哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2020 年春季学期

计算学部《机器学习》课程

Lab 3 实验报告

姓名	刘璟烁
学号	
班号	1803104
电子邮件	
手机号码	

目录

一、实验目的

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

二、实验要求及实验环境

实验要求

1.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)

用 k-means 聚类测试效果

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结果

2.应用

可以在 UCI 上找一个简单问题数据，用你实现的 GMM 进行聚类。

实验环境

操作系统：MacOS 10.15.7

python 版本：Python 3.8.3

三、设计思想及算法实现

1.算法原理

1.1 k-means 算法

对于一个无监督学习聚类任务，即给定训练样本集合 D = {x_1,x_2,x_3,...,x_n}，以及样本集的类别个数 k，希望得到对样本集合 D 的一个划分 C，其中 C = {C_1，C_2,...,C_k}为各个类的集合。 k-means 算法以样本点见的欧式距离为度量，使得划分在同一类中的点的欧式距离之和尽可能小，具体的，通过求解下述最优化问题得到训练结果：

$$\min_{\mu}\min_{C} F(\mu, C) = \min_{\mu}\min_{C}\sum_{i=1}^{k}\sum_{j:C(j)=i}||\mu_i - x_j||^2$$

其中，μ_i为第 i 类样本的均值。求解该式是一个 NP 难问题，k-means 算法使用了求近似最优解的方法，通过循环迭代，求出该最优化问题的最优解。具体的，k-means 算法首先随机初始化 k 个中心点作为 k 个类的初始均值μ,随后开始如下迭代：

1.将簇划分 C 初始化为 C_t=∅，t=1,2...k

2.对于 i=1,2...m,计算样本 x_i（i = 1,2,...n）和各个均值μ_j(j=1,2,...k)的距离：

$$d_{ij} = ||x_i - \mu_j||_2^2$$

,将 x_j 标记最小的为 d_{ij} 所对应的类别λ_i。此时更新$C_{\lambda_i} = C_{\lambda_i} \cup \{x_i\}$

3.对于 j=1,2,...,k,对 C_j 中所有的样本点重新计算新均值:

$$\mu_j = \frac{1}{C_j}\sum_{\mathbf{x}\in C_j}\mathbf{x}$$

4.如果所有的 k 个均值都没有发生变化，则停止循环

1.2 GMM 模型与 EM 算法

首先介绍 GMM（Gaussian Mixture Model）的基本原理，GMM 模型用高斯概率密度函数(正态分布曲线)精确地量化事物，将一个事物分解为若干的基于高斯概率密度函数（正态分布曲线）形成的模型，具体的，对于对于一个样本集合 $D = \{x_1, x_2, x_3, ..., x_n\}$，假设其中的样本属于多个类，且每个类中的样本符合该类具有的特定参数的高斯分布，例如第 k 类样本的概率密度函数为：

$$P(x|\mu_k, \Sigma_k) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|_k^{\frac{1}{2}}} \exp(-\frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k))$$

那么对于这个由 k 个类的分布混合成的样本集，GMM 希望得到总样本集上的概率密度函数：

$$p(x_j) = \sum_{i=1}^{k} \alpha_i p(x_j|\mu_i, \Sigma_i)$$

其中 α_i 为每个类相应的混合系数，相当于该类的先验概率，满足 $\sum_{i=1}^{k} \alpha_i = 1$；$\mu_i$，$\Sigma_i$ 为第 i 个高斯分布的均值和协方差矩阵。

认为样本集由混合高斯分布给出，并且假设每个类的先验概率 α_i 为任一个样本属于类别 i 的概率，而每个类的样本均满足特定的高斯分布。则给定一个样本 x_j，该样本属于第 i 类（也就是该样本由第 i 类生成的）的后验概率为：

$$\gamma_{ji} = P(z_j = i|x_j) = \frac{p(z_j = i)p(x_j|z_j = i)}{p(x_j)} = \frac{\alpha_i p(x_j|\mu_i, \Sigma_i)}{\sum_{l=1}^{k} \alpha_l p(x_j|\mu_l, \Sigma_l)}$$

求得该公式后，可以通过寻找后验概率最大的类将任何一个样本划分到类集合 C = {C_1，C_2,...,C_k}中的某一个类中，求出该后验概率公式需要首先求解其中的参数α、μ、Σ，于是用到下述极大似然估计方法:

$$lnL(\mu,\Sigma,\alpha)=\sum_{j=1}^{N}ln\sum_{l=1}^{k}\alpha_l N(x_j|\mu_l,\Sigma_l)$$

