机器学习 assignment3 实验报告

21307347
陈欣宇

一、问题描述

1.1 实验内容

探索 K-Means 和 GMM 这两种聚类算法的性能。

1.2 实验要求

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。

二、实验过程

2.1 基本思路：对 MNIST 数据集实现 K-Means 和 GMM 聚类算法。

K-Means 首先初始化 K 个聚类中心（①随机选择 K 个点 ②随机选择一个点，再不断计算离当前聚类中心们最远的点，直至选择了 K 个点），训练过程主要有两个步骤，划分样本点和更新聚类中心，每次划分样本点后计算损失函数（样本点与聚类中心距离的平均数），更新聚类中心后计算聚类中心移动距离，经测试，移动距离的变化更直观地反应了训练有无收敛，未收敛则继续划分样本点，否则跳出循环，得到最终聚类中心。

GMM 称为高斯混合模型，模型表达式如下：p 为权重，Φ表示单个高斯模型，μ和Σ分别表示均值和协方差。

$$P(x|\theta)=\sum_{i}^{K}p_i\phi(x|\mu_i,\Sigma_i)$$

其中

$$\phi(x|\mu_i,\Sigma_i)=\frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma_i|^{\frac{1}{2}}}\exp\left(-\frac{(x-\mu_i)^T\Sigma_i^{-1}(x-\mu_i)}{2}\right)$$

首先确定初始的平均值集和协方差集（①μ取样本集上随机点，Σ使用样本生成；②随机生成Υ_{nk}，使用以下的 M 步计算μ和Σ。p 平分），训练过程采用 EM 算法，直接介绍更新公式：

E-step：

$$\gamma_{ij}=\frac{p_j\phi(x_i|\mu_j,\Sigma_j)}{\sum_{k=1}^{K}p_k\phi(x_i|\mu_k,\Sigma_k)}$$

M-step：

$$p_j^{(t+1)}=\frac{1}{N}\sum_{i=1}^{N}\gamma_{ij}\qquad\mu_j^{(t+1)}=\frac{\sum_{i=1}^{N}x_i\gamma_{ij}}{\sum_{i=1}^{N}\gamma_{ij}}$$

$$\Sigma_j^{(t+1)}=\frac{\sum_{i=1}^{N}(x_i-\mu_j)^T(x_i-\mu_j)\gamma_{ij}}{\sum_{i=1}^{N}\gamma_{ij}}$$

使用对数似然函数$L(\theta)=\ln P(X|\theta)$作为损失函数，训练不断进行 EM 步至$L(\theta)$收敛。

其他：在协方差矩阵初始化中，可加入微小数防止矩阵不可逆。

2.2 代码实现：

2.2.1 数据预处理：读取并使用 PCA 进行降维操作

```
    train_data = pd.read_csv("data/mnist_train.csv")
    test_data = pd.read_csv("data/mnist_test.csv")
    TrainData = train_data.iloc[:,1:].values
    train_labels = train_data.iloc[:,0].values
    TestData = test_data.iloc[:,1:].values
    test_labels = test_data.iloc[:,0].values
    # 数据降维
    pcaModel = PCA(n_components=NEW_DIMENSION)
    pcaModel.fit(TrainData)
    TrainData = pcaModel.transform(TrainData)
    TestData = pcaModel.transform(TestData)
```

将 K-Means 和 GMM 封装为两个类分别进行调用

2.2.2 K-Means

K-means 成员函数如下：

```
class Kmeans():
    def __init__(self, data, K, cen_init):…
    def initCentroids(self, data, cen_init):    #初始化聚类中心…
    def getDistance(self, data):    #计算所有点到当前聚类中心的距离…
    def getClusters(self, data):    #划分样本点，计算loss…
    def getCentroids(self, data, clusters): #更新聚类中心…
    def getAccuracy(self, clusters, Labels):  #计算结果准确率…
    def train(self,data):   #训练函数…
    def test(self,data,labels): #测试函数…
```

训练过程：

```
k = Kmeans(TrainData,10,'random')
start_time = time.time()
for i in range(EPOCHS):
    loss,diff = k.train(TrainData)
    acc = k.getAccuracy(k.clusters, train_labels)
    print('epochs:{}\tloss = {:.4f}\tdiff={:.4f}\tacc =
{:.4f}'.format(i+1,loss,diff, acc))
    if diff < 1e-7:
        break
end_time = time.time()
acc, loss = k.test(TestData, test_labels)
print('test: loss = {:.4f}\tacc = {:.4f}'.format(loss, acc))
print('Total time:{:.2f}s.\n'.format(end_time-start_time))
```

