**智能信息系统综合实践**

**实验报告**

|  |  |
| --- | --- |
| **题 目：** | **聚类** |
| **年 级：** | **2021** |
| **专 业：** | **软件工程** |
| **学 号：** | **2021117405** |
| **姓 名：** | **孙潇桐** |

目录

[1. 题目 3](#_Toc163229191)

[2. 解题步骤 3](#_Toc163229192)

[2.1. 题目一：手动实现K-Means聚类 3](#_Toc163229193)

[2.1.1. 对数据的处理 3](#_Toc163229194)

[2.1.2. 使用不同的距离函数对样本进行聚类 3](#_Toc163229195)

[2.1.3. 聚类结果的展示 7](#_Toc163229196)

[2.1.4. 聚类结果分析 13](#_Toc163229197)

[2.2. 题目二：手动实现DBSCAN聚类 13](#_Toc163229198)

[2.2.1. DBSCAN的实现 13](#_Toc163229199)

[2.2.2. 使用DBSCAN聚类6个数据集的展示 16](#_Toc163229200)

[2.2.3. 对DBSCAN聚类的观察 18](#_Toc163229201)

[2.3. 将K-Means与DBSCAN以及GMM在6个数据集上进行对比 18](#_Toc163229202)

[2.3.1. GMM聚类的原理 18](#_Toc163229203)

[2.3.2. GMM聚类的结果展示 20](#_Toc163229204)

[2.3.3. 聚类结果的分析方法 22](#_Toc163229205)

[2.3.4. 聚类结果的展示和算法间的比较 23](#_Toc163229206)

[3. 总结 26](#_Toc163229207)

[4. 附件 27](#_Toc163229208)

[4.1. 完整代码 27](#_Toc163229209)

[4.1.1. 不调库实现K-Means分类器：kmeans.py 27](#_Toc163229210)

[4.1.2. 不调库实现DBSCAN分类器：dbscan.py 30](#_Toc163229211)

[4.1.3. 使用sklearn提供的库测试GMM分类器：gmm.py 32](#_Toc163229212)

[4.1.4. 测试上面提到的分类器，画图并计算评价系数：test.py 33](#_Toc163229213)

# 题目

1. 不调用工具包，实现K-Means聚类，尝试不同K值和距离度量，可视化聚类结果。
2. 不调用工具包，实现DBSCAN，并在这6个数据集上运行可视化聚类结果。
3. 将K-Means与DBSCAN在6个数据集上进行对比，可视化聚类结果并使用至少两种sklearn中的聚类评估指标进行评判。有能力的同学可以增加其他的聚类算法（如GMM等）

# 解题步骤

## 题目一：手动实现K-Means聚类

### 对数据的处理

这次的数据集比较多，足足有6个，所以我将这些数据集放在同一个文件夹。在计算的时候只需要遍历存放数据的文件夹就可以得到所有数据，不过需要注意的是os.listdir() 这个方法返回的只是文件的名字，并不是具体的路径。在读取的时候还需要文件名和目录的名字拼起来，不然的话np.loadtxt() 会因为工作目录不在存储数据集的文件夹而找不到我们的数据文件，下面是我的实现方式：

# 遍历data文件夹下所有数据集

for file in os.listdir(directory):

    # 拼接完整地址并读取当前数据集

    data = np.loadtxt(directory + file)

经过这样的处理，我们就可以很方便的访问所有的数据集。

### 使用不同的距离函数对样本进行聚类

距离度量是K-Means 聚类非常重要的一环，因为K-Means是通过计算样本与聚类中心的距离来判断当前样本属于的类别。接下来我将展示使用不同距离度量函数的K-Means聚类在全部6个数据集上所以选择不同的距离度量就会对最后聚类的结果产生一些的影响。为了探究不同距离度量和不同K值对聚类的具体影响，接下来我将展示使用不同距离度量函数分别使用不同的K值在全部数据集上的K-Means聚类表现。

K-Means的名字就告诉我们了，这个聚类方法的聚类依据是均值(means)。我们只需要不断的计算每个类别的平均点作为聚类中心，直到两次迭代的差小于阈值，我在这次实验中设置的阈值是曼哈顿距离小于，下面是我画的流程图：

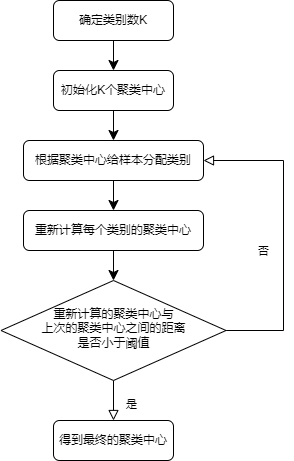


Figure 1 K-Means聚类流程图

为了能够更方便的适配不同的距离算法，我使用了函数引用的方式在计算类中心点的时候动态的将计算函数（实参：dist\_fun）传入训练的函数，下面是我的具体实现：

# K-Means聚类函数，计算聚类中心

def kmenas\_train(data, dist\_fun, k):

    # 选择前k个作为聚类中心

    centers = np.array(data[:k])

    while True:

        # 建立一个新数组存储中心点

        center\_new = np.zeros(centers.shape)

        # 根据聚类中心计算每个样本属于的类别

        res = dist\_fun(centers, data)

        # 计算新的聚类中心

        for idx in range(k):

            center\_new[idx] = np.mean(data[res == idx], axis=0)

        # 如果两次训练之间的聚类中心没有变化则停止

        if np.sum(np.abs(centers - center\_new)) < 1e-10:

            break

        centers = center\_new

    return centers

上面的训练函数有3个参数，每个参数都是需要改变的，只需要写一个三重循环就可以完成所有的测试，下面是我的训练和画图的过程：

# 距离函数列表

list\_dist\_fun = [Euclidean\_Distance, Manhattan\_Distance, Cosine\_Similarity]

# 距离函数名字

list\_dist\_fun\_name = ["欧氏距离", "曼哈顿距离", "余弦相似度"]

# K值列表

list\_k = [2, 3, 4, 5, 6, 10]

# 遍历data文件夹下所有数据集

for file in os.listdir(directory):

    # 读取当前数据集

    data = np.loadtxt(directory + file)

    for dist\_fun, fun\_name in zip(list\_dist\_fun, list\_dist\_fun\_name):

        fig = plt.figure(figsize=(17, 8))

        # 计算子图列数

        num\_col = (len(list\_k) + 1) >> 1

        for idx, k in enumerate(list\_k):

            # 计算当前参数下的聚类中心

            centers = kmenas\_train(data, dist\_fun, k)

            result = dist\_fun(centers, data)

            # 创建子图

            ax = fig.add\_subplot(2, num\_col, idx + 1)

            ax.set\_title(f"k = {k}")

            # 将每一类用不同的颜色标记

            for i in range(k):

                ax.scatter(data[result == i, 0], data[result == i, 1])

        plt.suptitle(f"数据集{file}使用{fun\_name}聚类")

        # 调整布局以防止重叠

        plt.tight\_layout()

        plt.show()

接下来我将介绍一下我用到的三个距离度量以及我的实现方式，分别是欧氏距离，曼哈顿距离，余弦相似度。没有使用闵氏距离的原因是：欧氏距离和曼哈顿距离其实都是闵氏距离的特殊情况（当p = 1的时候是曼哈顿距离，p = 2 的时候是欧氏距离）。

**欧氏距离：**

欧氏距离表示了两个点的直线距离，对每个样本取距离最小的类型中心作为聚类的类别，下面是我的实现：

# 欧式距离聚类

def Euclidean\_Distance(centers, data):

    dist = np.zeros((data.shape[0], centers.shape[0]))

    # dist= sqrt(sum((x-center)^2))

    for idx, center in enumerate(centers):

        dist[:, idx] = np.sqrt(np.sum((data - center) \*\* 2, axis=1))

    # 根据最小距离选择类别

    return np.argmin(dist, axis=1)

**曼哈顿距离：**

曼哈顿距离是两点之间坐标差值的绝对值的和，对每个样本取距离最小的类型中心作为聚类的类别，下面是我的实现：

# 曼哈顿距离聚类

def Manhattan\_Distance(centers, data):

    dist = np.zeros((data.shape[0], centers.shape[0]))

    # dist= sum(|x-center|)

    for idx, center in enumerate(centers):

        dist[:, idx] = np.sum(np.abs(data - center), axis=1)

    # 根据最小距离选择类别

    return np.argmin(dist, axis=1)

**余弦相似度：**

余弦相似度通过计算两个向量之间夹角的余弦，得到两个向量之间的方向差异。当余弦值为1的是，两个向量的方向是一致的，达到了相似度的最大值。对每个样本取距离1最近的类型中心作为聚类的类别，下面是我的实现：

# 余弦相似度

def Cosine\_Similarity(centers, data):

    # 分别计算中心点和样本的模长

    mag\_center = np.sqrt(np.sum(centers\*\*2, axis=1))

    mag\_data = np.sqrt(np.sum(data\*\*2, axis=1))

    # 计算余弦相似度

    dot\_product = data @ centers.T

    dist = dot\_product / np.outer(mag\_data, mag\_center)

    # 选择最接近1的类别

    return np.argmin(np.abs(dist - 1), axis=1)

### 聚类结果的展示

为了能够更好的对比不同的距离函数和K值对聚类的影响，我将以数据集为单位展示改变这些参数的聚类结果。我选取了2, 3, 4, 5, 6, 10这几个值作为K值，距离函数就是2.1.2.节中展示的三个距离函数。