关于上式，分别对μ、Σ求偏导为 0 并进行化简得到相应方程：

$$\mu_i=\frac{\sum_{j=1}^{N}\gamma_{ji}x_j}{\sum_{j=1}^{N}\gamma_{ji}}$$

以及：

$$\Sigma_i=\frac{\sum_{j=1}^{N}\gamma_{ji}(x_j-\mu_i)(x_j-\mu_i)^T}{\sum_{j=1}^{N}\gamma_{ji}}$$

另外由于各高斯分布对应的权重α满足一定的条件约束，即非负、累加和为 1，因此对数似然函数可以利用拉格朗日乘数法求得关于α的方程：

$$lnL(\mu,\Sigma,\alpha)=\sum_{j=1}^{N}ln\sum_{l=1}^{k}\alpha_l N(x_j|\mu_l,\Sigma_l)+\lambda\sum_{l=1}^{k}\alpha_l$$

令其关于α的偏导得 0 并作化简可得：

$$\alpha_i=\frac{1}{N}\sum_{j=1}^{N}\frac{\gamma_{ji}}{\sum_{l=1}^{k}\alpha_l\gamma_{ji}}$$

有上述公式可以看出，混合高斯分布中某一类的均值为以样本属于该类的后验概率加权的平均值，而某类的混合系数为样本属于该类的平均后验概率。

GMM 中求解上述似然方程式的主要思想为 EM 似然估计算法，在统计问题中被用于寻找依赖于不可观察的隐性变量的概率模型中，用于求取参数的最大似然估计。EM 算法经常用在机器学习和计算机视觉得数据聚类领域，它的标准计算框架由 E 步（expectation）和 M 步(maximization)交替组成，算法的收敛可以确保迭代至少逼近局部极大值。具体的，通过如下迭代即可完成：

1、初始化分布参数

2、E-step：根据参数计算每个样本属于的概率(也就是我们的 Q)

3、M-step：根据 Q，求出含有的似然函数的下界并最大化它，得到新的参数

4、不断地迭代更新下去，直到收敛

2.算法实现

1.k-means 算法实现：

根据上述算法原理，给定输入样本集 $D = \{x_1, x_2, x_3, ..., x_n\}$，以及样本集的类别个数 k，输出一个对样本集合的划分 $C = \{C_1, C_2, ..., C_k\}$，则实现 k-means 算法有如下伪代码：

1）从数据集 D 中随机选择 k 个样本作为初始的 k 个均值：$\mu_i \{i = 1,2,...,k\}$

2）对于 n=1,2,...,N

a) 将簇划分 C 初始化为 $C_t=\varnothing$，t=1,2...k

b) 对于 i=1,2...m,计算样本 x_i（i = 1,2,...n）和各个均值μ_j(j=1,2,...k)的距离：

$$d_{ij} = ||x_i - \mu_j||_2^2$$

,将 x_j 标记最小的为 d_{ij} 所对应的类别λ_i。此时更新$C_{\lambda_i} = C_{\lambda_i} \cup \{x_i\}$

c) 对于 j=1,2,...,k,对 C_j 中所有的样本点重新计算新均值:

$$\mu_j = \frac{1}{C_j} \sum_{\mathbf{x} \in C_j} \mathbf{x}$$

e) 如果所有的 k 个均值都没有发生变化，则转到步骤 3）

3） 输出簇划分 $C=\{C_1, C_2, ...C_k\}$

具体实现中，可以通过从样本集中随机选取 k 个样本点作为初始均值：

```
means = []
for i in range(k):
    means.append(X_train.T[(sam_size//k)*i])
```

K-means 算法每一次迭代的距离计算以及样本划分（对应 EM 算法中 E 步的思想）：

```
for i in range(k):
    Cla[i].clear()
for sam_point in X_train.T:  # 选一个样本点
    min_dis = (sam_point[0] - means[0][0])**2 + (sam_point[1] - \
means[0][1])**2
    min_cla = 0
    for j in range(k):  # 关于该样本点，对每一个类均值进行相应距离的计算
        this_dis = (sam_point[0] - means[j][0])**2 + (sam_point[1] -\
means[j][1])**2
        if this_dis < min_dis:  # 如果该类距离小于之前最小距离，更新
            min_dis = this_dis
            min_cla = j
    Cla[min_cla].append(sam_point)
```

