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： 多项式拟合正弦函数

学号：1170301007

姓名：沈子鸣

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

实验要求:

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）。

可以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:

Windows 10 专业教育版；python 3.7.4；jupyter notebook 6.0.1

三、设计思想（本程序中的用到的主要算法及数据结构）

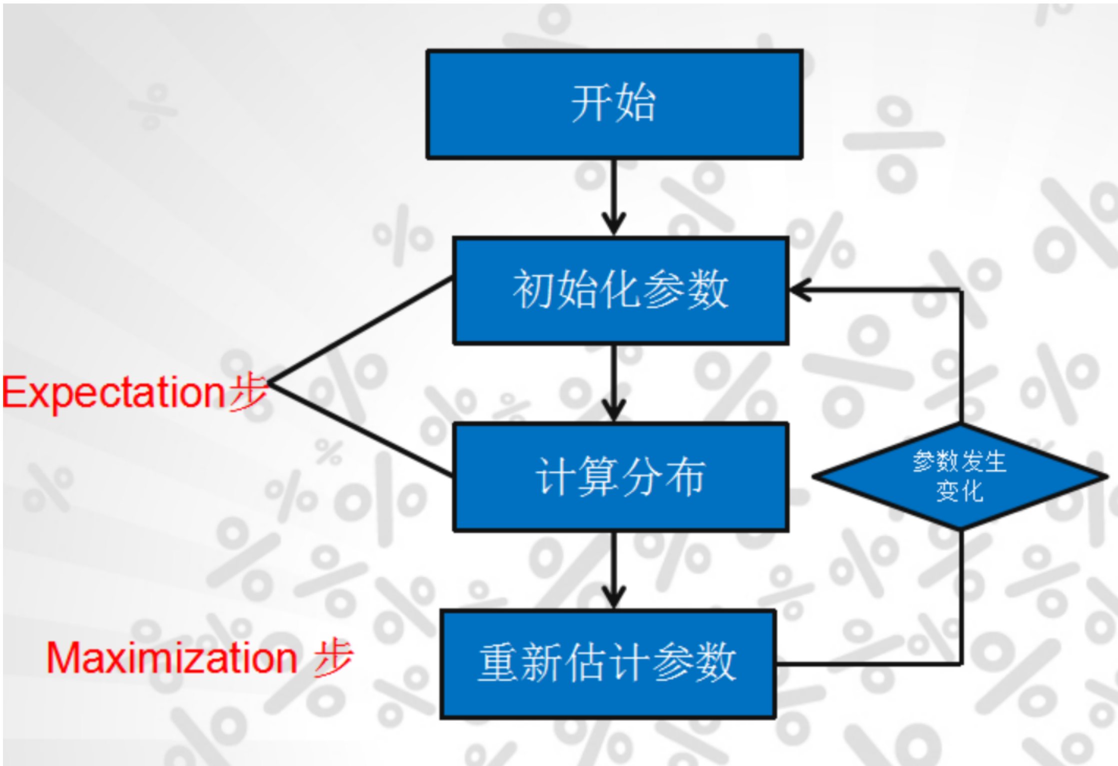
1. 算法原理

   本次实验主要分两部分进行。两个算法K-means和GMM的实现本质上都是EM算法的应用。先介绍EM算法的思想:

   EM算法分为两步，

   > E步：expectation step，求期望步
   >
   > M步：maximization step，最大化似然步

   可以用下图表示



   E步是调整分布，M步是根据调整的分布，求使得目标函数最大化的参数，从而更新了参数，接着参数的更新又可以调整分布，不断循环，直到参数的变化收敛，这一过程目标函数是向更优的方向趋近的，但是在某些情况下会陷入局部最优而非全局最优的问题。

1) K-means算法

K-means算法是EM算法的一个简单体现。伪码如下：

```
k ← 输入期望的聚类数
centers ← 从样本中初始化k个聚类中心（可以随机选取）
while true:
    E步：遍历所有样本，根据样本到k个聚类中心的距离度量，判断该样本的标签（类别）labels[i]
    M步：根据标签划分结果labels，重新计算k个聚类中心（centers）
    if k个聚类中心的变化全部小于误差值eps:
        break
return centers, labels
```

假设我们有一组样本X，想要把他们分为3类（这个类别数是根据实际情况选择的）。在初始化聚类中心时，一个简单的方法是随机选取X中的3个点，作为三个类别的样本中心，因为是随机选取的，所以不能避免初始化的三个中心点之间距离过近的情况，糟糕的初始值选取可能会导致糟糕的分类结果。

然后开始不断迭代，迭代主要分为两步，这两部分别对应着EM算法的E步和M步。其中E步是标签的划分，遍历X中的所有点，对任意一个点计算它到3个聚类中心的度量距离（比如欧式距），取距离最小的聚类中心所代表的那一类，作为该点的标签。E步过后，所有样本都更新了标签，然后进入M步，根据E步更新的标签，得到了每一类的样本，然后分别对这三类样本，计算均值（可以分别对每一个维度计算均值，得到均值中心）。得到的均值中心又可以在下一轮迭代中，作为标签更新的参考。

循环过程直到所有的聚类中心收敛。

2）GMM算法

GMM相对K-means是比较复杂的EM算法的应用实现。与K-means不同的是，GMM算法在E步时没有使用“最近距离法”来给每个样本赋类别（hard assignment），而是增加了隐变量γ。γ是(N,K)的矩阵，$\gamma[n,k]$表示第n个样本是第k类的概率，因此，γ具有归一性。即γ的每一行的元素的和值为1.

GMM算法是用混合高斯模型来描述样本的分布，因为在多类情境中，单一高斯分布肯定是无法描绘数据分布。多个高斯分布的简单叠加也无法描绘数据分布的。只有混合高斯分布才能较好的描绘一组由多个高斯模型产生的样本。对于样本中的任一个数据点，任一高斯模型能够产生该点的概率，也就是任一高斯模型对该点的生成的贡献（contribution）是不同的，但可以肯定的是，这些贡献的和值是1.

γ的推导过程如下

$$\gamma(z_{nk}) = p(z_{nk}=1|x) = \frac{p(x,z_k=1)}{p(x)}$$
$$= \frac{\pi_k N(x|\mu_k,\Sigma_k)}{\Sigma_{j=1}^{K}\pi_j N(x|\mu_j,\Sigma_j)}$$
$$\text{其中，}\pi_k = p(z_k=1)\text{为第}k\text{个高斯模型的权重，}N\text{为概率密度函数}$$

上式中可以看作π为先验概率，γ为后验概率。

而我们的目标函数，即对数极大似然估计为：

$$\ln p(X|\pi,\mu,\Sigma) = \Sigma_{n=1}^{N}\ln\Sigma_{k=1}^{K}\pi_k N(x|\mu_k,\Sigma_k)$$