**数据集aniso.txt:**

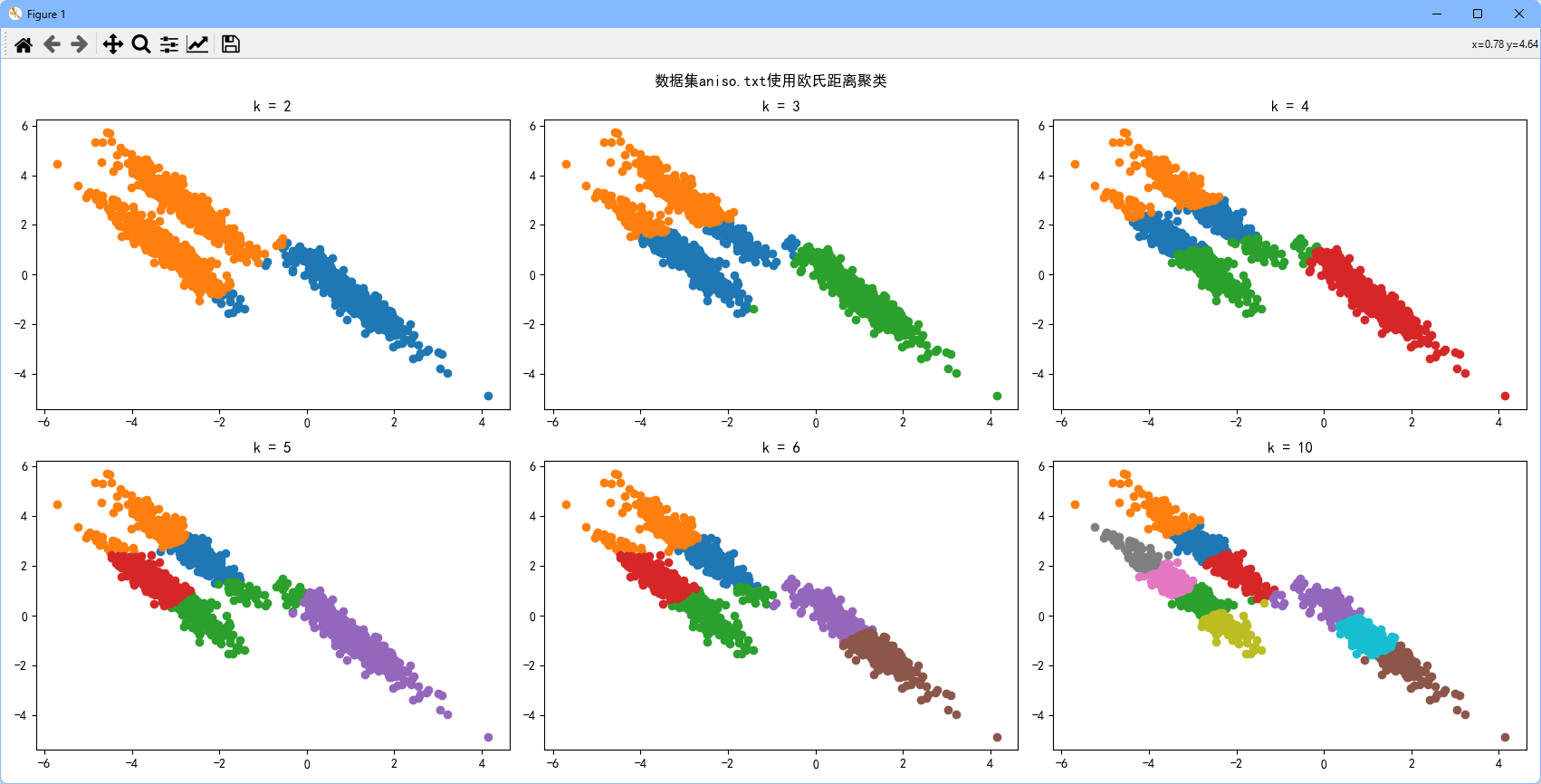


Figure 2 数据集aniso使用欧氏距离聚类



Figure 3 数据集aniso使用曼哈顿距离聚类

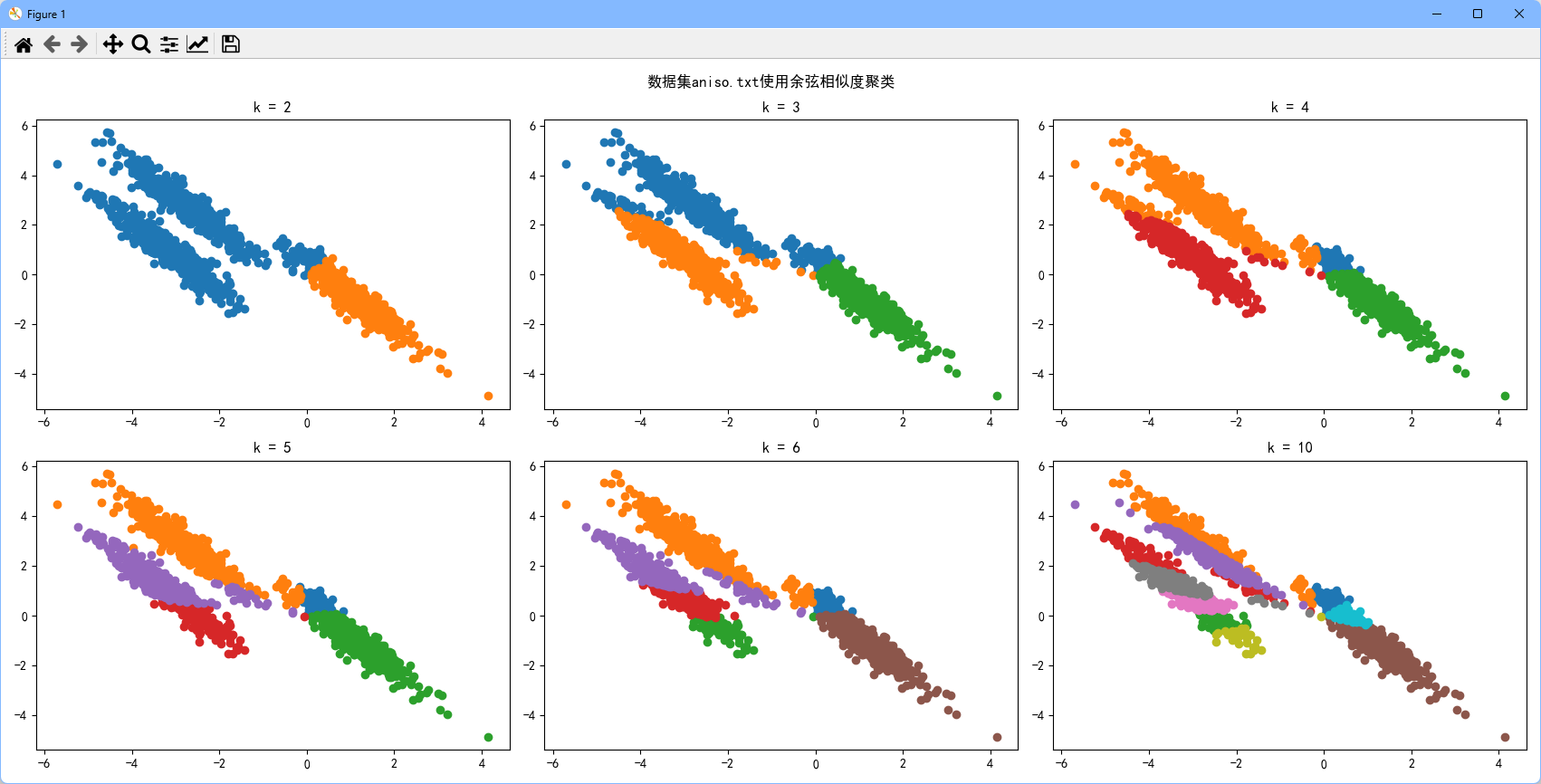


Figure 4 数据集sniso使用余弦相似度聚类

**数据集blobs.txt:**

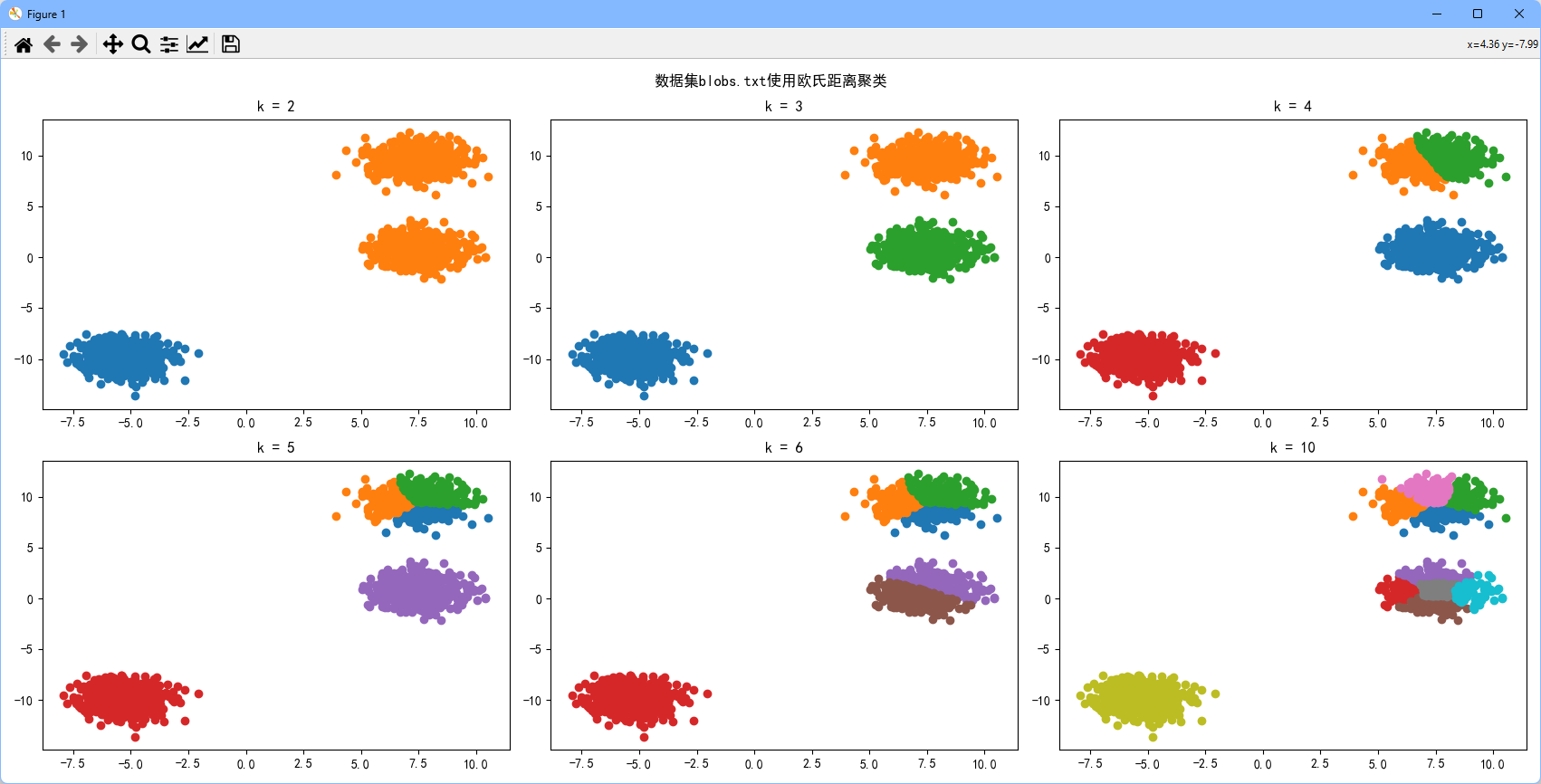


Figure 5 数据集blobs使用欧式距离聚类

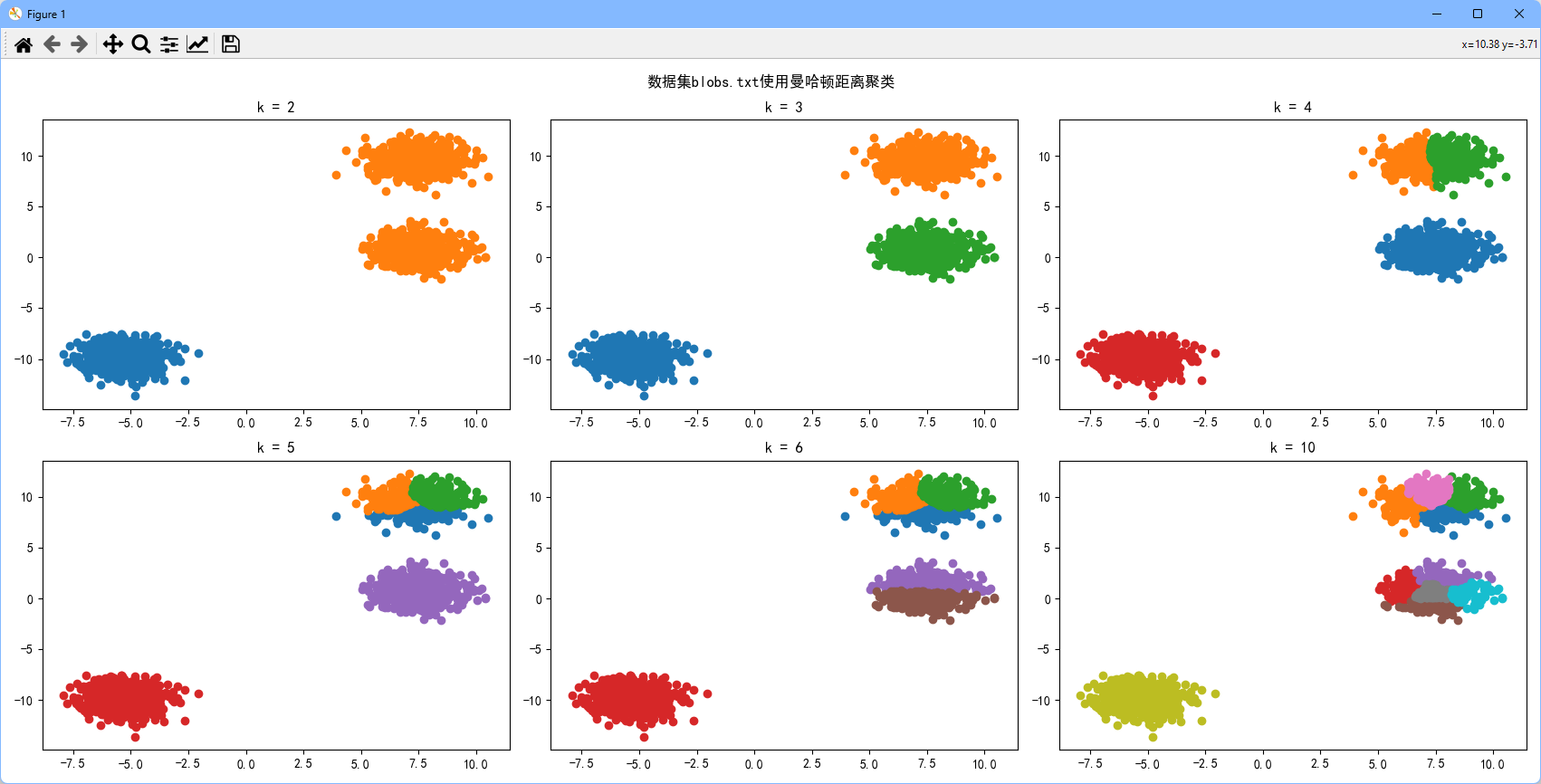


Figure 6数据集blobs使用曼哈顿距离聚类

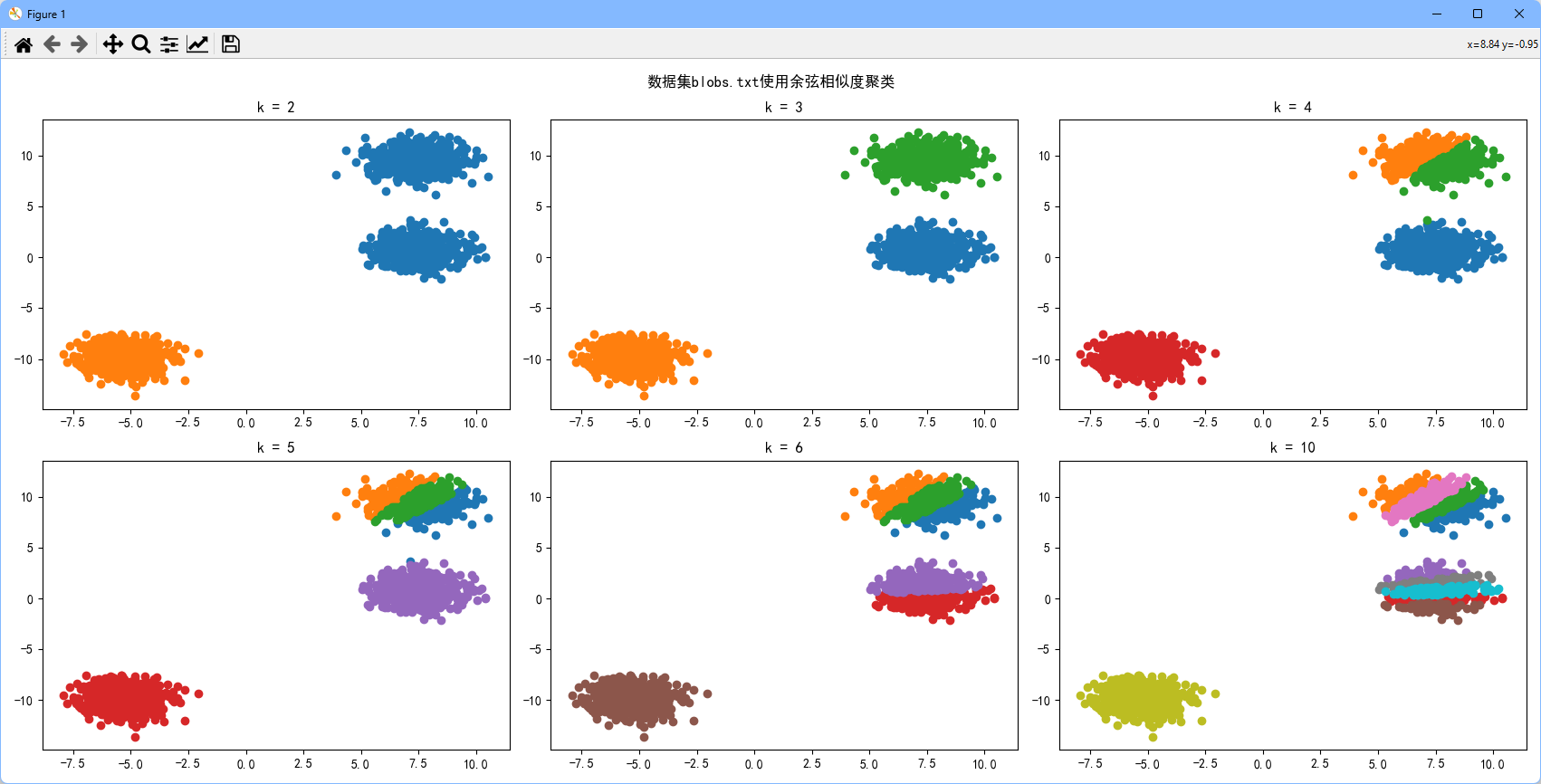


Figure 7数据集blobs使用余弦相似度聚类

**数据集noisy\_circle.txt:**

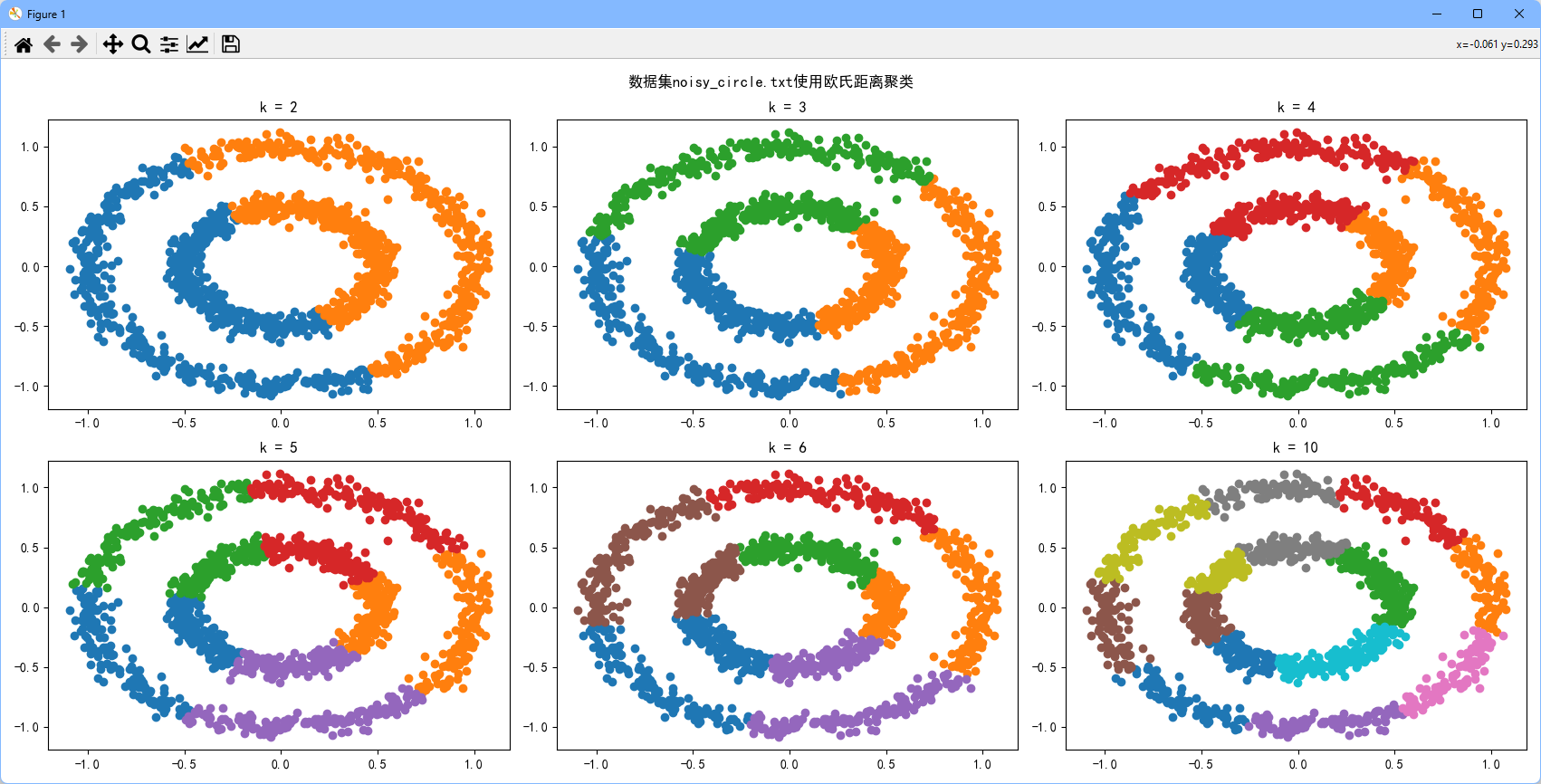


Figure 8 数据集noisy\_circle使用欧式距离聚类

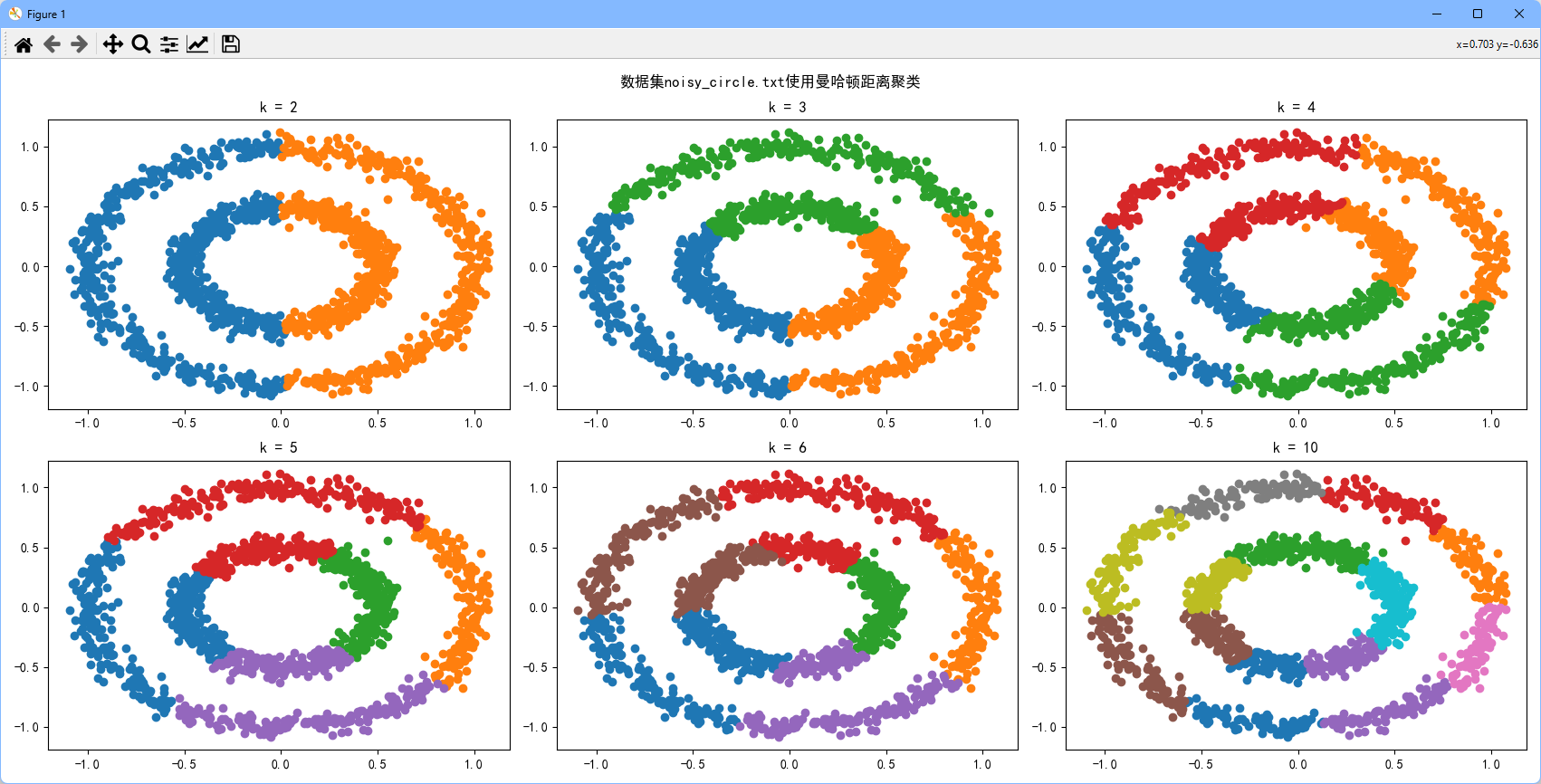


Figure 9 数据集noisy\_circle使用曼哈顿距离聚类

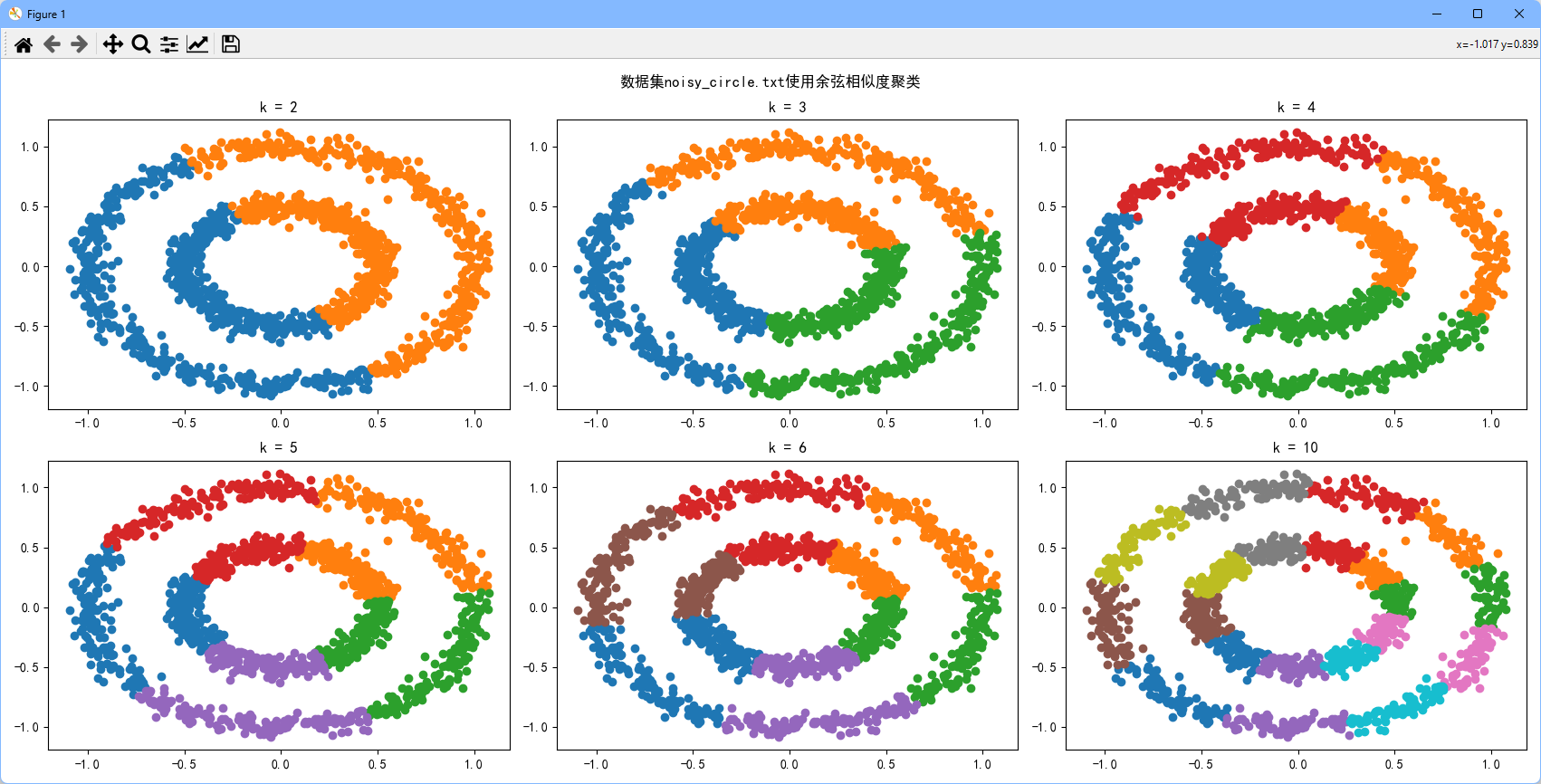


Figure 10 数据集noisy\_circle使用余弦相似度聚类

**数据集noisy\_moons.txt:**

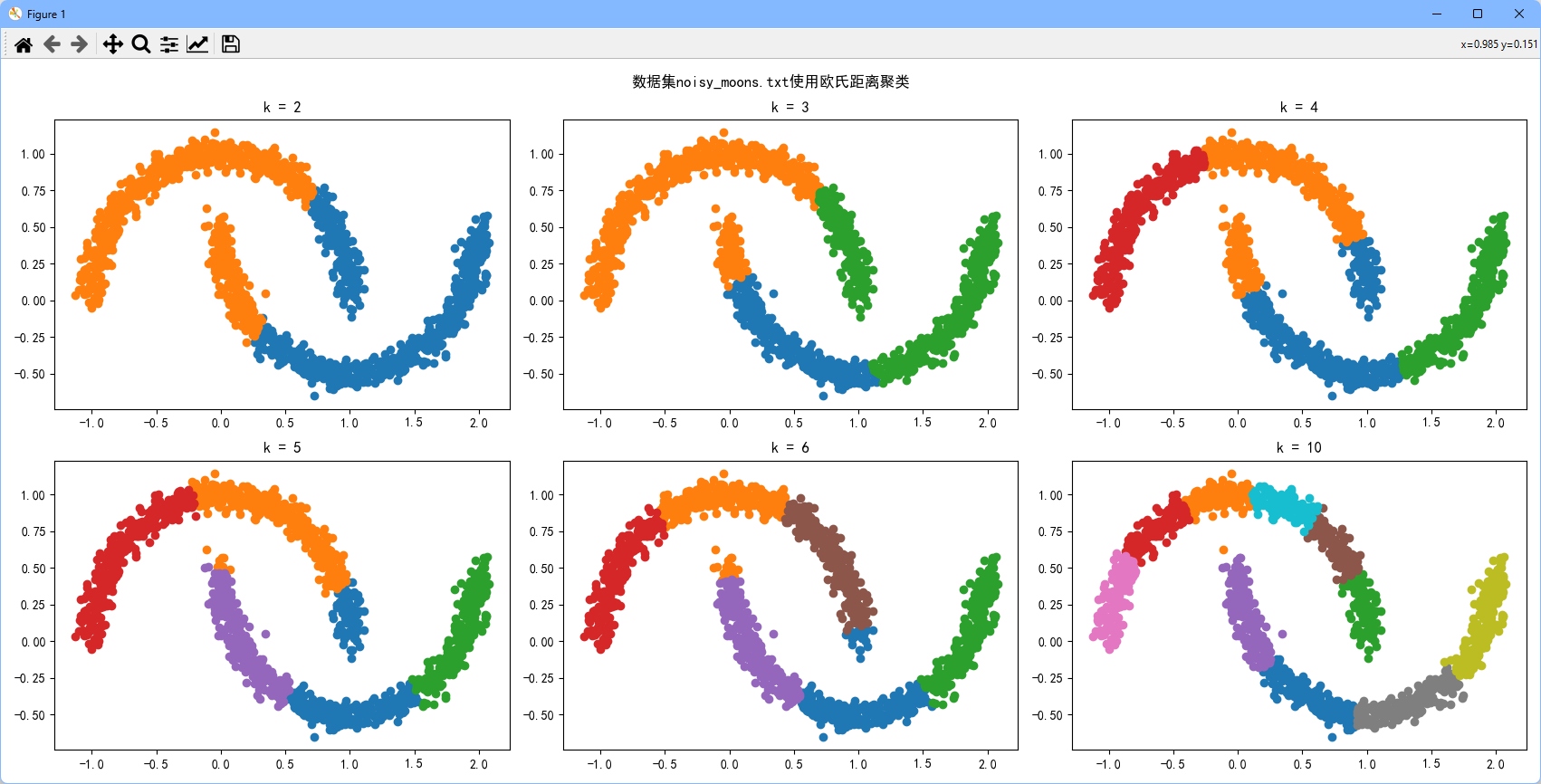


Figure 11 数据集noisy\_moons使用欧式距离聚类

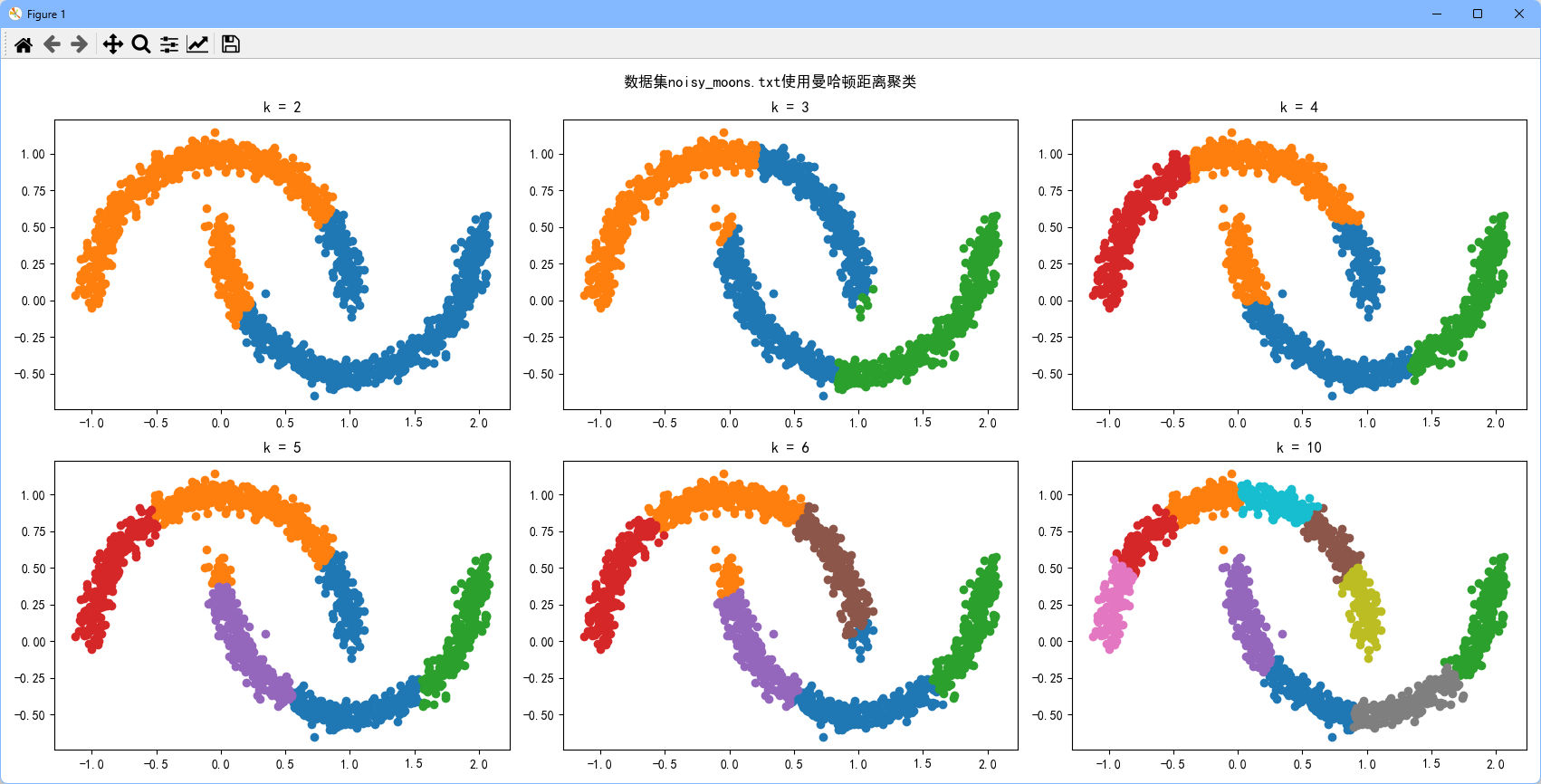


Figure 12 数据集noisy\_moons使用曼哈顿距离聚类

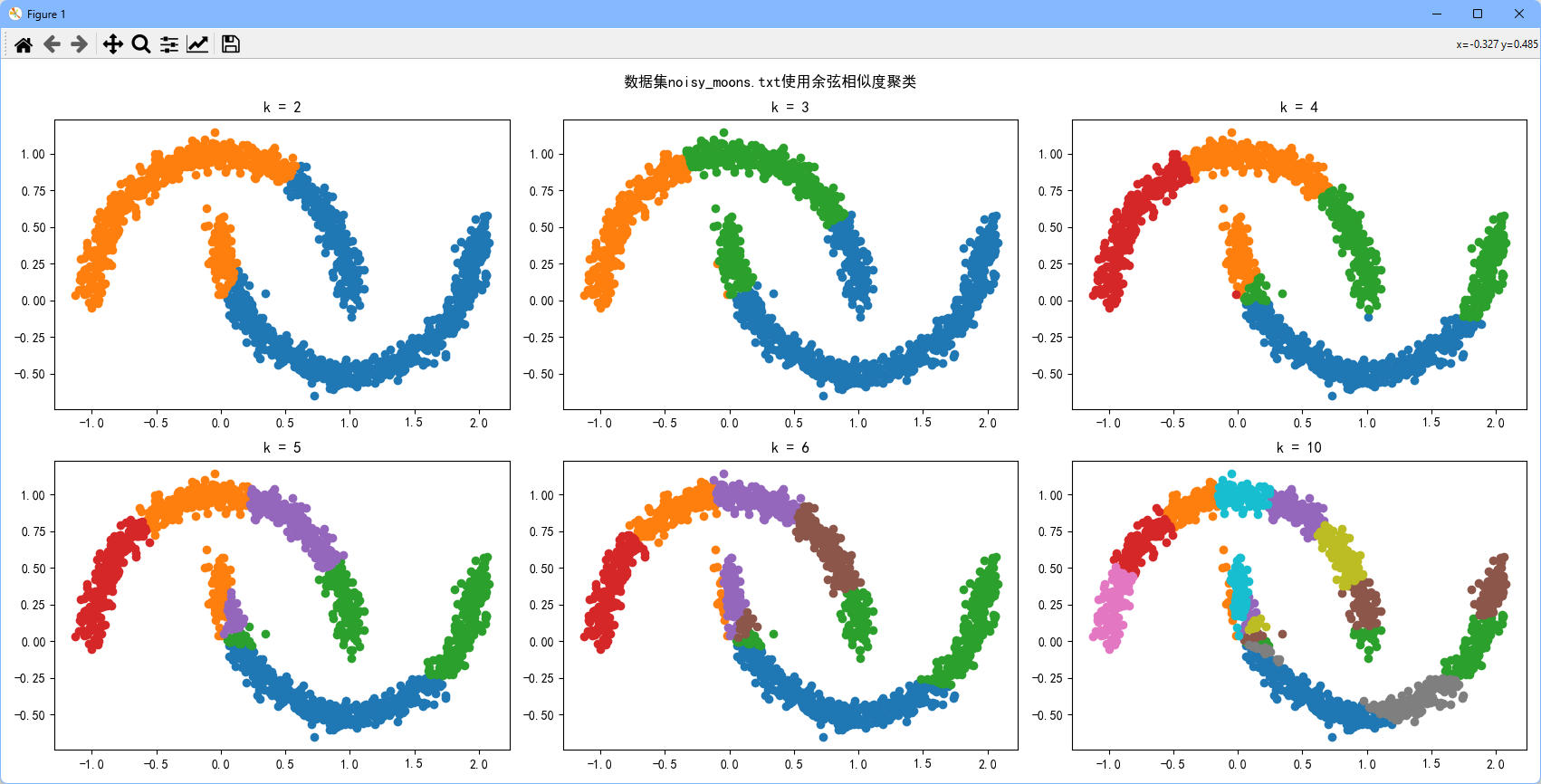


Figure 13 数据集noisy\_moons使用余弦相似度聚类

**数据集no\_structure.txt:**

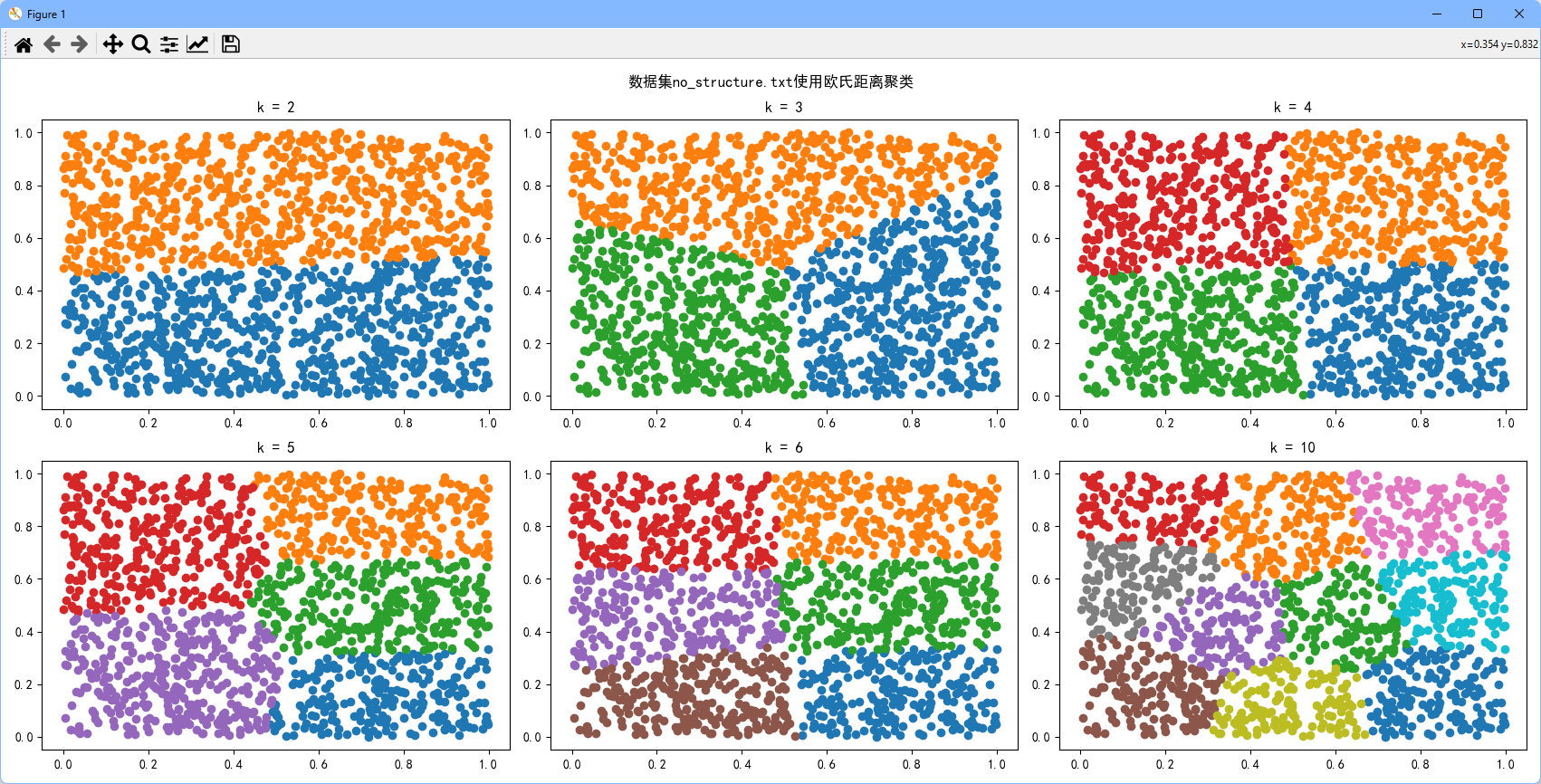


Figure 14 数据集no\_structure使用欧式距离聚类

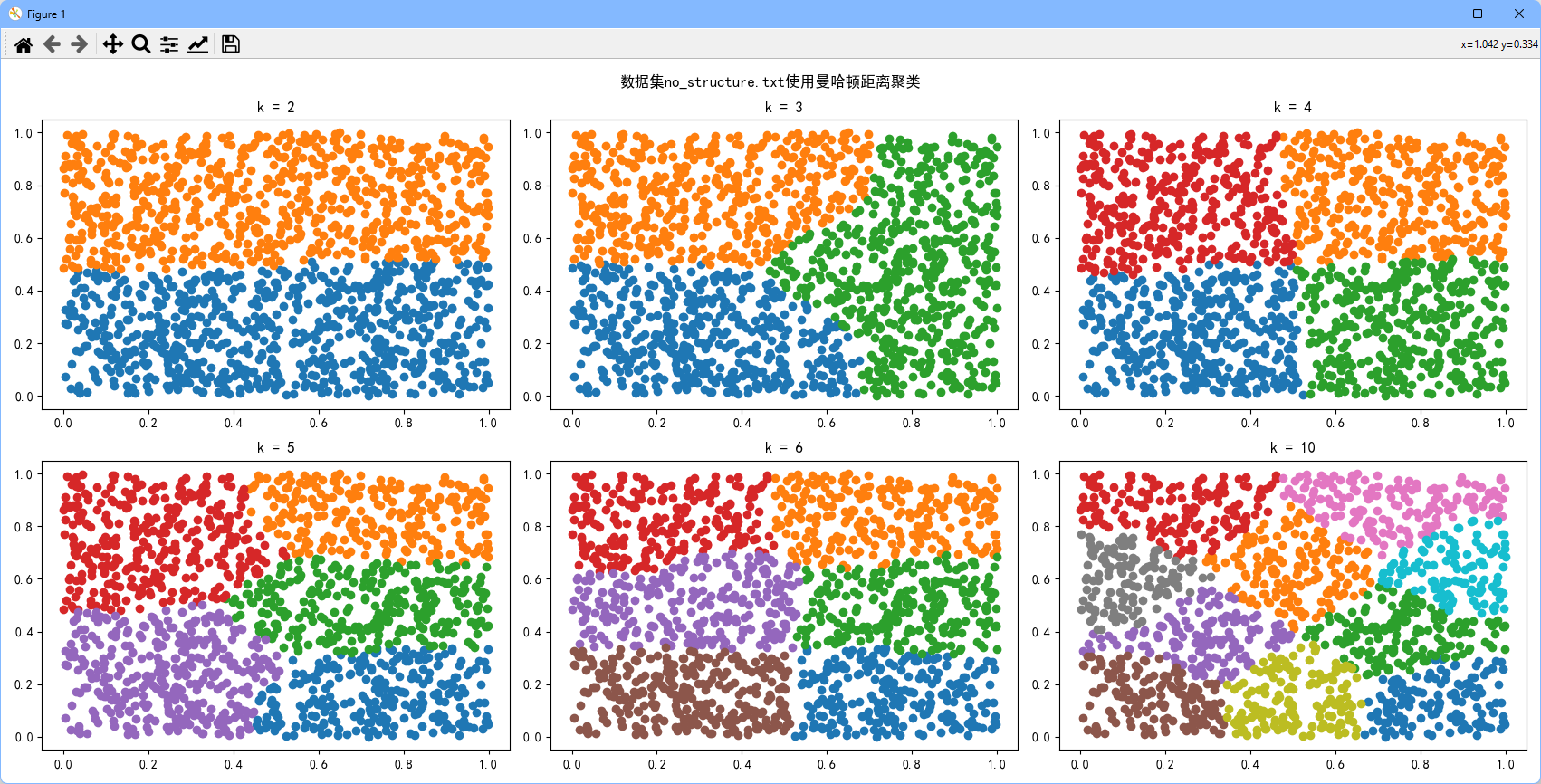


Figure 15数据集no\_structure使用曼哈顿距离聚类