随后进行该次迭代中均值的更新（对应 EM 算法中 M 步的思想）：

```
means_changed = False
for i in range(k):
```

```
    # 计算 k 类均值:
    this_mean = np.zeros(X_train.shape[0],dtype=float)
    for point in Cla[i]:
        this_mean += point
    this_mean /= len(Cla[i])
    if not (np.array_equal(this_mean,means[i])):
        means_changed = True
        means[i] = this_mean
if not means_changed:
    break
```

2.Gaussian 算法实现：

给定输入样本集 D = {x_1,x_2,x_3,...,x_n}，以及样本集的类别个数 k，输出一个对样本集合的划分 C = {C1，C2,...,Ck}，则实现 GMM 算法有如下伪代码：

1). 随机初始化参数α、μ、Σ,并将簇划分 C 初始化为 C_t=∅ （t = 1,2,...,k）

2). 开始迭代至带到迭代次数或者是参数值不再发生变化：

a). 计算每个样本由各个混合高斯成分（即γ_{ji}）生成的后验概率 ，即 EM 算法中的 E 步

b). 根据 E 步确定的后验概率γ,利用上述原理中的公式更新参数α、μ、Σ，即 EM 算法的 M 步

3). 根据最终的后验概率γ将每个样本划分到特定的类别 C_t 中

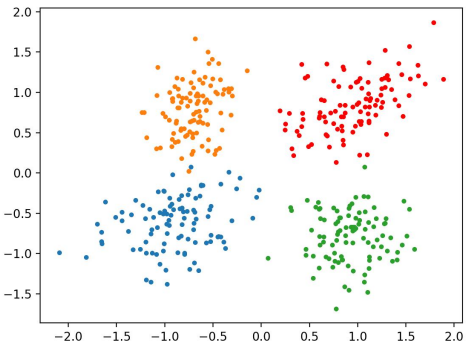
4） 输出簇划分 $C=\{C_1,C_2,...C_k\}$

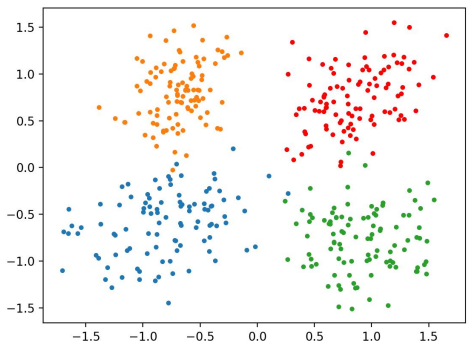
具体的，我们可以通过调用先前实现的 k-means 算法生成的均值作为 GMM 算法的初始值，这样可以一定程度上提高收敛效果。

四、实验结果分析

1.数据生成

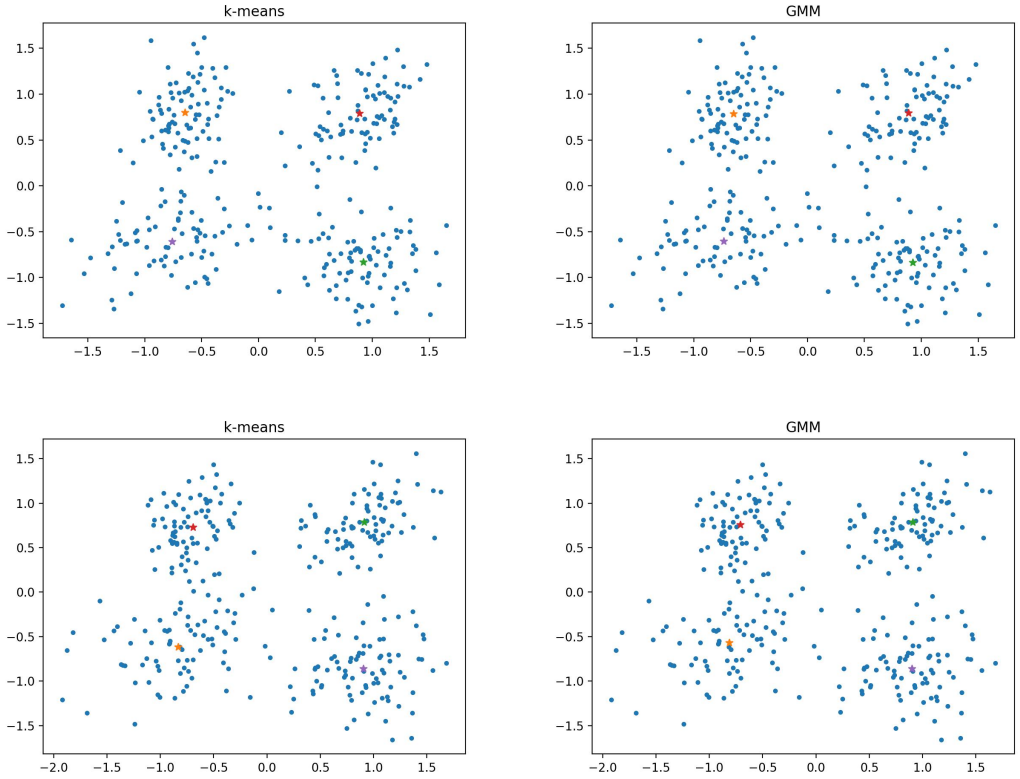
本次实验手动数据生成部分设定类的个数 k = 4， 并且为了直观展示两个算法的聚类效果，我们不妨设定特征属性的维度为 2，随机生成满足高斯分布的样本集，并且样本各特征属性之间不一定满足朴素贝叶斯分布，具体代码请见 gene_samples.py，如下为两次随机生成的样本点：





