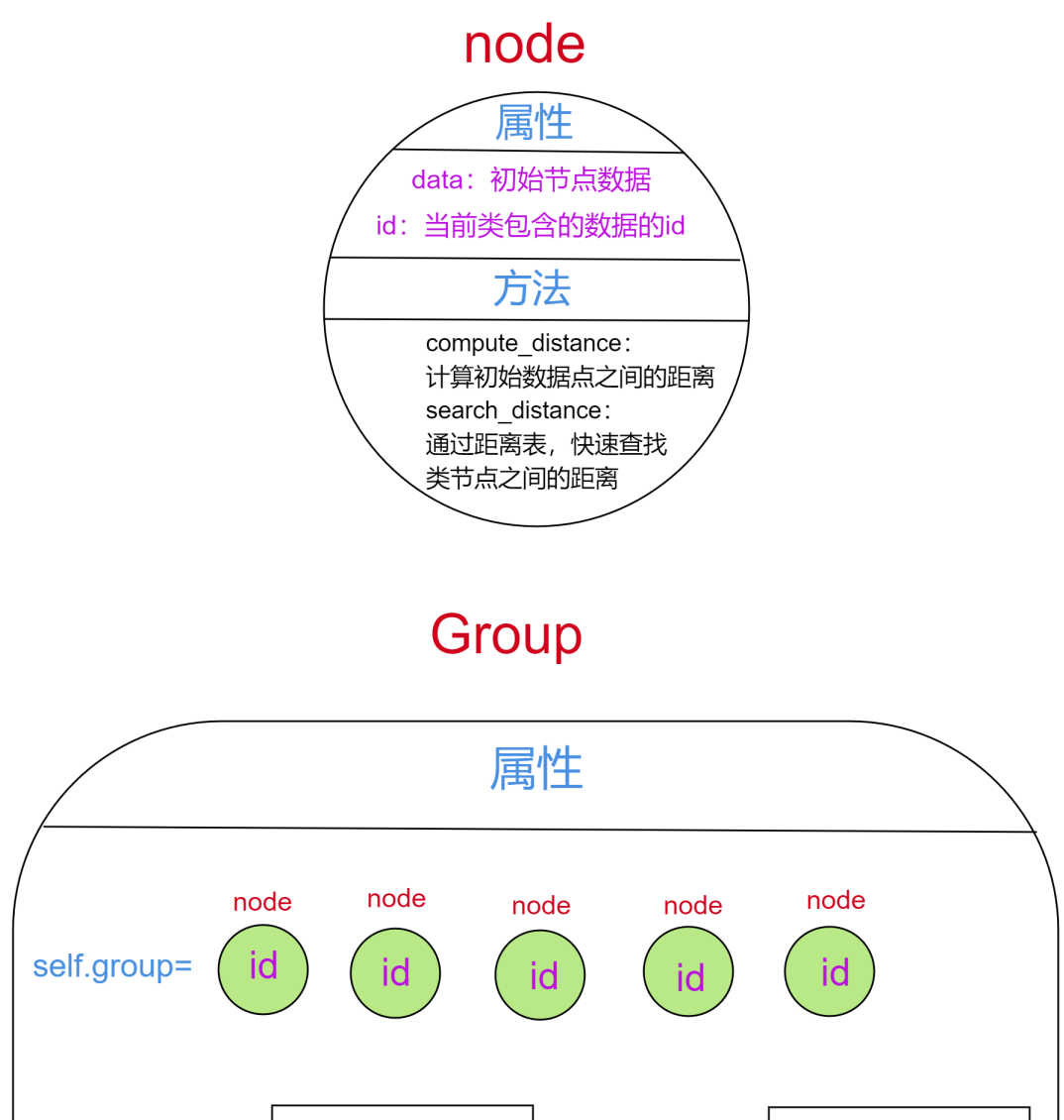
聚类算法

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
```

一、层次聚类(聚合算法)



通过如下操作，可以获得数组中最小元素的下标，记住这种方法，后面有用

In [2]:
```
a=np.array([[54,2,3],[1,6,7]])
i=int(a.argmin()/a.shape[1])
j=a.argmin()-a.shape[1]*i
print("最小值下标:",i,j)
print("最小值:",a[i][j])
```

```
最小值下标: 1 0
最小值: 1
```

In [3]:
```python
def get_argmin(a:np.array)->tuple:
    i=int(a.argmin()/a.shape[1])
    j=a.argmin()-a.shape[1]*i
    return i,j
```

定义欧几里得距离

In [4]:
```python
def Euild_distance(a,b):
    return np.sqrt(np.sum((a-b)**2))
```

In [5]:
```python
a=np.array([1,2,3])
b=np.array([4,5,6])
Euild_distance(a,b)**2
```

Out[5]:
```
27.0
```

定义节点，用来表示每一个数据点

In [6]:
```python
class Node_:
    def __init__(self,data=None,id=None,isleaf=False):
        self.data=data      # 初始节点的数据，用于计算初始距离
        self.id=[]
        self.id+=id
        self.left=None      # 本来想用递归算法的，但是由于每个非叶子节点可以动态更新id属性，所以left，right属性没用，这里没删
        self.right=None
        self.isleaf=isleaf

    @staticmethod
    def Euild_distance(a,b):
        return np.sqrt(np.sum((a-b)**2))

    # 第一次计算所有节点之间的相互距离
    def compute_distance(self,other):
        assert self is not other
        assert self.isleaf and other.isleaf
        return Euild_distance(self.data,other.data)

    # 通过查表获得节点之间的距离
    def search_distance(self,other,D):
        temp=np.array([D[i,j] for i in self.id for j in other.id])
        return temp.min()

    def __str__(self):
        return "id={}".format(self.id)
```

In [7]:
```python
a=Node_(np.array([1,2,3]),id=[],isleaf=True)
b=Node_(np.array([4,5,6]),id=[],isleaf=True)
a.compute_distance(b)**2
```

Out[7]:
```
27.0
```

定义一个Group，用来存放节点，并能对节点进行各种操作

In [8]:
```python
class Group_:
    def __init__(self,X):
        self.group=[Node_(data=X[i],id=[i],isleaf=True) for i in range(X.shape[0])]  # 把矩阵的每一行数据做成一个节点，并编上号
        self.initial_distance=None # 初始所有节点的相互距离
        self.distance=None         # 节点合并后，类和类之间的相互距离
        self.distance_initial()
        self.labels_=np.zeros(X.shape[0])

    @staticmethod
    def get_argmin(a:np.array)->tuple:
        i=int(a.argmin()/a.shape[1])
        j=a.argmin()-a.shape[1]*i
        return i,j

    # 计算初始距离矩阵，并把距离信息存储到节点中，每个节点都保存有它与其他节点的距离
    def distance_initial(self):
        size=len(self.group)
        self.initial_distance=np.full((size,size),np.inf)
        for i in range(size):
            for j in range(size):
                if i==j:continue
                self.initial_distance[i][j]=self.group[i].compute_distance(self.group[j])

    # 计算当前所有类节点的相互距离
    def compute_distance(self):
        size=len(self.group)
        self.distance=np.full((size,size),np.inf)
        for i in range(size):
            for j in range(size):
                if i==j:continue
                self.distance[i][j]=self.group[i].search_distance(self.group[j],self.initial_distance)
```

```
    def merge(self,node1,node2):
        newid=node1.id+node2.id
        newnode=Node_(id=newid)
        newnode.left=node1
        newnode.right=node2
        return newnode

    def cluster(self,k):
        while len(self.group)>k:
            self.compute_distance()
            i,j=get_argmin(self.distance)
            assert i!=j
            newnode=self.merge(self.group[i],self.group[j])
            del self.group[i]
            if i<j:
                del self.group[j-1]  # 注意前面删除i节点后，所有节点下标少1
            else:
                del self.group[j]   # 如果i在后面，j下标不受影响，这个很容易漏掉导致错误很难检查出来
            self.group.append(newnode)

        # 更新样本类别标签
        size=len(self.group)
        for i in range(size):
            for id in self.group[i].id:
                self.labels_[id]=i

    def result(self):
        size=len(self.group)
        for i in range(size):
            print("类别{}:".format(i))
            print(self.group[i])

    # 李航课本里面的例子，只有样本之间的距离
    def lihang_p262(self,D):
        self.group=[Node_(id=[i],isleaf=True) for i in range(D.shape[0])] # 每个样本点data=None,只有编号
        size=len(self.group)
        self.initial_distance=np.full((size,size),np.inf)
        for i in range(size):
            for j in range(size):
                if i==j:continue
                self.initial_distance[i][j]=D[i,j]                # 把距离矩阵赋值过来，并保存在每个节点当中
```

验证李航第二版课本p262上面的例题

注意算法里面的类别编号从0开始，李航书里面的类别编号从1开始，相差1

In [9]:
```
D=np.array([[0,7,2,9,3],[7,0,5,4,6],[2,5,0,8,1],[9,4,8,0,5],[3,6,1,5,0]])
```

验证第(2)步的结果，有4个类别{id=3,5}、{id=1}、{id=4}、{id=2}

因此应该输出4个类别：{id=2,4}、{id=0}、{id=3}、{id=1}

In [10]:
```
mm=Group_(D)
mm.lihang_p262(D)  # 这个接口直接把初始节点之间的距离矩阵传进去
mm.cluster(k=4)
mm.result()
```

```
类别0:
id=[0]
类别1:
id=[1]
类别2:
id=[3]
类别3:
id=[2, 4]
```

验证第(3)步的结果，有3个类别{id=1,3,5}、{id=4}、{id=2}
对应应该输出：{id=0,2,4}、{id=3}、{id=1}

In [11]:
```
mm.lihang_p262(D)
mm.cluster(k=3)
mm.result()
```

```
类别0:
id=[1]
类别1:
id=[3]
类别2:
id=[0, 2, 4]
```

验证第(4)步的结果，有2个类别{id=1,3,5}、{id=2,4}
对应应该输出：{id=0,2,4}、{id=1,3}

In [12]:
```
mm.lihang_p262(D)
mm.cluster(k=2)
mm.result()
```

```
类别0:
id=[0, 2, 4]
类别1:
id=[1, 3]
```

鸢尾花数据集聚类并可视化

In [13]:
```
data=pd.read_csv("data/Iris.csv",header=0,index_col="Id")
data.head()
```

Out[13]:

Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5.0	3.6	1.4	0.2	Iris-setosa

In [14]:
```
data=data.iloc[:,:2]
data.head(),data.values.shape
```

Out[14]:
```
(    SepalLengthCm  SepalWidthCm
 Id
 1             5.1           3.5
 2             4.9           3.0
 3             4.7           3.2
 4             4.6           3.1
 5             5.0           3.6,
 (150, 2))
```

In [15]:
```
mm=Group_(data.values)
mm.initial_distance[:5,:5]
```

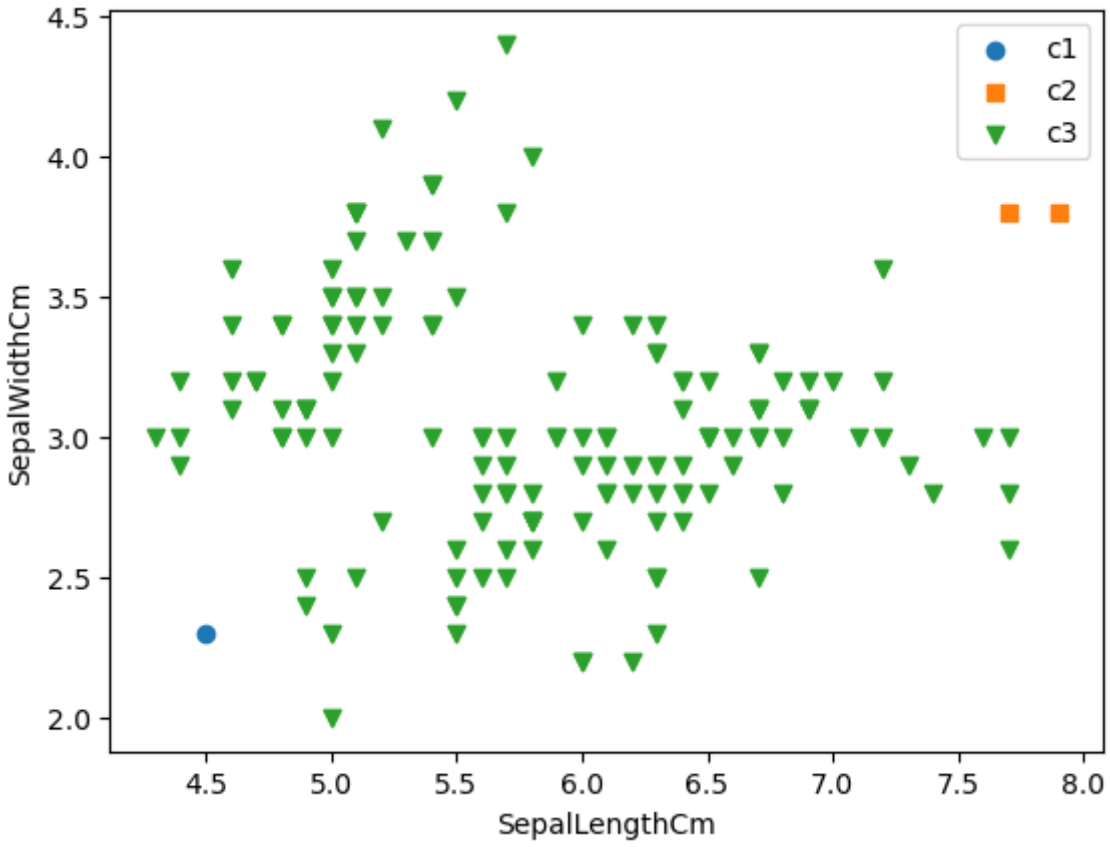
Out[15]:
```
array([[       inf, 0.53851648, 0.5       , 0.64031242, 0.14142136],
       [0.53851648,        inf, 0.28284271, 0.31622777, 0.60827625],
       [0.5       , 0.28284271,        inf, 0.14142136, 0.5       ],
       [0.64031242, 0.31622777, 0.14142136,        inf, 0.64031242],
       [0.14142136, 0.60827625, 0.5       , 0.64031242,        inf]])
```

In [16]:
```
mm.cluster(k=3)
mm.result()
print(mm.labels_)
```