主要代码：

聚类中心初始化：根据类型'random''distance'分别选择①随机选择 K 个点 ②随机选择一个点，再不断计算离当前聚类中心们最远的点，直至选择了 K 个点

```
def initCentroids(self, data, cen_init):    #初始化聚类中心
        indexes = np.arange(data.shape[0])
```

```
        np.random.shuffle(indexes)
        self.centroids = np.zeros((self.K, data.shape[1]))
        if cen_init=='random':
            for i in range(self.K):
                self.centroids[i] = data[indexes[i]]
        elif cen_init=='distance':
            usedIndexes = list()
            for i in range(self.K):
                if i==0:
                    self.centroids[i] = data[indexes[i]]
                    usedIndexes.append(indexes[i])
                else:
                    distances = self.getDistance(data)  #与聚类中心距离 60000 x K
                    totalDistances = np.sum(distances, axis=1) #距离和 60000 x 1
                    indexes = np.argsort(-totalDistances)
                    for index in indexes:
                        if index not in usedIndexes:
                            self.centroids[i] = data[index]
                            usedIndexes.append(index)
                            break
        self.clusters,_ = self.getClusters(data)
```

划分样本点:

```
    def getClusters(self, data):    #划分样本点，计算 loss
        distances = self.getDistance(data)
        clusters = np.argmin(distances, axis=1)
        avgDistances = np.sum(np.min(distances, axis=1))/data.shape[0]
        return clusters, avgDistances
```

更新聚类中心:

```
def getCentroids(self, data, clusters): #更新聚类中心
        oneHotClusters = np.zeros((data.shape[0], self.K))
        for i in range(data.shape[0]):
            oneHotClusters[i,clusters[i]] = 1
        CluSumDis = np.dot(oneHotClusters.T, data)
        CluNum = np.sum(oneHotClusters, axis=0).reshape((-1,1))
        return CluSumDis/CluNum
```

准确度计算：在分类结束后设计类与真实标签匹配的问题，直接调用 Munkres 匈牙利算法进行匹配操作，得到标签映射，便于最后矩阵计算

```
def getAccuracy(self, clusters, Labels):  #计算结果准确率
        clustersType = np.unique(clusters)
        LabelType = np.unique(Labels)
        labelNum = np.maximum(len(clustersType), len(LabelType))
        costMatrix = np.zeros((labelNum, labelNum)) # 代价矩阵
        for i in range(len(clustersType)):
            selclusters = (clusters==clustersType[i]).astype(float)
```

```
            for j in range(len(LabelType)):
                sellabels = (Labels==LabelType[j]).astype(float)
                costMatrix[i,j] = -np.sum(selclusters*sellabels)    # 越小匹配度越高
        m = Munkres()
        indexes = m.compute(costMatrix) # 匈牙利算法->索引映射
        maplabels = np.zeros_like(clusters, dtype=int)
        for index1,index2 in indexes:
            if index1<len(clustersType) and index2<len(LabelType):
                maplabels[clusters==clustersType[index1]] = LabelType[index2]
        return np.sum((maplabels==Labels).astype(float))/Labels.size
```

训练函数：训练过程不断调用 train，更新聚类中心并重新划分样本点，diff 记录聚类中心更新前后差距，据此判断训练是否收敛。

```
def train(self,data):   #训练函数
        newcentroids = self.getCentroids(data, self.clusters)
        diff = np.sum((newcentroids-self.centroids)**2)**0.5
        self.centroids = newcentroids
        self.clusters, loss = self.getClusters(data)
        return loss,diff
```

