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： GMM 模型

学号：1190200610
姓名： 张景阳

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

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）。

三、设计思想（本程序中的用到的主要算法及数据结构）

1. k-means 算法

原理：先随机选择 K 个质心，根据样本到质心的距离将样本分配到最近的簇中，然后根据簇中的样本更新质心，再次计算距离重新分配簇，直到质心不再发生变化，迭代结束。采用欧几里得距离，则一个簇中所有样本点到质心的距离的平方和。追求能够让簇内平方和最小化的质心

2. 高斯混合模型 GMM

1）高斯混合模型就是对高斯模型进行简单的扩展，GMM 使用多个高斯分布（而不是一个）的组合来刻画数据分布。GMM 同 K-means 算法一样也使用了 EM 算法进行迭代计算。GMM 假设每个簇的数据都是符合高斯分布的，当前数据呈现的分布就是各个簇的高斯分布叠加在一起的结果，即：

$$P(y|\theta)=\sum_{k=1}^{K}\alpha_k\phi(y|\theta_k)$$

其中：

$$\sum_{k=1}^{K}\alpha_k=1,\alpha_k\geq 0$$

$$\phi(y|\theta_k)=\frac{1}{\sqrt{2\pi}\sigma_k}\exp\left(-\frac{(y-\mu_k)^2}{2\sigma_k^2}\right),\theta_k=\left(\mu_k,\sigma_k^2\right)$$

一般情况下我们不能直接求解 GMM 的参数，而是在观察一系列数据的基础之上给出类别数量 K，希望求得最佳的 K 个高斯分布模型。此时问题转化为求得最佳的 K 个高斯分布模型的均值 u，方差 σ，以及混合权重 α 的求解。

通常这类问题可以通过最大似然估计来求解。遗憾的是，高斯混合模型如果直接使用最大似然估计将会得到一个复杂的非凸函数，其目标函数是和的对数，难以展开和对其求偏导。因此这里引出我我们的 EM 算法。在 GMM 中，我们使用 EM 算法来逐渐逼近最优解。下面我们就介绍 EM 算法。

2）EM 算法：从本质上说，EM 算法是最大似然估计的进阶版。对于上面我们提到的 GMM 中，我们需要直到每个类别中各个高斯分布所占的权值，进而我们可以抽象出这样一个模型：

假设我们有一个样本集 X 它是由 m 个样本构成的，可以写成 X={x1,x2…xm}，对于这 m 个样本当中，它们都有一个隐变量 z 是未知的。并且还有一个参数，就是我们希望通过极大似估计求解的参数。由于其中包含隐变量 z，所以我们没办法直接对概率函数求导求极值进行计算。

我们先写出含有隐变量的概率函数：

$$P_i = P(x_i, z_i; \theta)$$

我们希望找到对于全局最优的参数，所以我们希望找到使得最大，我们对这个式子求 log，可以得到（对数似然）：

$$\sum_{i=1}^{m} \log P_i = \sum_{i=1}^{m} \log \sum_{z_i} P(x_i, z_i; \theta)$$

我们假设隐变量 z 的概率分布是 Qi，所以上式可以变形为：

$$\sum_{i=1}^{m} \log P_i = \sum_{i=1}^{m} \log \sum_{z_i} Q_i(z_i) \frac{P(x_i, z_i; \theta)}{Q_i(z_i)}$$

接下来我们通过 Jensen 不等式：E[f(x)] >= f(E[x])，即函数的期望值大于等于期望值的函数值。而对数函数是广义上的凸函数，严格意义上的凹函数，它可以使用 Jensen 不等式，但是不等号的方向需要变号。

上式代入 Jensen 不等式可以得到：

$$\sum_{i=1}^{m} \log P_i \geq \sum_{i=1}^{m} \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i; \theta)}{Q_i(z_i)}$$

当我们固定 z 变量的时候，我们可以很方便的求解似然最大时的参数。同理，有了参数的取值之后，又可以来优化 z。

根据 Jensen 不等式，只有当自变量 x 时常数的时候才可以取等号，我们令：

$$\frac{P(x_i, z_i, \theta)}{Q_i(z_i)} = c$$

根据全概率公式我们可知：

$$\sum_{z_i} Q_i(z_i) = 1$$

因此：

$$\sum_{z_i} P(x_i, z_i, \theta) = c$$

整合一下，带入上式，我们可以得到：

$$Q_i(z_i) \cdot c = P(x_i, z_i, \theta)$$
$$Q_i(z_i) = \frac{P(x_i . z_i; \theta)}{c}$$
$$Q_i(z_i) = \frac{P(x_i . z_i; \theta)}{\sum_{z_i} P(x_i, z_i; \theta)}$$
$$Q_i(z_i) = \frac{P(x_i . z_i; \theta)}{P(x_i; \theta)}$$
$$Q_i(z_i) = P(z_i | x_i; \theta)$$

经过这一串变形之后，我们得到了计算公式其实是一个后验概率。这一步也就是我们刚才介绍的 E 步，之后再确定了之后，我们来求导求极限的方法求出函数最大时的参数，就是 M 步

整个 EM 算法就是重复上述过程直到收敛

3）再看 GMM

我们假设观测变量 yj 中来自第个模型的分量为隐变量，记为 γ jk。γ jk 是第 j 个观测数据来自第 k 个模型的概率，被称第 k 个分模型对观测数据 yj 的响应度并且满足：

$$n_k = \sum_{j=1}^{N} \gamma_{jk}$$
$$\sum_{k=1}^{K} n_k = N$$

其中 N 为样本数目，K 为分模型的树木。此时对视似然函数可以写为：

$$\log P(y, \gamma | \theta) = \sum_{k=1}^{K} n_k \log \alpha_k + \sum_{j=1}^{N} \gamma_{jk} \left[\log\left(\frac{1}{\sqrt{2\pi}}\right) - \log \sigma_k - \frac{1}{2\sigma_k^2}\left(y_j - \mu_k\right)^2 \right]$$