```
类别0:
id=[41]
类别1:
id=[117, 131]
类别2:
id=[109, 60, 59, 98, 93, 57, 106, 108, 105, 135, 118, 122, 62, 119, 68, 87, 125, 130, 107, 102, 129, 84, 53, 80, 81, 90, 69, 89, 94, 121, 64,
66, 88, 113, 79, 92, 67, 82, 101, 142, 114, 55, 99, 95, 96, 72, 146, 85, 76, 70, 56, 100, 136, 148, 134, 83, 91, 127, 138, 61, 149, 71, 73, 6
3, 78, 74, 111, 54, 128, 132, 97, 103, 126, 123, 133, 124, 144, 143, 141, 52, 139, 50, 120, 137, 110, 51, 115, 58, 75, 147, 104, 116, 140, 65,
86, 112, 77, 145, 15, 33, 14, 18, 32, 5, 16, 10, 48, 36, 20, 31, 13, 8, 38, 11, 24, 22, 6, 42, 21, 46, 19, 44, 27, 28, 4, 0, 17, 40, 43, 35, 2
3, 49, 39, 7, 26, 2, 29, 3, 47, 30, 12, 45, 37, 9, 34, 1, 25]
[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 0. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2.]
```

In [17]:
```
c1_x=data.iloc[mm.group[0].id,0]
c1_y=data.iloc[mm.group[0].id,1]

c2_x=data.iloc[mm.group[1].id,0]
c2_y=data.iloc[mm.group[1].id,1]

c3_x=data.iloc[mm.group[2].id,0]
c3_y=data.iloc[mm.group[2].id,1]

plt.scatter(c1_x,c1_y,marker="o",label="c1")
plt.scatter(c2_x,c2_y,marker="s",label="c2")
plt.scatter(c3_x,c3_y,marker="v",label="c3")
plt.xlabel("SepalLengthCm")
plt.ylabel("SepalWidthCm")
plt.legend()
```

Out[17]:
```
<matplotlib.legend.Legend at 0x23b5a899580>
```



使用sklearn的层次聚类

参考链接

In [18]:
```
cluster=AgglomerativeClustering(3)
cluster.fit(data.values)
cluster.labels_
```

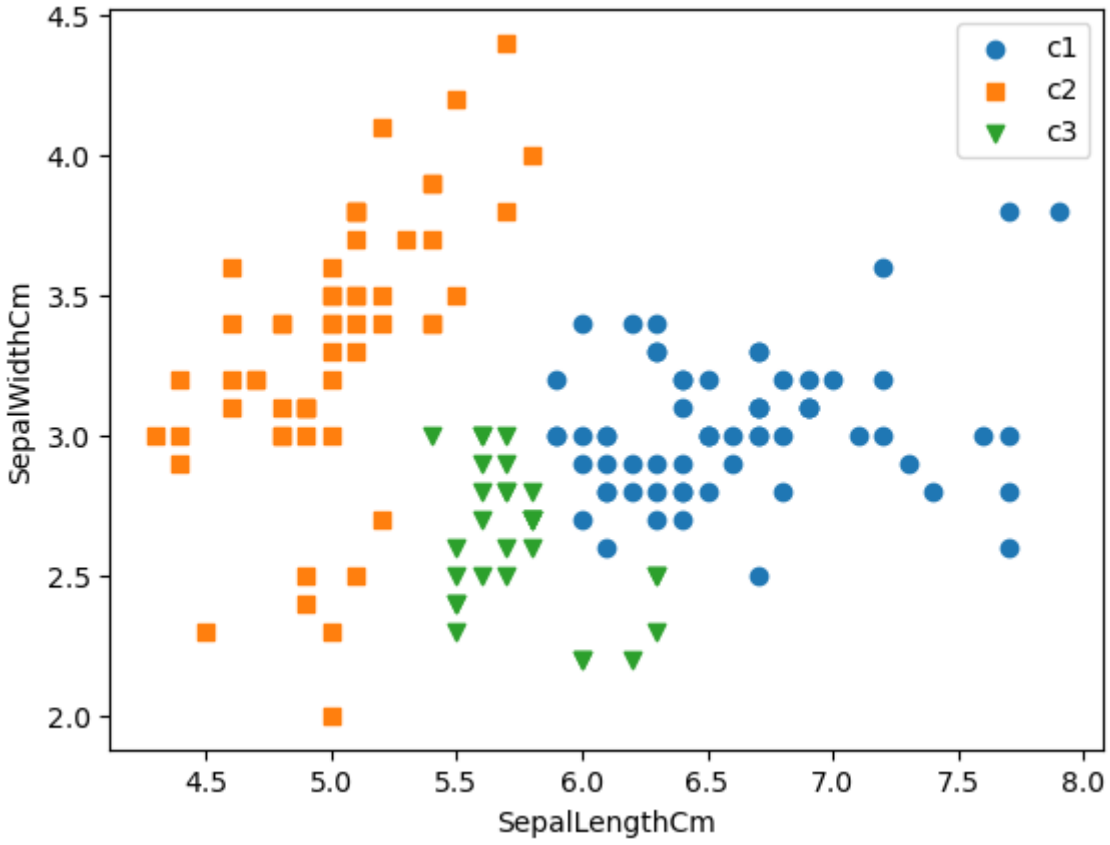
Out[18]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 0, 2, 0, 2, 0, 1, 0, 1, 1, 0, 2, 0, 2, 0,
       2, 2, 2, 2, 0, 0, 2, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0, 2, 0, 0, 2,
       2, 2, 2, 0, 2, 1, 2, 2, 2, 0, 1, 2, 0, 2, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0], dtype=int64)
```

In [19]:
```
c1=data.values[cluster.labels_==np.array(0),:]
c2=data.values[cluster.labels_==np.array(1),:]
c3=data.values[cluster.labels_==np.array(2),:]
c1.shape,c2.shape,c3.shape
```

Out[19]:
```
((64, 2), (56, 2), (30, 2))
```

In [20]:
```
plt.scatter(c1[:,0],c1[:,1],marker="o",label="c1")
plt.scatter(c2[:,0],c2[:,1],marker="s",label="c2")
plt.scatter(c3[:,0],c3[:,1],marker="v",label="c3")
plt.xlabel("SepalLengthCm")
plt.ylabel("SepalWidthCm")
plt.legend()
```

Out[20]:
```
<matplotlib.legend.Legend at 0x23b5a91a8b0>
```



为什么结果不一样？ 因为自己写的算法，计算类和类之间的距离的时候，用的是两个类当中最近的两个点的距离
而sklearn中的类间距离用的是linkage distance，具体请参考上面给的链接

我写算法的时候完全按照李航书上那个，例题的思路来写的，例题验证完全正确，
但是一用在鸢尾花数据集上，效果很差，和库里面

的完全不一样，所以各种检查错误，中途还重构了一次算法（第一次的算法利用递归算的距离）
翻过来倒过去，各种检查bug，愣是没找到，最后发现问题出在类间距离不同。

所以，书上给的代码，以及你网上查找的代码，以及库里面的代码，实现的可能都不是一个东西
意识到这一点再去看结果，查bug

算法重构

类间距离：类中心之间的距离

In [21]:
```
class Node:
    def __init__(self,data=None,id=None,isleaf=False):
        self.data=data
        self.id=[]
        self.id+=id
        self.center=None
        self.left=None
        self.right=None
        self.isleaf=isleaf

    @staticmethod
    def Euild_distance(a,b):
        return np.sqrt(np.sum((a-b)**2))

    def center_distance(self,other):
        return Euild_distance(self.center,other.center)

    def compute_center(self,X):
        for id in self.id:
            if self.center is None:
                self.center=X[id]
            else:
                self.center=self.center+X[id] # 这里有个+=原地操作的坑
        self.center=self.center/len(self.id)

    def __str__(self):
        return "id={}".format(self.id)
```

In [22]:
```
class Group:
    def __init__(self,X):
        self.X = X
        self.group=[Node(data=X[i],id=[i],isleaf=True) for i in range(X.shape[0])]
        self.distance=None
        self.labels_=np.zeros(X.shape[0])

        # 第一次初始化后计算各类的中心节点
        for node in self.group:
            node.compute_center(X)

    @staticmethod
    def get_argmin(a:np.array)->tuple:
        i=int(a.argmin()/a.shape[1])
        j=a.argmin()-a.shape[1]*i
        return i,j

    def compute_distance(self):
        size=len(self.group)
        self.distance=np.full((size,size),np.inf)
        for i in range(size):
            for j in range(size):
                if i==j:continue
                self.distance[i][j]=self.group[i].center_distance(self.group[j])

    def merge(self,node1,node2):
        newid=node1.id+node2.id
        newnode=Node(id=newid)
        newnode.compute_center(self.X)
        newnode.left=node1
        newnode.right=node2
        return newnode

    def cluster(self,k):
        while len(self.group)>k:
            self.compute_distance()
            i,j=get_argmin(self.distance)
            assert i!=j
            newnode=self.merge(self.group[i],self.group[j])
            del self.group[i]
            if i<j:
                del self.group[j-1]
            else:
                del self.group[j]
            self.group.append(newnode)

        size=len(self.group)
        for i in range(size):
            for id in self.group[i].id:
                self.labels_[id]=i

    def result(self):
        size=len(self.group)
```