令

$$\mu,\Sigma,\pi = argmax_{\mu,\Sigma,\pi}\ln p(X|\mu,\Sigma,\pi)$$

给定均值、方差和模型权重的初值，便可以从E步开始（计算γ），进入EM算法的迭代过程。得到γ后，可以对目标函数求导数（参考正态函数求导），令导数为0，得到参数更新：

$$\frac{\partial \ln p(X|\pi,\mu,\Sigma)}{\partial \mu_k} = \Sigma_n \frac{\pi_k N(x_n|\mu_k,\Sigma_k)}{\Sigma_j \pi_j N(x_n|\mu_j,\Sigma_j)} \Sigma_K^{-1}(x_n-\mu_k) = 0$$
$$\Sigma_k \gamma(z_{nk}) \Sigma_k^{-1}(x_n-\mu_k) = 0$$
$$\Sigma_n \gamma(z_{nk}) x_n = \Sigma_n \gamma(z_{nk}) \mu_k$$
$$\mu_k = \frac{1}{N_k} \Sigma_n \gamma(z_{nk}) x_n, N_k = \Sigma_n \gamma(z_{nk})$$

$$\frac{\partial \ln p(X|\pi,\mu,\Sigma)}{\partial \Sigma_k} = \Sigma_n \frac{1}{\Sigma_j \pi_j N(x|\mu_j,\Sigma_j)} (\pi_k N(x|\mu_k,\Sigma_k)(\Sigma_k^{-1} - \Sigma_k^{-1}(x_n-\mu_k)(x_n-\mu_k)^T \Sigma_k^{-1})) = 0$$
$$\Sigma_n \gamma(z_{nk})(\Sigma_k^{-1} - \Sigma_k^{-1}(x_n-\mu_k)(x_n-\mu_k)^T \Sigma_k^{-1}) = 0$$
$$\Sigma_n \gamma(z_{nk}) \Sigma_k^{-1} = \Sigma_n \gamma(z_{nk}) \Sigma_k^{-1}(x_n-\mu_k)(x_n-\mu_k)^T \Sigma_k^{-1}$$
$$\Sigma_k = \frac{1}{N_k} \Sigma_n \gamma(z_{nk})(x_n-\mu_k)(x_n-\mu_k)^T$$

至于π_k的更新，用$\Sigma_k \pi_k = 1$约束求导数=0可得

$$\pi_k = \frac{N_k}{N}$$

3）综述，K-means和GMM都是EM算法的表现。两者的区别在于E步，K-means在E步中计算参数分布时采用hard assignment的方式，即选取距离样本最近的聚类中心的类别，作为该样本的类别。而GMM在E步中计算参数分布时，是评估隐变量$\gamma(z_{nk})$，即每一类样本属于每一类聚类的概率。其实K-means的E步中也是隐变量，只不过在K-means中，在γ矩阵中，每一行只有一个元素为1，就是离样本最近的那个类。

此外，K-means还做出了每一类模型对总的贡献相同等比较强的假设。从实际表现来看，GMM的表现更贴近实际，K-means则过于简单。

2. 算法的实现

首先配置好自己生成数据时的高斯分布参数：

```
config = {
    'k':3,
    'n':200,
    'dim':2,
    'mus':np.array([
        [1,3], [7,8], [8,4]
    ]),
    'sigmas':np.array([
        [[1,0],[0,2]], [[3,0],[0,2]], [[3,0],[0,1]]
    ])
}
```

生成数据时：

data：shape=(k,n, dim)

data_：shape=(k*n, dim)

return data_

kmeans算法数据变量如下：

```
classes = np.zeros((N, dim+1)) # classes array
center = np.zeros((K, dim)) # center is the cluster center, it has K classes
distance = np.zeros(K) # in each iterate distance stores the distance between each data and the present cluster center
num = np.zeros(K) # num stores how many data in each class
new_center = np.zeros((K,dim)) # new cluster center
```

GMM算法数据变量如下:

E步中:

```
def e_step(data, mus, sigmas, pis, N, K, dim):
    gammas = np.zeros((N, K))
    for n in range(N):
        marginal_prob = 0
        for j in range(K):
            marginal_prob += pis[j] * multivariate_normal.pdf(data[n], mean=mus[j], cov=sigmas[j])
        for j in range(K):
            gammas[n,j] = pis[j] * multivariate_normal.pdf(data[n], mean=mus[j], cov=sigmas[j]) / marginal_prob
    return gammas
```

M步中:

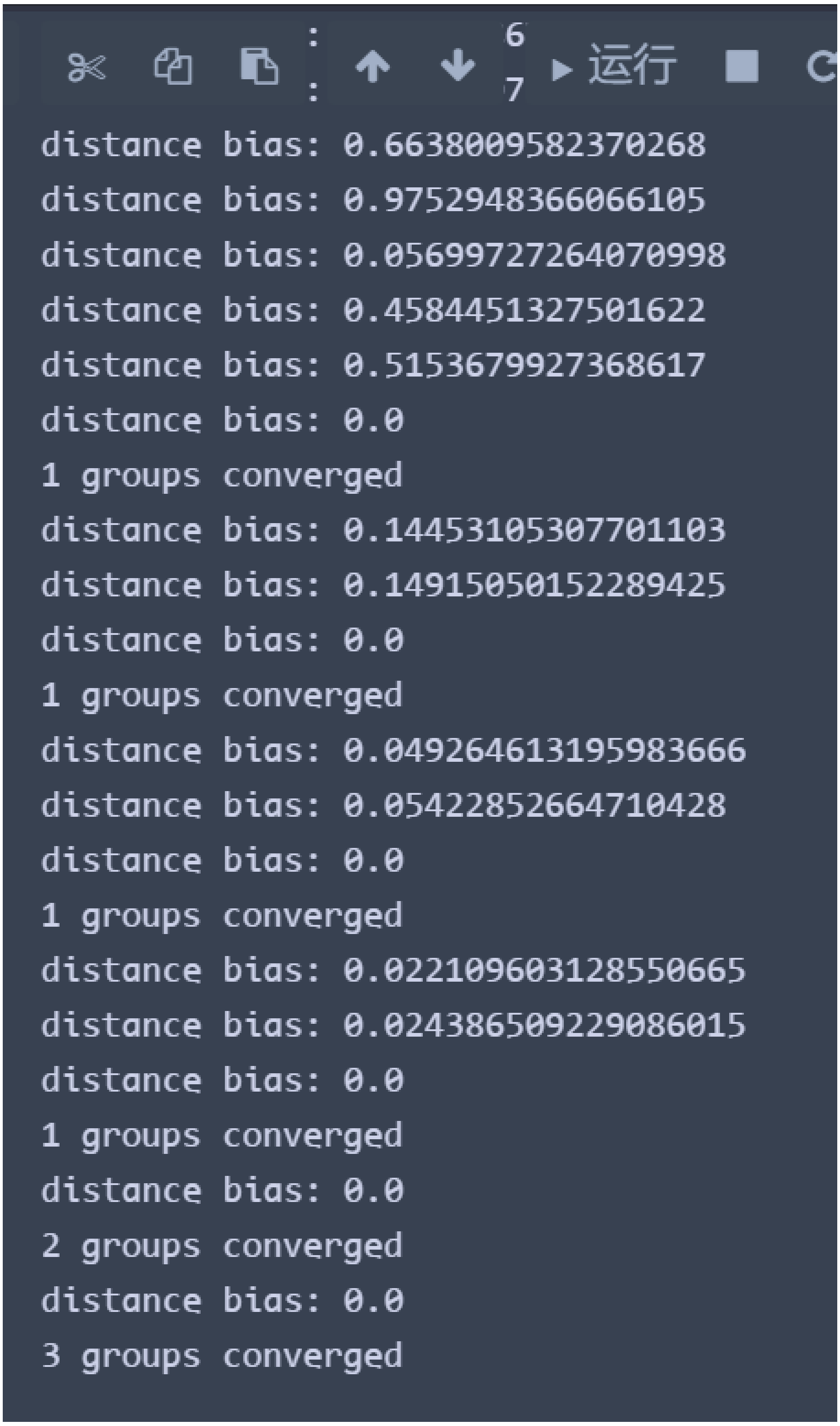
```
def m_step(data, gammas, mus, N, K, dim):
    mus_ = np.zeros((K, dim))
    sigmas_ = np.zeros((K, dim, dim))
    pis_ = np.zeros(K)
    for k in range(K):
        nk = 0
        for n in range(N):
            nk += gammas[n,k]
        mu_temp = np.zeros(dim)
        for n in range(N):
            mu_temp += gammas[n,k] * data[n]
        mus_[k] = mu_temp / nk

        sigma_temp = np.zeros(dim)
        for n in range(N):
            dis = data[n] - mus[k] # one-dimension array can not be transposed
            sigma_temp += dis**2 * gammas[n,k]
        sigma_temp_ = np.eye(dim)
        sigma_temp_[0,0] = sigma_temp[0]
        sigma_temp_[1,1] = sigma_temp[1]
        sigmas_[k] = sigma_temp_ / nk

        pis_[k] = nk / N
    return mus_, sigmas_, pis_