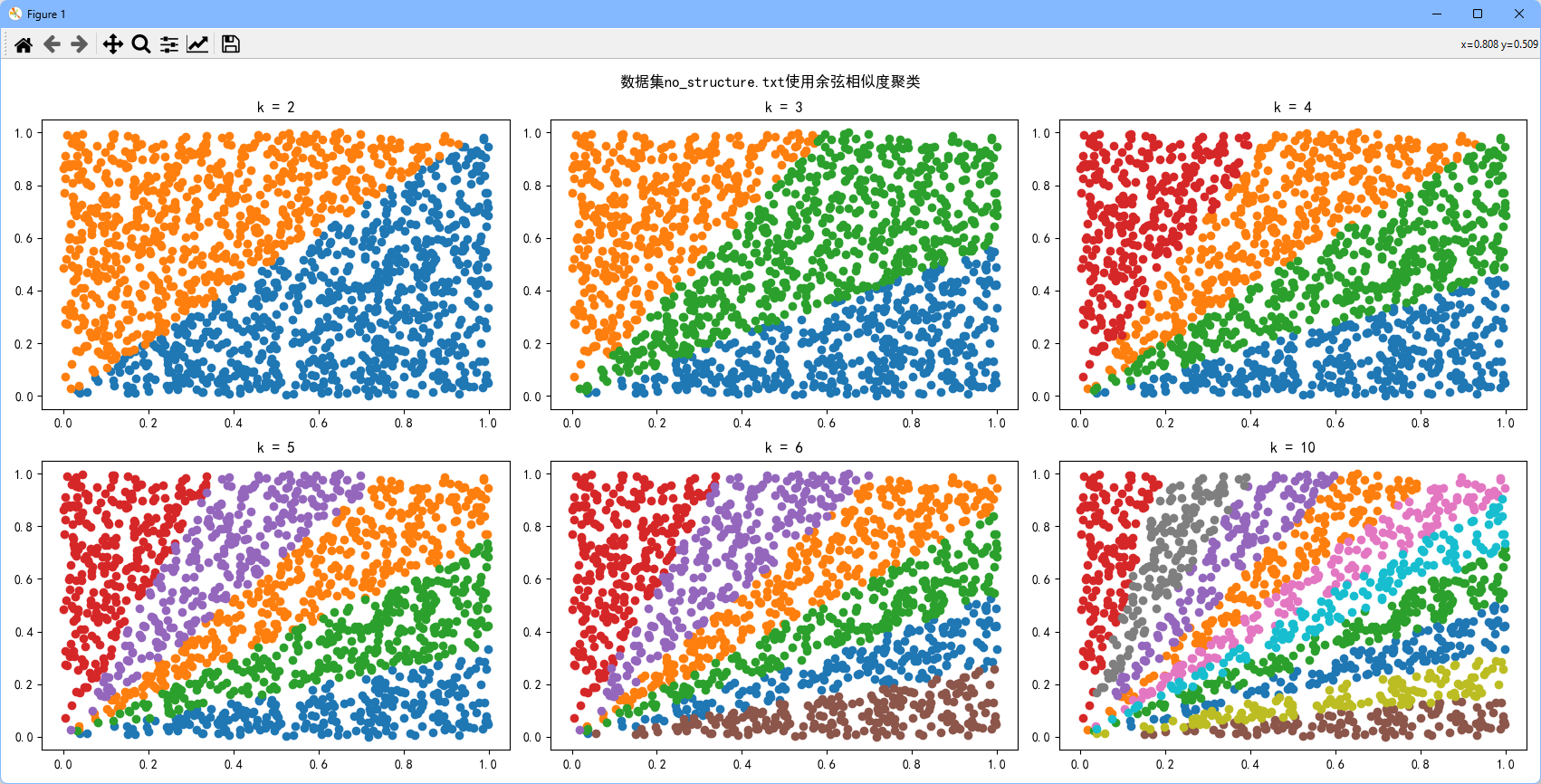


Figure 16 数据集no\_structure使用余弦相似度聚类

**数据集varied.txt:**

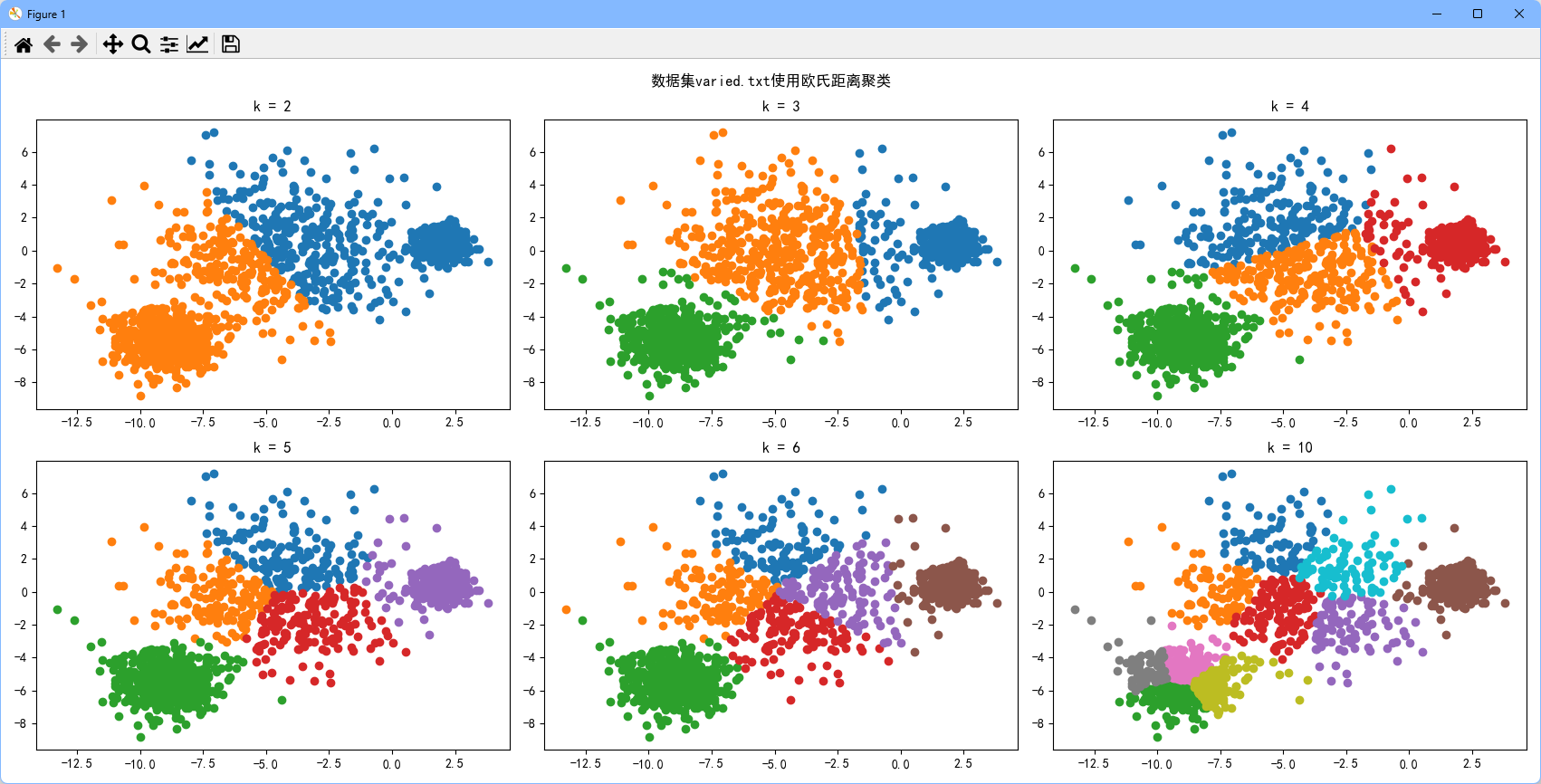


Figure 17 数据集varied使用欧式距离聚类

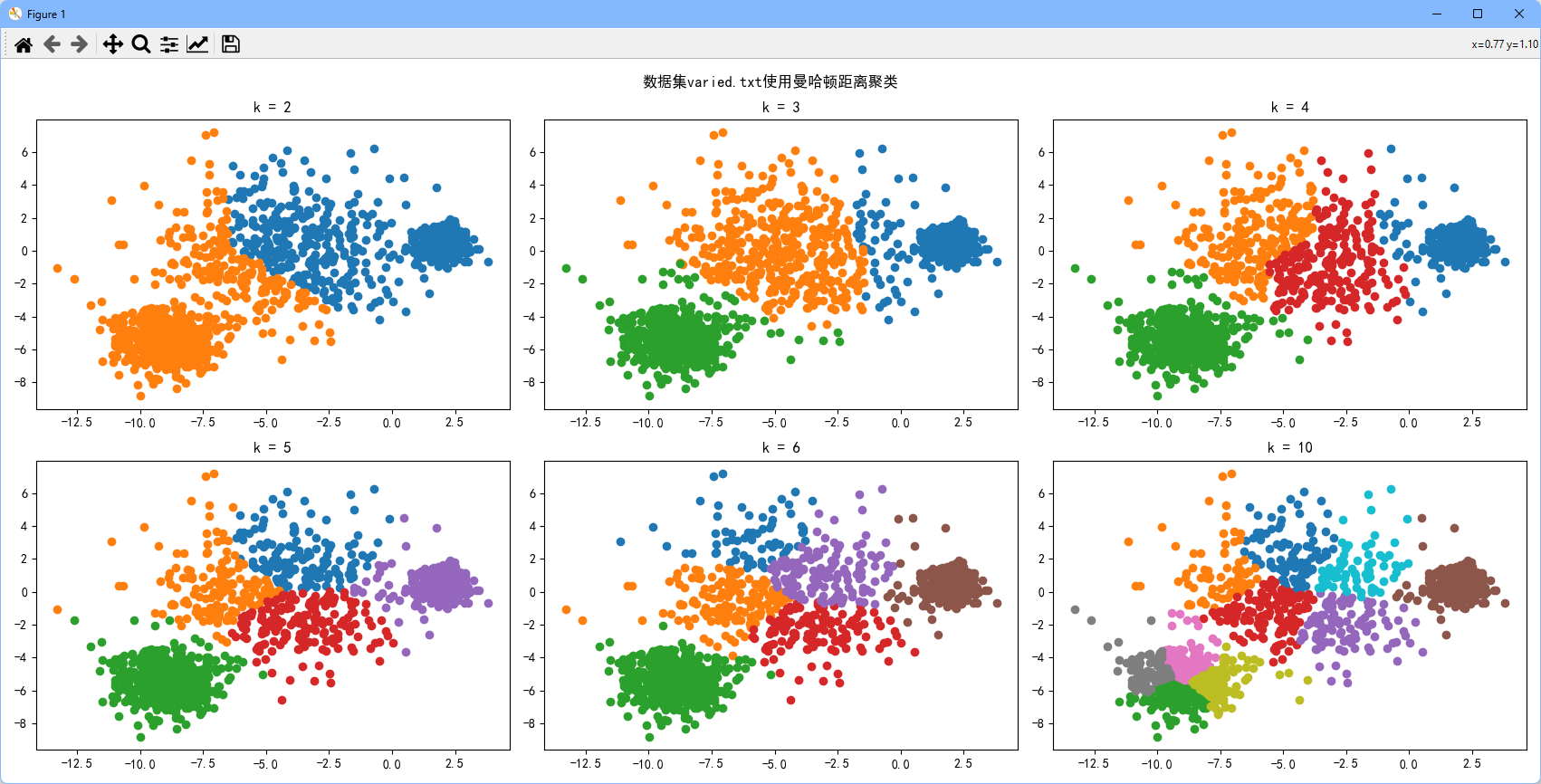


Figure 18 数据集varied使用曼哈顿距离聚类



Figure 19 数据集varied使用余弦相似度聚类

### 聚类结果分析

从上面的结果可以很轻易的看出，欧氏距离和曼哈顿距离作为闵氏距离的特殊情况，在使用相同的K值的情况下，分类的情况十分的相似。不过可能由于曼哈顿距离是沿着坐标轴计算的，而欧氏距离是计算直线距离，所以使用**曼哈顿距离**聚类的时候，类别之间的边界更加横平竖直一些。而使用余弦相似度计算的时候，由于非常独特的计算方式，聚类的结果也十分的不同。余弦相似度更倾向于将与聚类中心的相同方向的一系列样本聚在一起，所以呈现出类似扇形的图像。

## 题目二：手动实现DBSCAN聚类

### DBSCAN的实现

第二题所使用的DBSCAN的英文全称是Density-Based Spatial Clustering of Applications with Noise，也就是基于密度的聚类算法。这个算法是通过从满足一些条件的核心对象开始，不断的向周围扩展满足条件的样本，直到没有样本符合条件，为了更清晰的描述，我画了一个流程图：

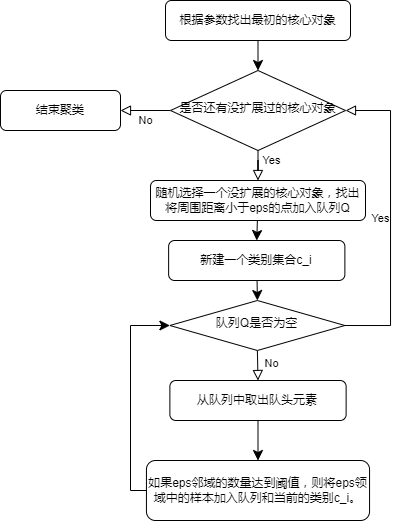


Figure 20 DBSCAN算法的流程图

这个算法只有两个参数，分别是领域阈值和邻域样本数量阈值。这两个参数影响了-领域的计算，下面是-领域的定义：

下面是我的python实现：

# 计算每个点周围的密度可达点的集合

def N(epsilon, data, x):

    data = np.array(data)

    return [tuple(point) for point in data[np.sum((data - x) \*\* 2, axis=1) < epsilon]]

有了-领域函数，接下来就需要初始化最初的核心对象集合：

# 核心对象集合

omega = queue.Queue()

# 初始化类别数和未访问样本集合

k, L = 0, set(data)

for x in L:

    # 将密度直达对象个数大于阈值的对象加入核心对象集合

    if len(N(epsilon, data, x)) >= minpts:

        omega.put(x)

此时我们就得到了最初的核心样本集合，我们只需要对每个核心样本不断的计算在其-领域以及其-领域内样本的-领域（密度可达对象）的点加入到当前的集合中就可以完成聚类：

# 储存每个聚类的集合

C = []

# print(L)

while not omega.empty():

    L\_old = L.copy()

    # 选择最后一个核心对象并去除这个对象

    o = omega.get()

    if o in L:

        L.remove(o)

    # 当前点被访问过的话跳过这个核心对象

    else:

        continue

    q = queue.Queue()

    q.put(o)

    while not q.empty():

        now = q.get()

        # 计算当前点密度可达的集合

        reachable = set(N(epsilon, data, now))

        if len(reachable) >= minpts:

            delta = reachable & L

            # 将密度可达的集合加入队列

            for d in delta:

                q.put(d)

            # 从未访问的数据集中去除这些可达的部分

            L = L - delta

    k += 1

    C.append(L\_old - L)

    if len(L) == 0:

        break

# 返回聚类结果和聚类数目

return C, k

在得到了聚类的结果之后，我们需要将其可视化出来，下面是我的实现，结果我将展示在下一小节。

# 聚类并展示结果

def dbcan\_and\_show(file, epsilon, minpts):

    data = np.loadtxt(directory + file)

    # 中间用到了set，不转为元组无法哈希

    data\_as\_tuples = [tuple(point) for point in data]