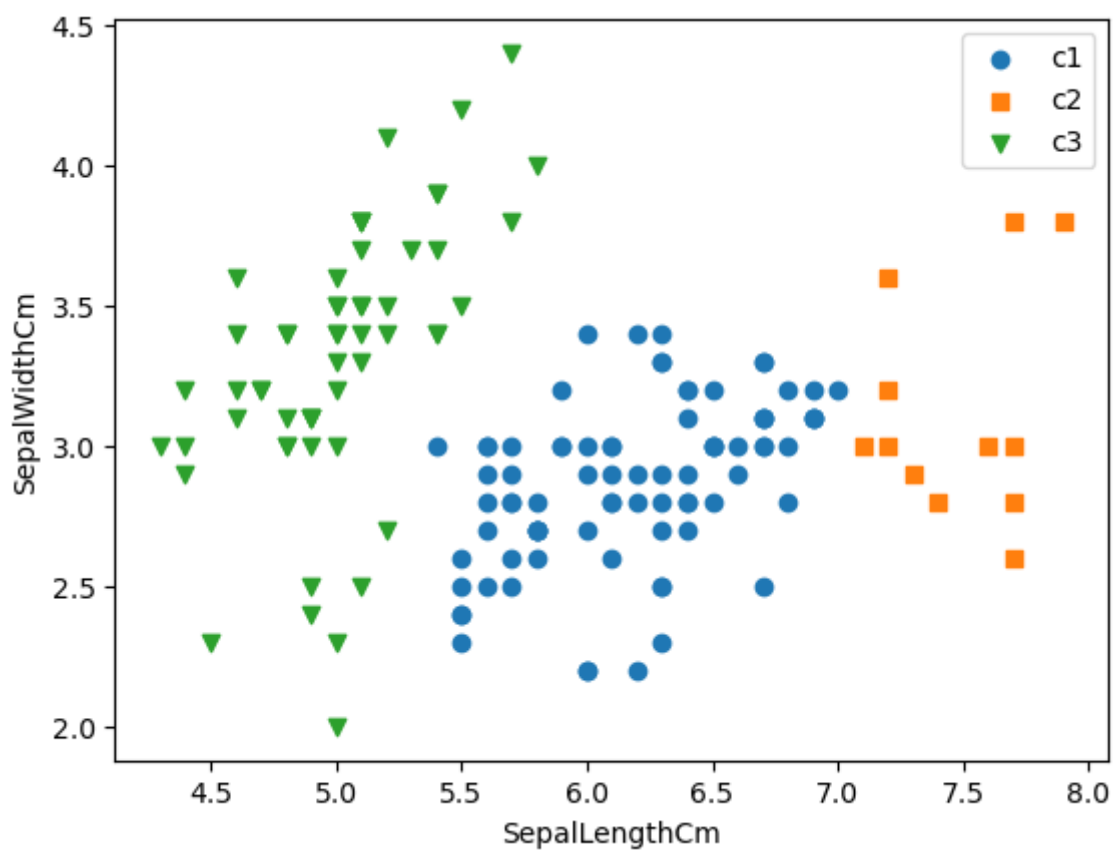
```
        for i in range(size):
            print("类别{}:".format(i))
            print(self.group[i])
```

In [23]:
```
data=pd.read_csv("data/Iris.csv",header=0,index_col="Id")
data=data.iloc[:,:2]
mm=Group(data.values)
mm.cluster(k=3)
mm.result()
print(mm.labels_)
```

```
类别0:
id=[108, 76, 140, 65, 86, 112, 77, 145, 58, 75, 147, 104, 116, 124, 144, 143, 141, 52, 139, 50, 120, 83, 134, 138, 61, 149, 91, 127, 63, 78, 9
7, 103, 126, 71, 73, 72, 146, 123, 133, 74, 111, 54, 128, 132, 70, 85, 137, 110, 51, 115, 56, 100, 136, 148, 62, 119, 68, 87, 53, 80, 81, 90,
69, 89, 113, 79, 92, 94, 121, 67, 82, 101, 142, 114, 55, 99, 84, 64, 66, 88, 95, 96]
类别1:
id=[107, 130, 125, 102, 129, 105, 135, 118, 122, 109, 117, 131]
类别2:
id=[14, 18, 15, 33, 41, 60, 59, 98, 93, 57, 106, 32, 21, 46, 19, 44, 5, 16, 10, 48, 36, 20, 31, 27, 28, 23, 49, 39, 7, 26, 4, 0, 17, 40, 43, 4
2, 13, 8, 38, 11, 24, 6, 22, 2, 29, 3, 47, 35, 30, 12, 45, 37, 9, 34, 1, 25]
[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 0. 0. 0. 0. 0. 0. 0. 2. 0. 2. 2. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 2. 0. 0.
 0. 0. 2. 0. 0. 0. 1. 0. 0. 1. 2. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0.
 0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
```

In [24]:
```
c1_x=data.iloc[mm.group[0].id,0]
c1_y=data.iloc[mm.group[0].id,1]

c2_x=data.iloc[mm.group[1].id,0]
c2_y=data.iloc[mm.group[1].id,1]

c3_x=data.iloc[mm.group[2].id,0]
c3_y=data.iloc[mm.group[2].id,1]

plt.scatter(c1_x,c1_y,marker="o",label="c1")
plt.scatter(c2_x,c2_y,marker="s",label="c2")
plt.scatter(c3_x,c3_y,marker="v",label="c3")
plt.xlabel("SepalLengthCm")
plt.ylabel("SepalWidthCm")
plt.legend()
```

Out[24]:
```
<matplotlib.legend.Legend at 0x23b5a7e0070>
```



In [25]:
```
data.head() #检查数据有没有被修改
```

Out[25]:

Id	SepalLengthCm	SepalWidthCm
1	5.1	3.5
2	4.9	3.0
3	4.7	3.2
4	4.6	3.1
5	5.0	3.6

二、前面的聚类算法存在什么问题?

请先看下面这个Rings的两个实验

案例：Rings的聚类

数据集简单一览

In [26]:
```
data=pd.read_csv("data/rings.csv",header=0)
data.head()
```

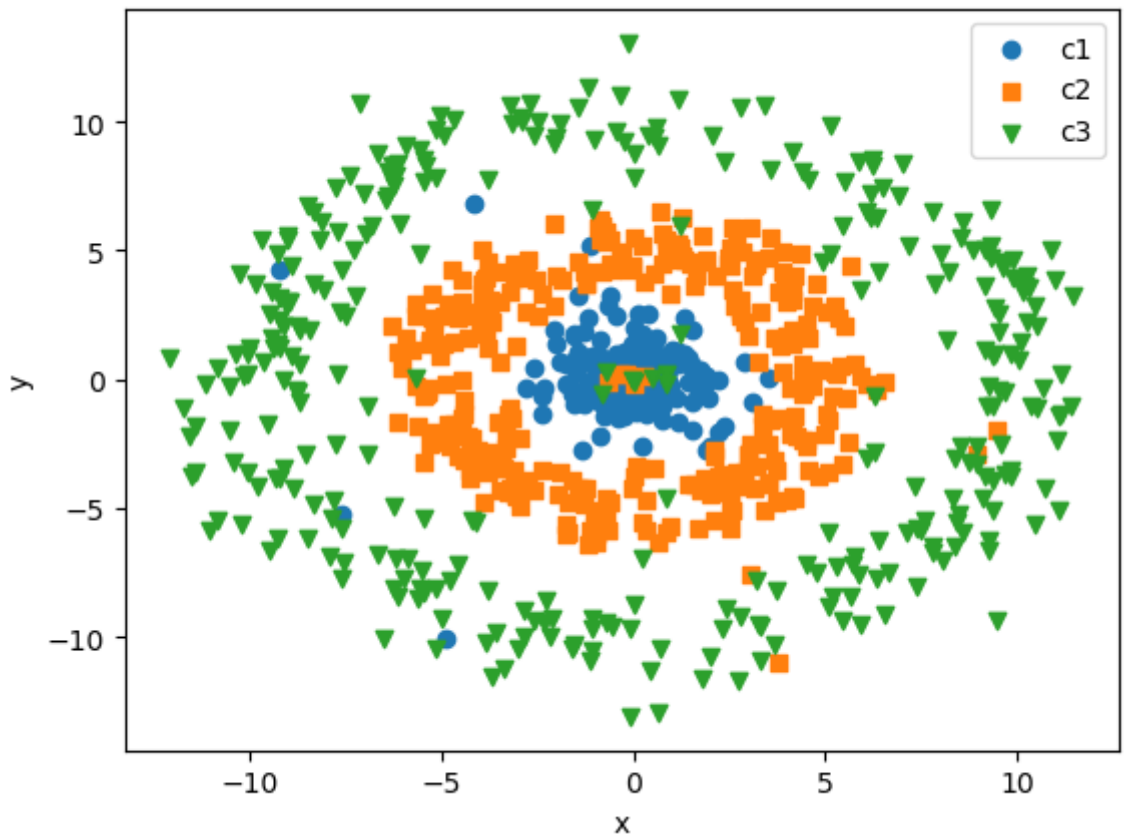
Out[26]:

	x	y	class
0	-0.127192	-0.194999	0
1	-10.070178	0.973600	2
2	1.909451	-4.072703	1
3	0.585665	9.816200	2
4	-5.459372	8.536418	2

按给的标签划分数据并可视化

In [27]:
```
# 取标签
index1=data["class"]==np.array(0)
index2=data["class"]==np.array(1)
index3=data["class"]==np.array(2)

# 取出每个类别的横坐标和纵坐标
data=data.values[:,:2]

c1_x=data[index1,0]
c1_y=data[index1,1]

c2_x=data[index2,0]
c2_y=data[index2,1]

c3_x=data[index3,0]
c3_y=data[index3,1]

#画图
plt.scatter(c1_x,c1_y,marker="o",label="c1")
plt.scatter(c2_x,c2_y,marker="s",label="c2")
plt.scatter(c3_x,c3_y,marker="v",label="c3")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
```

Out[27]:
```
<matplotlib.legend.Legend at 0x23b5a9e2f70>
```

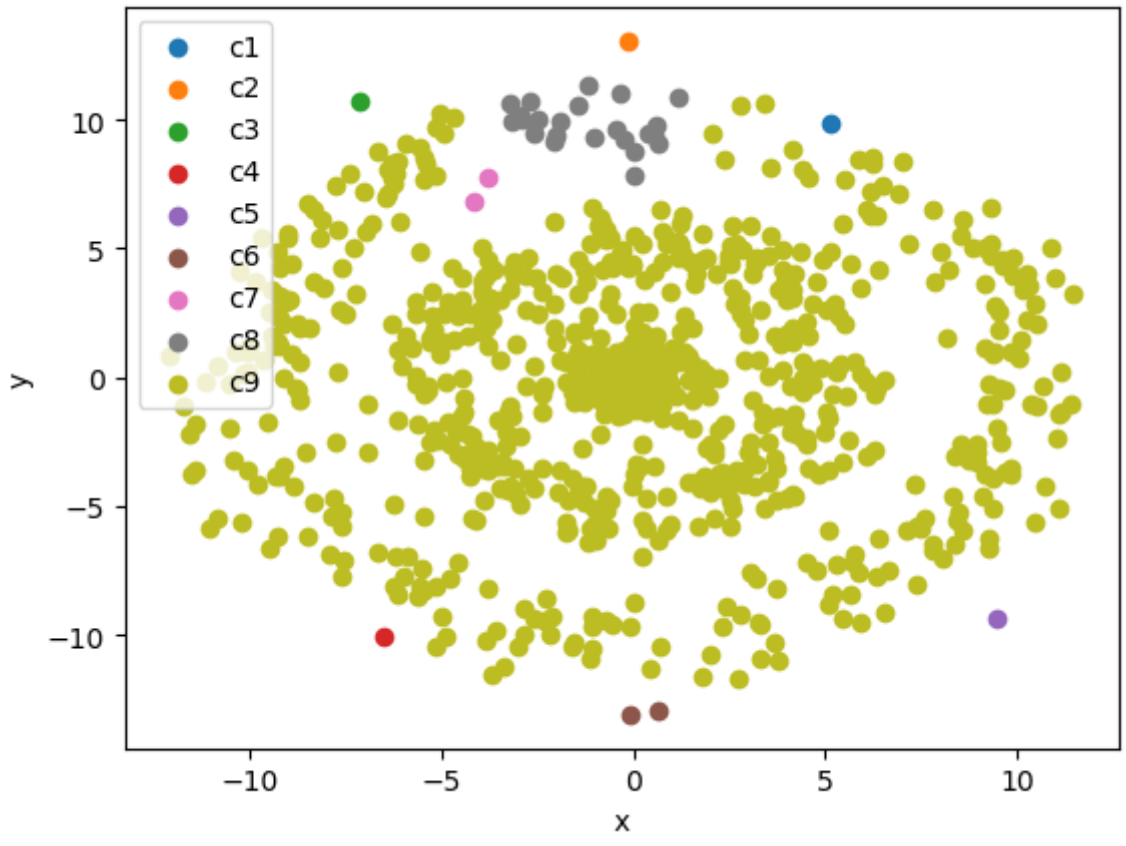


实验一：使用自己写的聚类算法分成9类并可视化（一）

类间距离采用两个类中的点之间的最小距离
由下图可以看出，只要是挨着一块的，都会被自动归到一个类别当中
所以会存在很多噪音点自成一类

In [28]:
```
data=pd.read_csv("data/rings.csv",header=0).iloc[:,[0,1]]
mm=Group_(data.values)
mm.cluster(k=9)
labels=["c1","c2","c3","c4","c5","c6","c7","c8","c9","c10","c11","c12"]
for i in range(9):
    plt.scatter(data.iloc[mm.group[i].id,0],data.iloc[mm.group[i].id,1],marker="o",label=labels[i])
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
```