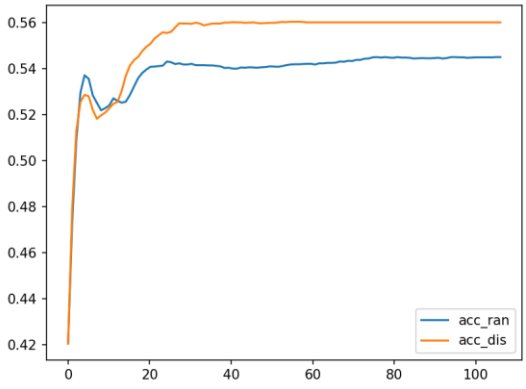
K-Means 测试过程：

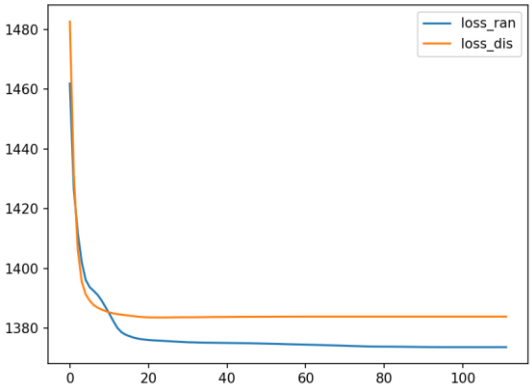
因结果具有一定随机性，调参先根据 10 次测试结果的平均值，比对结果取较优参数，便于之后比较。初始未降维数据总体只能到达 55%左右的准确率，使用 PCB 对最终结果影响不大，但大大加快了训练速度，因此在后续可以使用降维后数据进行性能比较。其中降至 40-70 时准确度处于正常数值，考虑效率和准确性，下面实验均将样本维度降至 60 进行比对。

初始化方式	Random①			Distance②		
PCB 维度	准确度	收敛轮数	时间/s	准确度	收敛轮数	时间/s
未降维(784)	0.5486	112.1	129.08	0.5275	58.9	66.75
40	0.5375	89.7	45.68	0.5469	71.2	40.98
50	**0.5630**	**89.2**	**49.52**	0.5449	74.2	36.51
60	0.5371	88.7	53.03	**0.5531**	**82.4**	**41.94**
70	0.5326	80.4	42.39	0.5395	67.6	34.71

对不同的聚类中心初始化方式进行性能比较：由上表可知 Distance 效果优于 random，二者准确度无明显差别，distance 的收敛速度较快。

下图分别为 random 和 distance 初始化方式的准确率 acc 和 loss 变化图，random 模式下的损失函数降到较低值，但相应的准确度不如 distance 模式，可能是陷入局部最优的之中。





2.2.3 GMM

GMM 成员函数如下：

```
class GMM():
    def __init__(self, K, data, init_type='random', cov_type='commom', reg_covar=1e-6):…
    def _init_parameters(self, data, init_type='random', cov_type='full'):  #初始化均值、协方差、权重
    def EStep(self, data):…
    def MStep(self, data, gamma):…
    def gaussfunc(self, x, mean, cov):  #计算高斯函数…
    def getAccuracy(self, clusters, Labels): #计算结果准确率…
    def train(self, data,gam):  #训练函数…
    def test(self, data, labels):   #测试函数…
    def getfullcov(self):   #计算完整协方差矩阵…
```

训练过程：

```
G = GMM(10,TrainData,init_type='random',cov_type='ellipse')
    start_time = time.time()
    gamma = np.zeros((TrainData.shape[0], 10))  #用于计算是否收敛
    for i in range(EPOCHS):
        gamma,diff = G.train(TrainData,gamma)
        acc = G.test(TrainData, train_labels)
        print('epochs:{}\tdiff:{:.4f}\tacc = {:.4f}'.format(i+1,diff,acc))
        if diff < 1e-4:
            break
    end_time = time.time()
    acc = G.test(TestData, test_labels)
    print('Test: acc = {:.4f}'.format(acc))
    print('Total time:{:.2f}s.\n'.format(end_time-start_time))
```

主要代码：

参数初始化：两种模型初始化方法'random''randgamma'分别为①μ取样本集上随机点，Σ使用样本生成；②随机生成Υ_{nk}，使用以下的 M 步计算μ和Σ。其中 random 方法中协方差矩阵使用 3 中初始结构（①common:普通矩阵②circle:对角且元素值都相等③ellipse:对角但元素值不要求相等）。此处因为协方差矩阵只存在对角元素甚至对角元素相等，因此 circle 类型每个协方差矩阵只存储一个值，ellipse 类型每个协方差矩阵使用一维列表存储对角数据。

```
def _init_parameters(self, data, init_type='random', cov_type='full'):
        # init_type:'random', 'randgamma'
        # cov_type:'commom', 'circle', 'ellipse'
        if init_type=='random':
            indexes = np.arange(data.shape[0])
            np.random.shuffle(indexes)
            self.means = np.zeros((self.K, self.dimension))
            self.means = data[indexes[:self.K]]
            self.weights = np.ones(self.K)/self.K
            tempCov = np.cov(data, rowvar=False)
            tempCov += np.eye(self.dimension)*self.reg_covar
            if cov_type=='commom':
                self.cov = tempCov[np.newaxis,:].repeat(self.K, axis=0)
            elif cov_type=='circle':
                self.cov = np.ones(self.K)*np.diag(tempCov).mean()
```

```
            elif cov_type=='ellipse':
                self.cov = np.diag(tempCov)
                self.cov = self.cov[np.newaxis, :].repeat(self.K, axis=0)
        elif init_type=='randgamma':
            gamma = np.random.rand(data.shape[0], self.K)
            gamma /= np.sum(gamma, axis=1).reshape(-1,1)
            self.MStep(data, gamma)
```