    C, k = dbscan(data\_as\_tuples, epsilon, minpts)

    plt.figure()

    for c in C:

        x\_values = [point[0] for point in c]

        y\_values = [point[1] for point in c]

        plt.scatter(x\_values, y\_values)

    plt.title(f"数据集{file}使用DBSCAN聚类分为{k}类")

    plt.figure()

    plt.scatter(data[:, 0], data[:, 1])

### 使用DBSCAN聚类6个数据集的展示

因为每个数据集的数据范围都不一样，如果使用相同的参数的话有的样本的效果就会非常差，所以我这次没有使用统一的参数，我会将每次聚类的参数在结果之前说明。

**数据集：aniso.txt，**



Figure 21 数据集aniso.txt使用DBSCAN聚类分为3类

**数据集：blobs.txt，**

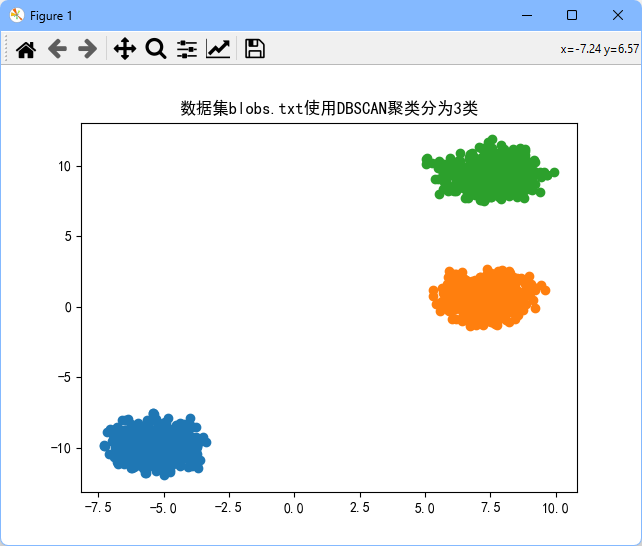


Figure 22 数据集blobs.txt使用DBSCAN聚类分为3类

**数据集：no\_structure.txt，**

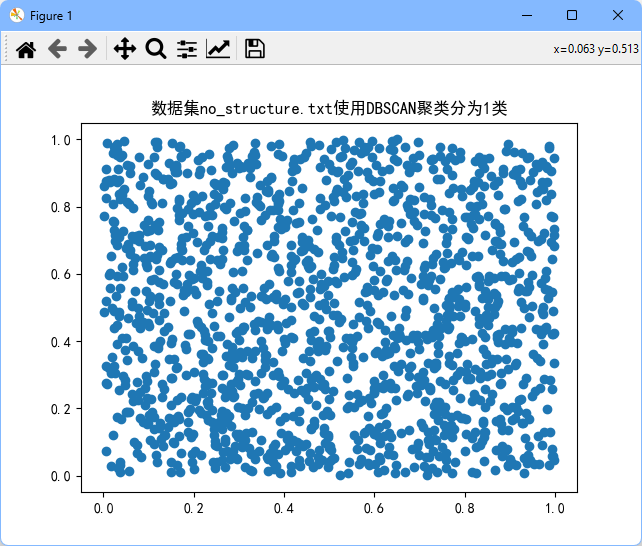


Figure 23 数据集no\_structure.txt使用DBSCAN聚类分为1类

**数据集：noisy\_circle.txt，**

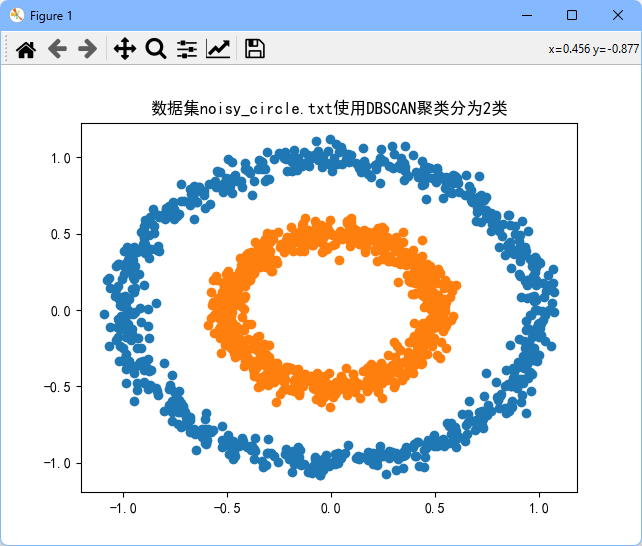


Figure 24 数据集noisy\_circle.txt使用DBSCAN聚类分为2类

**数据集：noisy\_moons.txt，**

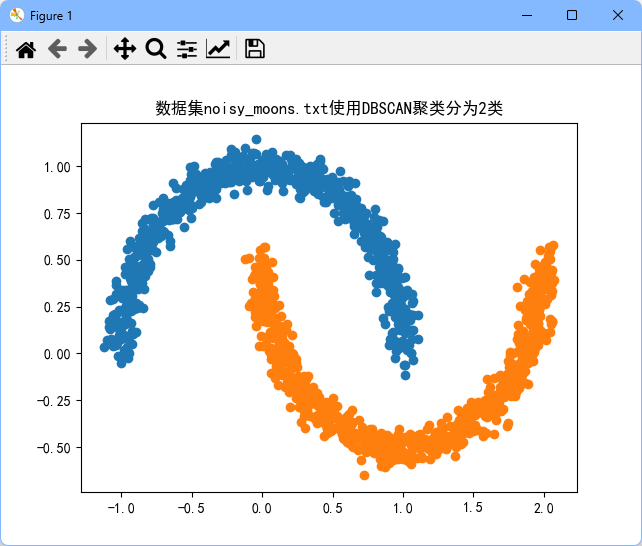


Figure 25 数据集noisy\_moons.txt使用DBSCAN聚类分为2类

**数据集：varied.txt，**

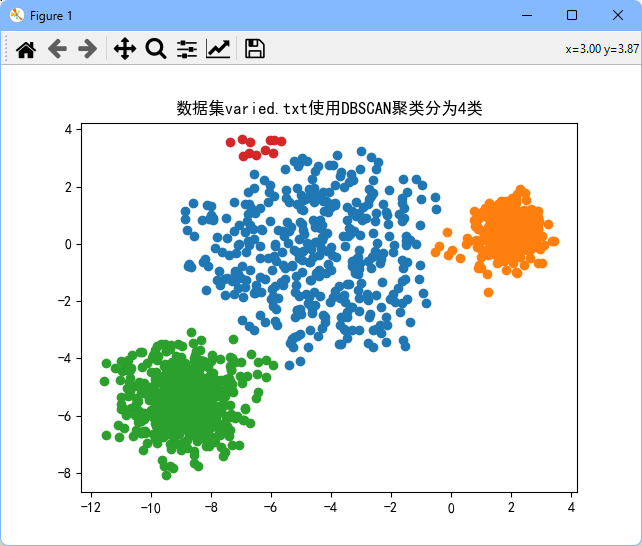


Figure 26 数据集varied.txt使用DBSCAN聚类分为4类

### 对DBSCAN聚类的观察

从上面的结果我们可以很轻易的看出来，对于一些非突的数据集，仅仅使用计算类别均值的方式难以完成真正的聚类。对于每个类别之间泾渭分明的数据集来说，使用密度聚类是更好的选择。

