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：必修
实验题目：GMM模型

学号：1181000420
姓名：韦昆杰

一、实验目的

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）。

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。

实验环境：

- Windows10
- PyCharm
- VSCode

三、设计思想(本程序中的用到的主要算法及数据结构)

3.1 k-means算法

3.1.1 算法原理

给定样本集$D=\{x_1,x_2,...,x_m\}$，$k-means$算法针对聚类所得簇划分$C=\{C_1,C_2,...,C_K\}$最小化平方误差

$$E=\sum_{i=1}^{k}\sum_{\mathbf{x}\in\mathbf{C_i}}||\mathbf{x}-\mu_\mathbf{i}||_2^2 \tag{1}$$

其中$u_i=\frac{1}{|C_i|}\sum_{\mathbf{x}\in C_i}\mathbf{x}$是簇$C_i$的均值向量，式$(1)$刻画了簇内样本围绕簇均值向量的紧密程度，$E$值越小则簇内样本相似度越高

然而我们直接求E的最小值并不容易，这是一个NP难的问题，因此k均值算法采用贪心策略，通过迭代优化来近似求解。

迭代优化的策略如下：

1. 从D中随机选择k个样本作为初始均值向量

2. 根据初始均值向量给出样本集D的一个划分，样本距离那个簇的均值向量距离最近，则将该样本划入相应的簇
3. 再根据当前簇划分来计算每个簇的新均值向量，如果新均值向量与当前均值向量相同，假设正确，保持不变；否则，更新均值向量，继续迭代求解直到当前均值向量均未更新。

3.1.2 算法实现

本实验中，产生k个中心点的python代码如下：

```
centers = mean + std * np.random.randn(k, dimension)  # (k,dimension)
```

k-means算法欧几里得距离的计算python代码如下：

```
# 计算每个样本离每个中心点的距离
for j in range(k):
  distances[:, j] = np.linalg.norm(data - centers[j], axis=1)
```

计算样本类别的python代码如下：

```
# 样本对应的类别为距离最近的中心点
clusters = np.argmin(distances, axis=1)
```

更新样本类别的python代码如下：

```
# 更新每个类别的中心点
for j in range(k):
  centers[j] = np.mean(data[clusters == j], axis=0)
```

3.2 混合高斯模型

3.2.1 算法原理

许多数据集都可以通过高斯分布来刻画，因此我们假设聚类结构中不同的簇遵循不同的高斯分布，以此来通过概率模型来表达聚类原型

若样本X满足一维高斯分布，其概率密度函数为

$$G(X|\mu,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2} \tag{2}$$

其中μ为均值，σ为方差
若样本X满足多维高斯分布，其概率密度函数为

$$G(X|\mu,\Sigma)=\frac{1}{\sqrt{(2\pi)|\boldsymbol{\Sigma}|}}\exp\left(-\frac{1}{2}(X-\mu)^T\boldsymbol{\Sigma}^{-1}(X-\mu)\right) \tag{3}$$

其中μ为n维均值向量，Σ为协方差矩阵

假设当前有K个簇，我们可以定义其高斯混合分布

$$p(X)=\sum_{k=1}^{K} \pi_k G\left(X \mid \mu_k, \Sigma_k\right) \tag{4}$$

其中π_k是高斯混合分布的混合系数

对于式(4)，我们采用极大似然估计求解，即最大化(对数)似然

$$\begin{aligned} & \ln p(X \mid \mu, \Sigma, \pi) \\ = & \sum_{i=1}^{N} p\left(X_i\right) \\ = & \sum_{i=1}^{N} \ln \sum_{k=1}^{K} \pi_k G\left(X_i \mid \mu_k, \Sigma_k\right) \end{aligned} \tag{5}$$

下面我们定义$\gamma_k(X)=p(k \mid X)$
根据贝叶斯公式

$$\begin{aligned} & \gamma_k(X) \\ = & \frac{p(X \mid k) p(k)}{\sum_{k=1}^{K} p(k) p(X \mid k)} \\ = & \frac{p(X \mid k) \pi_k}{\sum_{k=1}^{K} \pi_k p(X \mid k)} \end{aligned} \tag{6}$$

为了最大化对数似然函数，其对应μ_k, Σ_k, π_k偏导必须全为0，这样我们就分别得到

$$\mu_k=\frac{\sum_{n=1}^{N} \gamma_k\left(x_n\right) x_n}{\sum_{n=1}^{N} \gamma_k\left(x_n\right)} \tag{7}$$

$$\Sigma_k=\frac{\sum_{n=1}^{N} \gamma_k\left(x_n\right)\left(x_n-\mu_k\right)\left(x_n-\mu_k\right)^T}{\sum_{n=1}^{N} \gamma_k\left(x_n\right)} \tag{8}$$

$$\pi_k=\frac{1}{N} \sum_{n=1}^{N} \gamma_k\left(x_n\right) \tag{9}$$

由上述推导即可获得高斯混合模型的EM算法：每次迭代中，现根据当前参数计算样本属于每个高斯分布的后验概率$\gamma_k(X)$，再根据式(7)、(8)、(9)更新模型参数μ_k, Σ_k, π_k

