哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称： 机器学习

课程类型： 选修

实验题目： 实现 k-means 聚类和混合高斯模型

学号：1190201008
姓名： 周凡

一、 实验目的

实现一个 k-means 算法和混合高斯模型，并且用 EM 算法估计模型中的参数。

二、 实验要求及实验环境

1. 实验要求

测试：

用高斯分布产生 k 个高斯分布的数据（不同均值和方差）（其中参数自己设定）。

（1）用 k-means 聚类，测试效果；

（2）用混合高斯模型和你实现的 EM 算法估计参数，看看每次迭代后似然值变化情况，考察 EM 算法是否可以获得正确的结果（与你设定的结果比较）。

应用：可以 UCI 上找一个简单问题数据，用你实现的 GMM 进行聚类。

2. 实验环境

Python3.7, Windows10, Spyder, Anaconda 3, VScode

三、 设计思想（本程序中的用到的主要算法及数据结构）

1. 算法原理

1.1 EM 算法

最大期望（EM）算法是一类通过迭代进行极大似然估计的算法，用于对包含隐变量或缺失数据的概率模型进行参数估计。EM 算法的迭代主要由 E 步和 M 步组成。E 步主要是利用隐变量的现有估计值计算最大似然估计值。M 步主要是通过最大化 E 步求得的最大似然估计值来计算参数，然后将得到的参数用到下一轮的 E 步中。如此迭代，直到最终收敛。

1.2 K-Means 算法

给定样本集 D 以及需要划分的聚类数量 k，给出一个划分 $C=\{C_1, C_2,\ldots,C_k\}$ 使得该划分的平方误差 E 最小，其中：

$$E=\sum_{i=1}^{k}\sum_{x\in C_i}\|x-\mu_i\|^2$$

$$\mu_i=\frac{1}{|C|}\sum_{x\in C_i} x$$

K-Means 迭代策略如下：

1) 初始化一组均值向量（随机或者采取某种特定策略）；
2) 根据初始化的均值向量给出样本集 D 的一个划分，其中每个样本点归到距离其最近的均值点的那一类当中。
3) 根据上一步的划分，重新计算出每个聚类的均值向量，在回到第一步迭代。
4) 当发现各个均值点几乎不发生明显变化时，说明已经收敛，结束迭代。

1.3 GMM 算法

在 n 维样本空间中的随机变量 x 服从高斯分布的密度函数为

$$\mathrm{p}(x \mid \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{D}{2}} \mid \Sigma \mid^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}$$

则混合高斯分布的定义：

$$\mathrm{p}_M = \sum_{i=1}^{k} \alpha_i p(x \mid \mu_i, \Sigma_i)$$

该分部由 k 类高斯分布混合而成。其中 μ_i，Σ_i 为第 i 个分布的均值和协方差矩阵，α_i 为混合系数，总和为 1。

先给定样本集 D，可通过极大似然估计确定参数 α_i，μ_i，Σ_i，$\mathrm{i} \in \{1, 2, ..., k\}$：

$$L(D) = \ln\left(\prod_{j=i}^{m} p_M(x_j)\right) = \sum_{j=1}^{m} \ln\left(\sum_{i=1}^{k} \alpha_i p(x_j \mid \mu_i, \Sigma_i)\right)$$

添加拉格朗日项：

$$L(D) + \lambda\left(\sum_{i=1}^{k} \alpha_{\mathrm{i}} - 1\right) = \sum_{j=1}^{m} \ln\left(\sum_{i=1}^{k} \alpha_i p(x_j \mid \mu_i, \Sigma_i)\right) + \lambda\left(\sum_{i=1}^{k} \alpha_{\mathrm{i}} - 1\right)$$

