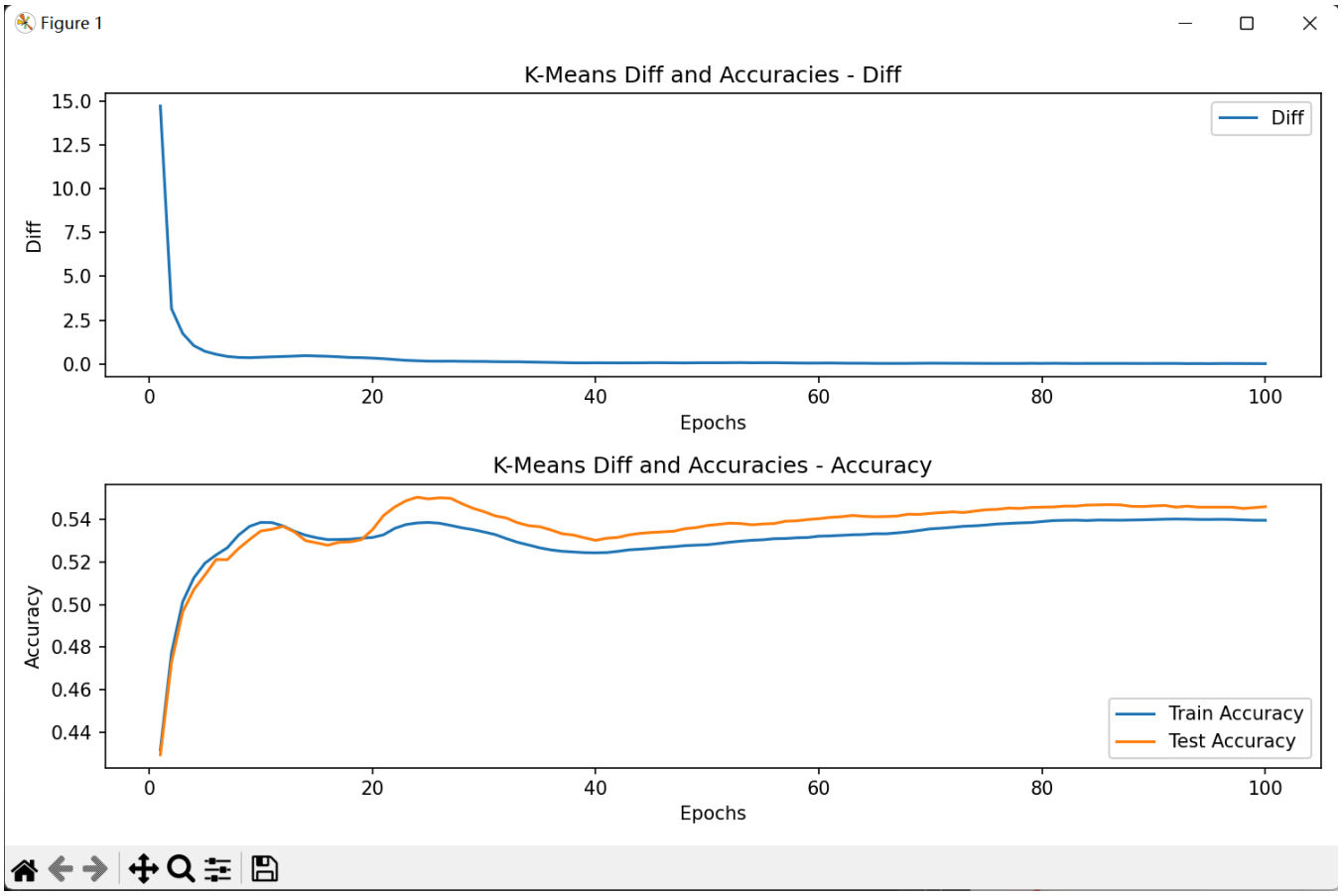
```
diff = np.sum((newCentroids - self.centroids)**2)**0.5
```

accuracy表示通聚类结果与真实类别标签的匹配程度。
分别用两种初始化方法迭代100次，运行结果如下：
随机初始化：



```
  Test  - Acc: 0.5834
Epoch 64/100:
  Diff: 0.000000
  Train - Acc: 0.5794
  Test  - Acc: 0.5834
K-Means Training Completed in 41.12 seconds.
```