3.2.2 算法实现

GMM首先需要初始化模型参数，对应python代码如下：

```
# 初始化均值向量
mu = get_centers(data, K)
# 初始化协方差矩阵
cov = np.zeros((K, dimension, dimension))
for i in range(K):
    cov[i, :, :] = np.identity(dimension) / 10
# 初始化混合系数
alpha = np.ones(size) / size
# 初始化后验概率矩阵
gamma = np.zeros((size, K))
```

Expectation step对应的python代码如下：

```
for k in range(K):
    # Expectation step
    # 计算后验概率矩阵
    gamma[:, k] = alpha[k] * multivariate_normal(mu[k], cov[k]).pdf(data)
sum = np.sum(gamma, axis=1).reshape(-1, 1)
gamma /= sum
```

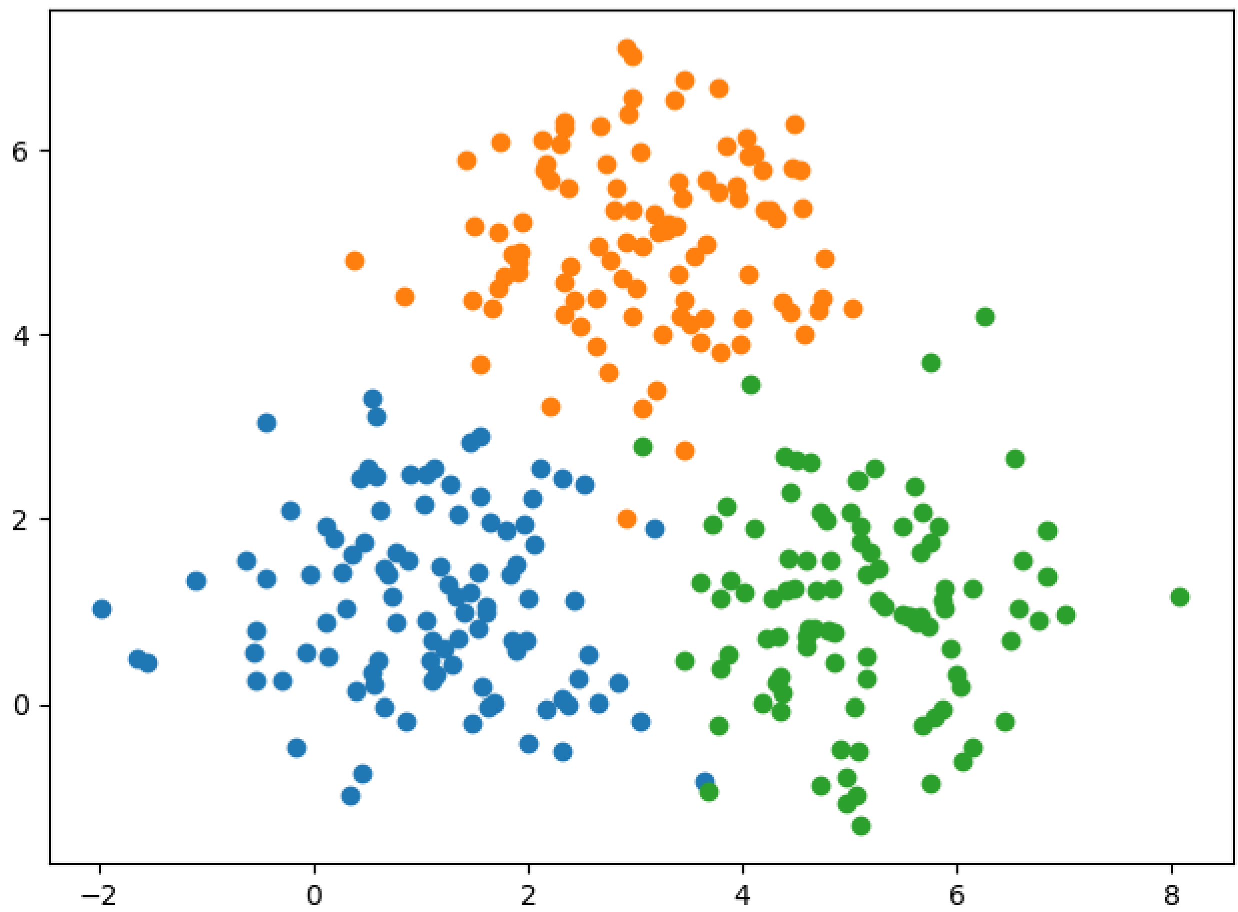
Maximization step对应的python代码如下：

```
# Maximization step
for k in range(K):
    gamma_k = np.sum(gamma[:, k], axis=0)
    # 计算新均值向量
    mu[k] = np.sum(gamma[:, k].reshape(-1, 1) * data, axis=0) / gamma_k
    # 计算新协方差矩阵
    cov[k] = (gamma[:, k].reshape(-1, 1) * (data - mu[k])).T @ (data - mu[k]) / gamma_
k
    # 计算新混合系数
    alpha[k] = gamma_k / K
```