```

极大似然估计:

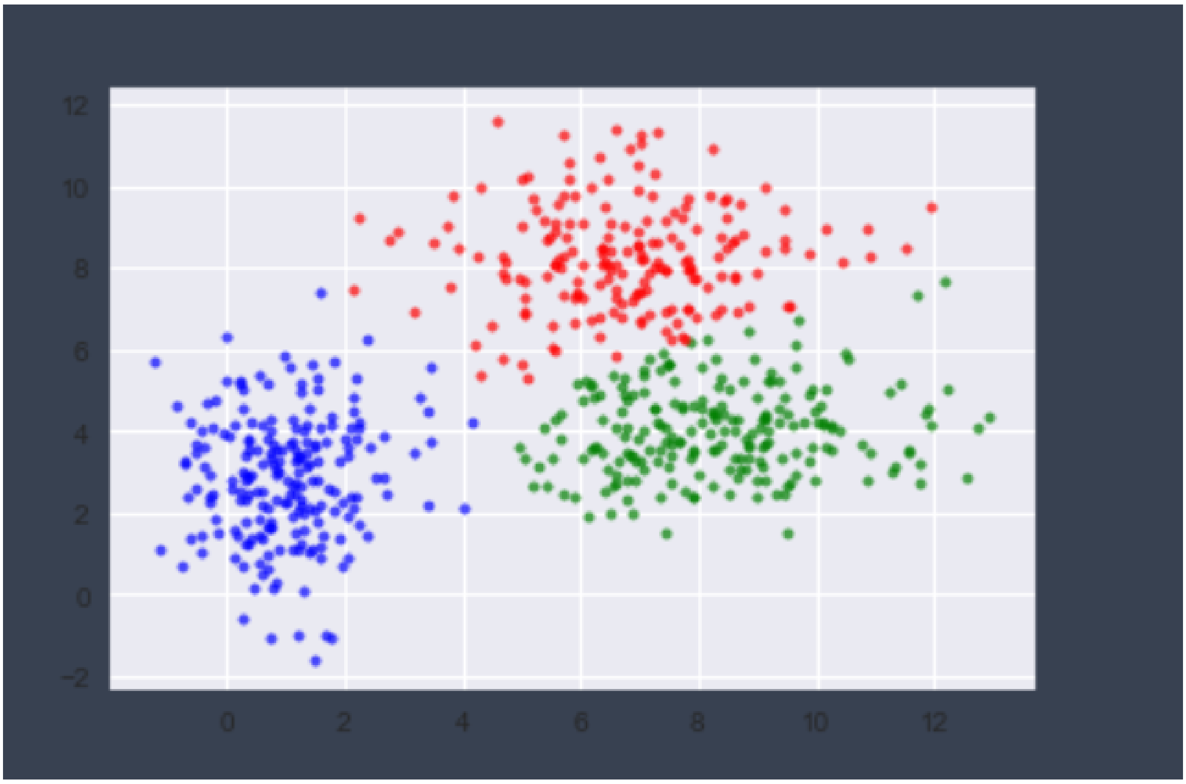
```python
def log_likelihood(data, mus, sigmas, pis, N, K, dim):
    l = 0
    for n in range(N):
        temp = 0
        for k in range(K):
            temp += pis[k] * multivariate_normal.pdf(data[n], mean=mus[k], cov=sigmas[k])
        l += math.log(temp)
    return l
```

四、实验结果与分析

自生成数据，K-means

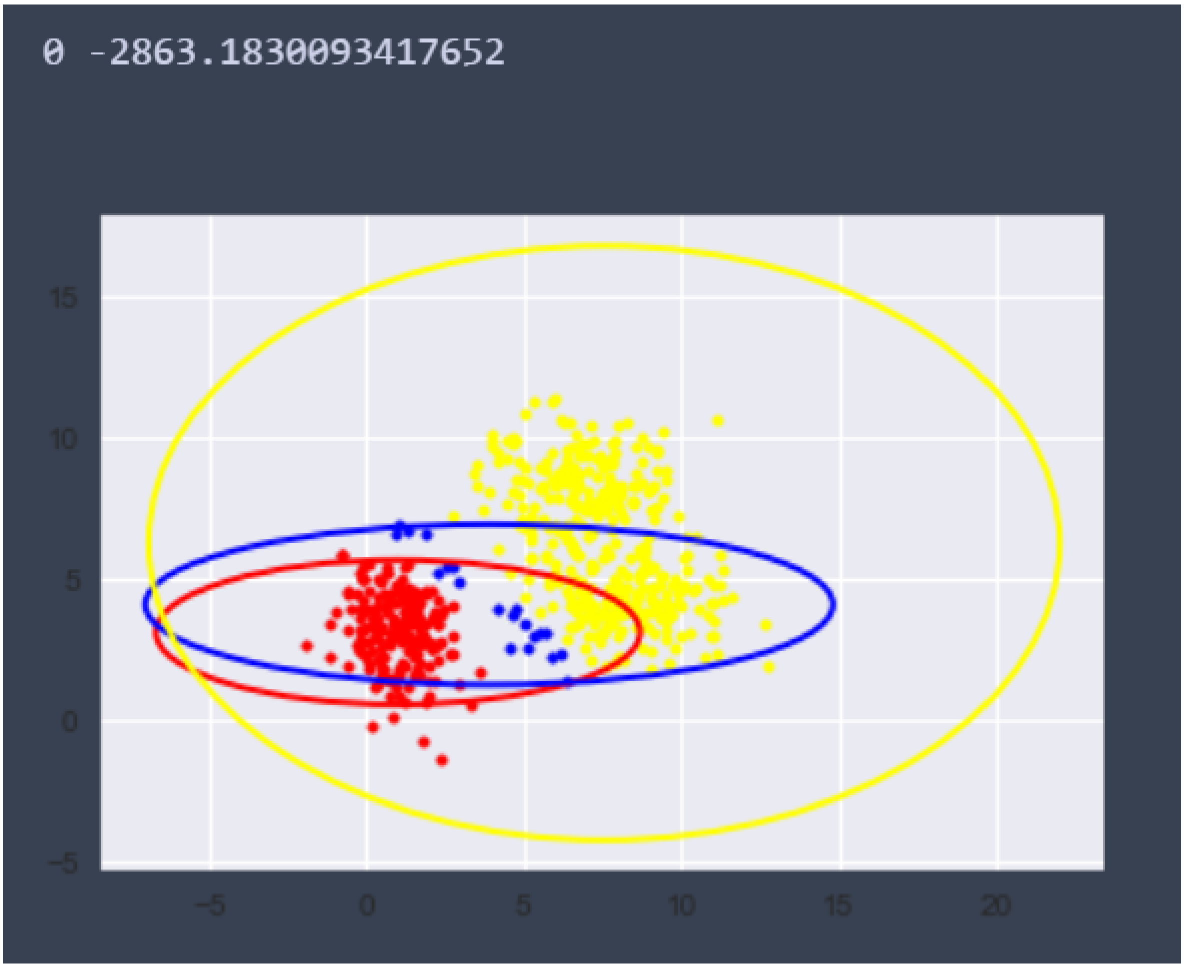


```
distance bias: 0.6638009582370268
distance bias: 0.9752948366066105
distance bias: 0.05699727264070998
distance bias: 0.4584451327501622
distance bias: 0.5153679927368617
distance bias: 0.0
1 groups converged
distance bias: 0.14453105307701103
distance bias: 0.14915050152289425
distance bias: 0.0
1 groups converged
distance bias: 0.049264613195983666
distance bias: 0.05422852664710428
distance bias: 0.0
1 groups converged
distance bias: 0.022109603128550665
distance bias: 0.024386509229086015
distance bias: 0.0
1 groups converged
distance bias: 0.0
2 groups converged
distance bias: 0.0
3 groups converged
```