K-Means++初始化：




```
Epoch 100/100:
  Diff: 0.021661
  Train - Acc: 0.5396
  Test  - Acc: 0.5460
K-Means Training Completed in 63.54 seconds.
```

1.对比可以看出用Kmeans++初始化训练时间比随机初始化训练时间长，因为如果数据本身的聚类结构简单，随机初始化可能快速选择到接近真实簇中心的点，导致快速收敛。而K-Means++初始化在这种情况下可能多此一举，花费额外时间计算距离分布。

2.accuracy：对比可以看出Kmeans++初始化准确率比随机初始化准确率低，K-Means++ 的初始化策略倾向于将簇中心分布得尽可能远，这种方式在大多数情况下效果更好，但如果数据分布不均匀（如某些簇非常密集），K-Means++ 的初始中心可能会偏离实际最佳中心。

3.而这两种初始化方法对应的diff都随着迭代次数逐渐减小并趋于0，说明其聚类中心都不再变化。

4 GMM 训练过程

4.1 初始化

1.输入参数：
n_components: 高斯分量的数量K。
init_data: 用于初始化的数据集。
init_type: 初始化类型（支持 random 和 kmeans）。
cov_type: 协方差矩阵类型（支持 full, diag, tied 和 spherical）。
reg_covar: 协方差矩阵的正则化项，防止矩阵不可逆。
isUsedPi: 是否在高斯分布计算中考虑常数项。
2.参数初始化：
随机初始化每个高斯分量的均值means。
初始化每个分量的权重 weights 为均匀分布：$\pi_k=1/K$。
根据协方差矩阵类型初始化每个分量的协方差矩阵cov：
full: 每个分量有独立的完整协方差矩阵。
diag: 每个分量有独立的对角协方差矩阵。
tied: 所有分量共享一个完整协方差矩阵。
代码如下：

```
def __init__(self, n_components, init_data, init_type='random', cov_type='full', reg_covar=1e-6, isUsedPi=False):
    self.n_components = n_components
    self.dimension = init_data.shape[1]
    self.weights = None
    self.means = None
    self.cov = None
    self.cov_type = cov_type  # 协方差矩阵类型
    self.reg_covar = reg_covar
    self.isUsedPi = isUsedPi
    self._init_parameters(init_type=init_type, cov_type=cov_type, init_data=init_data)

def _init_parameters(self, init_data, init_type='random', cov_type='full'):
    indexes = np.arange(init_data.shape[0])
    np.random.shuffle(indexes)
    self.means = init_data[indexes[:self.n_components]]  # 随机选择均值
    self.weights = np.ones(self.n_components) / self.n_components  # 初始化权重

    # 根据 cov_type 初始化协方差矩阵
    tempCov = np.cov(init_data, rowvar=False) + np.eye(self.dimension) * self.reg_covar
    # 每个分量独立的完整协方差矩阵
    if cov_type == 'full':
        self.cov = tempCov[np.newaxis, :].repeat(self.n_components, axis=0)
    # 对角矩阵
    elif cov_type == 'diag':
        self.cov = np.diag(tempCov).repeat(self.n_components, axis=0).reshape(self.n_components, -1)
     # 所有分量共享一个协方差矩阵
    elif cov_type == 'tied':
        self.cov = tempCov
```

3.2 训练(EM算法)