M-step 函数：先对高斯混合模型的平均值和权重序列进行更新，再根据协方差矩阵类型进行不同类型的更新，计算中加入对角非负正则化

```
def MStep(self, data, gamma):
        self.means = np.dot(gamma.T, data)/np.sum(gamma, axis=0).reshape(-1,1)
        self.weights = np.sum(gamma, axis=0)/data.shape[0]
        if self.cov_type=='commom':
            self.cov = np.zeros((self.K, self.dimension, self.dimension))
            for k in range(self.K):
                diff = data - self.means[k]
                self.cov[k] = np.dot(gamma[:,k]*diff.T, diff)
                self.cov[k] /= np.sum(gamma[:,k])
                self.cov[k] += np.eye(self.dimension)*self.reg_covar
        elif self.cov_type=='circle':
            self.cov = np.zeros((self.K))
            for k in range(self.K):
                diff = data - self.means[k]
                temp = np.dot(gamma[:,k]*diff.T, diff)
                temp /= np.sum(gamma[:,k])
                temp += np.eye(self.dimension)*self.reg_covar
                self.cov[k] = np.diag(temp).mean()
        elif self.cov_type=='ellipse':
            self.cov = np.zeros((self.K, self.dimension))
            for k in range(self.K):
                diff = data - self.means[k]
                temp = np.dot(gamma[:,k]*diff.T, diff)
                temp /= np.sum(gamma[:,k])
                temp += np.eye(self.dimension)*self.reg_covar
                self.cov[k] = np.diag(temp)
```

E-step 函数：计算Υ_{nk}，调用高斯函数以及权重进行计算

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

高斯函数：

```
def gaussfunc(self, x, mean, cov):
    diff = x - mean
    expon = -0.5*(np.sum(np.dot(diff,np.linalg.pinv(cov))*diff, axis=1))
    return np.exp(expon)/(((2*np.pi)**(self.dimension/2))*(np.sqrt(np.linalg.det(cov))))
```

准确度函数与 K-Means 基本一致

train 函数：调用 EStep 和 Mstep 即可，其中加入 gam 参数为上一轮 train 计算 gamma 值，用于计算 gamm 变化大小以判断是否收敛。

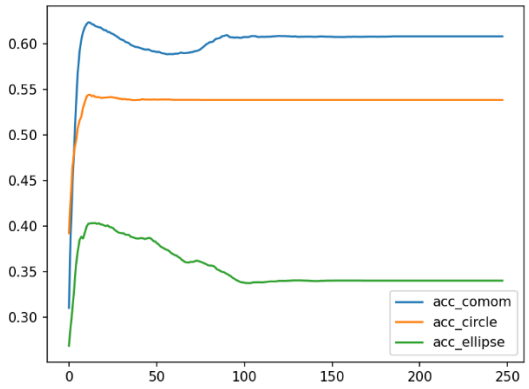
```
def train(self, data,gam):
        gamma = self.EStep(data)
        diff = np.linalg.norm(gamma-gam)
        self.MStep(data, gamma)
        return gamma,diff
```

GMM 测试过程：

这里直接使用上面测试的降维至 60，与 K-Means 明显的不同就是收敛轮数变多了，且收敛轮数波动较大，容易出现很多轮未判定收敛的情况，但准确性基本在 100 轮以后保持稳定。

比较不同协方差矩阵结构的性能：

训练过程会随机性出现难以收敛的情况，但三种协方差结构的性能比较都呈现在以下的准确率图像上，其中主要普通矩阵容易出现 gamma 值难收敛的情况，但普通矩阵能达到的准确率最高 60%以上，而对角且元素相等的矩阵保持与 K-Means 差不多的准确率，特点是收敛速度快，而对角且元素不等的矩阵准确率只有 33%左右，对角矩阵因为在不同维度之间保持独立，虽然能够加快训练速度，但准确度也遭到了很大限制。