对 α_i，μ_i，Σ_i，$\mathrm{i} \in \{1, 2, ..., k\}$ 分别求导令导数为 0 得到

$$\mu_i = \frac{\sum_{j=1}^{m} p_M(z_j = i \mid x_j) x_j}{\sum_{j=1}^{m} p_M(z_j = i \mid x_j)}$$

$$\Sigma_i = \frac{\sum_{j=1}^{m} p_M(z_j = i \mid x_j)\left(\mathrm{x}_j - \mu_i\right)\left(\mathrm{x}_j - \mu_i\right)^T}{\sum_{j=1}^{m} p_M(z_j = i \mid x_j)}$$

$$\alpha_i = \frac{1}{\mathrm{m}} \sum_{j=1}^{m} \frac{p(x_j \mid \mu_i, \Sigma_i)}{\sum_{i=1}^{k} \alpha_i p(x \mid \mu_i, \Sigma_i)}$$

说明各混合成分的均值可以通过样本加权平均来估计，权重样本式每个样本属于该成分的后验概率，每个高斯成分的混合系数也由样本属于该成分的平均后验概率确定。

2. 算法的实现

2.1 K-Means 算法

首先随机确定 k 个中心点，然后进行迭代。每次迭代，先根据确定的中心点更新每个点的 label，然后在根据每一类 label 的点更新中心，如此迭代，知道所有中心点两次迭代之间的距离足够小，即出现收敛，则说明聚类完成。

```
def classify(self):
    # not random
    # self.centers = self.center_confirm()
    # random
    for i in range(self.k):
        self.centers[i, :] = self.data[np.random.randint(low=0, high=self.n), :]
    # iterate
    while 1:
        distance = np.zeros(self.k)
        amou = np.zeros(self.k)
        new_centers = np.zeros((self.k, self.dim))
        # update labels
        for i in range(self.n):
            for j in range(self.k):
                distance[j] = np.linalg.norm(self.data[i, :] - self.centers[j, :])
            new_label = np.argmin(distance)
            self.label[i] = new_label

        converge_counter = 0
        sum_bias = 0
        # update centers
        for i in range(self.n):
            label = int(self.label[i])
            new_centers[label] += self.data[i]
            amou[label] += 1
        for i in range(self.k):
            if amou[i] != 0:
                new_centers[i] /= amou[i]
            bias = np.linalg.norm(new_centers[i]-self.centers[i])
            sum_bias += bias
            if bias < self.e:
                converge_counter += 1
        print(sum_bias)
        if converge_counter == self.k:
            break
        else:
            self.centers = new_centers
    return self.label, self.centers
```

2.2 GMM 算法

2.2.1 E-Step

根据样本和各分布的估计参数确定各分布的后验概率。

```
def e_step(self):
    gamma = np.zeros((self.n, self.k))
    for i in range(self.n):
        for j in range(self.k):
            gamma[i][j] = self.py[j] * \
                multivariate_normal.pdf(
                    self.data[i], self.mu[j], self.sigma[j])
        margin_pro = np.sum(gamma[i])
        gamma[i] /= margin_pro
    self.gamma = gamma
```

2.2.2 M-Step

更新各分布的参数。

```
def m_step(self):
    for i in range(self.k):
        gamma_i = np.expand_dims(self.gamma[:, i], axis=1)
        gamma_sum = np.sum(gamma_i)
        self.py[i] = gamma_sum/self.n
        self.mu[i] = (self.data*gamma_i).sum(axis=0)/gamma_sum
        self.sigma[i] = np.dot(
            (self.data-self.mu[i]).T, (self.data-self.mu[i])*gamma_i)/gamma_sum
```