训练阶段通过E步骤（Expectation）和M步骤（Maximization）迭代进行，直到达到最大迭代次数或收敛条件。
E步骤：对于每个样本x_i和每个分量k，计算样本属于该分量的后验概率（责任值）

```
def EStep(self, data):
        gamma = np.zeros((data.shape[0], self.n_components))
        for k in range(self.n_components):
            gamma[:, k] = self.weights[k] * self.gaussianFunc(data, self.means[k],
                                                              self.cov[k])
        gamma /= np.sum(gamma, axis=1).reshape(-1, 1)
        return gamma
```

M步骤：更新均值，权重和协方差矩阵

```
def MStep(self, data, gamma):
        """
        M 步，更新 GMM 模型参数
        :param data: 样本数据
        :param gamma: 样本属于各个高斯分量的责任值
        """
        self.means = np.dot(gamma.T, data) / np.sum(gamma, axis=0).reshape(-1, 1)  # 更新均值
        self.weights = np.sum(gamma, axis=0) / data.shape[0]  # 更新权重

        if self.cov_type == 'full':
            # 每个分量有独立的完整协方差矩阵
            for k in range(self.n_components):
                diff = data - self.means[k]
                self.cov[k] = np.dot(gamma[:, k] * diff.T, diff) / np.sum(gamma[:, k]) +
                                        np.eye(self.dimension) * self.reg_covar
        elif self.cov_type == 'diag':
            # 每个分量有独立的对角协方差矩阵
            for k in range(self.n_components):
                diff = data - self.means[k]
                self.cov[k] = np.sum(gamma[:, k][:, np.newaxis] * diff**2, axis=0) / np.
                                        sum(gamma[:, k]) + self.reg_covar
        elif self.cov_type == 'tied':
            # 所有分量共享一个完整协方差矩阵
            tied_cov = np.zeros((self.dimension, self.dimension))
            for k in range(self.n_components):
                diff = data - self.means[k]
                tied_cov += np.dot((gamma[:, k][:, np.newaxis] * diff).T, diff)
            self.cov = tied_cov / data.shape[0] + np.eye(self.dimension) * self.reg_covar
```

4.3 测试

在测试阶段，GMM根据样本的后验概率对其进行分类，并计算性能指标（准确率和对数似然）
准确率： 构建一个代价矩阵 costMatrix，矩阵的每个元素表示某预测簇和某真实簇之间的重叠样本数的负，使用匈牙利算法找到代价矩阵中最优的标签映射关系，然后根据映射关系，将predLabels转换为与真实标签对齐的形式，最后统计映射后的预测标签与真实标签相等的样本比例，即准确率。

```
def getAccuracy(self, predLabels, trueLabels):
        # 获取预测标签和真实标签的种类
        predLabelType = np.unique(predLabels)
        trueLabelType = np.unique(trueLabels)

        # 确定代价矩阵的大小，取预测标签和真实标签种类数的较大值
        labelNum = np.maximum(len(predLabelType), len(trueLabelType))
        costMatrix = np.zeros((labelNum, labelNum))

        # 构造代价矩阵，表示预测标签和真实标签之间的匹配代价
        for i in range(len(predLabelType)):
            # 当前预测标签对应的样本掩码
            chosenPredLabels = (predLabels == predLabelType[i]).astype(float)
            for j in range(len(trueLabelType)):
                # 当前真实标签对应的样本掩码
                chosenTrueLabels = (trueLabels == trueLabelType[j]).astype(float)
                # 代价矩阵中存储负的交集样本数
                costMatrix[i, j] = -np.sum(chosenPredLabels * chosenTrueLabels)

        # 使用 Munkres（匈牙利算法）计算最优匹配
        m = Munkres()
        indexes = m.compute(costMatrix)

        # 根据最优匹配结果映射预测标签到真实标签
        mappedPredLabels = np.zeros_like(predLabels, dtype=int)
        for index1, index2 in indexes:
            if index1 < len(predLabelType) and index2 < len(trueLabelType):
                mappedPredLabels[predLabels == predLabelType[index1]] = trueLabelType[index2]

        # 计算准确率：映射后的预测标签与真实标签相等的样本比例
        return np.sum((mappedPredLabels == trueLabels).astype(float)) / trueLabels.size
```