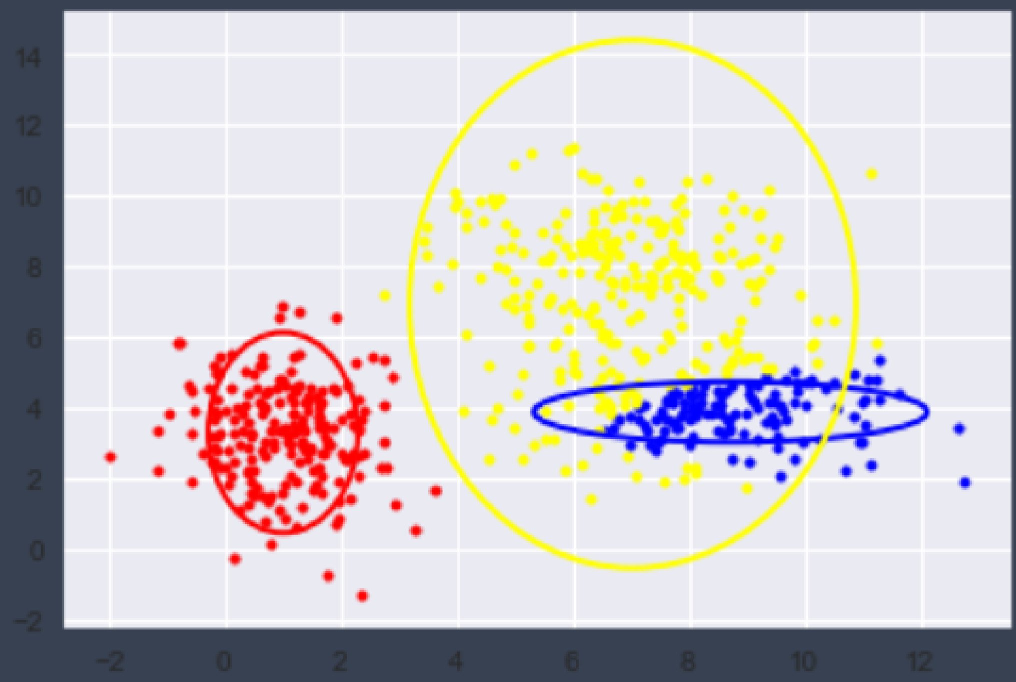


自生成数据，GMM

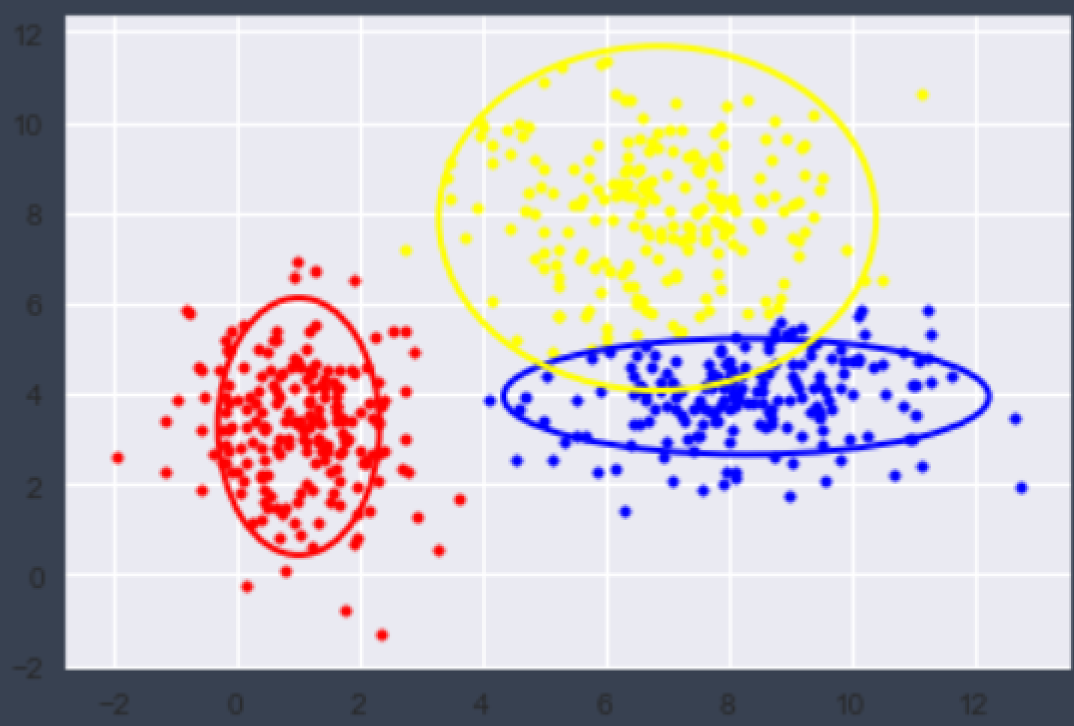
图中给出了每一阶段的样本分类情况和置信椭圆。最后带有填充的透明置信椭圆为生成数据的真实椭圆。