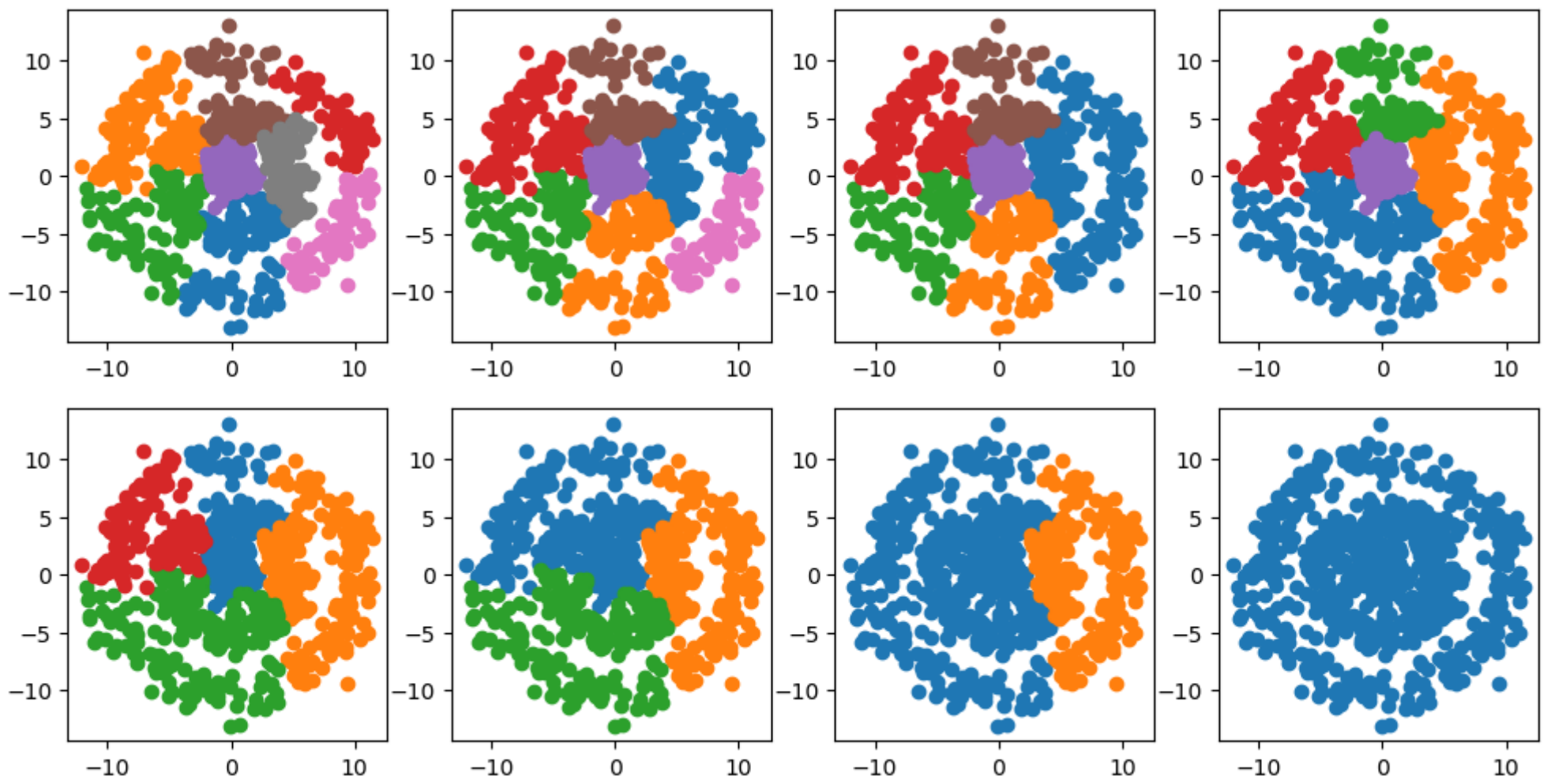
Out[28]: <matplotlib.legend.Legend at 0x23b5cb674f0>



实验二：使用sklern库里面的层次聚类并可视化

可以认为类间距离是类中心之间的距离
下图是分别聚类成8，7，6，5，4，3，2，1类的可视化
从图中可以看出，两层圆环都被分成了若干类别，那么就会产生
一个疑问，为什么圆环上的点挨得这么近，还是被分成多类别了呢?
这就需要思考类和类的合并规则，那就是使用类中心距离，虽然两个类
表面挨得很近，但是类中心距离不一定很近，如果你打算换个思路，使用两个类
中最近的两个点的距离作为合并规则，就会产生上面实验一的问题，如果真实的
类和类之间的距离不够明显，就会被直接合并成一个巨无霸大类。

In [29]:
```
fig,axes=plt.subplots(2,4)
fig.set_size_inches(12, 6)
labels=["c1","c2","c3","c4","c5","c6","c7","c8"]
k=8
for i in range(2):
    for j in range(4):  # 定位到某个坐标图
        cluster=AgglomerativeClustering(k)
        cluster.fit(data.values)
        for m in range(k):
            axes[i][j].scatter(data.values[cluster.labels_==np.array(m),0],data.values[cluster.labels_==np.array(m),1],label=labels[m])
        k=k-1
```



解决思路介绍（纯个人理解）

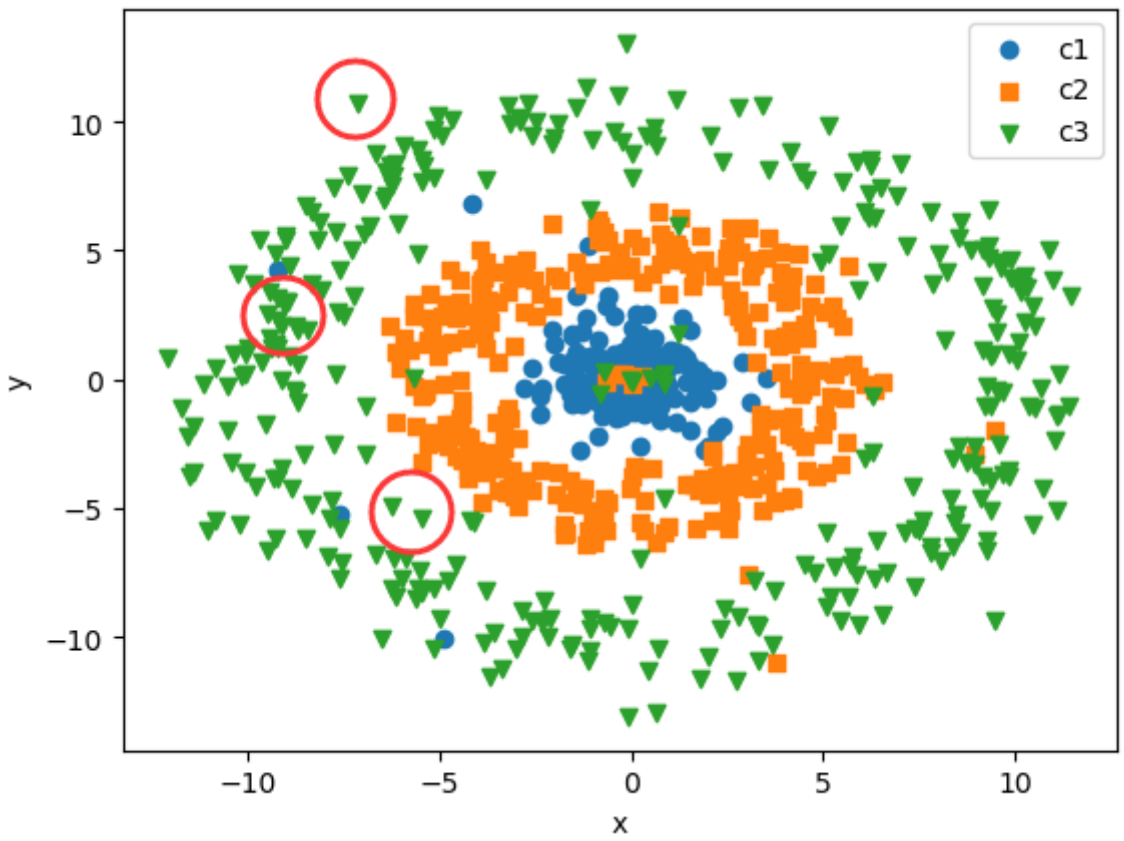
前面的算法各有缺点，如何克服这些缺点呢？这是下手点，对于上面两个圆环，我们希望中间的圆环是一类，外面的圆环是另一类，
但是前面的算法一不小心会把两层圆环合并，而单个圆环被切开，比如上面倒数第二张图，被分成左右两个部分。
这些算法的特点是每次会形成若干漩涡中心（重力中心），如果按照边界挨得近不近来合并漩涡，会形成巨无霸漩涡，不是我们要的，

如果按照漩涡中心来合并，就会忽略了边界的信息，会把看上去是一个整体的东西（比如中外两层圆环）给割裂开来。
无论是哪一种，形成的类都是漩涡这种球形，一团一团的形状，没办法形成各式各样的形状

如果合并的时候基于边界处的密度考虑呢，就是说，如果边界的地方点特别"密集"，我们就合并为一类。
首先这样会不会产生巨无霸，比如我们中间环一块密集的区域变成一个漩涡后，我们对这个区域周围的点
的密集程度进行考察，由于中间环与外环中间的点比较稀疏，基于我们定下的某种合并规则，是不会让中间环
与外环的任何一部分合并的，同理中间环与正中央有个稀疏带，这两个也不会合并

三、基于密度的聚类算法——DBSCAN

三个基本定义



以一个点x为中心，半径ϵ(超参数)为半径画圆，有下面三种情况:

情形一：如果周围包含N(也是超参数)个样本点，称x是核心点
情形二：周围没点，这个圆里面就x一个点，称x是噪音点
情形三：周围的点数量在1~N-1之间，称x是边界点
由定义可知，核心点周围的点可能是核心点，也可能是边界点，不可能是噪音点
上图中分别用红圈框住了三个圆，这三个圆的中心点就是核心点，边界点，噪音点