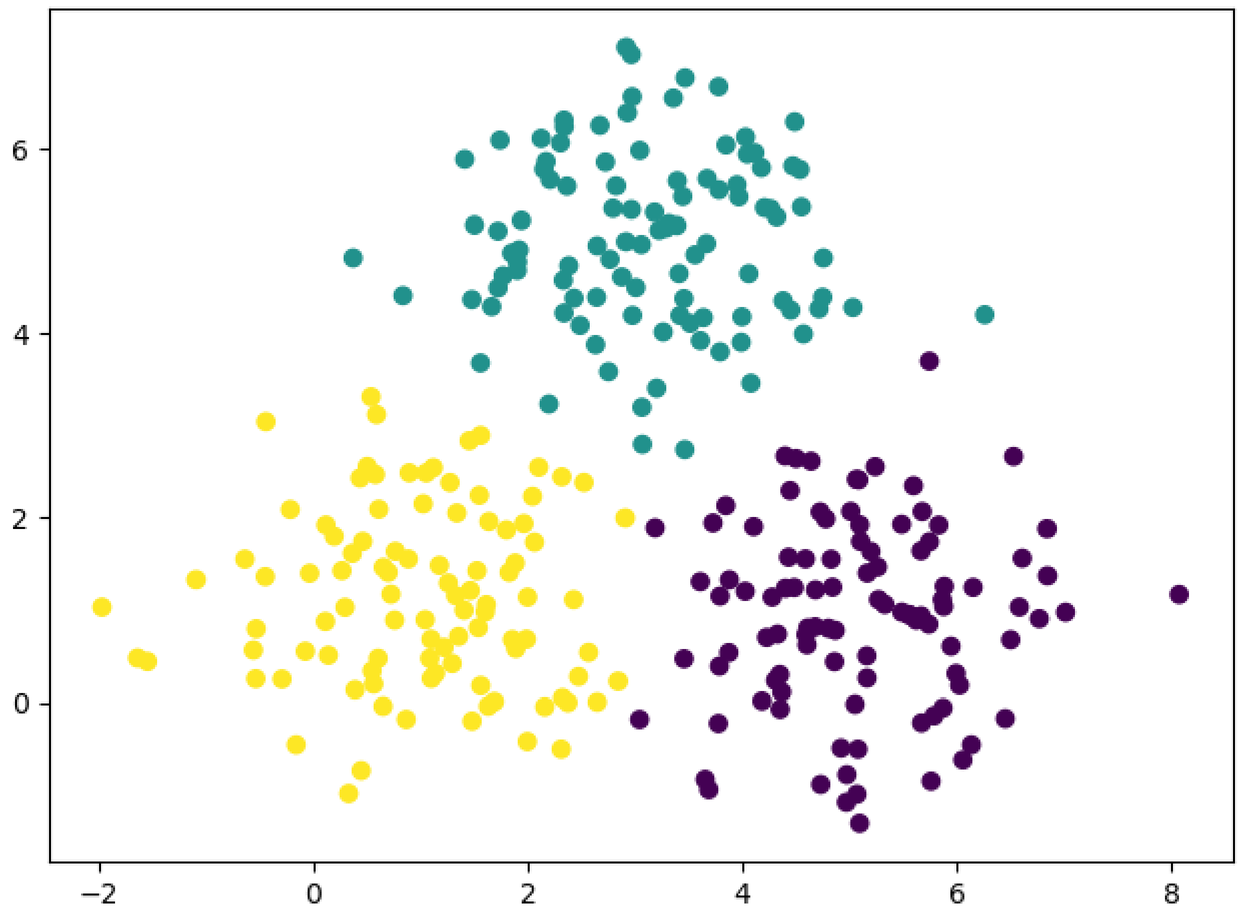
四、实验结果分析

4.1 k-means算法

首先生成数据，k=3，共生成300个二维数据点



然后使用k-means算法对生成数据进行分类，分类的结果如下：

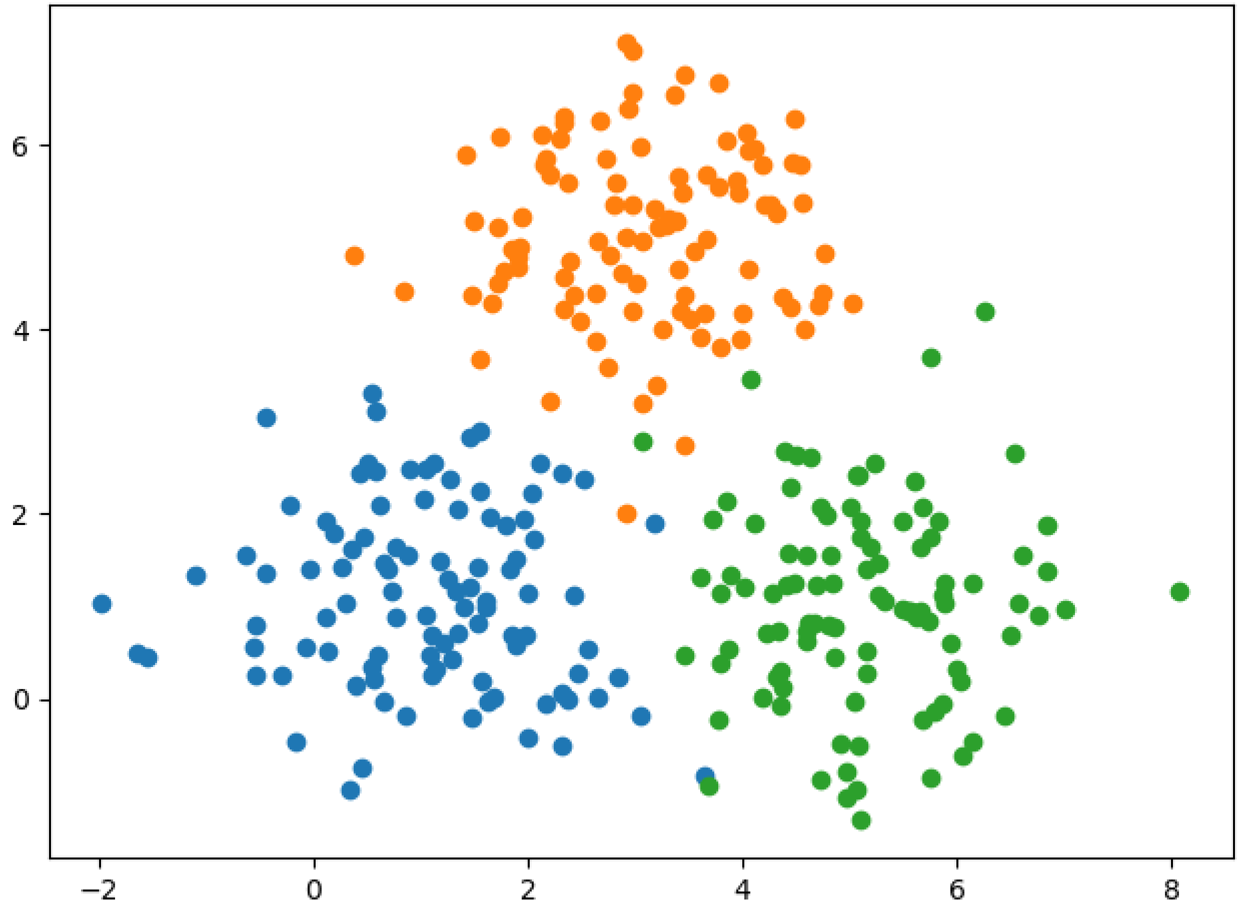