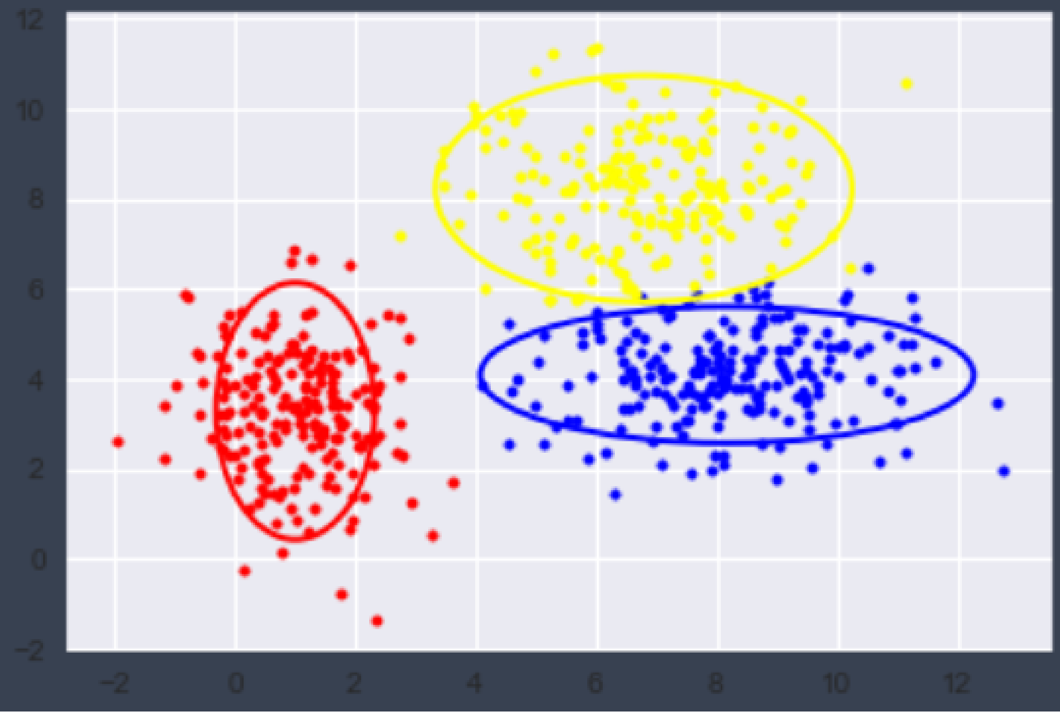
```
10 -2637.9494669651363
```

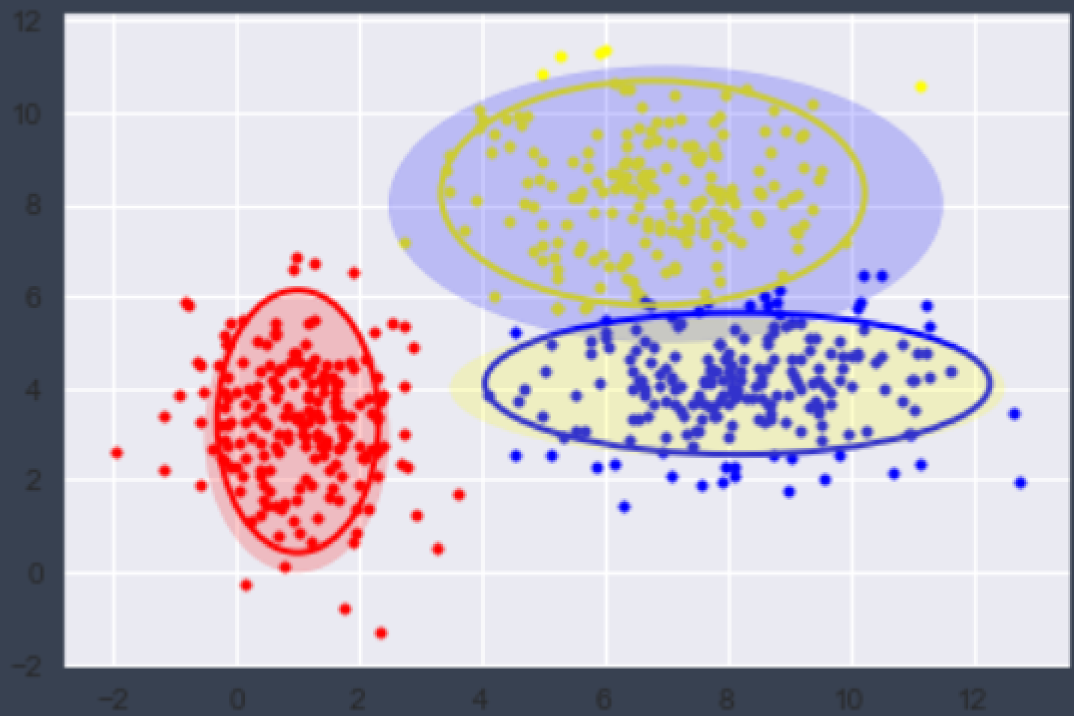



```
20 -2616.727708794242
```



```
30 -2611.366835733978
```





uci数据，K-means求初值，GMM

```
1 groups converged
distance bias: 0.0
2 groups converged
distance bias: 0.0
3 groups converged
0 -7190.317449853984
10 -9533.006152887352
20 -9147.862208694027
30 -9129.631863621678
[71. 65. 73.]
```

uci种子数据是有标签的，共210组真实数据，分为3类，每类70个样本。

分类结果（丢弃了第一个样本，故总数为209）显示，分为3类的结果中，每类个数分别为71，65和73. 打印分类结果，与真实标签情况符合较好。

五、结论

1. K-means和GMM都是EM算法的体现。两者共同之处都有隐变量，遵循EM算法的E步和M步的迭代优化。不同之处在于K-means给出了很多很强的假设，比如假设了所有聚类模型对总的贡献是相等的（平均的），假设一个样本由某一个特定聚类模型产生的概率是1，其他为0. 而GMM用混合高斯模型来描述聚类结果。假设多个高斯模型对总模型的贡献是有权重的，且样本属于某一类也是由概率的。两者都能较好的解决简单的分类问题，但存在着可能只取到局部最优的问题。
2. 初值的选取对K-means和GMM的效果影响较大。K-means的初值选取通常是给定聚类个数k和随机选取初始聚类中心。而对于GMM来说，如果初始高斯模型的均值和方差选取不好的话，可能会出现极大似然值为0的情况，即该样本几乎不可能由我们初始的高斯模型生成。另外在实验过程中还会出现协方差矩阵不可逆的情况。

六、参考文献

七、附录：源代码（带注释）

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
from scipy.stats import multivariate_normal
import math
import pandas as pd
plt.style.use('seaborn')
COLOR_LIST = ['red','blue','yellow','green']
config = {
    'k':3,
    'n':200,
    'dim':2,
    'mus':np.array([
        [1,3], [7,8], [8,4]
    ]),
    'sigmas':np.array([
        [[1,0],[0,2]], [[3,0],[0,2]], [[3,0],[0,1]]
    ])
```

```
}
def gen_self_data(k, n, dim, mus, sigmas):
    data = np.zeros((k, n, dim))
    for i in range(k):
        data[i] = np.random.multivariate_normal(mus[i], sigmas[i], n)
    data_ = np.zeros((k * n, dim))
    for i in range(k):
        data_[i * n:(i + 1) * n] = data[i]
    return data_

def gen_seeds_data(file_name):
    df = pd.read_csv(file_name)
    data = df.values
    return data
```

```
def kmeans(data, K, N, dim):
    classes = np.zeros((N, dim+1)) # classes array
    classes[:,0:dim] = data[:,:]
    # center is the cluster center, it has K classes
    # use a random value to initialize the K centers
    center = np.zeros((K, dim))
    for i in range(K):
        center[i,:] = data[np.random.randint(0, high=N),:]
    # K-means
    while True:
        distance = np.zeros(K) # in each iterate distance stores the distance between each data and the present cluster center
        num = np.zeros(K) # num stores how many data in each class
        new_center = np.zeros((K,dim)) # new cluster center
        # classify Expectation
        for i in range(N):
            for j in range(K):
                distance[j] = np.linalg.norm(data[i,:] - center[j,:]) # calculate the Eculidean distance between data and present center
            arg = np.argmin(distance) # the index of the min of distance
            classes[i,dim] = arg; # tag the data with class

        k = 0 # counter
        # update the new center Maximize
        for i in range(N):
            if classes[i,dim] >= K:
                print('There may be a wrong class: %d'%(K))
            else:
                c = int(classes[i,dim])
                new_center[c,:] = new_center[c,:] + classes[i,:dim]
                num[c] += 1
        for i in range(K):
            if num[i] != 0:
                new_center[i,:] /= num[i]
            distance_bias = np.linalg.norm(new_center[i,:] - center[i,:])
            print('distance bias:', distance_bias)
            if distance_bias < 1e-9:
                k += 1
                print('%d groups converged'%(k))
        # when K classes all converge then the progress is done
        if k == K:
            break
```

```
        else:
            center = new_center # update center, next iterate
    return classes
```