2.2.3 迭代

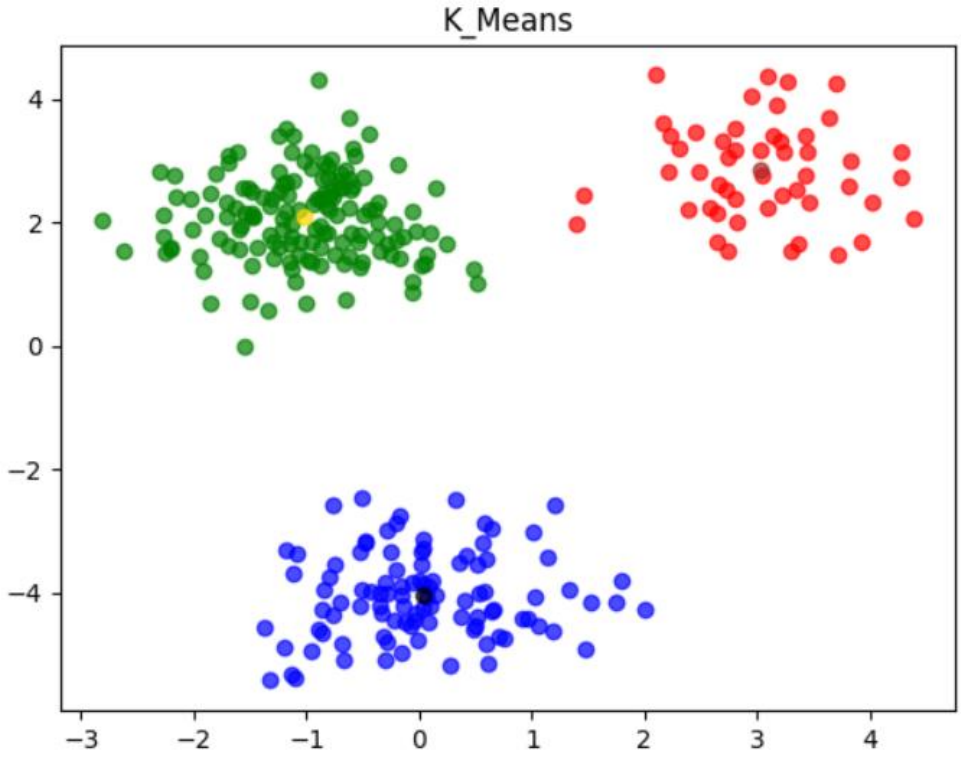
```
def classify(self):
    loss = []
    counter = []
    i = 0
    while 1:
        old_loss = self.likelihood()
        self.e_step()
        self.m_step()
        loss.append(abs(old_loss))
        counter.append(i)
        i += 1
        new_loss = self.likelihood()
        print(old_loss)
        # print(abs(new_loss - old_loss))
        if abs(new_loss - old_loss) < self.e:
            break
    self.labels = np.argmax(self.gamma, axis=1)
    return loss, counter
```