2.k-means、GMM 在生成数据集上的聚类效果

对于上面手动生成的数据集合，我们分别用 k-means 和 GMM 模型给出聚类得到的均值：



上面两组图分别为两次随机生成的数据集上 k-means 和 GMM 生成的结果均值，其中各类中心用星号在图中标出，下面为其数值表示：

```
result of k-means:
[[-0.64549674  0.79798249]
 [ 0.92008525 -0.83258181]
 [ 0.88603427  0.79242378]
 [-0.75745787 -0.60637208]]
```

```
result of GMM:
[[-0.64950582  0.78722504]
 [ 0.9256927  -0.83358637]
 [ 0.88775937  0.79409861]
 [-0.73740903 -0.60517702]]
```

```
result of k-means:
[[-0.8320002  -0.61404851]
 [ 0.91018967  0.78866289]
 [-0.69443757  0.72868027]
 [ 0.90547777 -0.85958325]]
```

```
result of GMM:
[[-0.81343925 -0.56695091]
 [ 0.90816296  0.78755651]
 [-0.70983507  0.75775259]
 [ 0.89767751 -0.86082036]]
```

可以看到 k-means 和 GMM 产生的结果相差不大，根据 Gene_samples.py 中设定的协方差，可以看出二者均比较接近正确结果。

3.k-means、GMM 在 UCI 数据集上的聚类效果

3.1 在 iris 数据集上的聚类效果

导入数据集 iris，它有三个类别，我们分别调用 k-means 和 GMM 对其进行聚类并评判正确率，具体的正确率判断方法为为原始样本的每个类别生成一个标签，在聚类完成后，按照标签的所有排列组合计算分类正确率，取正确率最高的那一组合的正确率为聚类正确率，运行可得：

```
the GMM's accuracy for iris is :0.9333333333333333
the k-means' accuracy for iris is :0.8866666666666667
```

3.2 在 seeds 数据集上的聚类效果

类似前者，导入数据集 seeds，我们不妨取其中两个类别的数据进行判断，调用 k-means 和 GMM 对其进行聚类并评判正确率，正确率判断方法如上，运行结果如下：

```
the GMM's accuracy for seeds is :0.95
the k-means' accuracy for seeds is :0.9214285714285714
```

可以看到 GMM 效果相对略有优势。

五、结论

根据上述实验结果以及算法原理的分析，我们可以得出以下结论：

1. k-means 使用欧式距离度量样本与各个类别中心的相似度，为硬聚类；而 GMM 模型使用后验概率判断样本所属类别并聚类，为软聚类。

2. 两个模型均为迭代执行的算法，且迭代的策略也相同：算法开始执行时先对需要计算的参数赋初值，然后交替执行两个步骤，一个步骤是对数据的估计（k-means 是估计每个点所属簇；GMM 是计算隐含变量的期望；）;第二步是用上一步算出的估计值重新计算参数值，更新目标参数（k-means 是计算簇心位置；GMM 是计算各个高斯分布的中心位置和协方差矩阵）。

3.k-means 算法与 GMM 均应用了 EM 算法的思想。

六、参考资料

吴恩达：机器学习

Christopher Bishop. Pattern Recognition and Machine Learning.