```
def kmeans_self():
    data = gen_self_data(config['k'], config['n'], config['dim'], config['mus'], config['sigmas'])
    K = 3
    N = np.size(data, axis=0)
    dim = 2
    classes = kmeans(data, K, N, dim)
    # scatter
    fig = plt.figure()
    ax = fig.add_subplot(111)
    for i in range(N):
        if classes[i,dim] == 0:
            ax.scatter(classes[i,0], classes[i,1], alpha=0.7, c='blue', marker='.')
        elif classes[i,dim] == 1:
            ax.scatter(classes[i,0], classes[i,1], alpha=0.7, c='green', marker='.')
        elif classes[i,dim] == 2:
            ax.scatter(classes[i,0], classes[i,1], alpha=0.7, c='red', marker='.')
        elif classes[i,dim] == 3:
            ax.scatter(classes[i,0], classes[i,1], alpha=0.7, c='yellow', marker='.')
        elif classes[i,dim] == 4:
            ax.scatter(classes[i,0], classes[i,1], alpha=0.7, c='magenta', marker='.')
    plt.show()
```

```
def e_step(data, mus, sigmas, pis, N, K, dim):
    gammas = np.zeros((N, K))
    for n in range(N):
        marginal_prob = 0
        for j in range(K):
            marginal_prob += pis[j] * multivariate_normal.pdf(data[n], mean=mus[j], cov=sigmas[j])
        for j in range(K):
            gammas[n,j] = pis[j] * multivariate_normal.pdf(data[n], mean=mus[j], cov=sigmas[j]) / marginal_prob
    return gammas

def m_step(data, gammas, mus, N, K, dim):
    mus_ = np.zeros((K, dim))
    sigmas_ = np.zeros((K, dim, dim))
    pis_ = np.zeros(K)
    for k in range(K):
        nk = 0
        for n in range(N):
            nk += gammas[n,k]
        mu_temp = np.zeros(dim)
        for n in range(N):
            mu_temp += gammas[n,k] * data[n]
        mus_[k] = mu_temp / nk
```

```

        sigma_temp = np.zeros(dim)
        for n in range(N):
            dis = data[n] - mus[k] # one-dimension array can not be transposed
            sigma_temp += dis**2 * gammas[n,k]
        sigma_temp_ = np.eye(dim)
        sigma_temp_[0,0] = sigma_temp[0]
        sigma_temp_[1,1] = sigma_temp[1]
        sigmas_[k] = sigma_temp_ / nk

        pis_[k] = nk / N
    return mus_, sigmas_, pis_
```

```
def log_likelihood(data, mus, sigmas, pis, N, K, dim):
    l = 0
    for n in range(N):
        temp = 0
        for k in range(K):
            temp += pis[k] * multivariate_normal.pdf(data[n], mean=mus[k], cov=sigmas[k])
        l += math.log(temp)
    return l
```

```
def show_gmm_result(data, mus, sigmas, classes, real_ce):
    fig = plt.figure()
    ax = plt.subplot()
    K = np.size(mus, 0)
    N = np.size(data, 0)
    for i in range(K):
        ellipse = Ellipse(
            xy=mus[i], width=3*sigmas[i,0,0], height=3*sigmas[i,1,1], edgecolor=COLOR_LIST[i], lw=2, fill=False)
        ax.add_patch(ellipse)
    if real_ce:
        for i in range(K):
            ellipse = Ellipse(
                xy=config['mus'][i], width=3*config['sigmas'][i,0,0], height=3*config['sigmas'][i,1,1], color=COLOR_LIST[i], alpha=0.2)
            ax.add_patch(ellipse)
    for i in range(N):
        plt.scatter(data[i,0], data[i,1], marker='.', color=COLOR_LIST[int(classes[i])])
#     plt.scatter(data[:,0], data[:,1], marker='.', color=COLOR_LIST[classes])
    plt.show()
```

```
def gmm_self():
    data = gen_self_data(config['k'], config['n'], config['dim'], config['mus'], config['sigmas'])
    # print(data)
    N = np.size(data, axis=0)
    K = 3
    dim = 2
    mus = np.array([
```

```
        [3,3], [4,4], [5,5]
    ])
    sigmas = np.array([
        [[1,0],[0,1]]
    ] * K)
    pis = np.array([1 / K] * K)
    epoch = 100
    eps = 1e-3

    classes = np.zeros(N)
    for i in range(epoch):
        old_loss = log_likelihood(data, mus,sigmas, pis, N, K, dim)
        gammas = e_step(data, mus, sigmas, pis, N, K, dim)
        mus, sigmas, pis = m_step(data, gammas, mus, N, K, dim)
        new_loss = log_likelihood(data, mus,sigmas, pis, N, K, dim)
        if i % 10 == 0:
            print(i, new_loss)
            argmaxs = np.argmax(gammas, axis=1)
            for ii in range(N):
                classes[ii] = argmaxs[ii]
            show_gmm_result(data, mus, sigmas, classes, False)
        if (abs(new_loss - old_loss) < eps):
            break
    argmaxs = np.argmax(gammas, axis=1)
    for ii in range(N):
        classes[ii] = argmaxs[ii]
    show_gmm_result(data, mus, sigmas, classes, True)
```

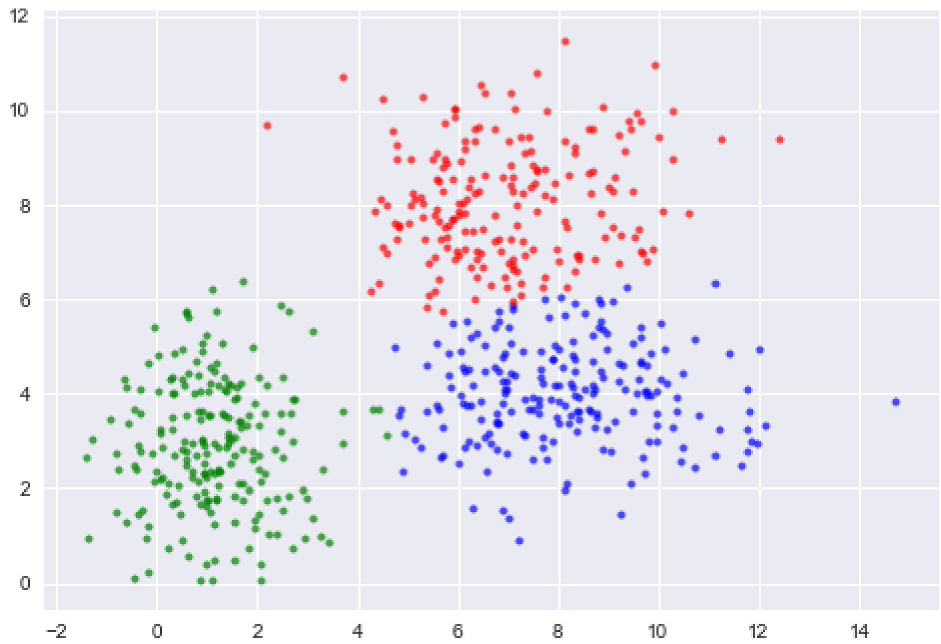
```
def uci():
#     data = frog_gen_data('./Frogs_MFCCs.csv')
    data = gen_seeds_data('./seeds_dataset.csv')
    labels = data[:,-1]
    np.delete(data, -1, axis=1)
    N = np.size(data, axis=0)
    dim = np.size(data, axis=1)
    K = 3
    classes = kmeans(data, K, N, dim)
    mus = np.array([
        [0] * dim
    ] * K)
    sigmas = np.array([
        np.eye(dim)
    ] * K)
    temp = np.array([
        [0] * dim
    ] * K)
    temp_counts = np.array([0] * K)
    for i in range(N):
        c = int(classes[i,-1])
        temp_counts[c] += 1
        for j in range(dim):
            temp[c,j] += classes[i,j]
    for i in range(K):
        for j in range(dim):
            mus[i,j] = temp[i,j] / temp_counts[i]
    for i in range(K):
#         c = int(temp_counts[i])
```