接着我们再用 EM 算法中的 E 步，求 Q 函数：

$$Q\left(\theta, \theta^{(j)}\right) = E\left[\log P(y, \gamma | \theta) | y, \theta^{(j)}\right]$$
$$= \sum_{k=1}^{K} \left\{ \sum_{j=1}^{N} \left(E\gamma_{jk}\right) \log \alpha_k + \sum_{j=1}^{N} \left(E\gamma_{jk}\right) \left[\log\left(\frac{1}{\sqrt{2\pi}}\right) - \log \sigma_k - \frac{1}{2\sigma_k^2}\left(y_j - \mu_k\right)^2 \right] \right\}$$

其中

$$\hat{\gamma}_{jk}=\frac{\alpha_k\phi\left(y_j\mid\theta_k\right)}{\sum_{k=1}^{K}\alpha_k\phi\left(y_j\mid\theta_k\right)}$$

EM 算法中的 M 步，极大化 Q 函数

$$Q\left(\theta,\theta^{(j)}\right)=\sum_{k=1}^{K}n_k\log\alpha_k+\sum_{j=1}^{N}\gamma_{jk}\left[\log\left(\frac{1}{\sqrt{2\pi}}\right)-\log\sigma_k-\frac{1}{2\sigma_k^2}\left(y_j-\mu_k\right)^2\right]\}$$

并且分别对 μ k, σ k, α k 求偏导，并令偏导等于 0，可以求得

$$\hat{\mu}_k=\frac{\sum_{j=1}^{N}\hat{\gamma}_{jk}y_j}{\sum_{j=1}^{N}\hat{\gamma}_{jk}}\qquad\hat{\sigma}_k=\frac{\sum_{j=1}^{N}\hat{\gamma}_{jk}\left(y_j-\mu_k\right)^2}{\sum_{j=1}^{N}\hat{\gamma}_{jk}}\qquad\hat{\alpha}_k=\frac{\sum_{j=1}^{N}\hat{\gamma}_{jk}}{N}$$

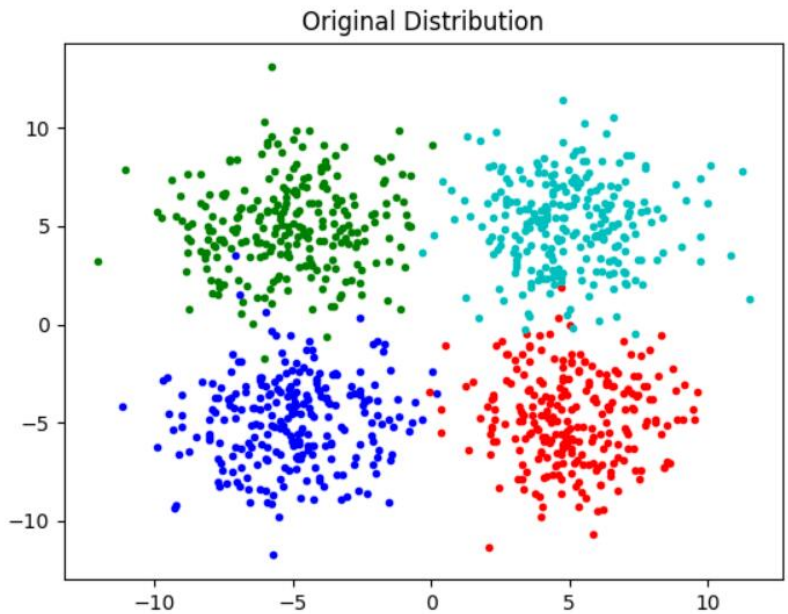
不断重复 E 步和 M 步直到对数似然函数收敛即完成训练。

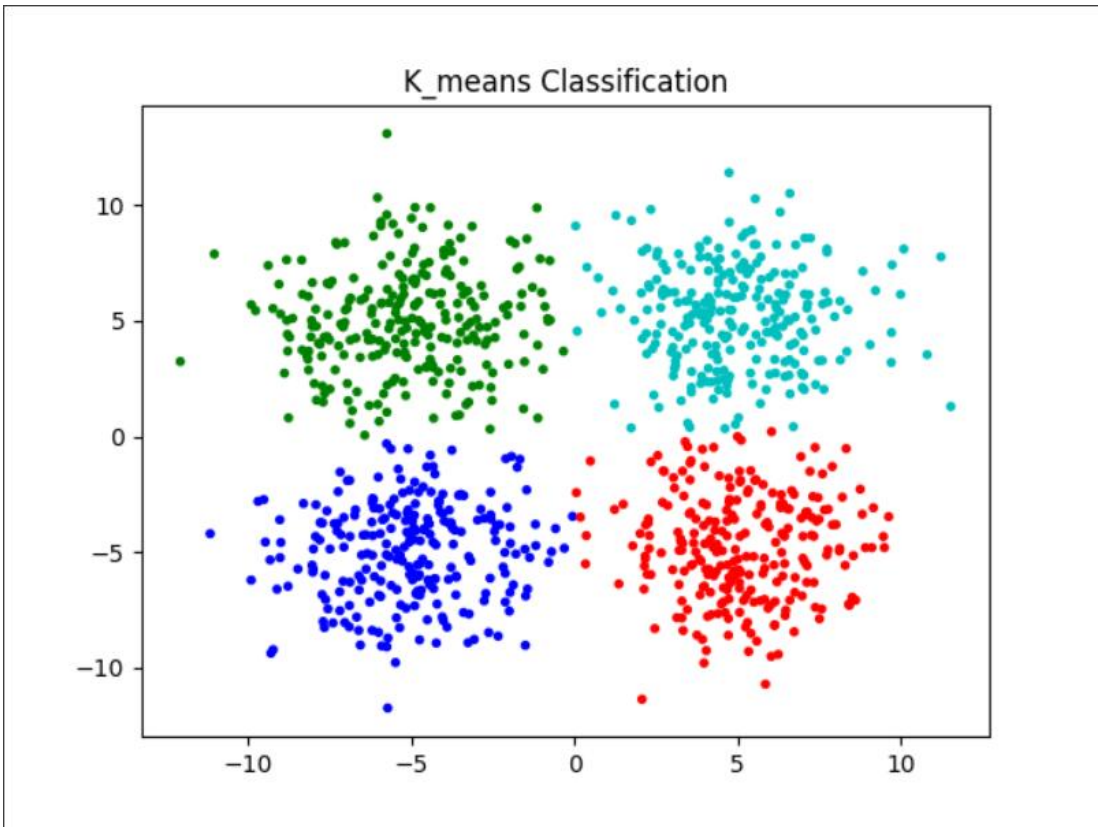
四、实验结果与分析

对于我的实验，样本数 N=1000，分类数 K=4，维数 D=2

1. k_means 算法：

当样本分布均匀且相对较离散时:

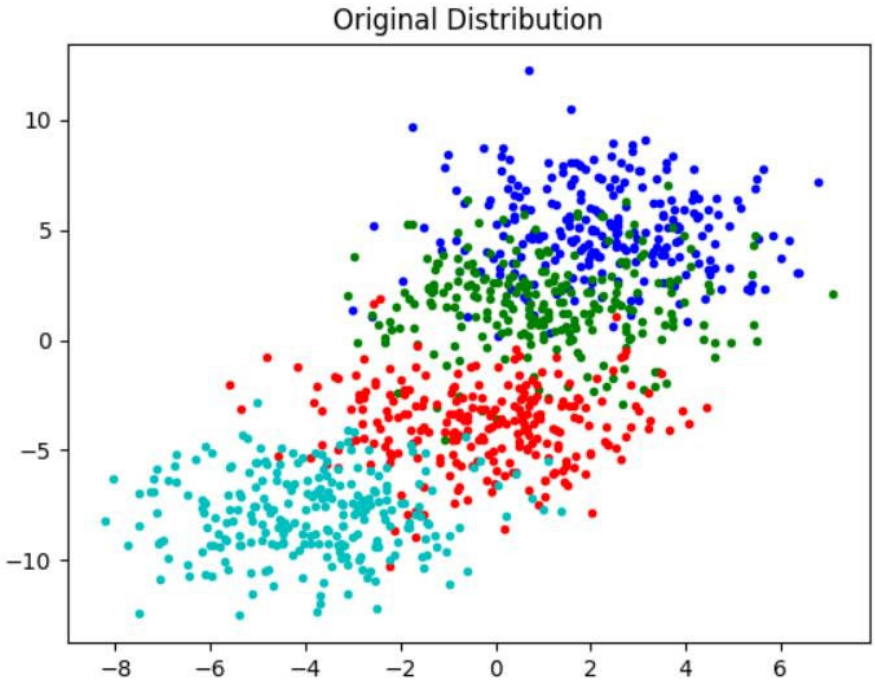


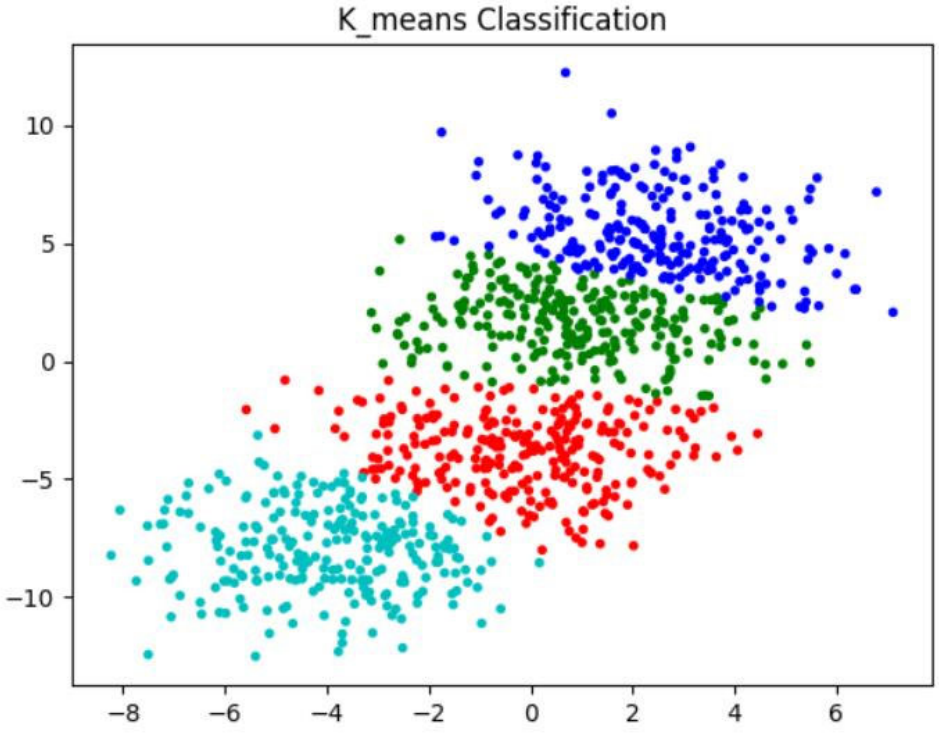



```
准确率为：0.985
```

从分类的图中以及我们所计算出的准确率来说，可以说效果非常好

当样本分布随机时：



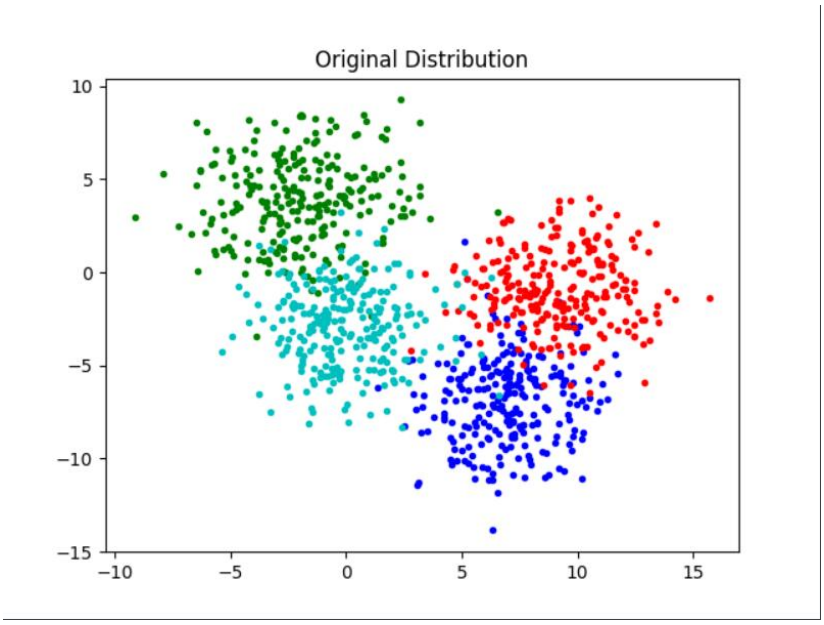


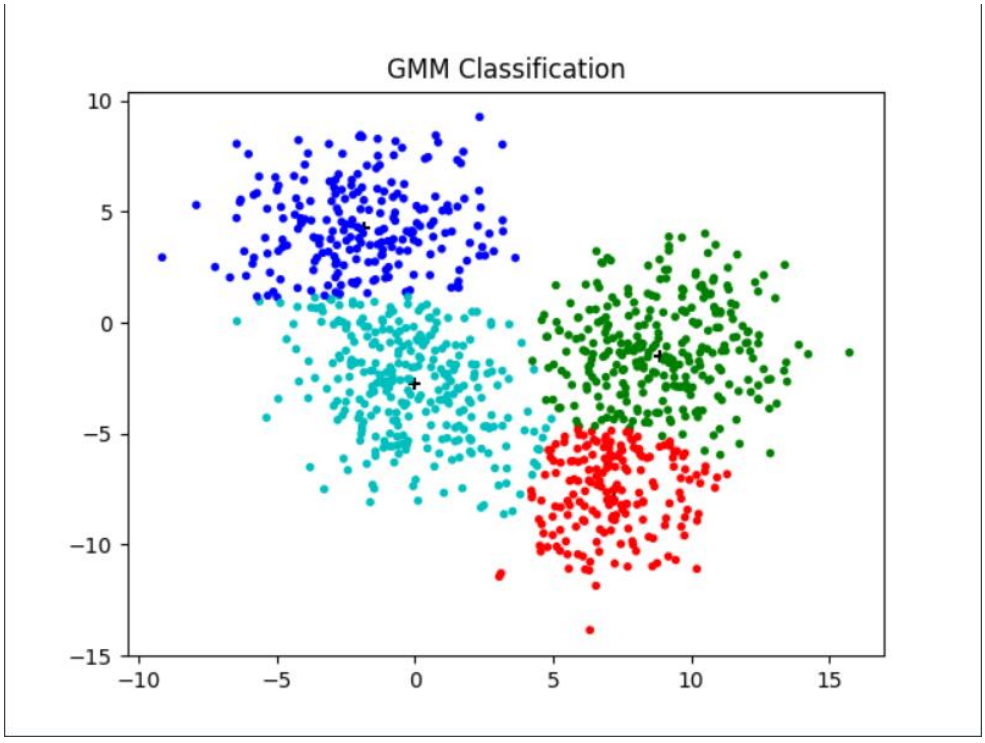

```
准确率为：0.834
```

我们发现，分类效果没有离散的时候好，准确率也下降不少。这是因为对于 k_means 来说，它只按照欧式距离来进行分类。由于类是服从高斯分布的，因此对于向源分布那样交叉的情况，k_means 识别的并不是很好。