周志华：机器学习

七、源代码

1.main.py:

该部分为 client 端，可以分别通过调用 hand_gene_samples()、test_on_iris()、test_on_seeds()方法观察两种模型分别在三种数据集上的效果：

```
import Gaussian_mix
import gene_samples
```

```
import matplotlib.pyplot as plt
import read_data



def hand_gene_samples():
    """
    手动生成数据并观测 k-means 和 GMM 结果
    :return:
    """
    k = 4
    X_train,X_test = gene_samples.gene_sam()
    means_1,y = Gaussian_mix.gaussian_mix(X_train,k)    # (k,m)
    means_2 = k_means.k_means_func(X_train,k)
    print("result of k-means: ")
    print(means_2)
    print()
    print("result of GMM: ")
    print(means_1)
    print()
    plt.figure(1)
    plt.scatter(X_train[0,:],X_train[1,:],marker = '.')
    for i in range(k):
```

```
            plt.scatter(means_1[i][0],means_1[i][1],marker='*')
        plt.title("GMM")
        plt.figure(2)
        plt.scatter(X_train[0,:],X_train[1,:],marker = '.')
        for i in range(k):
            plt.scatter(means_2[i][0],means_2[i][1],marker='*')
        plt.title("k-means")
        plt.show()
        return



def test_on_iris():
        X, Y = read_data.iris_data_load()   # 使用 iris 做训练集
        means,y = Gaussian_mix.gaussian_mix(X,3 )   # (k,m)
        print("the GMM's accuracy for iris is :"+
str(Gaussian_mix.calc_acc(means,X,y,Y)))
        means_2 = k_means.k_means_func(X,3)
        print("the k-means' accuracy for iris is :"+
str(k_means.calc_acc(means_2,X,Y)))
        return


def test_on_seeds():
```

```
    X,Y =   read_data.seeds_data_load()   # 使用 seeds 作训练集
    means,y = Gaussian_mix.gaussian_mix(X,2 )   # (k,m)
    print("the GMM's accuracy for seeds is :"+
str(Gaussian_mix.calc_acc(means,X,y,Y)))
    means_2 = k_means.k_means_func(X, 2)
    print("the k-means' accuracy for seeds is :" +
str(k_means.calc_acc(means_2, X, Y)))
    return


# hand_gene_samples()
# test_on_iris()
test_on_seeds()
```

2.gene_samples.py

随机生成多类满足高斯分布的数据集：

```
import numpy as np
import sklearn.model_selection
import matplotlib.pyplot as plt


def gene_sam(alpha = [0.25,0.25,0.25,0.25]):
    """
    生成 k（自行确定）个满足高斯分布的类数据，每个类有独立的均值和方差，
```

```
并且根据
    :param alpha: 各类的成分系数
    :return:X_train,X_test with dim (2,n_train) 、 (2，n_test)
    """
    total_num = 400
    #means
    mean_CLass_1 = [0.9,0.8]
    mean_CLass_2 = [-0.8,-0.6]
    mean_CLass_3 = [-0.7,0.8]
    mean_CLass_4 = [0.9,-0.8]
    #covariances
    cov_Cla_1 = 0.05
    cov_Cla_2 = 0.04
    cov_Cla_3 = 0.02
    cov_Cla_4 = 0
    #generate X with (m,n) dim,where m = 2 n is the number of the
samples
    sam_C1 = np.random.multivariate_normal\
        (mean = mean_CLass_1,cov
=[[0.11,cov_Cla_1],[cov_Cla_1,0.12]],\
         size = int(total_num * alpha[0])).T
    sam_C2 = np.random.multivariate_normal \
```

```
        (mean=mean_CLass_2, cov=[[0.14, cov_Cla_2], [cov_Cla_2,
0.1]], \
         size=int(total_num * alpha[1])).T
    sam_C3 = np.random.multivariate_normal \
        (mean=mean_CLass_3, cov=[[0.05, cov_Cla_3], [cov_Cla_3,
0.1]], \
         size=int(total_num * alpha[2])).T
    sam_C4 = np.random.multivariate_normal \
        (mean=mean_CLass_4, cov=[[0.12, cov_Cla_4], [cov_Cla_4,
0.1]], \
         size=int(total_num * alpha[3])).T
    X = np.c_[sam_C1,sam_C2]
    X = np.c_[X,sam_C3]
    X = np.c_[X,sam_C4]
    X_train,X_test =
sklearn.model_selection.train_test_split(X.T,test_size= 0.2)

    return X_train.T,X_test.T
```