由于数据是由我们自行生成的，因此我们知道真实数据属于哪一类，所以我们可以求出k-means算法对生成数据的分类准确率：

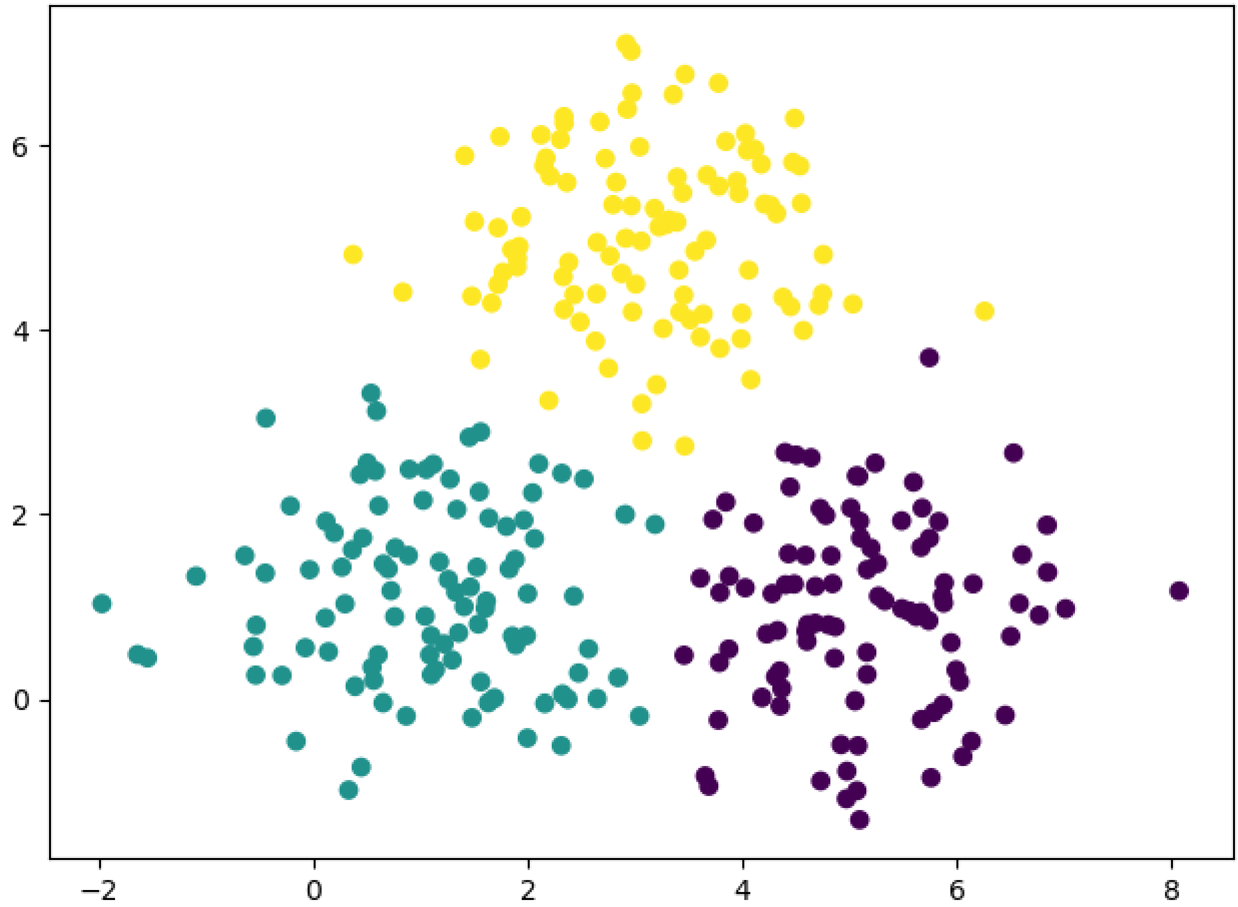
```
通过k-means算法对生成数据的分类准确率为0.9700000000000001
```