这样我们就引出我们的混合高斯模型 GMM 算法：

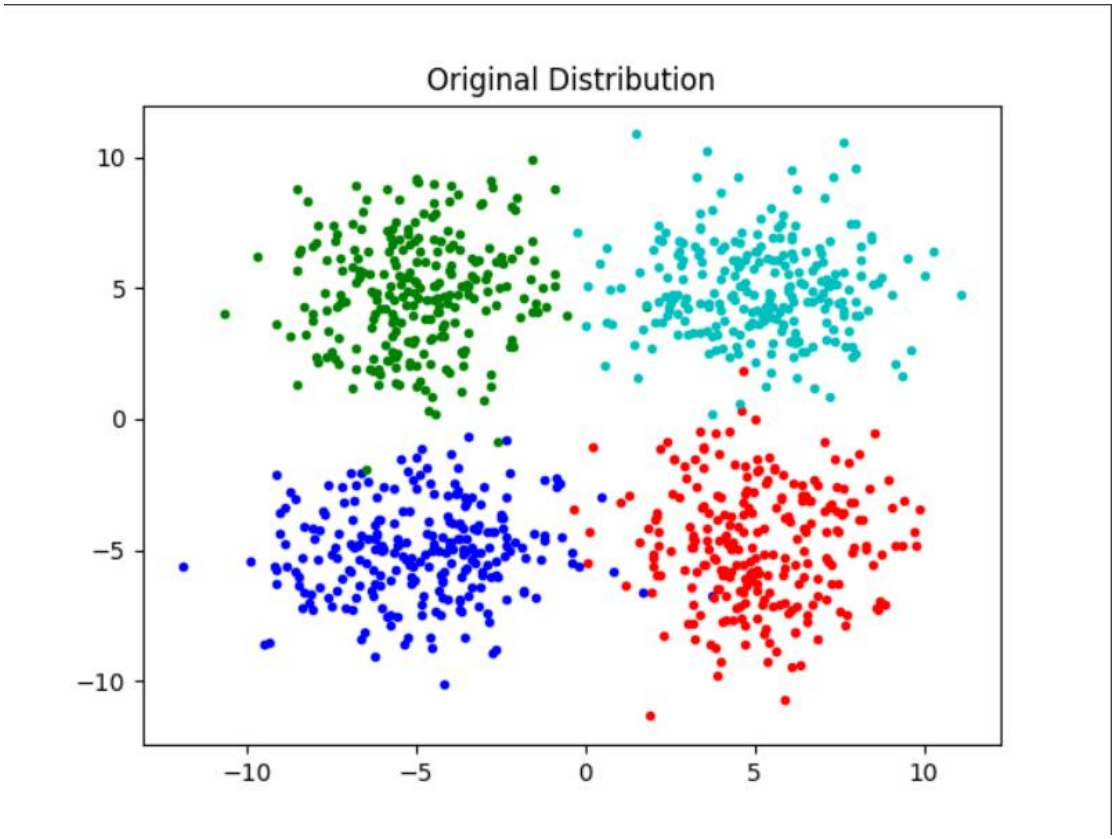
2. GMM 算法：

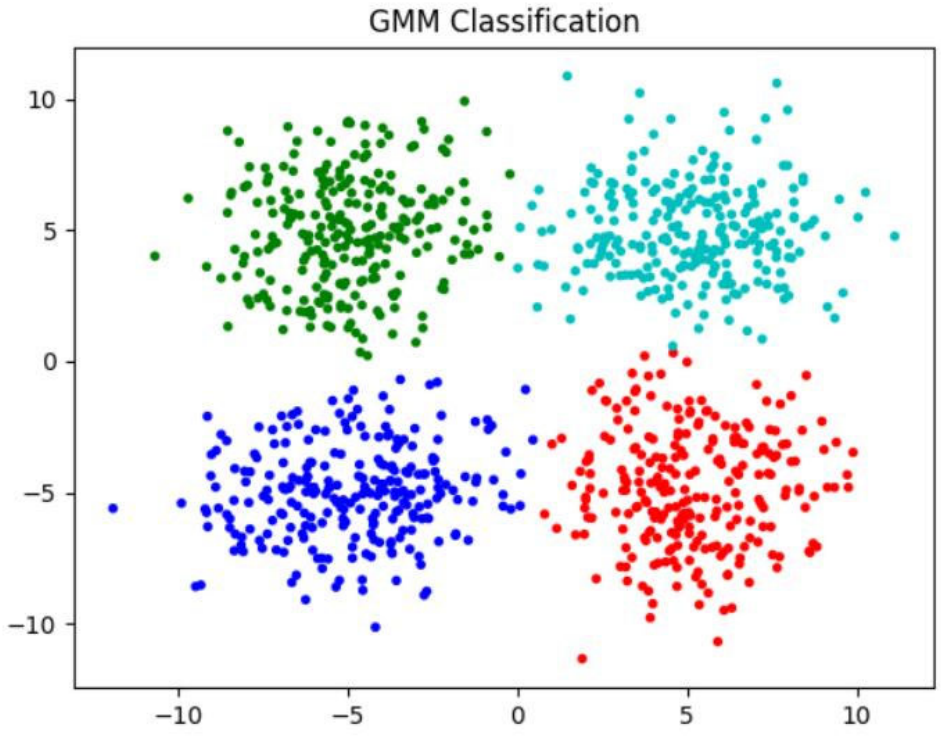




第一次写的时候，并没有对原始的标签进行识别，因此会导致两次分类的颜色不同，我通过对于标签的检测进行了一些改进

当数据均匀分布的时候:

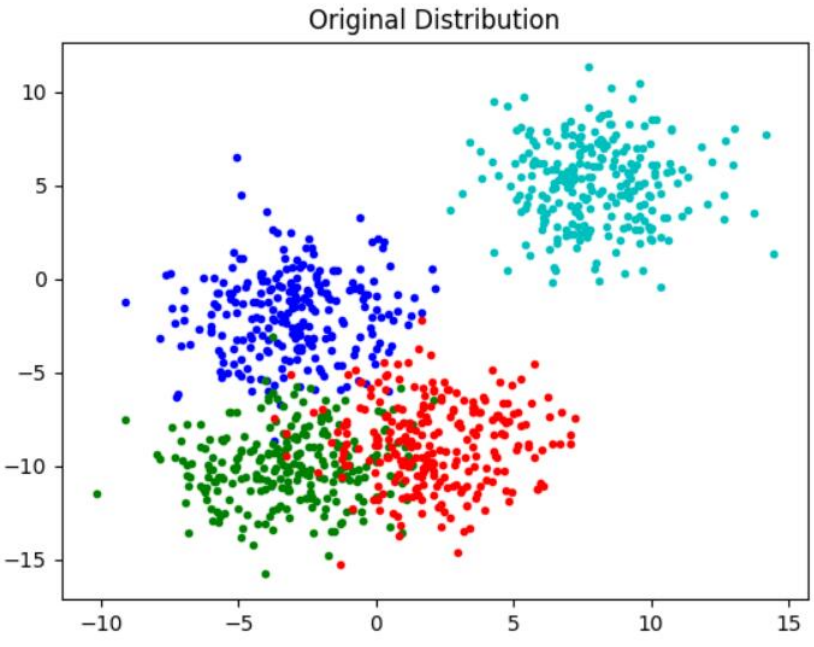


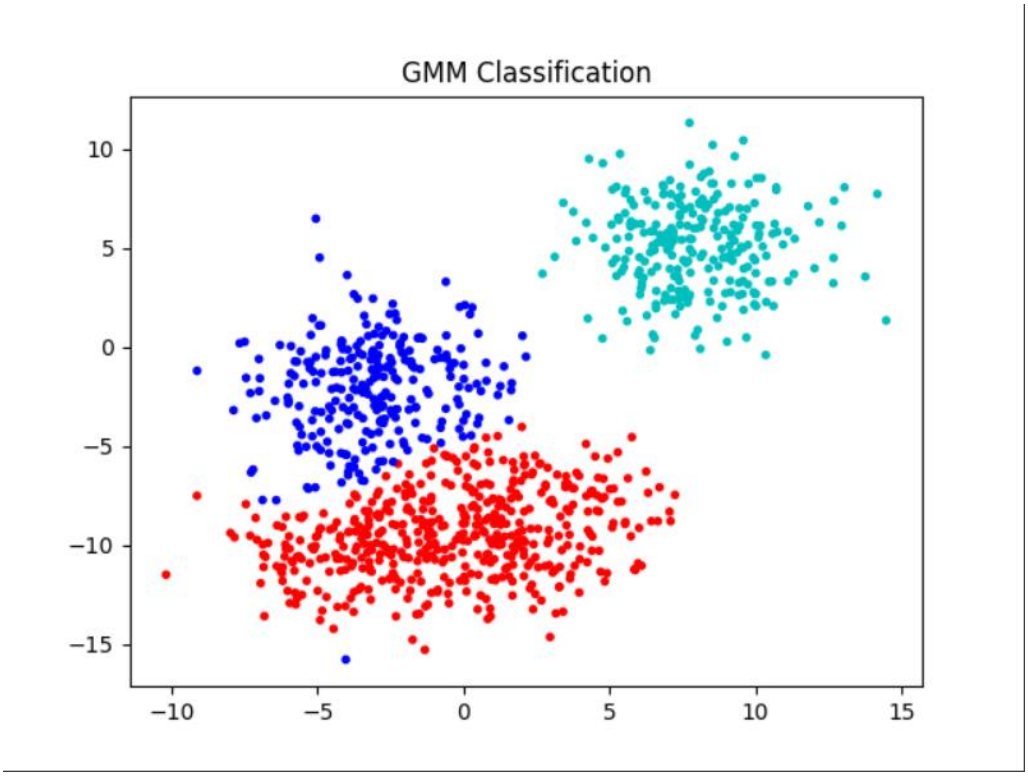



```
准确率为：0.983
```

和 k_means 相同，GMM 算法的准确性也是很高的

当数据不均匀时:

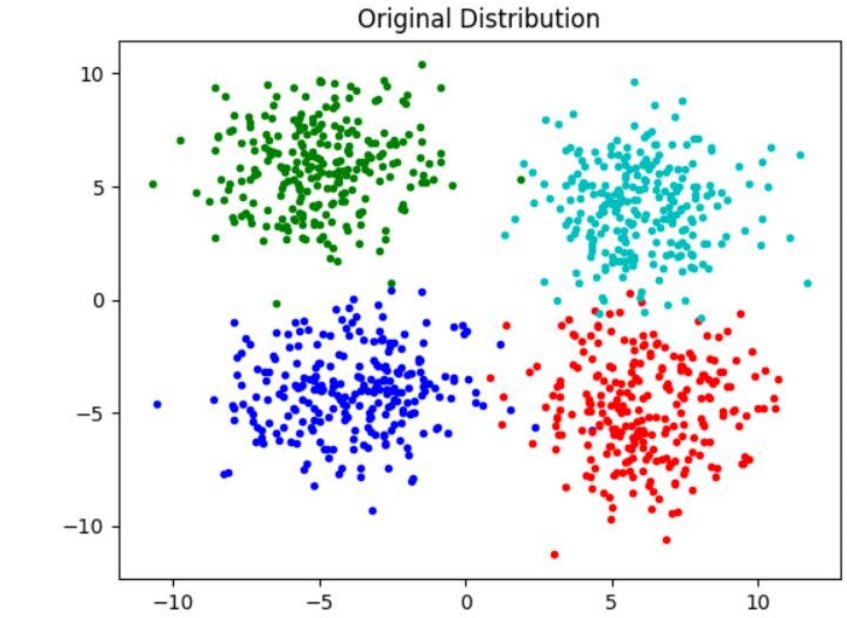


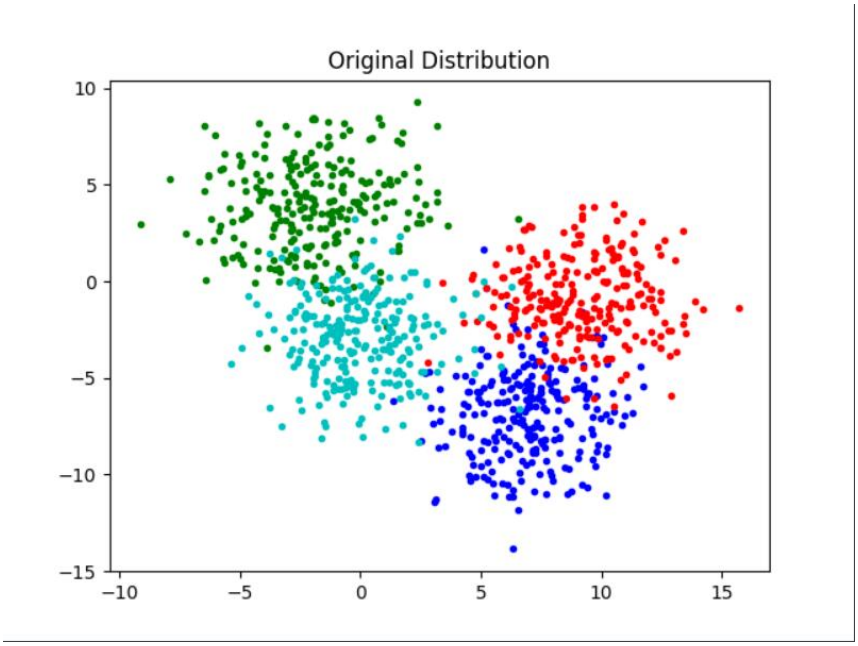



```
-4862.243368706835
-4862.697782821598
-4863.067458956338
-4863.365436261126
准确率为：0.73
```

我们可以看到，GMM 分类的效果并不是很理想十分理想,而且它只把它分成的三类

这跟初始 mean 的选取有关,只要我们将 k_mean 获得的 mean 作为 GMM 的初始值,我们将会得到很好的效果:





注意:对于 GMM 可能会导致奇异矩阵的问题,我们通过减少分类数,或者说调整初始的 mean 值并且调大协方差矩阵的值使其更加分散,就能尽可能避免这种情况

3. UCI 数据集:IRIS_DATA

有关 UCI 数据集的训练, 其中样本数 N=150,K=3,D=4 我们只需修改我们前面算法的接口就可以使用我们的算法进行训练:

k_means 算法:

```
k_means:
[[-1.00206653  0.89510445 -1.30297509 -1.25663117]
 [ 1.03359865  0.01388418  0.94369497  0.97226253]
 [-0.16840578 -0.97008147  0.25962078  0.17609756]]
准确率为：0.8533333333333334
```