## 将K-Means与DBSCAN以及GMM在6个数据集上进行对比

### GMM聚类的原理

GMM的英文全称是Gaussian Mixture Model，也就是高斯混合模型。故名思意，GMM是基于高斯分布的。GMM的原理是通过计算由多个带权重的高斯分布的多项式得到每个样本属于的类别，换言之，就是通过多个单高斯分布的线性组合近似样本的分布，下面是计算公式：

但是现在出现了一个问题，有K个高斯分布，同时有N个数据，但是在不知道哪个数据属于哪个高斯分布的同时也不知道每个高斯分布的三个参数（权重, 均值和方差），为了解决这个问题，我们将参数的集合定义为。接下来使用和K-Means类似的EM方法，不断的更新参数，直到两次迭代之间的参数差异小于阈值。

* E（期望）步骤：对每个样本计算其属于某个簇的概率：
* M（最大化）步骤：根据上面计算的概率，更新参数

最后就能得到与样本分布最为类似的模型，接下来我将使用sklearn完成GMM的分类并画出图像。下面是我的实现方式：

# 聚类并展示结果

def gmm\_and\_show(file, num\_class):

    data = np.loadtxt(directory + file)

    # 创建并拟合 GMM 模型

    gmm = GaussianMixture(n\_components=num\_class)

    gmm.fit(data)

    # 输出每个样本的所属聚类

    labels = gmm.predict(data)

    plt.figure()

    # 分类绘制聚类结果

    for label in np.unique(labels):

        ind = labels == label

        plt.scatter(data[ind, 0], data[ind, 1])

    plt.title(f"数据集{file}使用GMM聚类")

    print(f"数据集{file}使用GMM聚类")

### GMM聚类的结果展示

**数据集aniso.txt:**

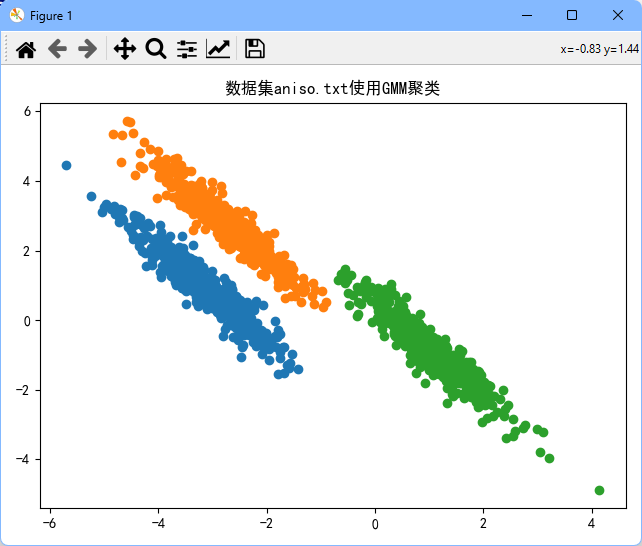


Figure 27数据集aniso.txt使用GMM聚类

**数据集blobs.txt:**

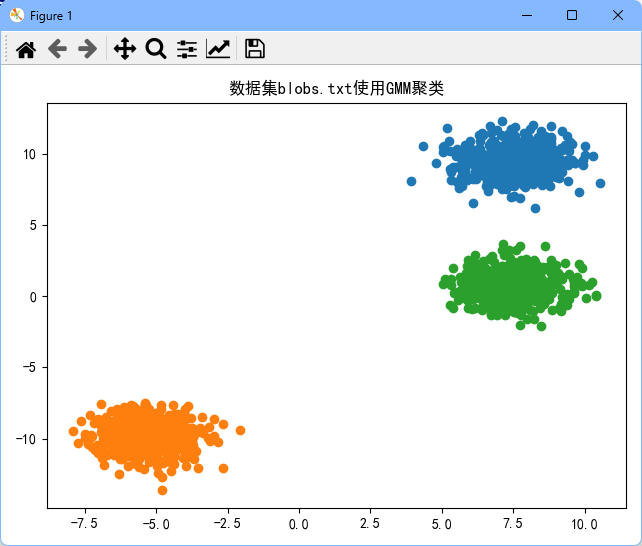


Figure 28 数据集blobs.txt使用GMM聚类

**数据集no\_structure.txt:**

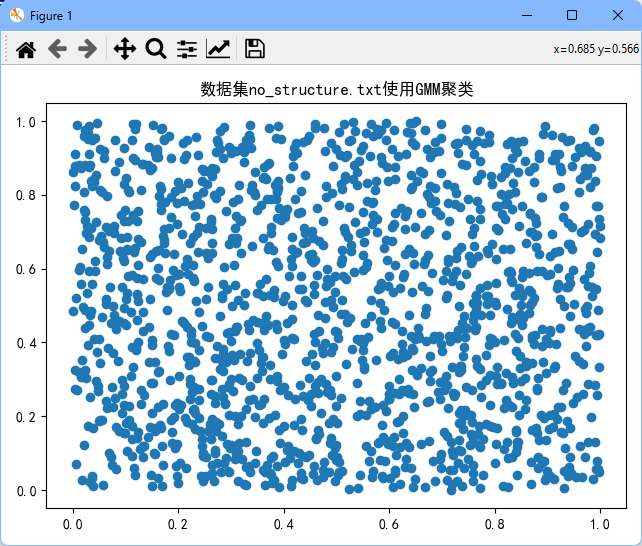


Figure 29 数据集no\_structure.txt使用GMM聚类

**数据集noisy\_circle.txt:**



Figure 30 数据集noisy\_circle.txt使用GMM聚类

**数据集noisy\_moons.txt:**

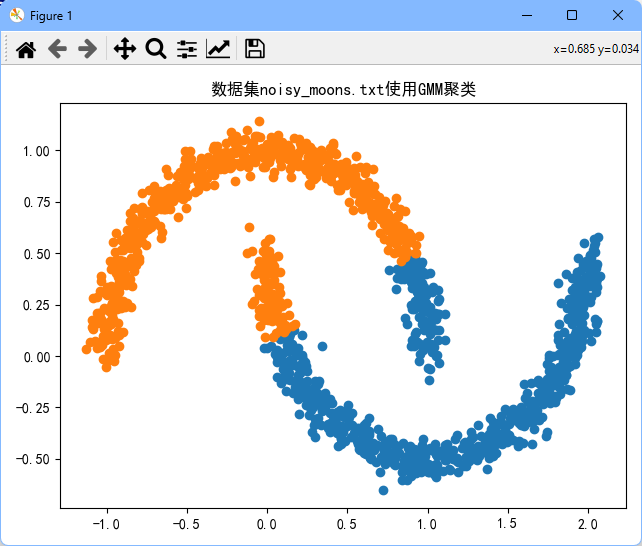


Figure 31 数据集noisy\_moons.txt使用GMM聚类

**数据集varied.txt:**

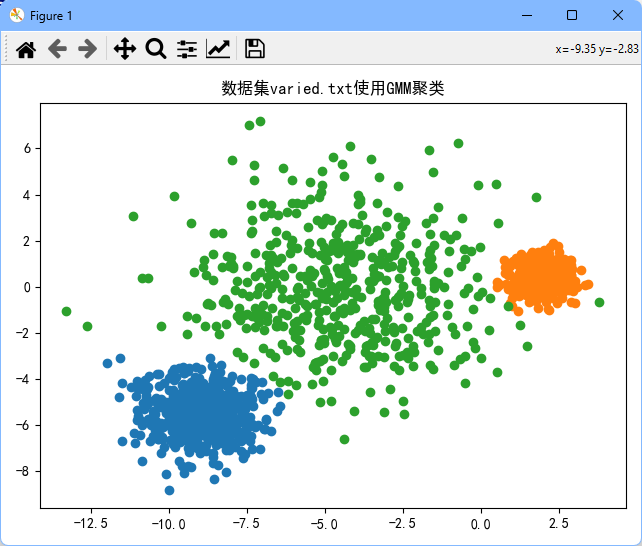


Figure 32 数据集varied.txt使用GMM聚类

### 聚类结果的分析方法

在完成了聚类之后，我们还需要对聚类结果进行分析，为了能够横向对比我上面提到的三个算法，我使用了sklearn提供的聚类评估指标其中的两个，我的选择是：Silhouette Score（轮廓系数）和Calinski-Harabasz Index（CH指数）。接下来我简单介绍一下这两个评估指标：

* Silhouette Score（轮廓系数）：这个算法结合了簇内的紧密度和簇间的分离度。对于每个数据点，计算它与其所属**簇内**所有其他点的**平均距离**，记为 。并找到离它**最近**的**另一个**簇，计算该簇内所有点到点的平均距离，记为。通过下面的公式：

最终的轮廓系数是所有的点轮廓系数的平均值，也就是。这个值在 -1和1之间:

* + 负值表示聚类结果差，数据点可能被分配到错误的簇中。
  + 0表示聚类结果重叠。
  + 正值表示聚类结果较好，数据点之间的距离大于这个点所属的簇内的距离。
* Calinski-Harabasz Index（CH指数）：这个算法基于簇内的紧密度和簇间的分离度。假设有K类，N个数据，先计算簇内的平均距离，再计算簇间平均距离 ，再使用CH指数公式：

这样就得到了CH指数，CH值越大取得的效果越好。

### 聚类结果的展示和算法间的比较

我新建了一个test.py测试文件，引入了我在前面自己实现的K-Means（曼哈顿距离和余弦相似度）和DBSCAN，以及sklearn库提供的GMM这三个聚类方式，具体代码我贴在附件（[4.1.4节](#_测试上面提到的分类器，画图并计算评价系数：test.py)）。接下来我会使用三种聚类方式，对每个文件进行四次聚类，不过由于文件no\_structure.txt在DBSCAN下只有一类，且评价指标没法评价只有一类的结果，所以就不在no\_structure.txt上测试了。接下来我将展示剩下的5个数据集聚类参数以及聚类结果。

**数据集aniso.txt：**

使用参数：K-Means (k = 3), DBSCAN (), GMM (k = 3)

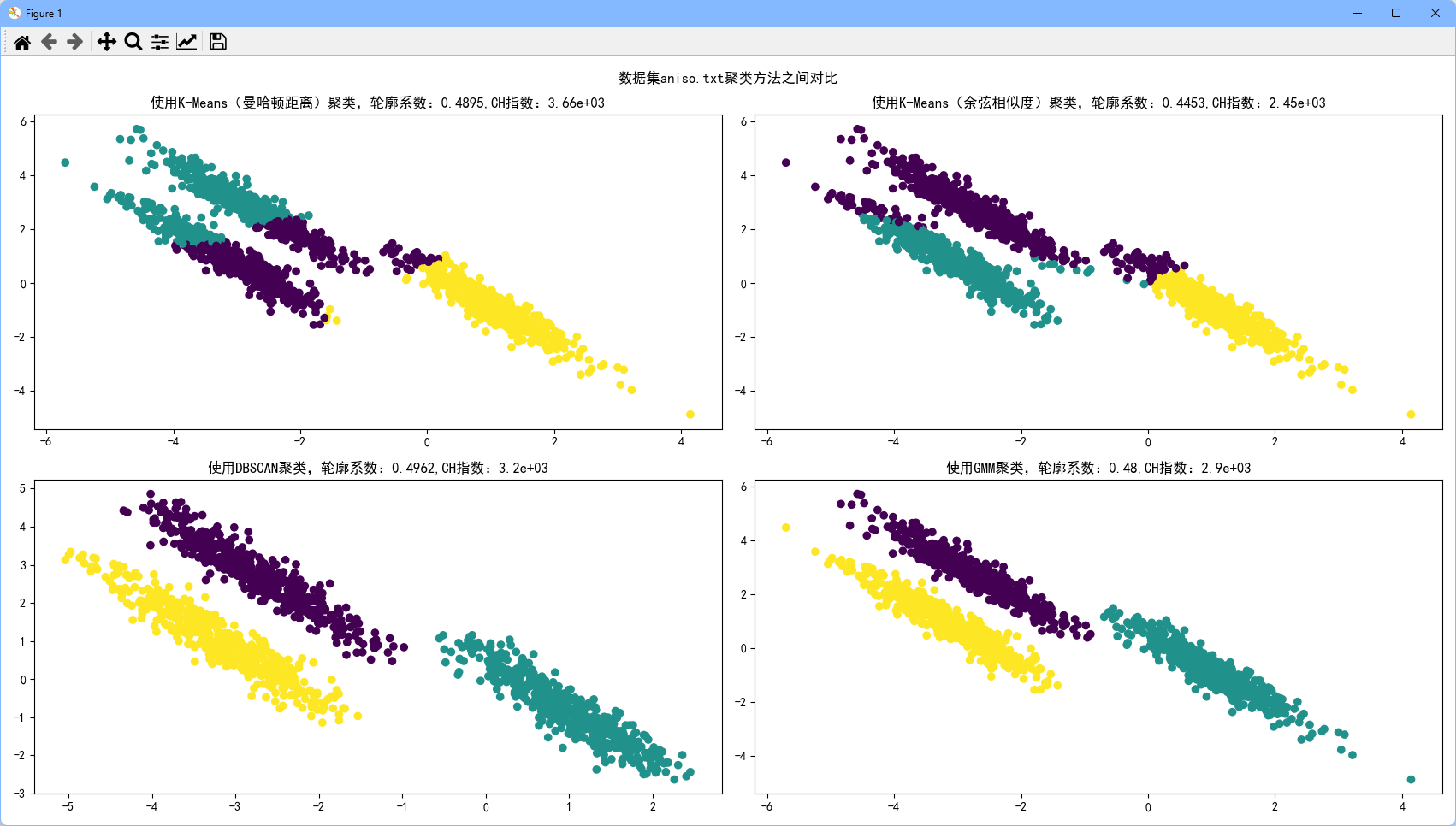


Figure 33 数据集aniso.txt四种聚类方案之间对比

可以从上面的结果中很轻易的看出来，DBSCAN不论是从轮廓系数还是CH指数上都领先其他聚类方法，因为这个数据集的相同类别之间间隔比较明显，排名第二的GMM因为数据集的三个类别都很贴合正态分布，所以效果也很不错。由于样本是长条的，比较符合余弦相似度的扇形分类，所以效果也算可以接受。但是当使用曼哈顿距离的时候，因为样本不是按照圆形分布的，导致无法让每个类别之中的关系更紧密而取得倒数第一的成绩。