4.2 混合高斯模型

首先生成数据，k=3，共生成300个二维数据点



然后使用混合高斯模型对生成数据进行分类，分类的结果如下：



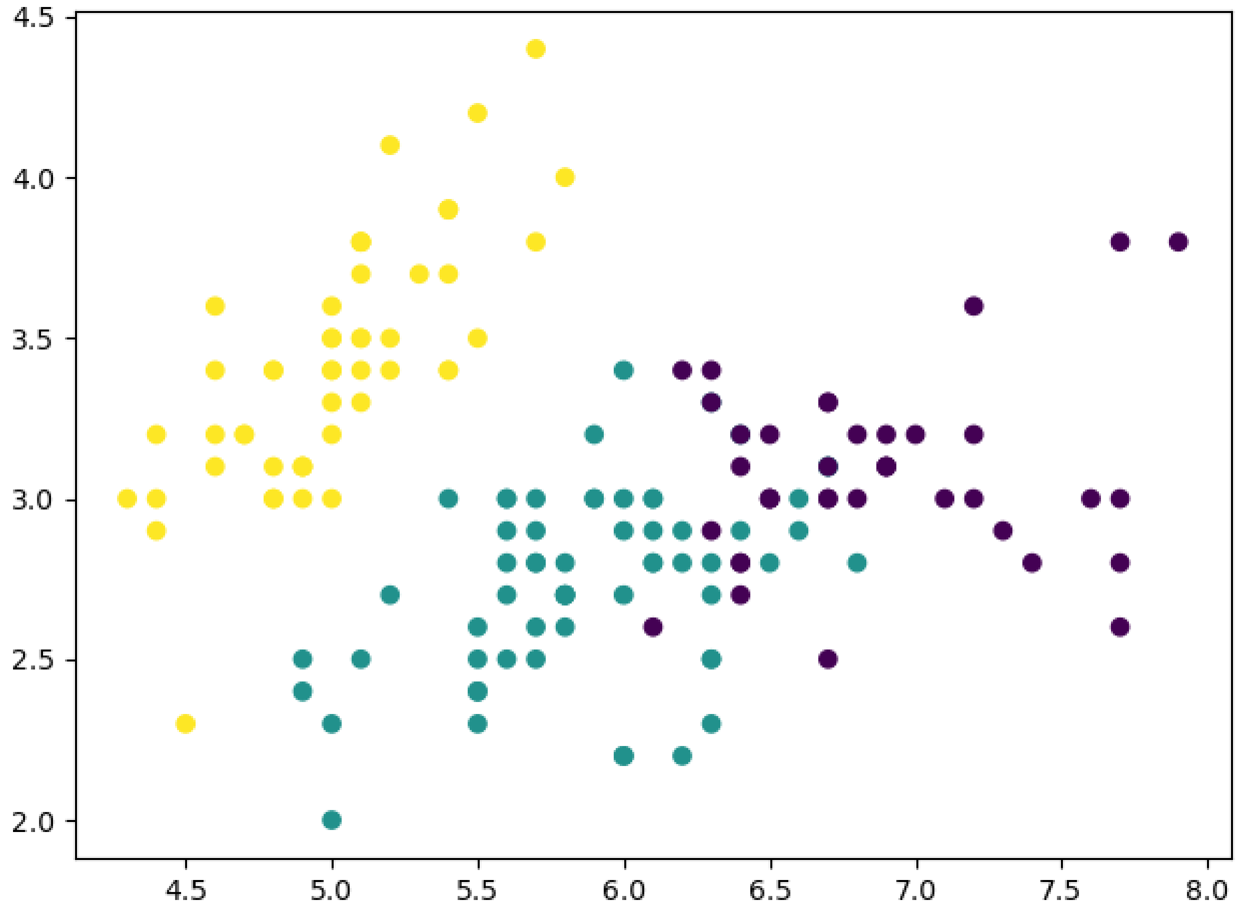
由于数据是由我们自行生成的，因此我们知道真实数据属于哪一类，所以我们可以求出混合高斯模型对生成数据的分类准确率：

```
通过GMM算法对生成数据的分类准确率为0.9899999999999999
```