GMM 算法:

```
GMM:
-156.55565384144967
-3.512810618523617
7.969447553337331
7.9951147883068705
7.996312392353475
7.996501890756416
7.996545431215151
7.99655461090216
[[ 0.90523345 -0.190544    1.01855103  1.08345208]
 [-1.01457897  0.8423068  -1.30487835 -1.25512862]
 [ 0.11777416 -0.64687837  0.29408174  0.18133246]]
准确率为：0.9866666666666667
```

我们发现,如果我们将 k_means 获得的均值作为 GMM 初始的 mean 那么会获得很好的效果

五、结论

1. K-Means 实际上假设数据式呈球状分布，假设使用的欧式距离来衡量样本与各个簇中心的相似度(假设数据的各个维度对于相似度计算的作用是相同的)，它的簇中心初始化对于最终的结果有很大的影响，如果选择不好初始的簇中心值容易使之陷入局部最优解；与之相比 GMM 使用更加一般的数据表示即高斯分布，GMM 使用 EM 算法进行迭代优化，因为其涉及到隐变量的问题，没有之前的完全数据，而是在不完全数据上进行
2. K-Means 的分类结果受初始中心点的影响，我们可以通过一些求均值或者别的的方式寻找初始点来尽量减少这种影响；GMM-EM 算法对初值的敏感程度更高，所以一般将 K-Means 得到的结果作为 GMM-EM 方法的初值
3. GMM-EM 算法可以达到很高的准确度，计算结果可以和 sklearn 自带的聚类算法计算的结果几乎相同
4. K-Means 和高斯混合模型比较：K-Means 其实就是一种特殊的高斯混合模型，我们假设每种类在样本中出现的概率相等，都为而且假设高斯模型中的每个变量之间是独立的，即变量间的协方差矩阵是对角阵，这样我们可以直接用欧氏距离作为 K-Means 的协方差去衡量相似性；K-Means

对响应度也做了简化，每个样本只属于一个类，即每个样本属于某个类，则响应度为 1，对于不属于的类，响应度直接设为 0，算是对 GMM 的一种简化。而在高斯混合模型中，每个类的数据出现在样本中的概率为 α \alpha α, 用协方差矩阵替代 K-Means 中的欧式距离去度量点和点之间的相似度，响应度也由离散的 0，1 变成了需要通过全概率公式计算的值。但是由于 GMM 并未增加如 K-Means 那么多假设条件，分类最终效果比 K-Means 好，但是 GMM-EM 算法过于细化，容易被噪声影响，所以适合作为优化算法，即适合对 K-Means 的分类结果进行进一步优化