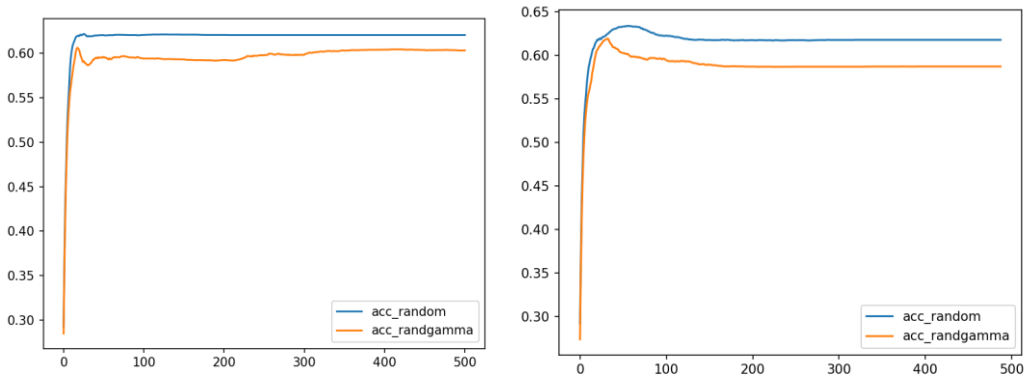
多次样例	收敛轮数	准确度
Commom	387.75	0.6052
Circle	126.24	0.5349
Ellipse	334.43	0.3447

注：该比对为 pcb 降维至 50 之后的结果

比较不同初始化方法的性能：

‘Random’模型协方差矩阵采用‘commom’模式，与‘randgamma’模型进行比较，在同样的打乱条件下，根据样本生成协方差比随机生成γ方案的准确率略好一些，二者的判定收敛轮数都具有很大随机性，但准确率在 100 轮之后基本稳定。

以下是两次重复测试结果：random 模式的准确率要略高于 randgamma 模式

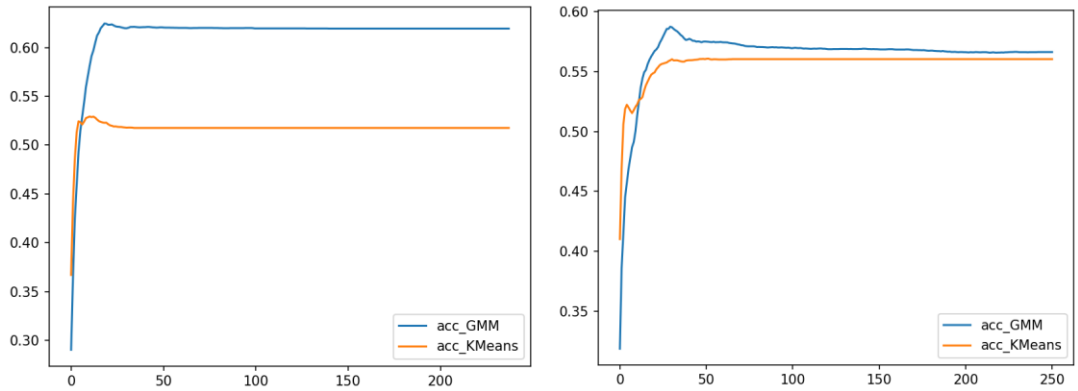


多个样例取平均测试结果表明 random 在准确率和收敛性上都要优于 randgamma

	准确率	收敛轮数	时间/s
random	0.6115	327.26	281.87
randgamma	0.5953	491	392.73

2.2.4 K-Means 与 GMM 性能比较

K-means 使用表现较优的‘Distance’模式，GMM 使用 random 的‘commom’模式，进行比较，以下是两次不同随机种子下的训练比较。



可以看出 GMM 算法的性能要明显优于 K-Means 算法，理论上来看，GMM 算法不仅通过更新聚类中心来优化模型，还有协方差、权重等更新来更好地拟合数据分布，多于 K-Means 的参数更新模型，具有优于 K-Means 的性能。

三、实验总结

本次实验实现了 K-Means 和 GMM 两类无监督学习算法。对其中不同初始化方式和结构的性能进行了比对，对模型的具体实现有了更好的理解。最终结果最多只能达到 60%的准确率，因为数据集是手写数字图像集，用无监督的聚类算法并不是很好的解决方案，就不再探究如何提升准确率的问题。但通过手动实现算法，对于这两类模型的细节处理和调参都有了新的体会。