```
            for j in range(dim):
                for k in range(N):
                    if classes[i,-1] == i:
                        sigmas[i,j,j] += pow((classes[i,j] - mus[i,j]), 2)
                sigmas[i,j,j] /= temp_counts[i]
        pis = np.array([1 / K] * K)
        epoch = 100
        eps = 1e-3
        gmm_classes = np.zeros(N)
        for i in range(epoch):
            old_loss = log_likelihood(data, mus,sigmas, pis, N, K, dim)
            gammas = e_step(data, mus, sigmas, pis, N, K, dim)
            mus, sigmas, pis = m_step(data, gammas, mus, N, K, dim)
            new_loss = log_likelihood(data, mus,sigmas, pis, N, K, dim)
            if i % 10 == 0:
                print(i, new_loss)
                argmaxs = np.argmax(gammas, axis=1)
                for ii in range(N):
                    classes[ii] = argmaxs[ii]
            if (abs(new_loss - old_loss) < eps):
                break
        argmaxs = np.argmax(gammas, axis=1)
        for ii in range(N):
            gmm_classes[ii] = argmaxs[ii]
        classes_count = np.zeros(K)
        for i in range(N):
            classes_count[int(gmm_classes[i])] += 1
        print(classes_count)
```

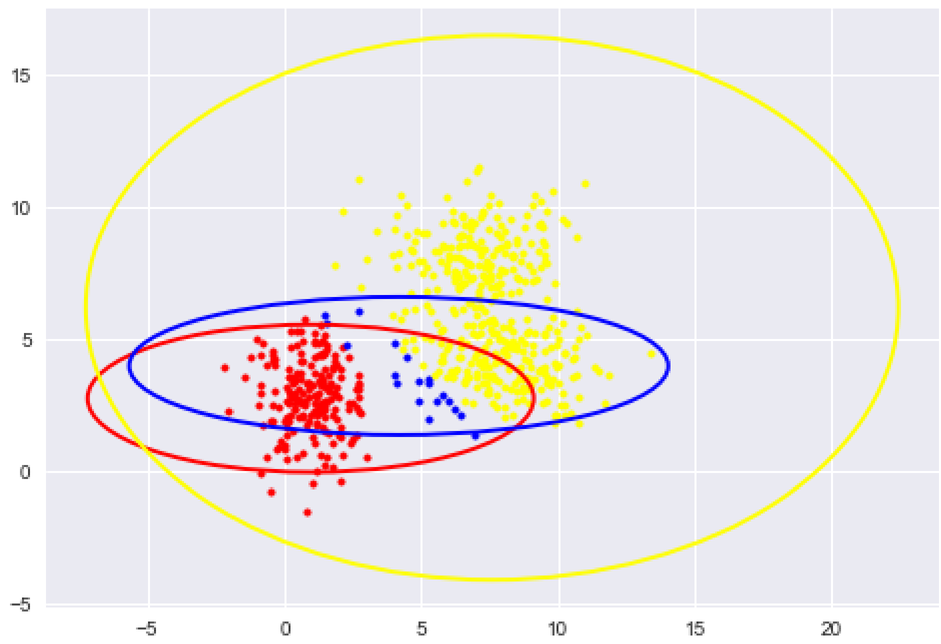
```
kmeans_self()
gmm_self()
uci()
```

```
distance bias: 3.083324312796322
distance bias: 2.972872294784163
distance bias: 0.39381867341463334
distance bias: 0.9704027729129306
distance bias: 1.6252317891992243
distance bias: 0.7972332267189581
distance bias: 0.6401858744851695
distance bias: 1.9718213279065842
distance bias: 0.6494934395673909
distance bias: 0.6028800089308306
distance bias: 0.9922104872872277
distance bias: 0.28004531054930387
distance bias: 1.1796145340363111
distance bias: 0.2886885356230116
distance bias: 0.06191595338758368
distance bias: 1.1163478530637279
distance bias: 0.09693196578158009
distance bias: 0.33583719948663376
distance bias: 1.1101375155934743
distance bias: 0.06308340360750482
distance bias: 0.810865292660083
distance bias: 0.4277347404588608
distance bias: 0.01617850862080447
```

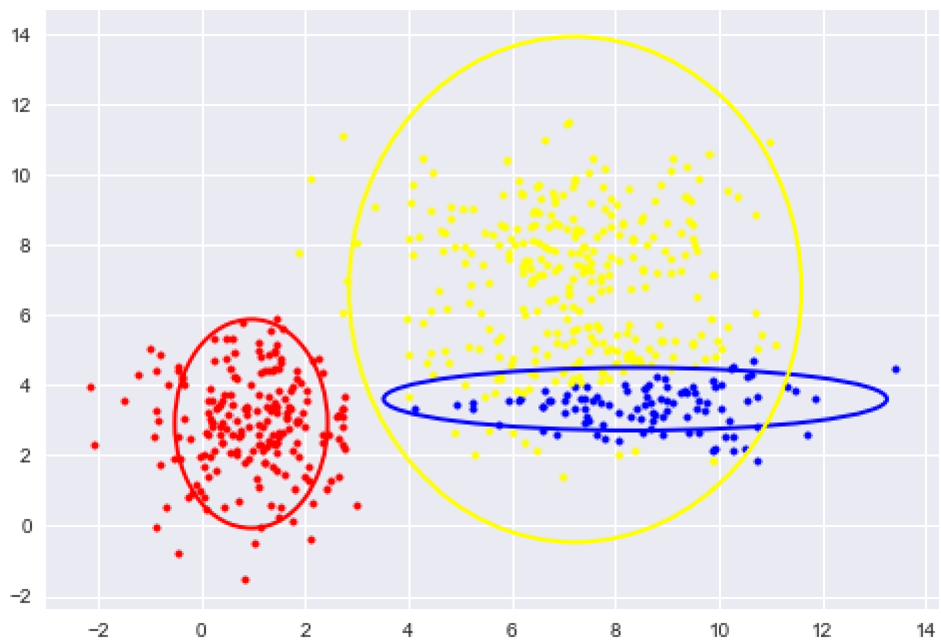
```
distance bias: 0.585744619426924
distance bias: 0.12618076253570384
distance bias: 0.03343857130709641
distance bias: 0.1324460281563676
distance bias: 0.06425522480472237
distance bias: 0.0
1 groups converged
distance bias: 0.07076655826196006
distance bias: 0.010418282879045436
distance bias: 0.0
1 groups converged
distance bias: 0.011771445508246074
distance bias: 0.0
1 groups converged
distance bias: 0.0
2 groups converged
distance bias: 0.0
3 groups converged
```



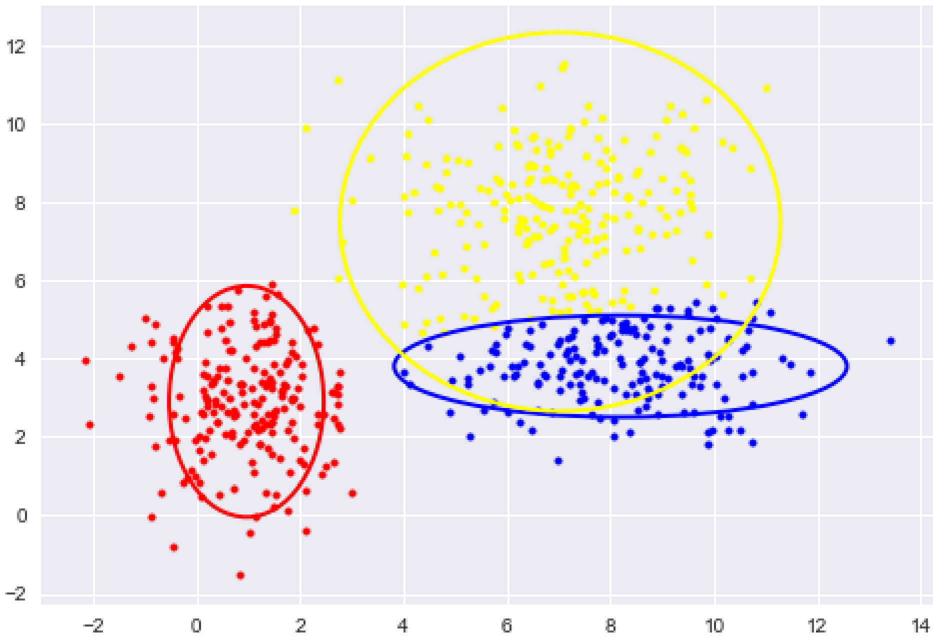