四、 实验结果与分析

1. K-Means 算法

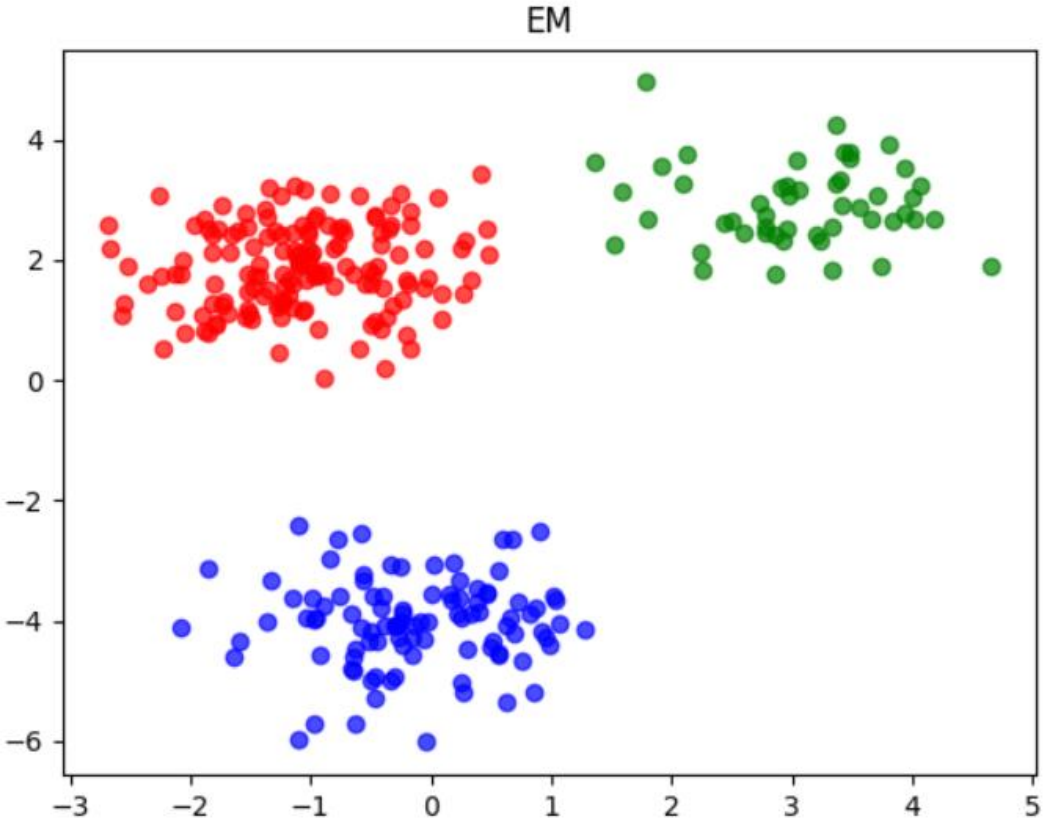
生成三个二维高斯分布的数据，均值分别为[3, 3], [0, -4], [-1, 2]，协方差矩阵都为[[0.5, 0.01], [0.01, 0.5]]。