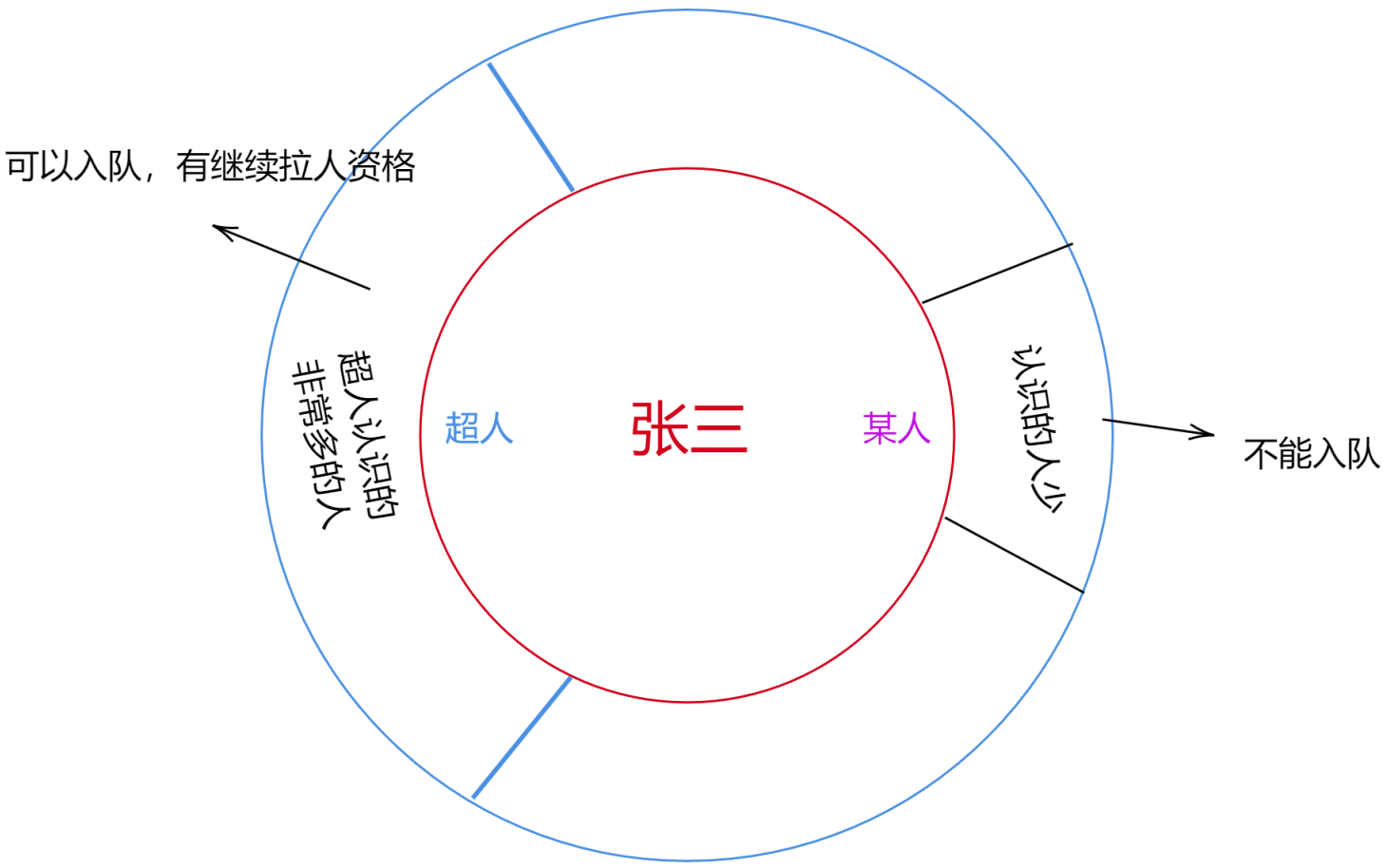
算法原理（定义核心点ϵ领域内的点都是同一类）：

简单起见，这里打个比方说明算法的执行过程，首先从一个点出发，设这个点是个近视眼张三，只能看到领域ϵ范围内的人（全是近视眼）
假如第一个人看到周围的人数超过阈值N=4，比如张三看到了6个人，那么就让张三和那6个人一起成立一个帮派，叫绿色帮，成立帮派的标志是给每个人胸前挂个牌子，写着绿色帮三个大字，然后刚成立帮派，张三就躺平了，后面拉新人的工作就交给另外6人了，我们设这个小队为拉人队，然后这个队列的第一个人李四开始拉人了，他看了一下周围，发现了5人（可以和之前的人重复），于是立马把这些人拉入绿色帮，并入队，然后李四躺平了（出队），接着排在李四后面的王二麻，开始观察周围，把看见的人胸前挂牌，拉入队伍，然后自己出队交给下一个人重复执行这个动作，当执行到一定时候，队伍里所有人周围都会没有新人了，这时候队头观察周围，看见0人，然后队头出队，最后拉人小队全部出队，队列为空的时候，绿色帮集合完毕。
比如从上面图中的最外环中的某个人开始，由于外环和中环中间有隔离带，所以外环的近视眼看不到中环的人的，所以按照上面这个步骤，外环的所有人会变成绿色帮。
接着从身上还没有牌子的人出发，比如小明身上还没牌子，小明四周看了下，空无一人，于是小明挂上了不入帮的牌子，表明自己不加入帮派，已经有立场了。
接着再从身上还没牌子的人出发，比如小华向四周看了下，看到了2个人，貌似2个人太少，直接成立帮派不够格，于是身上挂个"我待定"的牌子
后期如果其他人（其实就他旁边还没牌子的那两人）看到了他，并且有资格拉他进入帮派，他就可以换牌子，如果后期没人拉他入队，他就一直挂这个牌子，反正也是牌子。 接着再从身上没有牌子的人出发，比如小林，他刚好是中环的人，一次性把中环的人都变成了第二个帮派，橘色帮。
再接着从身上没牌子的人出发，比如小智，他是最里层的人，一次性把最里面的人变成了第三个帮派，蓝色帮。
至此，所有人都有了具体的帮派（染好了色），聚类完毕

备注：
有个小细节稍微需要改一下，就是张三拉入队的人再次去拉人有些条件限制，他如果能看到足够多的人，可以让这些人直接入队，如果他看到不够多的人，这些人
可以入帮，但不能入队。

这个概念叫密度可达，张三人脉多，称张三叫核心点，某人人脉少，称为边界点，如果边界点在核心点的邻域（某人是张三的朋友），某人可以从张三那里直接密度可达，
但是由于某人不是核心点，所以某人的朋友A不可以从某人出发直接密度可达。对于另外的核心点超人，他的朋友从超人出发密度可达，而超人从张三出发密度可达，所以
超人的朋友从张三密度可达。如果一段序列$x_1, x_2, \cdots, x_n$,有x_2从x_1密度可达，x_3从x_2密度可达，x_n从x_{n-1}密度可达，那么可以知道
$x_1, x_2, \cdots, x_{n-1}$均为核心点

DBSCAN的算法原理就是从一点x出发，找到与x密度可达的所有点，这些点的全体称为一个类别，也就是一个簇



张三直接认识的人，红圈中的人：直接入帮入队

张三第二层认识的人：

①.超人的朋友圈：全部可以入帮入队

②.某人的朋友圈，可以入帮，不可入队

算法实现的关键数据结构：

1.每个人用Node节点表示，牌子用id表示，默认为None，牌子的属性用0，1，2等数字表示
2.队伍queue使用一个列表来表示
3.画圈圈看圈圈里有多少人，主要通过查找距离表，比如距离张三的人的距离我都知道，看距离小于半径的有几个，就知道哪几个人在这个圈范围了
4.定义一个大Group类，统筹协调各个东西实现算法

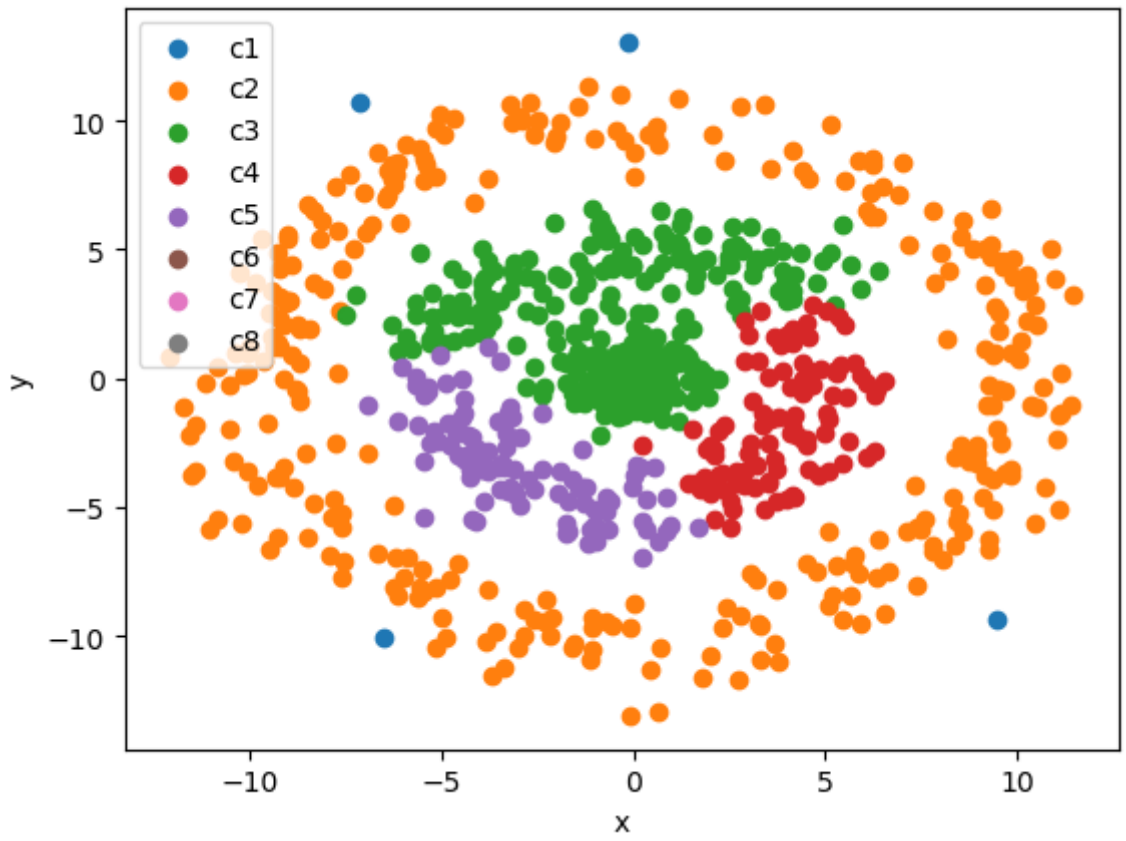
In [30]:
```python
class DBSCAN_Node:
    def __init__(self,data):
        self.data=data
        self.id=None
        self.tangping=False
        self.distance=None

    @staticmethod
    def Euild_distance(a,b):
        return np.sqrt(np.sum((a-b)**2))

    # 用于第一次计算所有节点之间的相互距离
    def compute_distance(self,other):
        assert self is not other
        return Euild_distance(self.data,other.data)

    # 在他近视眼范围内找人
    def neighborhood(self,epsilon,group):
        temp=np.argwhere(self.distance <= epsilon).squeeze(1).tolist()
        # 如果是空列表，直接返回
        if len(temp)==0:
            return temp
        else:
            neighbor=[group[i] for i in temp]
            return neighbor
```

In [31]:
```
class DBSCAN_Group:
    def __init__(self,X,epsilon=0.01,N=4):
        self.group=[DBSCAN_Node(data=X[i]) for i in range(X.shape[0])]  # 把矩阵的每一行数据做成一个节点，暂时不需要编号
        self.epsilon=epsilon
        self.N=N
        self.Distance=None # 所有人之间的相互距离
        self.distance_initial()
        self.queue=[]

        # 每个人身上获得他与别人的距离表
        for i in range(len(self.group)):
            self.group[i].distance=self.Distance[i]

        # 类别信息
        self.labels_=None

    # 计算初始距离矩阵
    def distance_initial(self):
        size=len(self.group)
        self.Distance=np.full((size,size),np.inf)
        for i in range(size):
            for j in range(size):
                if i==j:continue
                self.Distance[i][j]=self.group[i].compute_distance(self.group[j])

    # 把所有neighborhood成员加入到帮派中
    def join(self,neighborhood,bangpai_number):
        for x in neighborhood:
            x.id=bangpai_number


    def cluster(self):
        # 从每一个点依次出发
        size=len(self.group)
        bangpai_number=2
        for i in range(size):
            # 假如身上没牌子，从没牌子的人出发
            if self.group[i].id is None:
                # 找到附近所有人
                neighborhood=self.group[i].neighborhood(self.epsilon,self.group)
                # 附近没有人，牌子变成0，并且今后不会加入其他帮派
                if len(neighborhood) == 0:
                    self.group[i].id=0
                # 附近有人，但是不够变成一个帮派，暂时定为1，今后可以被其他人拉入帮派
                elif len(neighborhood)>0 and len(neighborhood)<self.N:
                    self.group[i].id=1
                # 附近有N及以上的人,首先自己变成帮派的人，然后把这群人也变成帮派的人
                else:
                    assert len(neighborhood)>=self.N
                    self.group[i].id=bangpai_number
                    self.group[i].tangping=True
                    self.join(neighborhood,bangpai_number)
                    # 把附近的人拉入拉人小队，继续拉人
                    self.queue+=neighborhood
                    while len(self.queue)>0:
                        if self.queue[0].tangping: # 如果队头元素躺平，也就是先前找过人，那么就不再找人，否则会变成死循环
                            del self.queue[0]
                        else:
                            neighborhood_2=self.queue[0].neighborhood(self.epsilon,self.group) # 张三朋友的朋友圈，第二层朋友圈
                            if len(neighborhood_2) >= self.N:                                  # queue[0]相当于超人，他的朋友圈neighborhood_2
                                self.queue[0].tangping=True                                    # 队头元素躺平
                                del self.queue[0]                                              # 超人找过人后就出队,后面即便再次入队，由于躺平
                                self.join(neighborhood_2,bangpai_number)                       # 由于超人的朋友圈密集，所以每个人都有入帮入队的
                                self.queue+=neighborhood_2                                     # 入队
                            if len(neighborhood_2) < self.N and len(neighborhood_2)>0 :        # queue[0]相当于某人，他的朋友圈neighborhood_2
                                self.queue[0].tangping=True                                    # 某人躺平
                                del self.queue[0]                                              # 某人出队
                                self.join(neighborhood_2,bangpai_number)                       # 至少是第二层朋友，和张三也算间接认识，有资格入

                    bangpai_number+=1
        # 更新样本标签
        self.labels_=[self.group[i].id for i in range(size)]
```