对数似然的负值：

表示为负的对数似然，计算过程中只考虑每个样本的最大后验概率

其计算公式如下：

$$\text{Loss} = -\sum_{i=1}^{N} \log\left(\max_{k}(\gamma_{ik})\right)$$

```
-np.sum(np.log(np.max(gamma, axis=1)))
```

4.4 训练

训练在train方法中进行，每次调用EStep和MStep：

```
def train(self, data):
        gamma = self.EStep(data)
        self.MStep(data, gamma)
```

直到达到最大迭代次数或者模型参数的变化小于预设阈值时停止

4.5 结果与对比

输出最终模型参数和聚类结果，本次实验用准确率accuracy和对数似然的负值loss来评估GMM的性能
探索使用不同结构的协方差矩阵对聚类性能的影响，本次实验用了三种，分别是

默认协方差矩阵结构full:即每个分量有各自不同的标准协方差矩阵，完全协方差矩阵（元素都不为零）；

对角协方差矩阵diag:所有分量有相同的标准协方差矩阵;

共享协方差矩阵tied:每个分量有各自不同对角协方差矩阵（非对角为零，对角不为零）

默认协方差矩阵结构full：
初始化：

```
tempCov = np.cov(init_data, rowvar=False) + np.eye(self.dimension) * self.reg_covar
if cov_type == 'full':
    self.cov = tempCov[np.newaxis, :].repeat(self.n_components, axis=0)
```

更新时：

```
if self.cov_type == 'full':
    # 每个分量有独立的完整协方差矩阵
    for k in range(self.n_components):
        diff = data - self.means[k]
        self.cov[k] = np.dot(gamma[:, k] * diff.T, diff) / np.sum(gamma[:, k]) +
                              np.eye(self.dimension) * self.reg_covar
```

对角协方差矩阵diag：

初始化：

```
tempCov = np.cov(init_data, rowvar=False) + np.eye(self.dimension) * self.reg_covar
    # 对角矩阵
elif cov_type == 'diag':
    self.cov = np.diag(tempCov).repeat(self.n_components, axis=0).reshape(
                self.n_components, -1)
```

更新时：

```
 elif self.cov_type == 'diag':
            # 每个分量有独立的对角协方差矩阵
            for k in range(self.n_components):
                diff = data - self.means[k]
                self.cov[k] = np.sum(gamma[:, k][:, np.newaxis] * diff**2, axis=0) / np.
                                        sum(gamma[:, k]) + self.reg_covar
        elif self.cov_type == 'tied':
```