3.k-means.py

实现 k-means 算法：

```
import numpy as np
import itertools


def calc_acc(mean,X,T):
    """
    计算准确率
    :param mean: (k,m)
    :param X: (m,n)
    :param T: (n,1)
    :return:
    """
    samp_num = X.shape[1]
    k = mean.shape[0]  # 类别数
    all_perm = list(itertools.permutations(range(k)))
    all_acc = []
    y = []
    for i in range (samp_num):  # 求得每个样本的分类
        min_dis = np.linalg.norm(X[:,i] - mean[0])
        min_ind = 0
        for j in range(1,k):
            dis = np.linalg.norm(X[:,i] - mean[j])
            if dis < min_dis:
```

```
                min_dis = dis
                min_ind = j
        y.append(min_ind)

    for a_perm in all_perm:
        count = 0
        for i in range(samp_num):
            if a_perm[y[i]] == T[i,0]:
                count += 1
        all_acc.append(count)
    return np.max(all_acc) * 1.0 / samp_num


def k_means_func(X_train,k,random_init = True):
    """
    通过 k-means 无监督学习出聚类模型
    :param X_train: training data
    :param k: number of classes
    :param random_init: whether give a random init parameter or not
    :return: (k,m) np array
    """
    sam_size = X_train.shape[1]
```

```
    #initialization
    if random_init:
        means = []
        for i in range(k):
            means.append(X_train.T[(sam_size//k)*i])

    # 为每个类提供存放该类样本点的容器（集合构成的列表）
    Cla = []
    for i in range(k):
        Cla.append(list()) # 生成 k 个类

    # 计算每个点到四个类中心的距离
    while True:
        for i in range(k):
            Cla[i].clear()
        for sam_point in X_train.T:
            min_dis = (sam_point[0] - means[0][0])**2 + (sam_point[1]
- means[0][1])**2
            min_cla = 0
            for j in range(k):
                this_dis = (sam_point[0] - means[j][0])**2 +
(sam_point[1] - means[j][1])**2
```

```
                if this_dis < min_dis:
                    min_dis = this_dis
                    min_cla = j
            Cla[min_cla].append(sam_point)

        # 重新计算中心
        means_changed = False
        for i in range(k):
            # 计算 k 类均值：
            this_mean = np.zeros(X_train.shape[0],dtype=float)
            for point in Cla[i]:
                this_mean += point
            this_mean /= len(Cla[i])
            if not (np.array_equal(this_mean,means[i])):
                means_changed = True
                means[i] = this_mean
        if not means_changed:
            break

    return np.array(means)
```