实验一

In [32]:
```
data=pd.read_csv("data/rings.csv",header=0).iloc[:,[0,1]]
mm=DBSCAN_Group(data.values,epsilon=1.45,N=20)
mm.cluster()

labels=["c1","c2","c3","c4","c5","c6","c7","c8","c9","c10","c11","c12"]
plt.figure()
for i in range(8):
    plt.scatter(data.values[np.array(mm.labels_)==np.array(i),0],data.values[np.array(mm.labels_)==np.array(i),1],label=labels[i])
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
```

Out[32]:
```
<matplotlib.legend.Legend at 0x23b5a7c4f40>
```

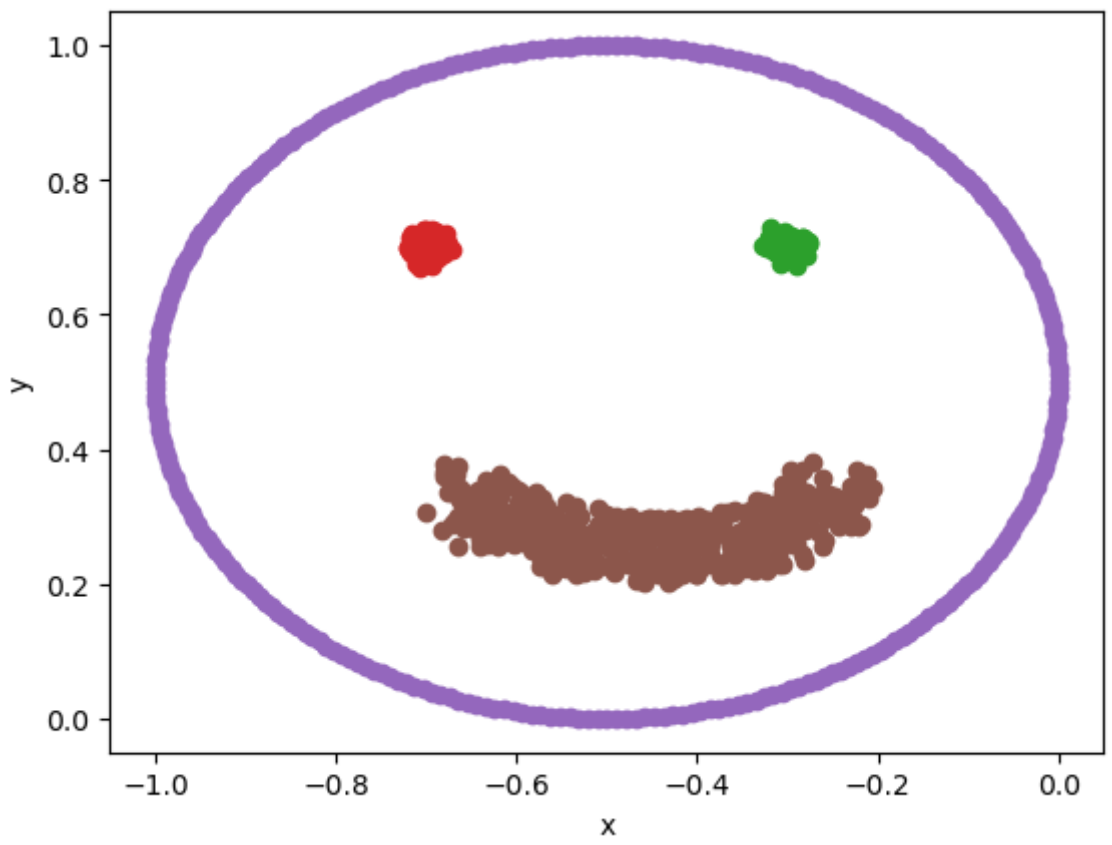


实验二

In [33]:
```
data=pd.read_csv("data/smile2.csv",header=0).iloc[:,[0,1]]
mm=DBSCAN_Group(data.values,epsilon=0.1,N=10)
mm.cluster()
```

In [34]:
```
labels=["c1","c2","c3","c4","c5","c6","c7","c8","c9","c10","c11","c12"]
plt.figure()
for i in range(8):
    plt.scatter(data.values[np.array(mm.labels_)==np.array(i),0],data.values[np.array(mm.labels_)==np.array(i),1],label=labels[i])
plt.xlabel("x")
plt.ylabel("y")
```

Out[34]:
```
Text(0, 0.5, 'y')
```

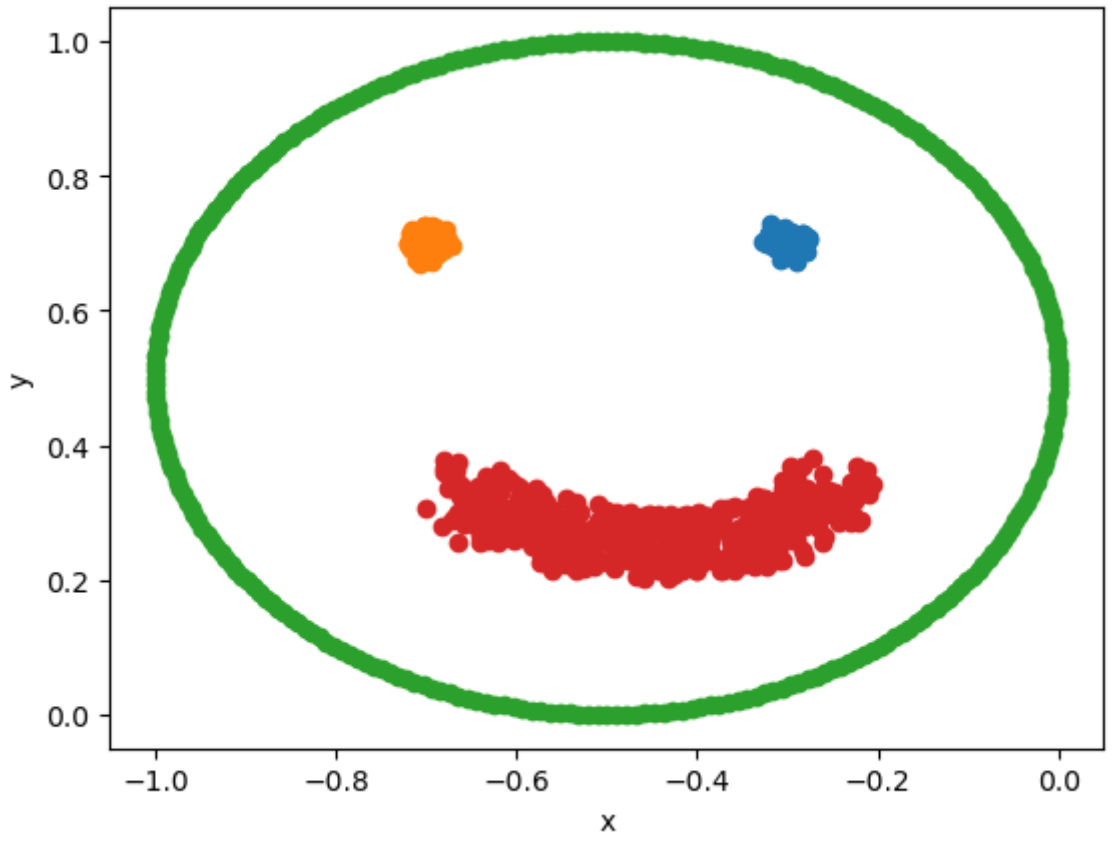


实验三：使用sklearn中的DBSCAN

In [35]:
```
from sklearn.cluster import DBSCAN
from sklearn import metrics
mm = DBSCAN(eps=0.1, min_samples=10).fit(data.values)     # 一行代码搞定
```

In [36]:
```
labels=["c1","c2","c3","c4","c5","c6","c7","c8","c9","c10","c11","c12"]
plt.figure()
for i in range(8):
    plt.scatter(data.values[np.array(mm.labels_)==np.array(i),0],data.values[np.array(mm.labels_)==np.array(i),1],label=labels[i])
plt.xlabel("x")
plt.ylabel("y")
```

Out[36]:
```
Text(0, 0.5, 'y')
```

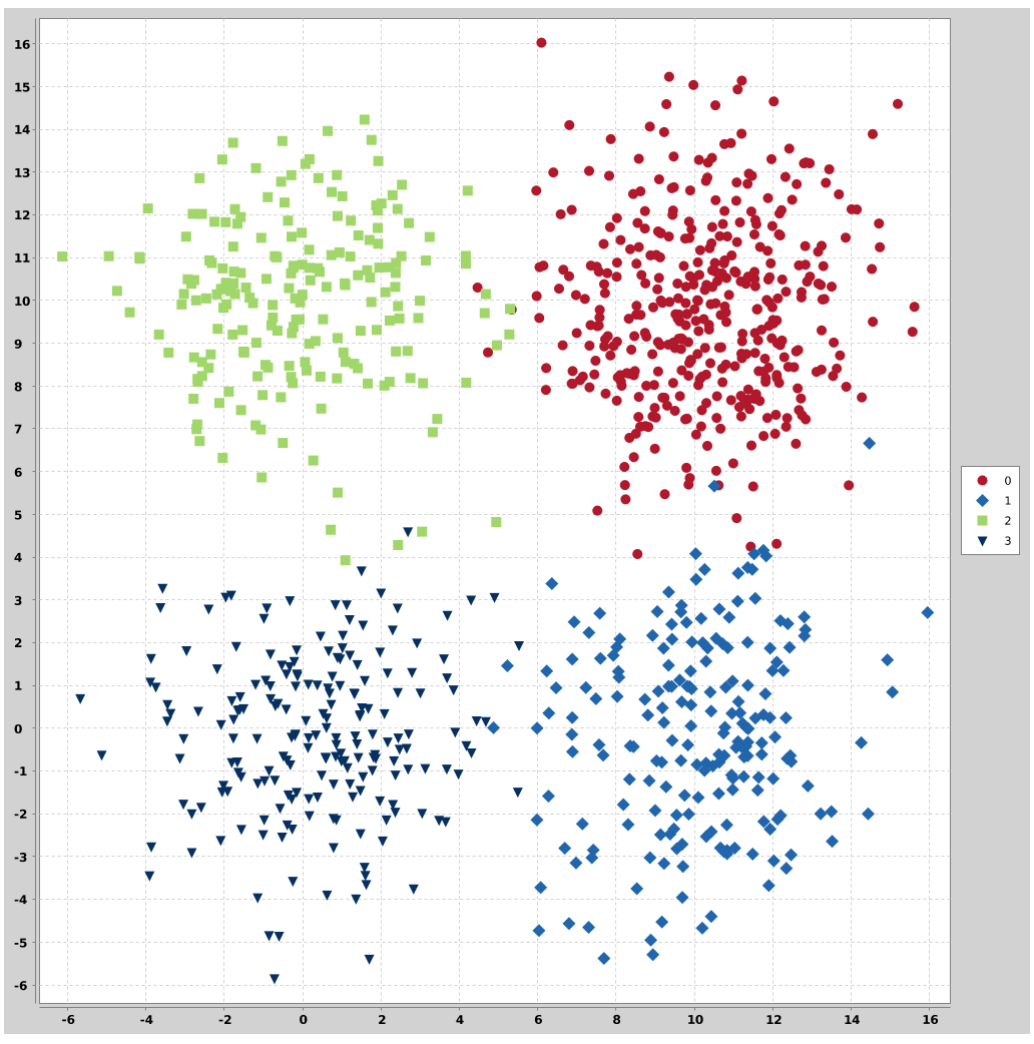


四、爬山算法——Mean Shift

这种算法也有前面层次聚类算法的缺点，就是形成的类都是球状的簇，之所以最后还是介绍这种算法，
是因为这种算法的原理比较有意思，是基于“漩涡”高斯概率分布的聚类，能把每个漩涡状的分布各自分成一个类别