共享协方差矩阵tied：

初始化：

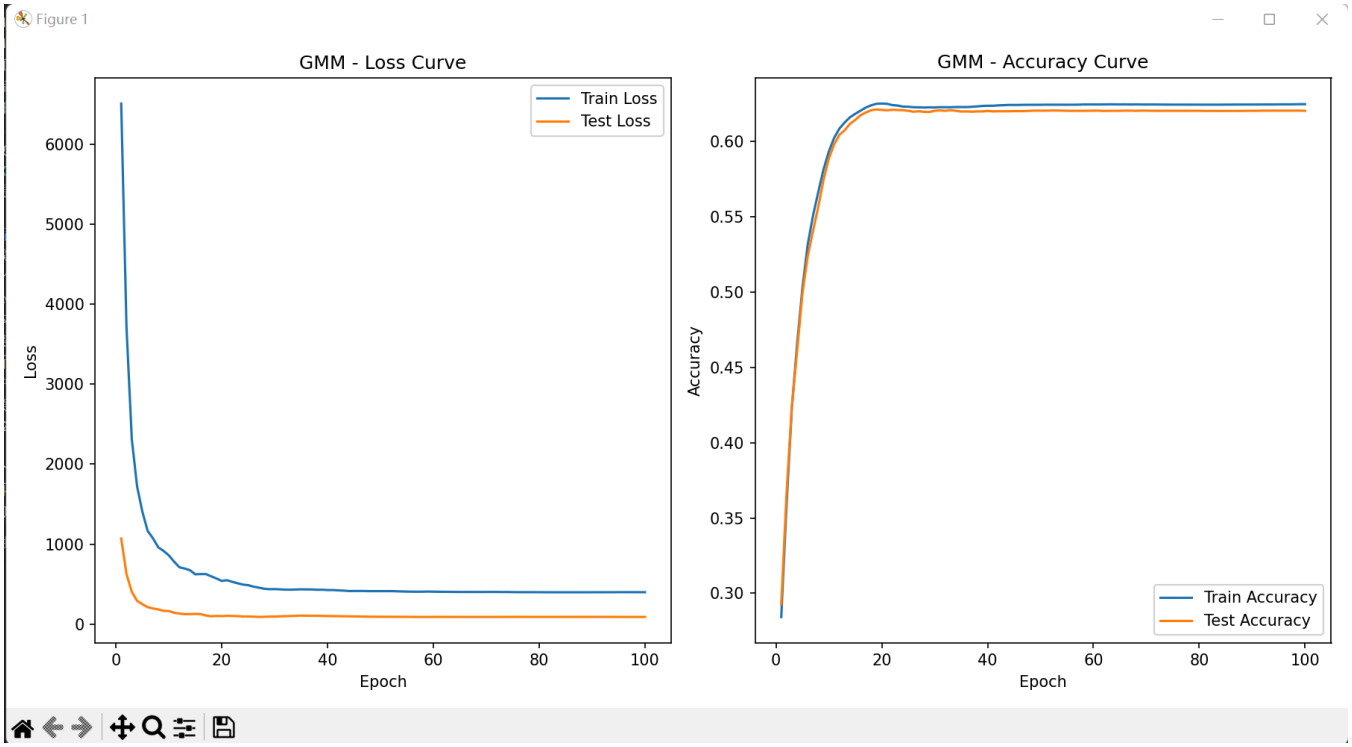
```
tempCov = np.cov(init_data, rowvar=False) + np.eye(self.dimension) * self.reg_covar
elif cov_type == 'tied':
    self.cov = tempCov  # 所有分量共享一个协方差矩阵
```

更新时：

```
elif self.cov_type == 'tied':
    # 所有分量共享一个完整协方差矩阵
    tied_cov = np.zeros((self.dimension, self.dimension))
    for k in range(self.n_components):
        diff = data - self.means[k]
        tied_cov += np.dot((gamma[:, k][:, np.newaxis] * diff).T, diff)
    self.cov = tied_cov / data.shape[0] + np.eye(self.dimension) * self.reg_covar
```