六、参考文献

【统计学习方法】李航
【机器学习】周志华

七、附录：源代码（带注释）

k_means_and_gmm

```
import itertools

import random



import numpy as np

import matplotlib.pyplot as plt

from sklearn import metrics

import scipy.stats as st

import pandas as pd

from itertools import permutations




K = 4

INF = 9999999

D = 2

N = 1000

n = 250

colors = "bgrcmykw"
```

```
def create_data():
    mean = create_random_mean(K, D)
    # mean = create_fixed_mean()
    cov = create_random_cov(K, D)
    X = np.zeros((N, D))
    color_index = 0
    for i in range(K):
        x = np.random.multivariate_normal(mean[i], cov=cov[i], size=n)
        plt.scatter(x[:, 0], x[:, 1], c=colors[color_index], marker=".
")
        color_index = color_index + 1
        for j in range(n):
            X[j + i * n] = x[j]
    plt.title("Original Distribution")
    plt.show()
    return X, mean



def create_random_cov(k, d):
    cov = np.zeros((k, d, d))
    for i in range(k):
        cov[i] = np.identity(d)
        for j in range(d):
            cov[i][j][j] = random.uniform(3, 5)
    return cov



def create_random_mean(k, f):
    mean = np.zeros((k, f))
    for i in range(k):
```

```
        for j in range(f):
            mean[i][j] = random.randint(-10, 10)
    return mean



def create_fixed_mean():
    # return [[-5, -5], [-5, 5], [5, -5], [5, 5]]
    return [[-4, -4], [-5, 6], [6, -5], [6, 4]]



def create_random_gmm_num(k, d):
    mean = create_random_mean(k, d)
    cov = np.zeros((k, d, d))
    for i in range(k):
        cov[i] = np.identity(d)
    ratio = np.ones(k) / k
    return mean, cov, ratio



def k_means(X, k, threshold=1e-20):
    X_flag = np.zeros(X.shape[0])
    mean = create_random_mean(k, X.shape[1])
    dis_start = cal_distance_all(X, k, mean)
    while True:
        dis_end = dis_start
        for i in range(X.shape[0]):
            flag = 0
            min_dis = INF
            for j in range(k):
                if min_dis > cal_distance(X[i, :], mean[j, :]):
```

```
                min_dis = cal_distance(X[i, :], mean[j, :])
                flag = j
            X_flag[i] = flag
        mean = update_mean(X, k, X_flag)
        dis_start = cal_distance_all(X, k, mean)
        if np.fabs(dis_end - dis_start) < threshold:
            break
    return mean, X_flag



def gmm(X, k, threshold=1e-5):
    mean, cov, ratio = create_random_gmm_num(k, X.shape[1])
    mean, x = k_means(X, k)
    last_log_likelihood = cal_log_likelihood(X, k, mean, cov, ratio)
    iters = 0
    while True:
        y_z = step_e(X, k, mean, cov, ratio)
        mean, cov, ratio = step_m(X, k, mean, cov, ratio, y_z)
        now_log_likelihood = cal_log_likelihood(X, k, mean, cov, ratio)
        print(now_log_likelihood)
        if last_log_likelihood < now_log_likelihood and (now_log_likeli
hood - last_log_likelihood) < threshold:
            break
        last_log_likelihood = now_log_likelihood
        iters = iters + 1
        if iters >= 50:
            break
    return X, y_z, mean
```

```
def draw_gmm(X, y_z, k, mean):
    label = np.zeros(X.shape[0])
    for i in range(k):
        label[i * n: (i + 1) * n] = i
    X = np.hstack((X, label.reshape(-1, 1)))
    for i in range(X.shape[0]):
        X[i, -1] = np.argmax(y_z[i, :])
    draw(X, k, mean, "GMM ")



def draw_k_means(X, X_flag, k, mean):
    X = np.hstack((X, X_flag.reshape(-1, 1)))
    draw(X, k, mean, "K_means ")



def draw(X, k, real_mean, method):
    for i in range(k):
        x = []
        for j in range(X.shape[0]):
            if X[j, -1] == i:
                x.append(X[j, 0:2])
        x = np.array(x)
        cal_mean = np.mean(x, axis=0)
        min = INF
        real_label = 0
        for p in range(k):
            if cal_distance(cal_mean, real_mean[p]) < min:
                min = cal_distance(cal_mean, real_mean[p])
                real_label = p
```

```
        colors_array = np.full(x.shape[0], colors[real_label], dtype=n
p.str_)
        plt.scatter(x[:, 0], x[:, 1], c=colors_array, marker=".")
    acc = cal_accuracy(X, k)
    plt.title(method + "Classification")