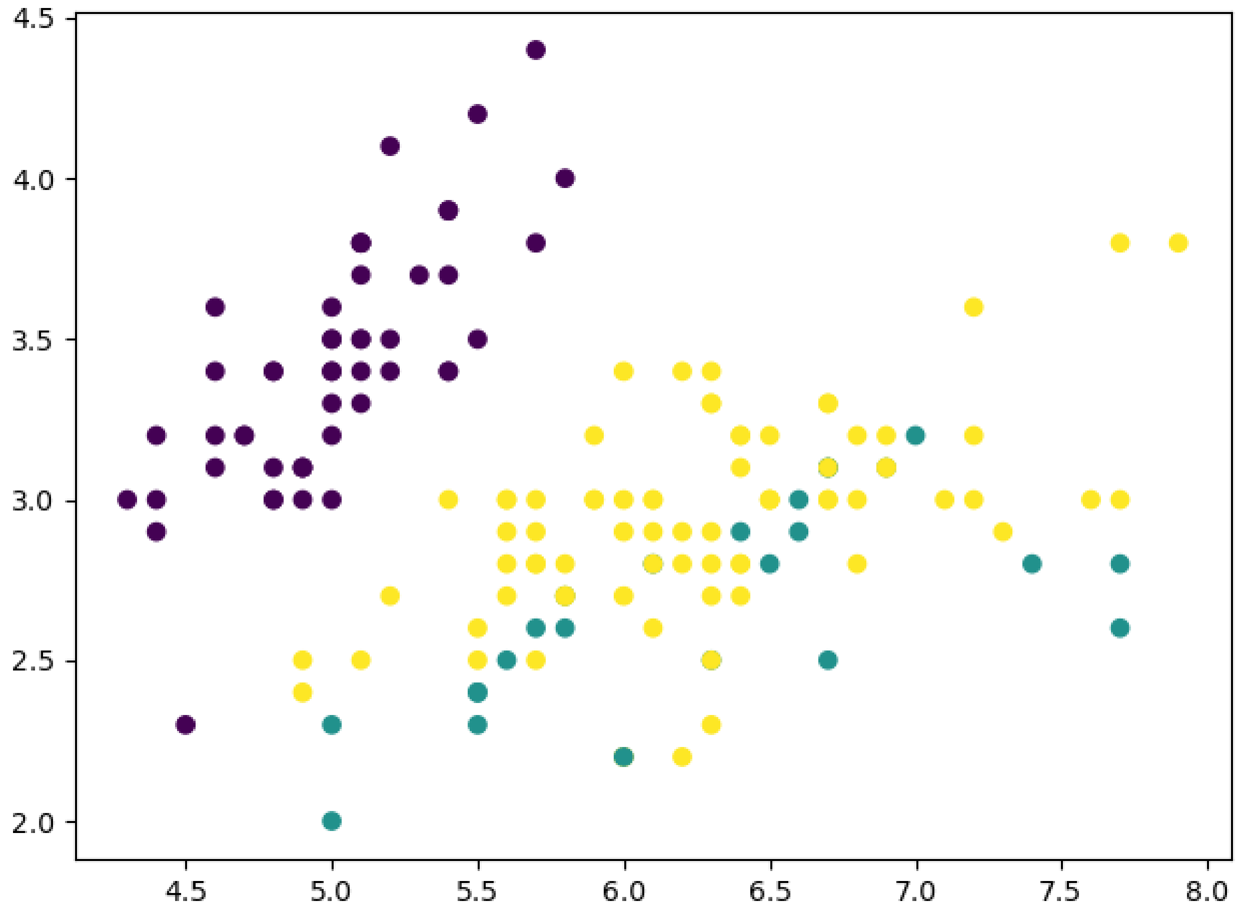
4.3 uci网站iris数据分类（k-means算法 混合高斯模型）

> 由于iris数据是多维的，这里展示前两维的分类效果

使用k-means算法对uci网站iris数据进行分类，分类的结果如下



使用混合高斯模型对uci网站iris数据进行分类，分类的结果如下



因为iris数据最后一列有真实数据的分类

```
5.0,3.5,1.3,0.3,Iris-setosa
4.5,2.3,1.3,0.3,Iris-setosa
4.4,3.2,1.3,0.2,Iris-setosa
5.0,3.5,1.6,0.6,Iris-setosa
5.1,3.8,1.9,0.4,Iris-setosa
4.8,3.0,1.4,0.3,Iris-setosa
5.1,3.8,1.6,0.2,Iris-setosa
4.6,3.2,1.4,0.2,Iris-setosa
5.3,3.7,1.5,0.2,Iris-setosa
5.0,3.3,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
5.7,2.8,4.5,1.3,Iris-versicolor
6.3,3.3,4.7,1.6,Iris-versicolor
4.9,2.4,3.3,1.0,Iris-versicolor
6.6,2.9,4.6,1.3,Iris-versicolor
5.2,2.7,3.9,1.4,Iris-versicolor
5.0,2.0,3.5,1.0,Iris-versicolor
5.9,3.0,4.2,1.5,Iris-versicolor
```

因此我们可以求出k-means算法和混合高斯模型对iris数据的分类准确率：

```
通过k-means算法对uci数据的分类准确率为1.0
通过GMM算法对生成数据的分类准确率为1.0
```

可以看出，k-means算法和混合高斯模型都实现了分类准确率100%

五、结论

- k-means算法利用数据点之间的欧式距离大小，将数据划分到不同的类别，欧式距离较短的点处于同一类。