图中 k-means 给出的结果中心点与所给样本的原均值很好地符合，对各个点的聚类表现也很好。

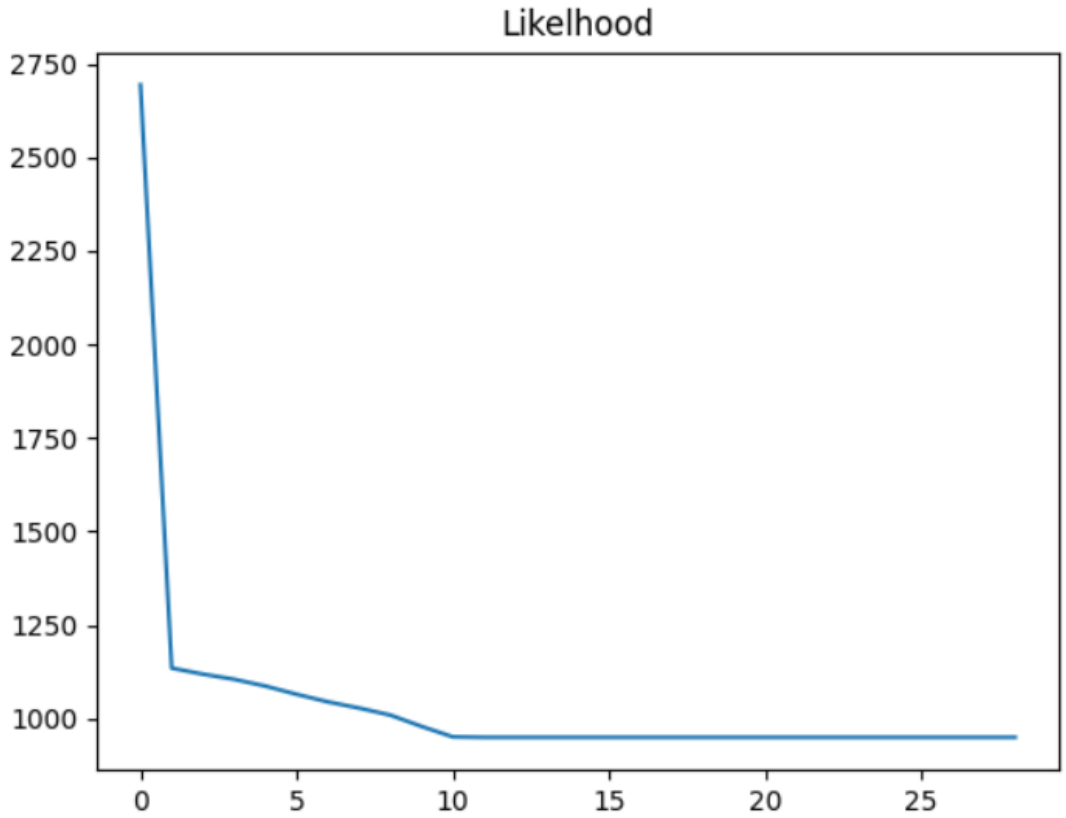


2. GMM 算法

聚类结果：



似然值在迭代过程中的变化：



最终得到的三个分布的均值和协方差矩阵：

```
means:  [[-0.02633694 -4.03200709]
 [-1.02557644  1.99139991]
 [ 3.17697153  2.96436246]]
cov:  [[[ 0.52318305  0.03068833]
  [ 0.03068833  0.48443315]]

 [[ 0.51506313 -0.02865405]
  [-0.02865405  0.57803766]]

 [[ 0.67459612 -0.02115003]
  [-0.02115003  0.34156883]]]
```

由以上结果可知，EM 算法的聚类效果也很好，得到的分布参数与实际样本的原参数也比较接近。

3. UCI 数据测试

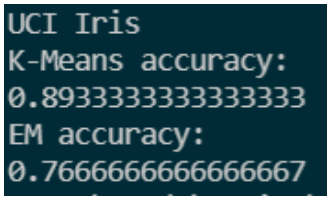
选取了 Iris 数据集进行测试，结果如下：

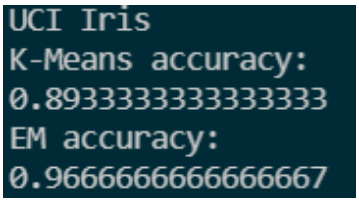
```
UCI Iris
K-Means accuracy:
0.8866666666666667
EM accuracy:
0.56
```

可见 k-means 算法的准确率较好，但是 GMM 模型下的 EM 算法性能较差，可能是 Iris 数据集的分布与 GMM 差异较大或者 EM 的初值选取不当导致的。

1. GMM 算法

在修改了 GMM 的初始均值后再次聚类，得到一下结果：

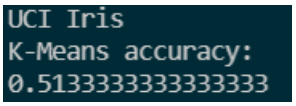




均值向量从原本的[1,2,…,k]分别变为[2,3,,…,k+1], [3,4,…,k+2]，可以发现准确率明显提高，尤其是后者准确率已经非常接近 1 了。由此可见，GMM 适用于 Iris 数据集的分布，且可以表现出很好的性能，要优于 k-means 算法，但其性能会明显收到初值的影响。

2. K-Means 算法

实验还发现有时 k-means 得到的结果性能也很差，由于每次计算只有初值不一样（随机生成），所以可以断定 k-means 算法的性能也明显收到初值的影响。



五、 结论

1. k-means 和 GMM 算法的性能都明显收到初始中心的影响，可以通过特定策略生成初始值或者利用随机生成多进行几次试验来得到较好的结果。
2. GMM 算法相对而言性能更好。可能的原因是 k-means 算法假设数据是呈球状均匀分布，而 GMM 算法假设数据呈现高斯分布，后者的假设相比之下更符合普遍规律。

六、 参考文献

- 周志华 著. 机器学习, 北京: 清华大学出版社, 2016.1
- Iris dataset: https://archive.ics.uci.edu/ml/datasets/Iris

七、 附录：源代码（带注释）

- 主程序见 lab3.py
- k-means 实现见 k_means.py
- GMM 实现见 em_gmm.py
- UCI 数据集见 iris.csv