先给个例子，说明我们想要干什么

如下图，有4堆点，很明显就是4个分布，我们就是想让这4个分布各自聚成1个类别
话不多说，直接利用sklearn里面的我们之前学过的两种算法跑一遍，看看什么效果



In [37]:
```
data=pd.read_csv("data/sizes1.arff",header=0).iloc[:,[0,1]]
data.head()
```

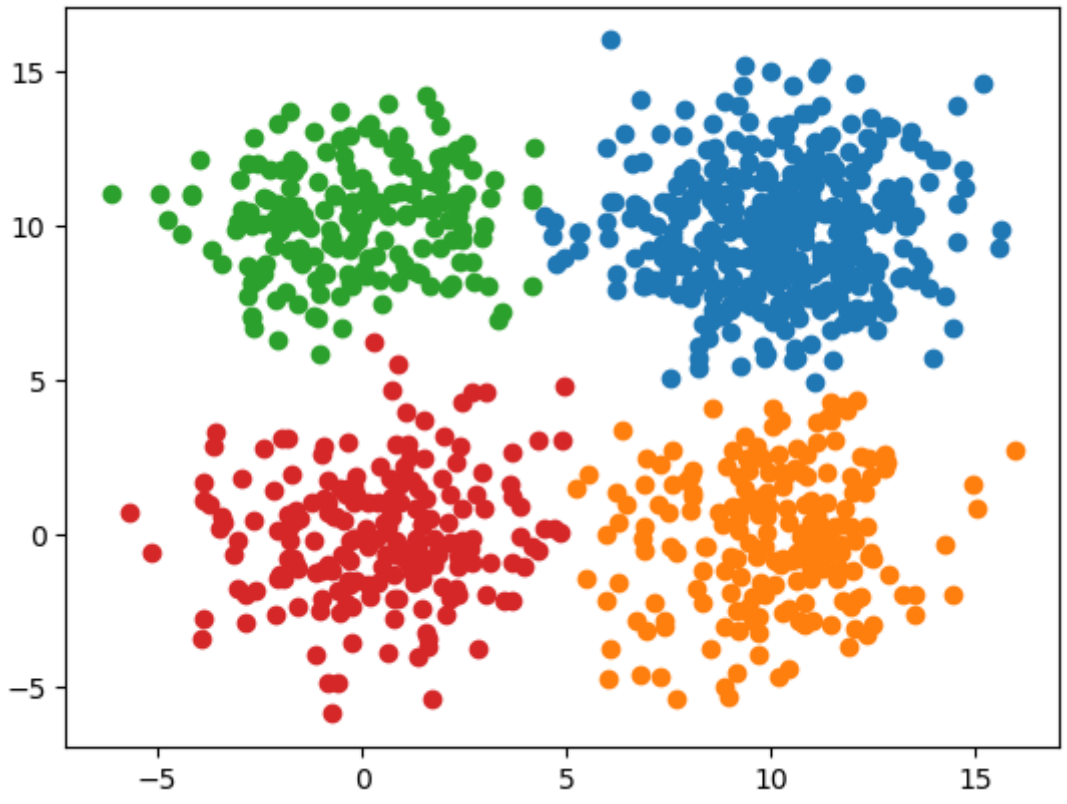
Out[37]:

	x	y
0	9.28531	14.58790
1	12.39770	8.44890
2	8.73624	11.68030
3	9.10197	12.83270
4	12.29480	8.67373

实验一：sklearn中的层次聚类

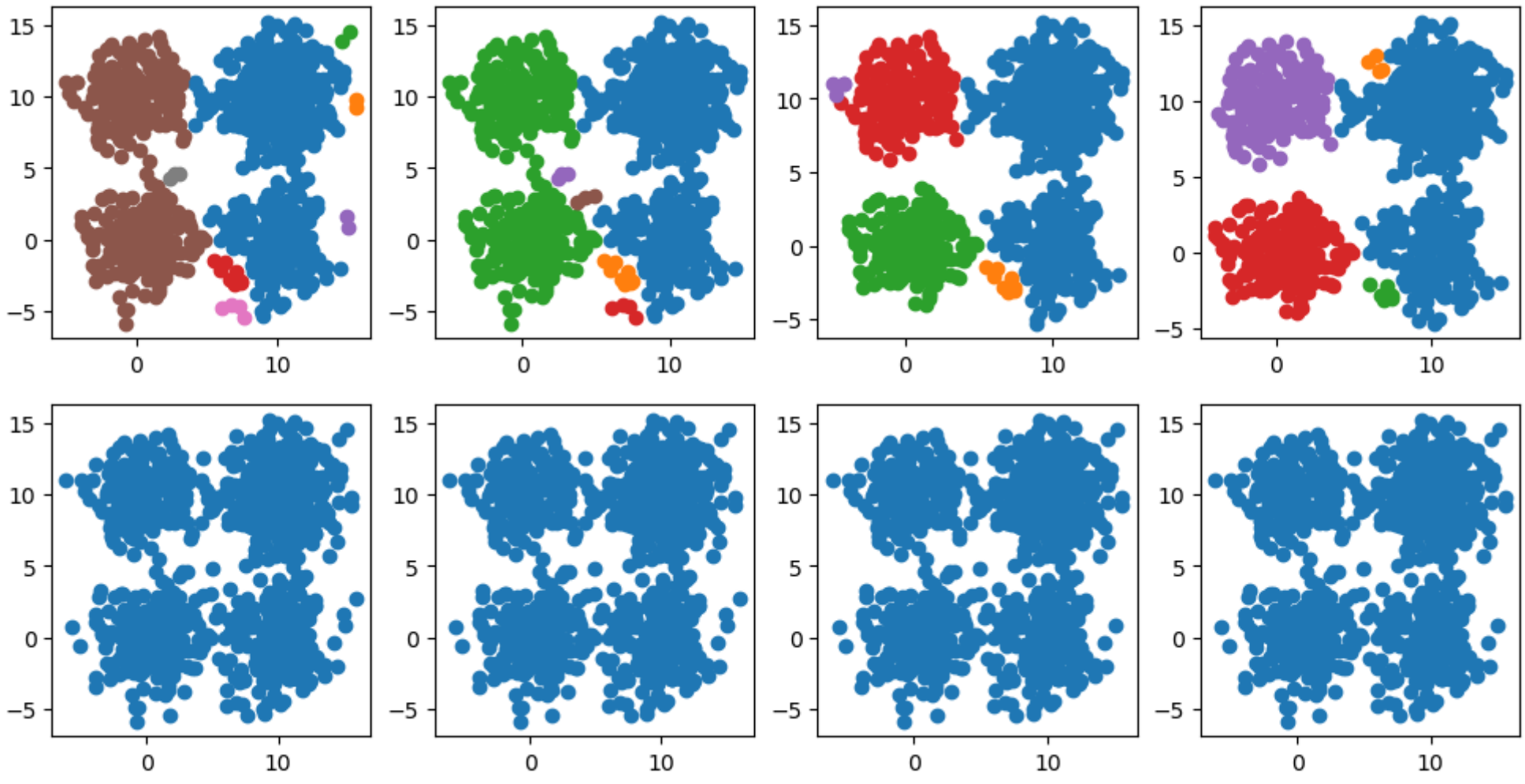
效果好像很不错，一团一团都分出来了

In [38]:
```
plt.figure()
labels=["c1","c2","c3","c4","c5","c6","c7","c8"]
cluster=AgglomerativeClustering(4)
cluster.fit(data.values)
for i in range(4):
    plt.scatter(data.values[cluster.labels_==np.array(i),0],data.values[cluster.labels_==np.array(i),1],label=labels[i])
```



实验二：使用sklearn中的DBSCAN

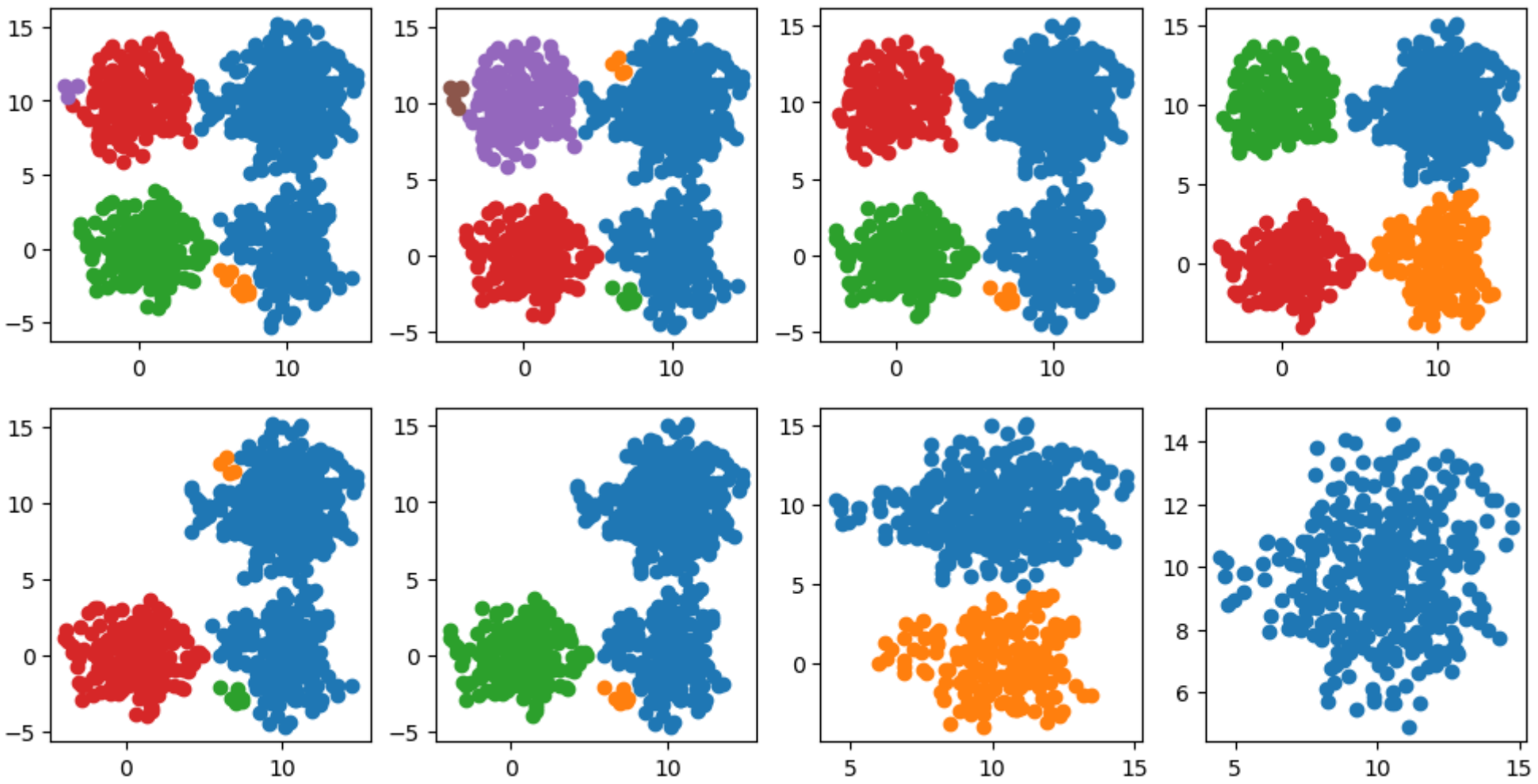
In [39]:
```
fig,axes=plt.subplots(2,4)
fig.set_size_inches(12, 6)
labels=["c1","c2","c3","c4","c5","c6","c7","c8"]
k=8
for i in range(2):
    for j in range(4):  # 定位到某个坐标图
        cluster = DBSCAN(eps=i+1, min_samples=j+2)
        cluster.fit(data.values)
        for m in range(k):
            axes[i][j].scatter(data.values[cluster.labels_==np.array(m),0],data.values[cluster.labels_==np.array(m),1],label=labels[m])
        k=k-1
```