**数据集blobs.txt：**

使用参数：K-Means (k = 3), DBSCAN (), GMM (k = 3)

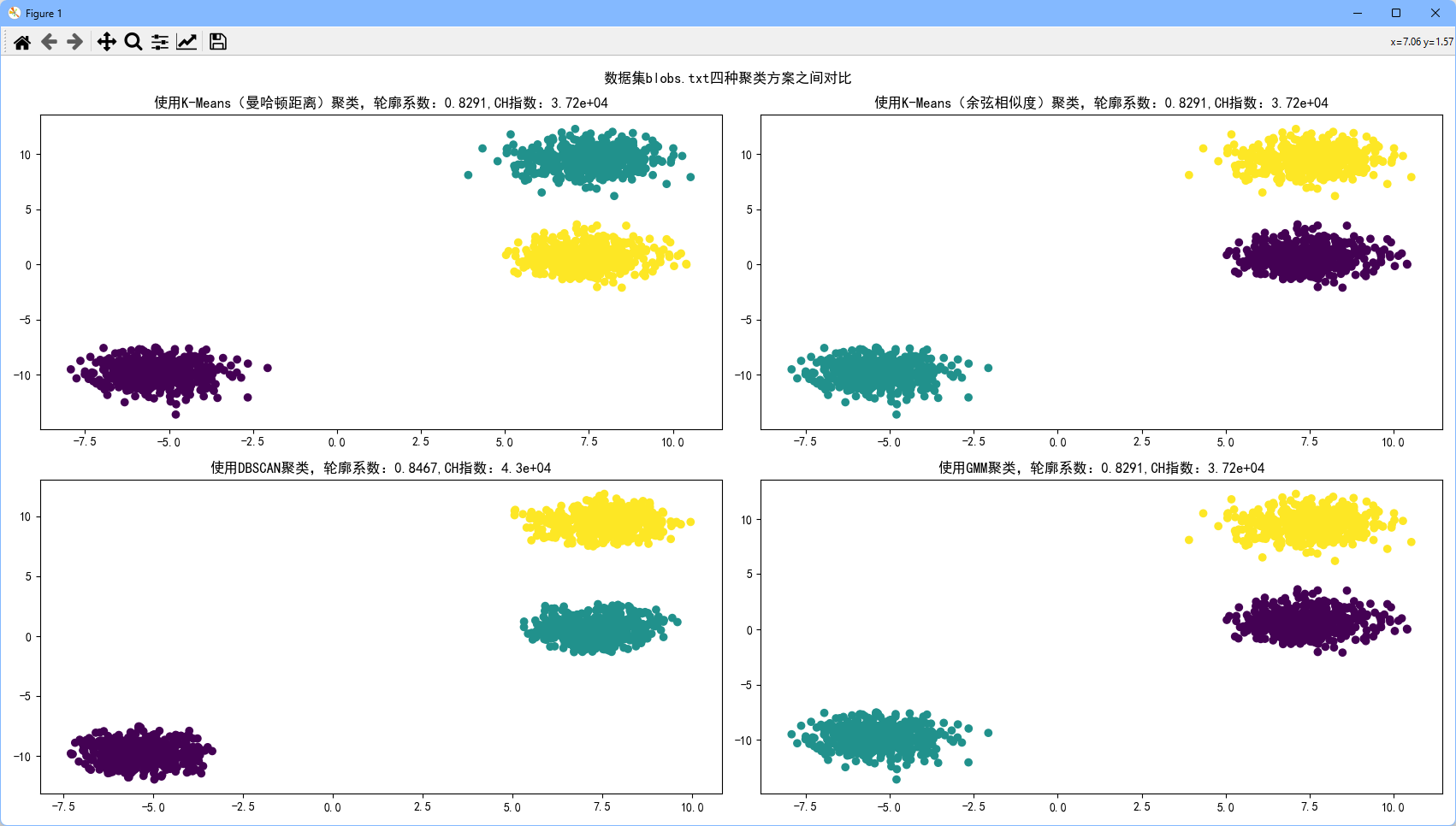


Figure 34 数据集blobs.txt四种聚类方案之间对比

这个数据集的每个类别的间距非常大，而且类之间很紧密，所以对每个分类方法都没什么挑战。在上个数据集表现不佳的曼哈顿距离K-Means因为这次样本和圆形较为相似而取得了不错的结果，不过这次所有数据集的分类结果都是一样的。

**数据集noisy\_circle.txt：**

使用参数：K-Means (k = 2), DBSCAN (), GMM (k = 2)

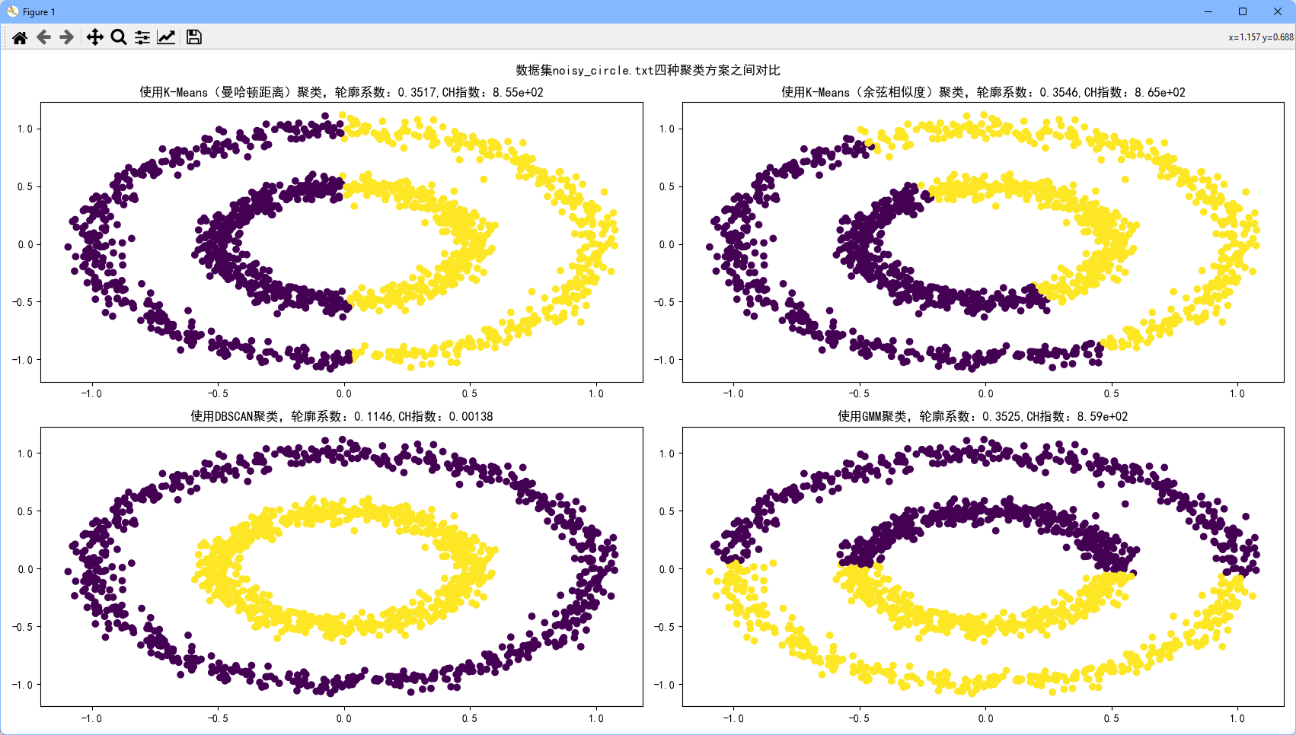


Figure 35 数据集noisy\_circle.txt四种聚类方案之间对比

这个数据集就能很明显的看出DBSCAN的不同，其他的聚类方法都在寻找一个比较集中的区域来进行分类，但是DBSCAN可以找到一个方法使得两类样本完全分离，且分离度是最高的。

**数据集noisy\_moons.txt：**

使用参数：K-Means (k = 2), DBSCAN (), GMM (k = 2)

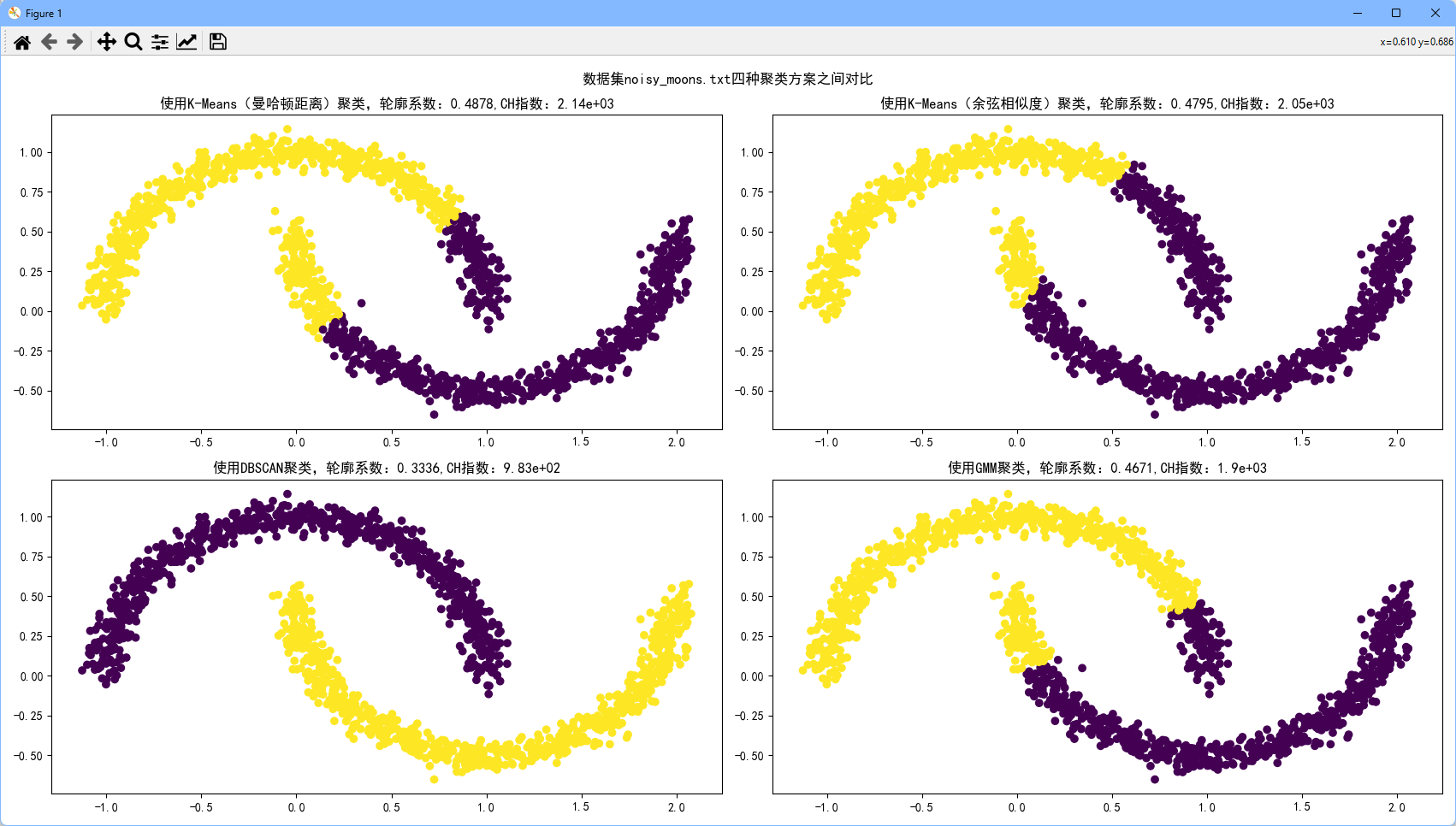


Figure 36 数据集noisy\_moons.txt四种聚类方案之间对比

这个数据集和上个数据集十分相似，能很明显的看出DBSCAN的不同，其他的聚类方法都在寻找一个比较集中的区域来进行分类，但是DBSCAN可以找到一个方法使得两类样本完全分离，最终的CH指数也是断崖式领先。

**数据集varied.txt：**

使用参数：K-Means (k = 3), DBSCAN (), GMM (k = 3)

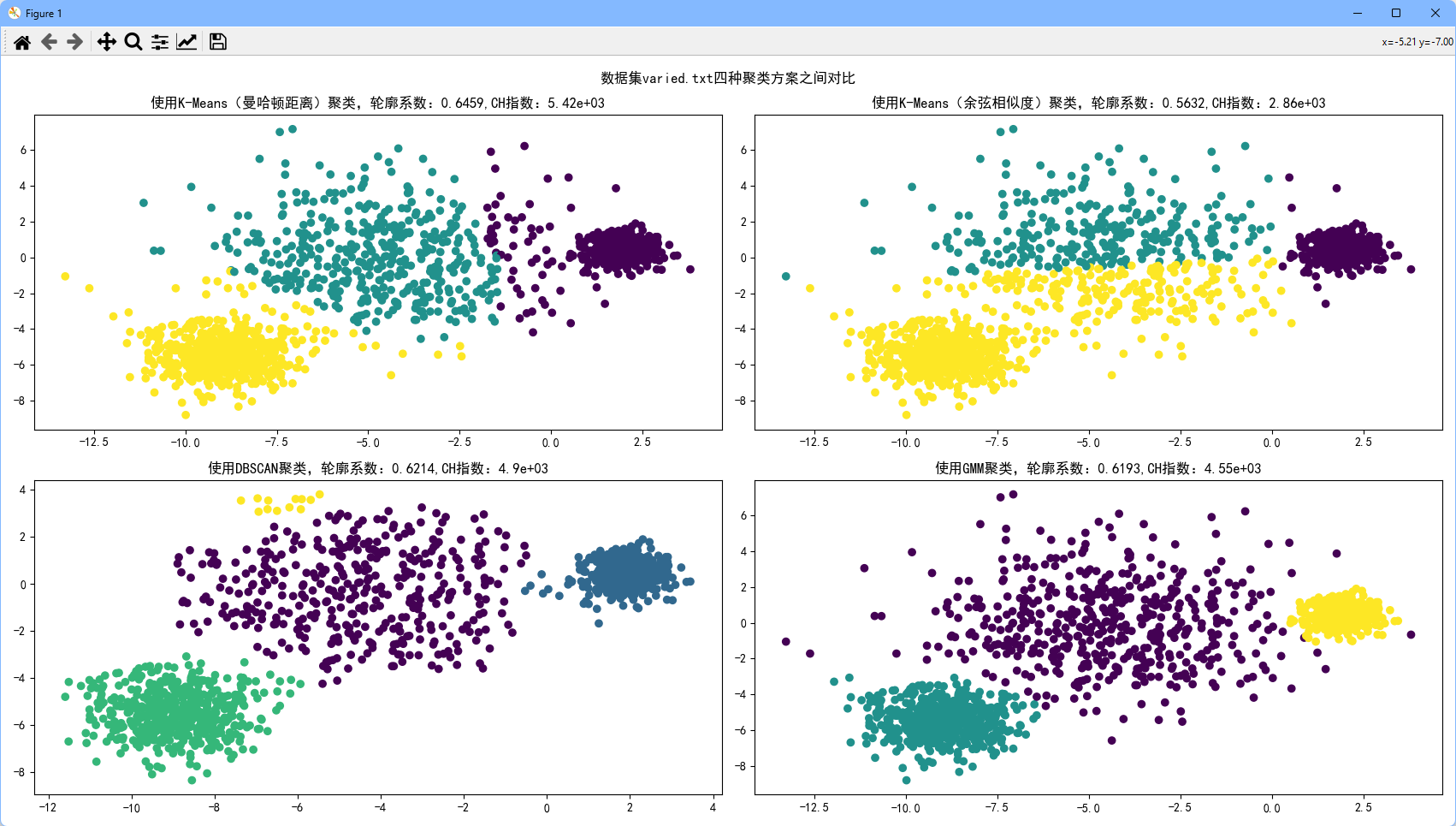


Figure 37 数据集varied.txt四种聚类方案之间对比

这个数据集很明显是三个高斯分布，这对于GMM来说是专业对口，所以理所应当的分类效果最好，而因为这三个高斯分布的 的差距比较大，所以DBSCAN的表现也不错。因为高斯分布是向着周围扩展的，而不是单方向扩展，导致余弦相似度的分类效果堪称灾难。曼哈顿距离则可以将高斯分布最密集的部分分为一类，所以效果还算可以。