4.Gaussian_mix.py

实现 GMM 模型：

```
mport numpy as np
import k_means
import itertools



def calc_acc(mean,X,y,T):
    """
    计算准确率
    :param mean: (k,m)
    :param X: (m,n)
    :param T: (n,1)
    :return:
    """
    samp_num = X.shape[1]
    k = mean.shape[0]  # 类别数
    all_perm = list(itertools.permutations(range(k)))
    all_acc = []
    print(T)
    print(y)
    for a_perm in all_perm:
        count = 0
```

```
            for i in range(samp_num):
                if a_perm[y[i]] == T[i,0]:
                    count += 1
            all_acc.append(count)
        return np.max(all_acc) * 1.0 / samp_num




def random_get_sigma(k,m):
    a = []
    for i in range(k):
        X = np.random.rand(m,m)
        X = np.diag(X.diagonal())
        a.append(X)
    return np.array(a)


def normal_prob(x, mu, sigma):
    """
    给定 x，高斯分布的参数，计算概率密度（pdf 的造轮子实现）
    :param x: (m,1)
```

```
    :param mu: array with (m,)
    :param sigma: (m,m)
    :return:
    """
    m = x.shape[0]
    sigma_inv = np.linalg.inv(sigma)
    x = np.array([x]).T  # 将 x、mu 转为列向量的标准格式
    mu_vec = np.array([mu]).T
    frac_s = np.exp(-1 / 2 * (x - mu_vec).T.dot(sigma_inv).dot(x -
mu_vec))  # 求出概率密度函数的分子
    frac_m = (2 * np.pi) ** (m / 2) * np.linalg.det(sigma) ** (1 / 2)  # 求
出概率密度的分母

    return frac_s / frac_m


def calc_likelihood(X, mu, sigma, alpha):
    """
    计算给定数据和参数的似然值
    :param X: (m,n) 需要计算似然函数值的训练集
    :param mu: （k,m）各个类的均值构成的矩阵
    :param sigma: (k,m,m) 各类协方差矩阵构成的张量
```

```
    :param alpha: (1,k) 各类分布概率构成的行向量
    :return: 给定参数与数据对应的似然函数值
    """
    samp_num = X.shape[1]  # 记录样本数
    k = alpha.shape[1]  # 记录类数
    total_sum = 0  # 存储返回结果的变量
    for i in range(samp_num):
        cla_sum = 0
        for l in range(k):
            cla_sum += alpha[0][l] * normal_prob(X[:, i], mu[l], sigma[l])
        total_sum += np.log(cla_sum)  # 对每个样本的分布概率密度的对数进行累加
    return total_sum


def gaussian_mix(X_train, k, delta=1e-13):
    """
    Gaussian Mixture Model
    :param X_train: (m,n) matrix,where m is the dim of a sample's vector
    :param k: the number of classes
    :param delta: used to justify whether should terminate the loop
    :return:
```

```
    """
    size = X_train.shape[1]  # 样本数
    m = X_train.shape[0]  # 特征维度
    alpha = np.random.dirichlet(np.ones(k), size=1)  # 初始化 alpha ，
(1,4) np array
    mu = k_means.k_means_func(X_train, k)  # 采用 k-means 进行初始
化  (k,m)
    sigma = random_get_sigma(k,m)

    while True:
        gamma = np.zeros((size, k),dtype=float)  # calculate gamma
with current parameters:the expectation of EM
        likelihood = calc_likelihood(X_train, mu, sigma, alpha)
        print("likelihood = " + str(float(likelihood)))
        for i in range(size):  # calc E pace
            x = X_train[:, i]
            sum = 0  # calculate the probability of sample j
            for l in range(k):
                sum += alpha[0][l] * normal_prob(x, mu[l], sigma[l])
            for j in range(k):
                gamma[i][j] = alpha[0][j] * normal_prob(x, mu[j],
sigma[j]) / sum
```

```
        new_alpha = np.zeros(alpha.shape,dtype=float)
        new_mu = np.zeros(mu.shape, dtype=float)
        new_sigma = np.zeros(sigma.shape,dtype=float)
        for i in range(k):   # calc M pace
            sum_of_gamma = 0
            for j in range(size):
                sum_of_gamma += gamma[j][i]
            new_alpha[0][i] = sum_of_gamma / size

            sum_frac_son_mu = np.zeros((1, m),dtype=float)
            for j in range(size):
                sum_frac_son_mu += gamma[j][i] * X_train[:, j].T
            new_mu[i] = sum_frac_son_mu / sum_of_gamma

            sum_frac_son_sigma = np.zeros((m, m),dtype=float)
            for j in range(size):
                x = np.array([X_train[:,j]]).T
                mu_d = np.array([new_mu[i]]).T
                sum_frac_son_sigma += gamma[j][i] * np.dot(x-mu_d ,
(x-mu_d).T)
            new_sigma[i] = sum_frac_son_sigma / sum_of_gamma
```

```
            this_delt = np.linalg.norm(new_alpha - alpha) +
np.linalg.norm(new_mu - mu) + \
                        np.sum([np.linalg.norm(new_sigma[i] - sigma[i]) for
i in range(k)])
            if this_delt < delta:  # 如果变化量小于设定阈值，停止迭代
                break
            else:  # 否则更新 M 步得到的参数
                alpha = new_alpha
                mu = new_mu
                sigma = new_sigma


        y = []
        for i in range(size):  # 重新计算最终的 gamma
            x = X_train[:, i]
            min_prob = 0
            min_ind = 0
            sum = 0  # calculate the probability of sample j
            for l in range(k):
                sum += alpha[0][l] * normal_prob(x, mu[l], sigma[l])
            for j in range(k):
                gamma[i][j] = alpha[0][j] * normal_prob(x, mu[j], sigma[j]) /
sum
```

```
            if gamma[i][j] > min_prob:
                min_prob = gamma[i][j]
                min_ind = j
        y.append(min_ind)
    return mu,y    # 返回各个 聚类中心
```