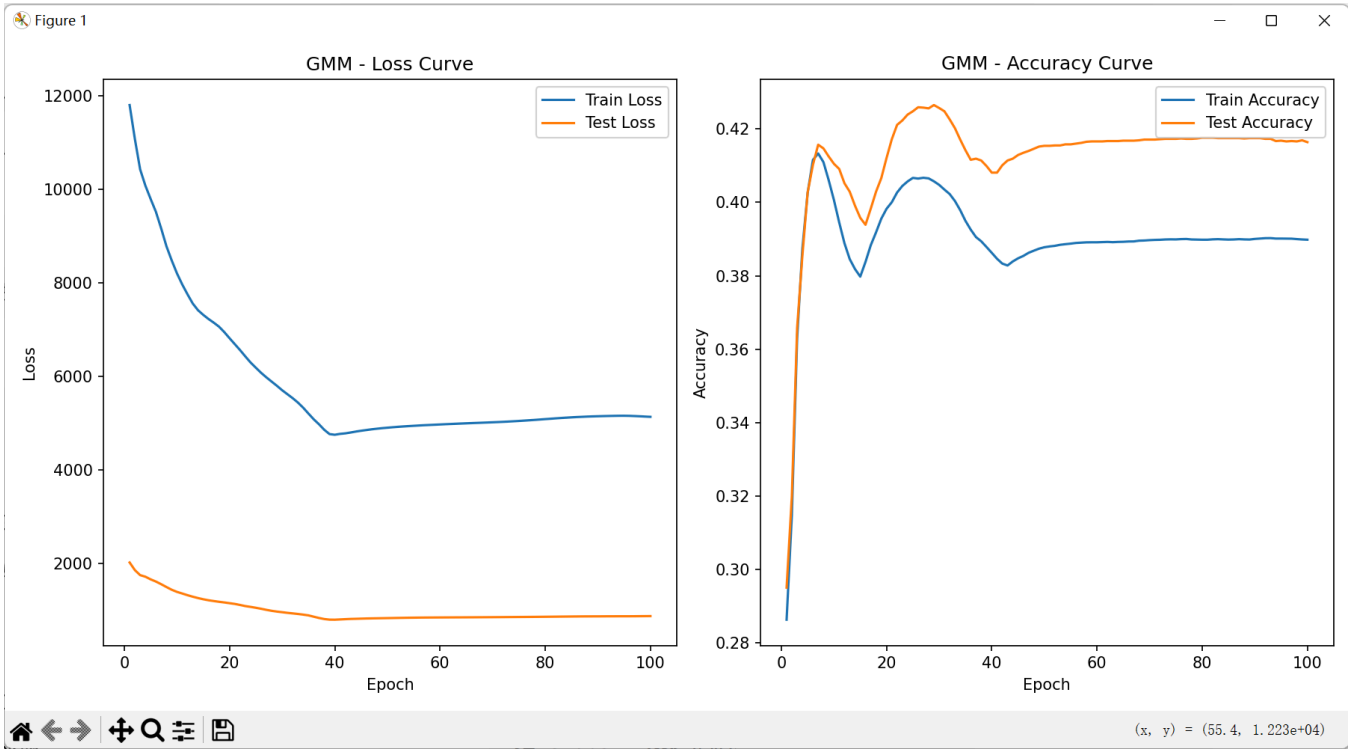
运行结果如下：

full：



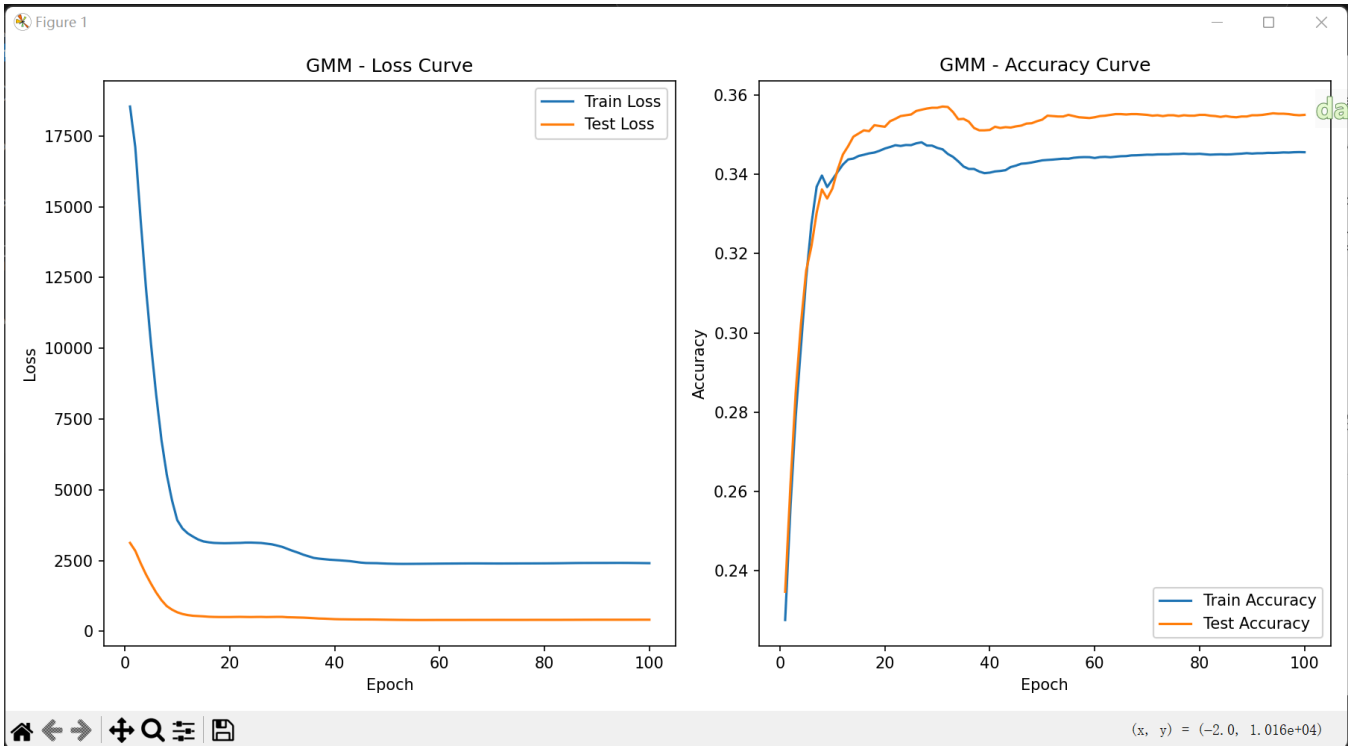

```
  Train - Loss: 400.8189, Acc: 0.6249
  Test  - Loss: 91.2605, Acc: 0.6206
Epoch 100/100:
  Train - Loss: 400.3540, Acc: 0.6249
  Test  - Loss: 91.2610, Acc: 0.6205
GMM Training Completed in 106.58 seconds.
```

diag：




```
Epoch 99/100:
  Train - Loss: 5139.1799, Acc: 0.3899
  Test  - Loss: 867.0017, Acc: 0.4169
Epoch 100/100:
  Train - Loss: 5133.7248, Acc: 0.3898
  Test  - Loss: 867.8331, Acc: 0.4164
GMM Training Completed in 108.94 seconds.
```

tied:




```
  Test  - Loss: 411.7502, Acc: 0.3550
Epoch 99/100:
  Train - Loss: 2411.3462, Acc: 0.3456
  Test  - Loss: 411.9805, Acc: 0.3549
Epoch 100/100:
  Train - Loss: 2409.0834, Acc: 0.3456
  Test  - Loss: 411.9969, Acc: 0.3550
GMM Training Completed in 98.75 seconds.
```

对比可以看出，

训练时间：

用tied矩阵的训练时间最短，diag训练时间最长。

因为tied所有高斯分量共享一个协方差矩阵，故协方差矩阵更新只需计算一次，因此计算量最小。

准确率：

用full矩阵准确率最高，tied最差。

对于full，因为每个分量有独立的完整协方差矩阵，能够捕捉数据的复杂分布特性，故其能够更好地拟合真实数据分布。

而对于tied，因为其所有高斯分量共享一个协方差矩阵，模型自由度较低，无法充分拟合簇间的

差异，导致聚类结果不够精确。

对数似然：

full矩阵的对数似然的负值达到最低，diag最高。

对数似然的负值越小，表示模型对数据的拟合程度越好，因为full协方差矩阵自由度最高，所以能最好地拟合数据的真实分布。

综合来看，采用full协方差结构矩阵的模型表现要比另外两个都要好，其对数据的拟合效果最好。