- GMM假设所有数据点来自多个参数不同的高斯分布，来自同一分布的数据点被划分为同一类。
- 通过k-means算法和GMM算法都可以实现聚类，但是两个算法在簇中心初始化不好时，容易陷入局部最优解，导致聚类效果不佳。
- GMM可以选择随机点作为初始均值向量，也可以用k-means算法的结果作为初始均值向量。

六、参考文献

- Christopher Bishop. Pattern Recognition and Machine Learning.
- 周志华 著. 机器学习, 北京: 清华大学出版社, 2016.1
- UCI Iris

七、附录:源代码(带注释)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
import pandas as pd


def generate_data():
    mean1 = [1, 1]
    cov1 = [[1, 0], [0, 1]]
    x1, y1 = np.random.multivariate_normal(mean1, cov1, 100).T
    mean2 = [3, 5]
    cov2 = [[1, 0], [0, 1]]
    x2, y2 = np.random.multivariate_normal(mean2, cov2, 100).T
    mean3 = [5, 1]
    cov3 = [[1, 0], [0, 1]]
    x3, y3 = np.random.multivariate_normal(mean3, cov3, 100).T
    plt.scatter(x1, y1)
    plt.scatter(x2, y2)
    plt.scatter(x3, y3)
    plt.show()

    x = np.concatenate((x1, x2, x3))
    y = np.concatenate((y1, y2, y3))
    data = np.zeros((len(x), 2))
    data[:, 0] = x
    data[:, 1] = y
    return data