    plt.show()



def cal_accuracy(X, k):
    # 计算准确率
    accuracy = []
    per = np.zeros(k)
    for i in range(k):
        per[i] = i
    per = np.array(list(itertools.permutations([0, 1, 2, 3])))
    for i in range(per.shape[0]):
        count = 0
        for j in range(k):
            for p in range(int(X.shape[0] / k)):
                if X[p + (j * int(X.shape[0] / k)), -1] == per[i][j]:
                    count = count + 1
        accuracy.append(count / X.shape[0])
    num_acc = np.argmax(accuracy)
    acc = accuracy[num_acc]
    print("准确率为: "+str(acc))
    return acc



def step_e(X, k, mean, cov, ratio):
    # e步
```

```
    # y_z: 样本混合高斯叠加之后的后验概率
    # ratio: 每种高斯分布所占的比例
    # mean: 每种高斯分布的均值
    # cov: 每种高斯分布的协方差矩阵
    y_z = np.zeros((X.shape[0], k))
    for i in range(X.shape[0]):
        # 计算每个样本在混合高斯之后的后验概率
        ratio_sum = 0
        ratio_pdf = np.zeros(k)
        for j in range(k):
            ratio_pdf[j] = ratio[j] * st.multivariate_normal.pdf(X[i], m
ean=mean[j], cov=cov[j])
            ratio_sum = ratio_sum + ratio_pdf[j]
        for j in range(k):
            y_z[i, j] = ratio_pdf[j] / ratio_sum
    return y_z



def step_m(X, k, mean, cov, ratio, y_z):
    # m步 更新数据
    new_mean = np.zeros(mean.shape)
    new_cov = np.zeros(cov.shape)
    new_ratio = np.zeros(ratio.shape)
    for j in range(k):
        new_ratio[j] = np.sum(y_z[:, j]) / X.shape[0]
        y = y_z[:, j].reshape(-1, 1)
        new_mean[j, :] = (y.T @ X) / np.sum(y)
        new_cov[j] = ((X - mean[j]).T @ np.multiply((X - mean[j]), y) /
np.sum(y))
    return new_mean, new_cov, new_ratio
```

```
def update_mean(X, k, X_flag):
    new_center = np.zeros((k, X.shape[1]))
    for i in range(k):
        count = 0
        for j in range(X.shape[0]):
            if X_flag[j] == i:
                new_center[i, :] = new_center[i, :] + X[j, :]
                count = count + 1
        if count != 0:
            new_center[i, :] = new_center[i, :] / count
    return new_center



def cal_log_likelihood(X, k, mean, cov, ratio):
    log_sum = 0
    for i in range(X.shape[0]):
        ratio_pdf_sum = 0
        for j in range(k):
            ratio_pdf_sum = ratio_pdf_sum + ratio[j] * st.multivariate_n
ormal.pdf(X[j], mean=mean[j], cov=cov[j])
        log_sum = log_sum + np.log(ratio_pdf_sum)
    return log_sum



def cal_distance(X, center):
    return np.sum(np.power((X - center), 2))



def cal_distance_all(X, k, center):
```

```
    dis = 0
    for i in range(X.shape[0]):
        for j in range(k):
            dis = dis + cal_distance(X[i, :], center[j, :])
    return dis



def main():

    # 获取数据 以及真实的均值 mean
    X, real_mean = create_data()

    # k_means
    mean_k_means, X_flag = k_means(X, K)
    draw_k_means(X, X_flag, K, real_mean)

    # GMM_EM
    X_gmm, y_z, mean_gmm = gmm(X, K)
    draw_gmm(X, y_z, K, real_mean)



if __name__ == '__main__':
    main()
```

iris_data

```
import random
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
import scipy.stats as st
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
import pandas as pd
import k_means_and_GMM as kg


K = 3
n = 50



def iris_data():
    # 处理数据 拿出数据和标签
    column = ["s_length", "s_width", "p_length", "p_width", "class"]
    data = pd.read_csv("data/iris.data.csv", engine='python', names=co
lumn)
    std = StandardScaler()
    X = std.fit_transform(np.array(data[column[0:4]]))
    y = np.array(data[column[4]])
    return X, y



def k_means(X):
    print("k_means:")
    mean, X_flag = kg.k_means(X, K)
    print(mean)
    X = np.hstack((X, X_flag.reshape(-1, 1)))
    kg.cal_accuracy(X, K)
    return mean



def gmm(X):
    print("GMM:")
```

```
    X, y_z, mean = kg.gmm(X, K)
    label = np.zeros(X.shape[0])
    for i in range(K):
        label[i * n: (i + 1) * n] = i
    X = np.hstack((X, label.reshape(-1, 1)))
    for i in range(X.shape[0]):
        X[i, -1] = np.argmax(y_z[i, :])
    print(mean)
    kg.cal_accuracy(X, K)


def main():
    X, y = iris_data()
    k_means(X)
    gmm(X)


if __name__ == '__main__':
    main()
```