```
0 -2877.3663996465784
```



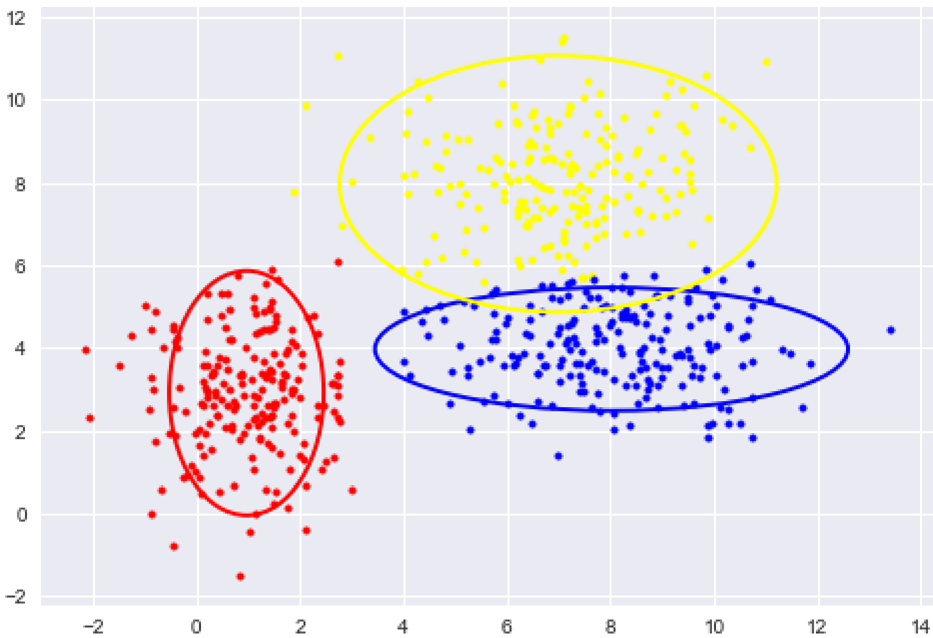
```
10 -2682.532886407031
```



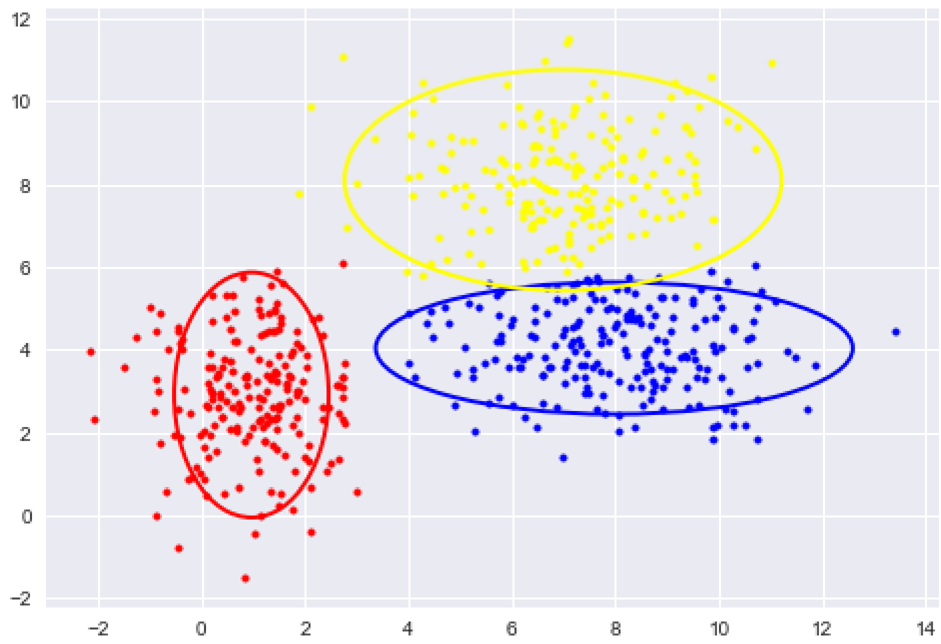
```
20 -2667.972207762223
```

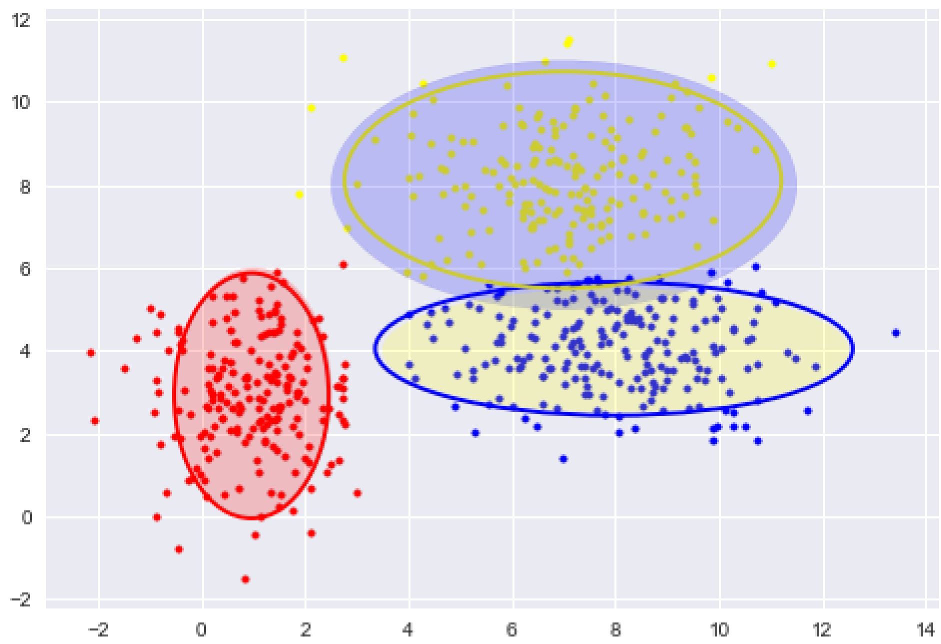



```
30 -2661.6199175566203
```




```
40 -2660.8746225441946
```






```
distance bias: 0.8217318680446959
distance bias: 0.9308262798579227
distance bias: 1.2717002187885862
distance bias: 0.8480798919941978
distance bias: 0.1637473971976366
distance bias: 1.3268732183141505
distance bias: 0.45564950396648163
distance bias: 0.5566088002629445
distance bias: 0.6643525920283189
distance bias: 0.12594429368098323
distance bias: 0.2026175820387384
distance bias: 0.3086453453165971
distance bias: 0.07944495115512543
distance bias: 0.15647780583361648
distance bias: 0.22470875878830027
distance bias: 0.0
1 groups converged
distance bias: 0.06218726955475801
distance bias: 0.059648809672817944
```

```
distance bias: 0.0
1 groups converged
distance bias: 0.0
2 groups converged
distance bias: 0.0
3 groups converged
0 -7190.317449853984
10 -9533.006152887352
20 -9147.862208694027
30 -9129.631863621678
[71. 65. 73.]
```