# 根据本实验生成数据，求出分类的准确率
def cluster_score(clusters, length=100):
    score = 0
    for k in range(3):
        zero_count = 0
```

```python
        one_count = 0
        two_count = 0
        for i in range(length):
            if clusters[i] == 0:
                zero_count += 1
            if clusters[i] == 1:
                one_count += 1
            if clusters[i] == 2:
                two_count += 1
        sum = zero_count + one_count + two_count
        if zero_count == max(zero_count, one_count, two_count):
            score += zero_count / sum
        if one_count == max(zero_count, one_count, two_count):
            score += one_count / sum
        if two_count == max(zero_count, one_count, two_count):
            score += two_count / sum
    return score / 3


def get_centers(data, k):
    dimension = data.shape[1]
    mean = np.mean(data, axis=0).reshape((1, dimension))
    std = np.std(data, axis=0).reshape((1, dimension))
    centers = mean + std * np.random.randn(k, dimension)  # (k,dimension)
    return centers


def k_means(data, k=3, iteration_times=1000):
    size = data.shape[0]
    centers = get_centers(data, k)
    distances = np.zeros((size, k))
    pre_clusters = np.zeros((size, 1))
    for i in range(iteration_times):
        # 计算每个样本离每个中心点的距离
        for j in range(k):
            distances[:, j] = np.linalg.norm(data - centers[j], axis=1)
        # 样本对应的类别为距离最近的中心点
        clusters = np.argmin(distances, axis=1)
        # 如果两次簇划分结果相同，停止迭代，返回结果
        if np.array_equal(clusters, pre_clusters):
            return clusters
        # 更新每个类别的中心点
        for j in range(k):
            centers[j] = np.mean(data[clusters == j], axis=0)
        pre_clusters = np.copy(clusters)
    return clusters


def GMM(data, K=3, iteration_times=1000):
    size, dimension = data.shape
    # 初始化均值向量
    mu = get_centers(data, K)
    # 初始化协方差矩阵
    cov = np.zeros((K, dimension, dimension))
    for i in range(K):
```

```python
            cov[i, :, :] = np.identity(dimension) / 10
    # 初始化混合系数
    alpha = np.ones(size) / size
    # 初始化后验概率矩阵
    gamma = np.zeros((size, K))
    for i in range(iteration_times):
        for k in range(K):
            # Expectation step
            # 计算后验概率矩阵
            gamma[:, k] = alpha[k] * multivariate_normal(mu[k], cov[k]).pdf(data)
        sum = np.sum(gamma, axis=1).reshape(-1, 1)
        gamma /= sum

        # Maximization step
        for k in range(K):
            gamma_k = np.sum(gamma[:, k], axis=0)
            # 计算新均值向量
            mu[k] = np.sum(gamma[:, k].reshape(-1, 1) * data, axis=0) / gamma_k
            # 计算新协方差矩阵
            cov[k] = (gamma[:, k].reshape(-1, 1) * (data - mu[k])).T @ (data - mu[k])
/ gamma_k
            # 计算新混合系数
            alpha[k] = gamma_k / K
    return mu, cov, alpha


def GMM_predict(data, K=3, iteration_times=1000):
    size, dimension = data.shape
    mu, cov, alpha = GMM(data, K, iteration_times)
    gamma = np.zeros((size, K))
    for k in range(K):
        # Expectation step
        # 计算后验概率矩阵
        gamma[:, k] = alpha[k] * multivariate_normal(mu[k], cov[k]).pdf(data)
    sum = np.sum(gamma, axis=1).reshape(-1, 1)
    gamma /= sum
    return np.argmax(gamma, axis=1)


if __name__ == '__main__':
    data = generate_data()
    clusters = k_means(data)
    print(f'通过k-means算法对生成数据的分类准确率为{cluster_score(clusters)}')
    plt.scatter(data[:, 0], data[:, 1], c=clusters)
    plt.show()
    clusters = GMM_predict(data)
    print(f'通过GMM算法对生成数据的分类准确率为{cluster_score(clusters)}')
    plt.scatter(data[:, 0], data[:, 1], c=clusters)
    plt.show()

    # read data from uci
    data = pd.read_csv('iris.data', header=None).iloc[:, :-1]
    data = pd.DataFrame(data, dtype=float)
    data = np.array(data)
```

```
    clusters = k_means(data)
    print(f'通过k-means算法对uci数据的分类准确率为{cluster_score(clusters,length=50)}')
    plt.scatter(data[:, 0], data[:, 1], c=clusters)
    plt.show()

    clusters = GMM_predict(data)
    print(f'通过GMM算法对生成数据的分类准确率为{cluster_score(clusters,length=50)}')
    plt.scatter(data[:, 0], data[:, 1], c=clusters)
    plt.show()
```