# 总结

详细的结论我直接写在了第二节（[2.3.4节](#_聚类结果的展示和算法间的比较)）中的每一次的结果后面，这里再对之前的结果进行一次简单的概括。这次我一共尝试了五种聚类的方案，分别是K-Means（曼哈顿距离），K-Means（欧氏距离），K-Means（余弦相似度），DBSCAN（密度聚类），GMM（混合高斯模型）。但是由于曼哈顿距离和欧式距离都属于闵氏距离，最后对比的时候只是用了曼哈顿模型。对于像 blobs.txt中的泾渭分明的团状数据，不论是使用哪种分类器都可以有非常好的效果。闵氏距离的K-Means，由于需要类似圆形的数据集，所以在上面多个数据集中表现不佳，但是类似圆形的数据集DBSCAN和GMM的表现同样不错，所以可能K-Means的优势只剩速度稍快了吧。而余弦相似度对数据集的要求更是苛刻，只扫从上面6个数据集来看感觉不适合用于聚类。DBSCAN由于更贴合“聚类”这个概念，所以在多数数据集上表现不错。GMM使用了多个正态分布来替代了K-Means中各种距离函数的计算，对正态分布的样本表现最佳。

综上所述，上面我提到的5个聚类方案没有银弹，都需要先对样本进行观察，才能选择出最适合的聚类方式，但是相对来说DBSCAN的适用性是最广的，就是需要多次调节参数。

这次实验花费了我非常多的时间和精力，从复习上个学习学过的各种聚类方式的原理到可视化模型的方法，到思考模型的测试策略。每一个环节都十分重要，我也都遇到了一些困难：

这次我花费最多时间解决的问题就是在亲手余弦相似度的时候遇到的运行效率问题。一开始我使用普通的循环来完成这个过程，但是感觉速度非常慢。通过思考，我想到了通过矩阵运算完成计算余弦相似度的方法。Numpy中的矩阵运算是经过高度优化的，在替换之后效率果然大幅提升，为我节省了很多等待的时间。

这次的实验让我受益匪浅，使我对numpy库和sklearn库的理解更进一步的同时也锻炼了我解决问题的能力。上面提到的两个库都是在机器学习领域非常重要的工具，只有用好他们才能更好的完成实验，并理解这些模型的底层原理。我将继续保持学习，多学，多写，争取有朝一日能成为机器学习大师。

# 附件

## 完整代码

### 不调库实现K-Means分类器：kmeans.py

import numpy as np

import matplotlib.pyplot as plt

import os

from matplotlib import rcParams

# 设置中文字体

rcParams["font.family"] = "SimHei"

# 解决负号问题

rcParams["axes.unicode\_minus"] = False

# 定义数据集目录

directory = "data/"

# 欧式距离聚类

def Euclidean\_Distance(centers, data):

    dist = np.zeros((data.shape[0], centers.shape[0]))

    # dist= sqrt(sum((x-center)^2))

    for idx, center in enumerate(centers):

        dist[:, idx] = np.sqrt(np.sum((data - center) \*\* 2, axis=1))

    # 根据最小距离选择类别

    return np.argmin(dist, axis=1)

# 曼哈顿距离聚类

def Manhattan\_Distance(centers, data):

    dist = np.zeros((data.shape[0], centers.shape[0]))

    # dist= sum(|x-center|)

    for idx, center in enumerate(centers):

        dist[:, idx] = np.sum(np.abs(data - center), axis=1)

    # 根据最小距离选择类别

    return np.argmin(dist, axis=1)

# 余弦相似度

def Cosine\_Similarity(centers, data):

    # 分别计算中心点和样本的模长

    mag\_center = np.sqrt(np.sum(centers\*\*2, axis=1))

    mag\_data = np.sqrt(np.sum(data\*\*2, axis=1))

    # 计算余弦相似度

    dot\_product = data @ centers.T

    dist = dot\_product / np.outer(mag\_data, mag\_center)

    # 选择最接近1的类别

    return np.argmin(np.abs(dist - 1), axis=1)

# K-Means聚类函数，计算聚类中心

def kmenas\_train(data, dist\_fun, k):

    # 选择前k个作为聚类中心

    centers = np.array(data[:k])

    while True:

        center\_new = np.zeros(centers.shape)

        # 根据聚类中心计算每个样本属于的类别

        res = dist\_fun(centers, data)

        # 计算新的聚类中心

        for idx in range(k):

            center\_new[idx] = np.mean(data[res == idx], axis=0)

        # 如果两次训练之间的聚类中心没有变化则停止

        if np.sum(np.abs(centers - center\_new)) < 1e-10:

            break

        centers = center\_new

    return centers

# 距离函数列表

list\_dist\_fun = [Euclidean\_Distance, Manhattan\_Distance, Cosine\_Similarity]

# 距离函数名字

list\_dist\_fun\_name = ["欧氏距离", "曼哈顿距离", "余弦相似度"]

# K值列表

list\_k = [2, 3, 4, 5, 6, 10]

# 测试函数

if \_\_name\_\_ == "\_\_main\_\_":

    # 遍历data文件夹下所有数据集

    for file in os.listdir(directory):

        # 读取当前数据集

        data = np.loadtxt(directory + file)

        # 记录最佳的参数：当前值，距离函数，k值

        best\_s\_s = [0, 0, 0]

        best\_c\_h = [0, 0, 0]

        for dist\_fun, fun\_name in zip(list\_dist\_fun, list\_dist\_fun\_name):

            fig = plt.figure(figsize=(17, 8))

            # 计算子图列数

            num\_col = (len(list\_k) + 1) >> 1

            for idx, k in enumerate(list\_k):

                # 计算当前参数下的聚类中心

                centers = kmenas\_train(data, dist\_fun, k)

                labels = dist\_fun(centers, data)

                # 创建子图

                ax = fig.add\_subplot(2, num\_col, idx + 1)

                ax.set\_title(f"k = {k}")

                """

                # 计算轮廓系数

                 s\_s = silhouette\_score(data, labels)

                if s\_s > best\_s\_s[0]:

                    best\_s\_s = [s\_s, fun\_name, k]

                c\_h = calinski\_harabasz\_score(data, labels)

                if c\_h > best\_c\_h[0]:

                    best\_c\_h = [c\_h, fun\_name, k

                """

                # 将每一类用不同的颜色标记

                for i in range(k):

                    ax.scatter(data[labels == i, 0], data[labels == i, 1])

            plt.suptitle(f"数据集{file}使用{fun\_name}聚类")

            # 调整布局以防止重叠

            plt.tight\_layout()

        """

        print(f"文件：{file}")

        print(f"使用轮廓系数评价的最佳参数：{best\_s\_s}")

        print(f"使用CH指数评价的最佳参数：{best\_c\_h}\n")

        """

        plt.show()

### 不调库实现DBSCAN分类器：dbscan.py

import numpy as np

import matplotlib.pyplot as plt

import queue

from matplotlib import rcParams

# 设置中文字体

rcParams["font.family"] = "SimHei"

# 解决负号问题

rcParams["axes.unicode\_minus"] = False

# 定义数据集目录

directory = "data/"

# 计算每个点周围的密度可达点的集合

def N(epsilon, data, x):

    data = np.array(data)

    return [tuple(point) for point in data[np.sum((data - x) \*\* 2, axis=1) < epsilon]]

# 聚类函数，返回所有类别

def dbscan(data, epsilon, minpts):

    # 核心对象集合

    omega = queue.Queue()

    # 初始化类别数和未访问样本集合

    k, L = 0, set(data)

    for x in L:

        # 将密度直达对象个数大于阈值的对象加入核心对象集合

        if len(N(epsilon, data, x)) >= minpts:

            omega.put(x)

    # 储存每个聚类的集合

    C = []

    # print(L)

    while not omega.empty():

        L\_old = L.copy()

        # 选择最后一个核心对象并去除这个对象

        o = omega.get()

        if o in L:

            L.remove(o)

        # 当前点被访问过的话跳过这个核心对象

        else:

            continue

        q = queue.Queue()

        q.put(o)

        while not q.empty():

            now = q.get()

            # 计算当前点密度可达的集合

            reachable = set(N(epsilon, data, now))

            if len(reachable) >= minpts:

                delta = reachable & L

                # 将密度可达的集合加入队列

                for d in delta:

                    q.put(d)

                # 从未访问的数据集中去除这些可达的部分

                L = L - delta

        k += 1

        C.append(L\_old - L)

        if len(L) == 0:

            break

    # print(k)

    return C, k

# 聚类并展示结果

def dbcan\_and\_show(file, epsilon, minpts):

    data = np.loadtxt(directory + file)

    # 中间用到了set，不转为元组无法哈希

    data\_as\_tuples = [tuple(point) for point in data]

    C, k = dbscan(data\_as\_tuples, epsilon, minpts)

    plt.figure()

    for c in C:

        x\_values = [point[0] for point in c]

        y\_values = [point[1] for point in c]

        plt.scatter(x\_values, y\_values)

    plt.title(f"数据集{file}使用DBSCAN聚类分为{k}类")

    print(f"数据集{file}使用DBSCAN聚类分为{k}类")

    # plt.figure()

    # plt.scatter(data[:, 0], data[:, 1])

# 测试函数

if \_\_name\_\_ == "\_\_main\_\_":

    # 对每个数据集进行聚类

    dbcan\_and\_show("aniso.txt", 0.1, 10)

    dbcan\_and\_show("blobs.txt", 0.2, 10)

    dbcan\_and\_show("no\_structure.txt", 0.01, 10)

    dbcan\_and\_show("noisy\_circle.txt", 0.01, 10)

    dbcan\_and\_show("noisy\_moons.txt", 0.01, 10)

    dbcan\_and\_show("varied.txt", 0.45, 8)

    plt.tight\_layout()

    plt.show()

### 使用sklearn提供的库测试GMM分类器：gmm.py

import numpy as np

from sklearn.mixture import GaussianMixture

import matplotlib.pyplot as plt

from matplotlib import rcParams

# 设置中文字体

rcParams["font.family"] = "SimHei"

# 解决负号问题

rcParams["axes.unicode\_minus"] = False

# 定义数据集目录

directory = "data/"

# 聚类并展示结果

def gmm\_and\_show(file, num\_class):

    data = np.loadtxt(directory + file)

    # 创建并拟合 GMM 模型

    gmm = GaussianMixture(n\_components=num\_class)

    gmm.fit(data)

    # 输出每个样本的所属聚类

    labels = gmm.predict(data)

    plt.figure()

    # 分类绘制聚类结果

    for label in np.unique(labels):

        ind = labels == label

        plt.scatter(data[ind, 0], data[ind, 1])

    plt.title(f"数据集{file}使用GMM聚类")

    print(f"数据集{file}使用GMM聚类")

# 测试函数

if \_\_name\_\_ == "\_\_main\_\_":

    gmm\_and\_show("aniso.txt", 3)

    gmm\_and\_show("blobs.txt", 3)

    gmm\_and\_show("no\_structure.txt", 1)

    gmm\_and\_show("noisy\_circle.txt", 2)

    gmm\_and\_show("noisy\_moons.txt", 2)

    gmm\_and\_show("varied.txt", 3)

    plt.tight\_layout()

    plt.show()

### 测试上面提到的分类器，画图并计算评价系数：test.py

# 导入轮廓系数和CH指数计算函数

from sklearn.metrics import silhouette\_score, calinski\_harabasz\_score

import numpy as np

from matplotlib import rcParams

from sklearn.mixture import GaussianMixture

import matplotlib.pyplot as plt

# 导入我之前对聚类算法的实现

import kmeans

import dbscan

# 设置中文字体

rcParams["font.family"] = "SimHei"

# 解决负号问题

rcParams["axes.unicode\_minus"] = False

# 定义数据集目录

directory = "data/"

def plot\_and\_compare(file\_name, k, epsilon, minpts, num\_class):

    print(f"数据集{file\_name}：\n")

    fig = plt.figure(figsize=(17, 9))

    data = np.loadtxt(directory + file\_name)

    # 使用K-Means聚类

    # 曼哈顿距离

    dist\_fun = kmeans.Manhattan\_Distance

    centers = kmeans.kmenas\_train(data, dist\_fun, k)

    labels\_k\_means = dist\_fun(centers, data)

    # 计算轮廓指数

    s\_s = silhouette\_score(data, labels\_k\_means)

    # 计算CH指数

    c\_h = calinski\_harabasz\_score(data, labels\_k\_means)

    # 创建子图

    ax = fig.add\_subplot(2, 2, 1)

    ax.scatter(data[:, 0], data[:, 1], c=labels\_k\_means)

    ax.set\_title(f"使用K-Means（曼哈顿距离）聚类，轮廓系数：{s\_s:.4},CH指数：{c\_h:.3}")

    # 余弦相似度

    dist\_fun = kmeans.Cosine\_Similarity

    centers = kmeans.kmenas\_train(data, dist\_fun, k)

    labels\_k\_means = dist\_fun(centers, data)

    # 计算轮廓指数

    s\_s = silhouette\_score(data, labels\_k\_means)

    # 计算CH指数

    c\_h = calinski\_harabasz\_score(data, labels\_k\_means)

    # 创建子图

    ax = fig.add\_subplot(2, 2, 2)

    ax.scatter(data[:, 0], data[:, 1], c=labels\_k\_means)

    ax.set\_title(f"使用K-Means（余弦相似度）聚类，轮廓系数：{s\_s:.4},CH指数：{c\_h:.3}")

    # 使用DBSCAN聚类

    data\_as\_tuples = [tuple(point) for point in data]

    C, k = dbscan.dbscan(data\_as\_tuples, epsilon, minpts)

    labels\_dbscan = []

    data\_dbscan = []

    # 从类别中取出标签

    for idx, c in enumerate(C):

        for x in c:

            a, b = x

            labels\_dbscan.append(idx)

            data\_dbscan.append([a, b])

    data\_dbscan = np.array(data\_dbscan)

    labels\_dbscan = np.array(labels\_dbscan)

    # 计算轮廓指数

    s\_s = silhouette\_score(data\_dbscan, labels\_dbscan)

    # 计算CH指数

    c\_h = calinski\_harabasz\_score(data\_dbscan, labels\_dbscan)

    # 创建子图

    ax = fig.add\_subplot(2, 2, 3)

    ax.scatter(data\_dbscan[:, 0], data\_dbscan[:, 1], c=labels\_dbscan)

    ax.set\_title(f"使用DBSCAN聚类，轮廓系数：{s\_s:.4},CH指数：{c\_h:.3}")

    # 使用GMM聚类

    gmm = GaussianMixture(n\_components=num\_class)

    gmm.fit(data)

    # 输出每个样本的所属聚类

    labels\_gmm = gmm.predict(data)

    # 计算轮廓指数

    s\_s = silhouette\_score(data, labels\_gmm)

    # 计算CH指数

    c\_h = calinski\_harabasz\_score(data, labels\_gmm)

    # 创建子图

    ax = fig.add\_subplot(2, 2, 4)

    ax.scatter(data[:, 0], data[:, 1], c=labels\_gmm)

    ax.set\_title(f"使用GMM聚类，轮廓系数：{s\_s:.4},CH指数：{c\_h:.3}")

    plt.suptitle(f"数据集{file\_name}四种聚类方案之间对比")

    print(f"数据集{file\_name}四种聚类方案之间对比")

    plt.tight\_layout()

    plt.show()

plot\_and\_compare("aniso.txt", 3, 0.1, 10, 3)

plot\_and\_compare("blobs.txt", 3, 0.2, 10, 3)

plot\_and\_compare("no\_structure.txt", 1, 0.01, 10, 1)

plot\_and\_compare("noisy\_circle.txt", 2, 0.01, 3, 2)

plot\_and\_compare("noisy\_moons.txt", 2, 0.01, 5, 2)

plot\_and\_compare("varied.txt", 3, 0.5, 8, 3)