可以看出当i=0,j=2的时候，此时eps=1,min_samples=4,这个时候只有右边两堆点区分性不太好，
如果此时固定eps，增加或减少min_samples，就会导致这张图的左右两张图的情况，
如果固定min_samples，增加eps，令eps=2，就会导致这张图的下方图的情况。
那我们选择哪种方案去改变参数呢，应该是固定eps，增加min_samples，为了把右边上下两堆点分开
就是要增加密度数量，让进入张三朋友圈（上面那堆点）的门槛变高

In [40]:
```
fig,axes=plt.subplots(2,4)
fig.set_size_inches(12, 6)
labels=["c1","c2","c3","c4","c5","c6","c7","c8"]
k=8
for i in range(2):
    for j in range(4):  # 定位到某个坐标图
```

```
        cluster = DBSCAN(eps=1, min_samples=i+j+4)
        cluster.fit(data.values)
        for m in range(k):
            axes[i][j].scatter(data.values[cluster.labels_==np.array(m),0],data.values[cluster.labels_==np.array(m),1],label=labels[m])
        k=k-1
```



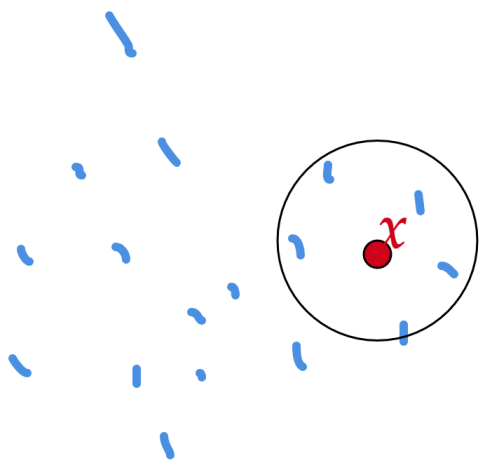
由此可见，当i=0，j=3的时候，此时eps=1，min_samples=7，就把四堆点完全分开了

跑完后，接下来做什么

由于效果都还不错，就不说什么缺点了，单独介绍mean shift基于什么来对点聚类的吧

首先这种算法也是基于数据分布的某种密度来设计的一种聚类算法，关于密度，我们前面给了衡量密度的一种方式，
那就是看张三认识多少人，张三认识的人里面，又认识了多少人，只有一连串人都认识很多人，我们才把这些人归为一类，
因为他们是一连串密度集中区。所以说上面说的密度，就是以x为中心画个圈，看圈里有多少点来度量x附近密度大不大。

然后关于mean shift，它的度量方式也类似，就是以x为中心画个圈，然后看圈里面所有点与x的距离求平均来衡量x的密度。



设x点的密度为$f(x)$，如果直接按照框个圆，

拿圆里面样本点与x的平均距离作为密度：

$f(x)=\frac{1}{N}\Sigma_{i=1}^{N}(x-x_i)^2$（需要归一化处理一下）

然后这个函数太陡了，我们换个形状类似，但是

高矮胖瘦可以调节的函数替换一下：

$$f(x)=\frac{1}{N}\Sigma_{i=1}^{N}\frac{1}{\sqrt{2p\pi\sigma}}exp\left(-\frac{(x-x_i)^2}{2\sigma^2}\right)$$

由于这个密度估计式子只是局部的，也就是N 只是圆圈中的点，

不是所有样本，那么N是否可以换成所有样本个数n呢?

$$f(x)=\frac{1}{n}\Sigma_{i=1}^{n}\frac{1}{\sqrt{2p\pi\sigma}}exp\left(-\frac{(x-x_i)^2}{2\sigma^2}\right)=\frac{1}{n}\left(\Sigma_{i=1}^{N}+\Sigma_{i=N+1}^{n}\right)\frac{1}{\sqrt{2p\pi\sigma}}exp\left(-\frac{(x-x_i)^2}{2\sigma^2}\right)$$

由于$\Sigma_{i=N+1}^{n}\frac{1}{\sqrt{2p\pi\sigma}}exp\left(-\frac{(x-x_i)^2}{2\sigma^2}\right)$是圈外点与$x$作差平方，

如果圈外点离x近，这个数值就大，离x越远，这个数值越小，

所以拿这部分点对度量x的密度是正向作用，可以使用

密度估计小实验(无参数估计)

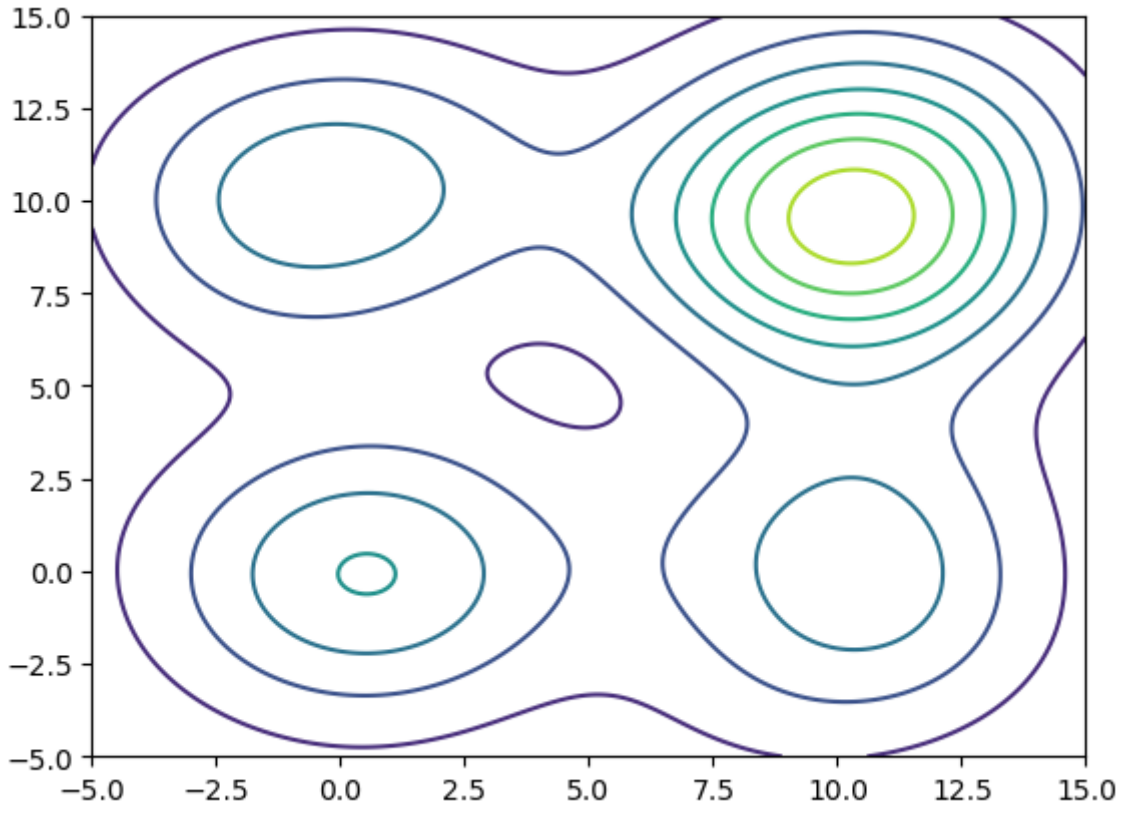
接下来就用上面这个密度估计对这4堆样本在二维平面上各个地方的密度做个估计，并画出等高线图

注：实际上这个估计有专门的名字，叫Parzen Window 密度估计,有兴趣可以看看它真正的证明，就是一种局部近似再近似扩展到全局

In [41]:
```
def parzen_window(x,y,data,sigma):
    t_0 = 1/np.sqrt(2*np.pi)/sigma
    for i in range(data.shape[0]):
        if i==0:
            a=-((x-data[i,0])**2+(y-data[i,1])**2)/2/sigma**2
            t_1=np.exp(a)
        else:
            a=-((x-data[i,0])**2+(y-data[i,1])**2)/2/sigma**2
            t_1=t_1+np.exp(a)
    t_2 = t_0*t_1/data.shape[0]
    return t_2
```

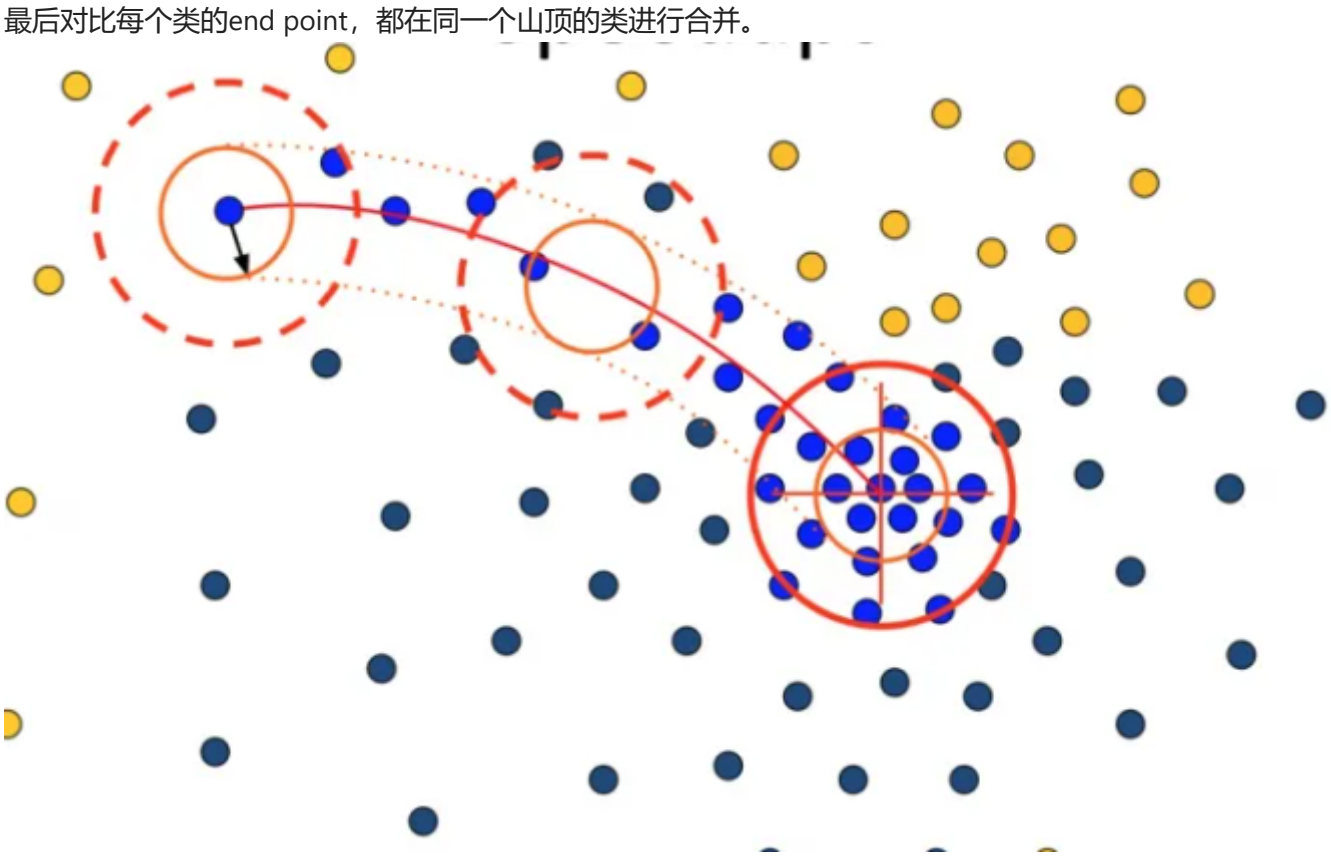
In [42]:
```
plt.figure()
x=np.linspace(-5,15,1000)
y=np.linspace(-5,15,1000)
X,Y=np.meshgrid(x,y)
Z=parzen_window(X,Y,data=data.values,sigma=2)
plt.contour(X,Y,Z)
```

Out[42]:
```
<matplotlib.contour.QuadContourSet at 0x23b5c87d100>
```



利用这个等高线图可以干什么?

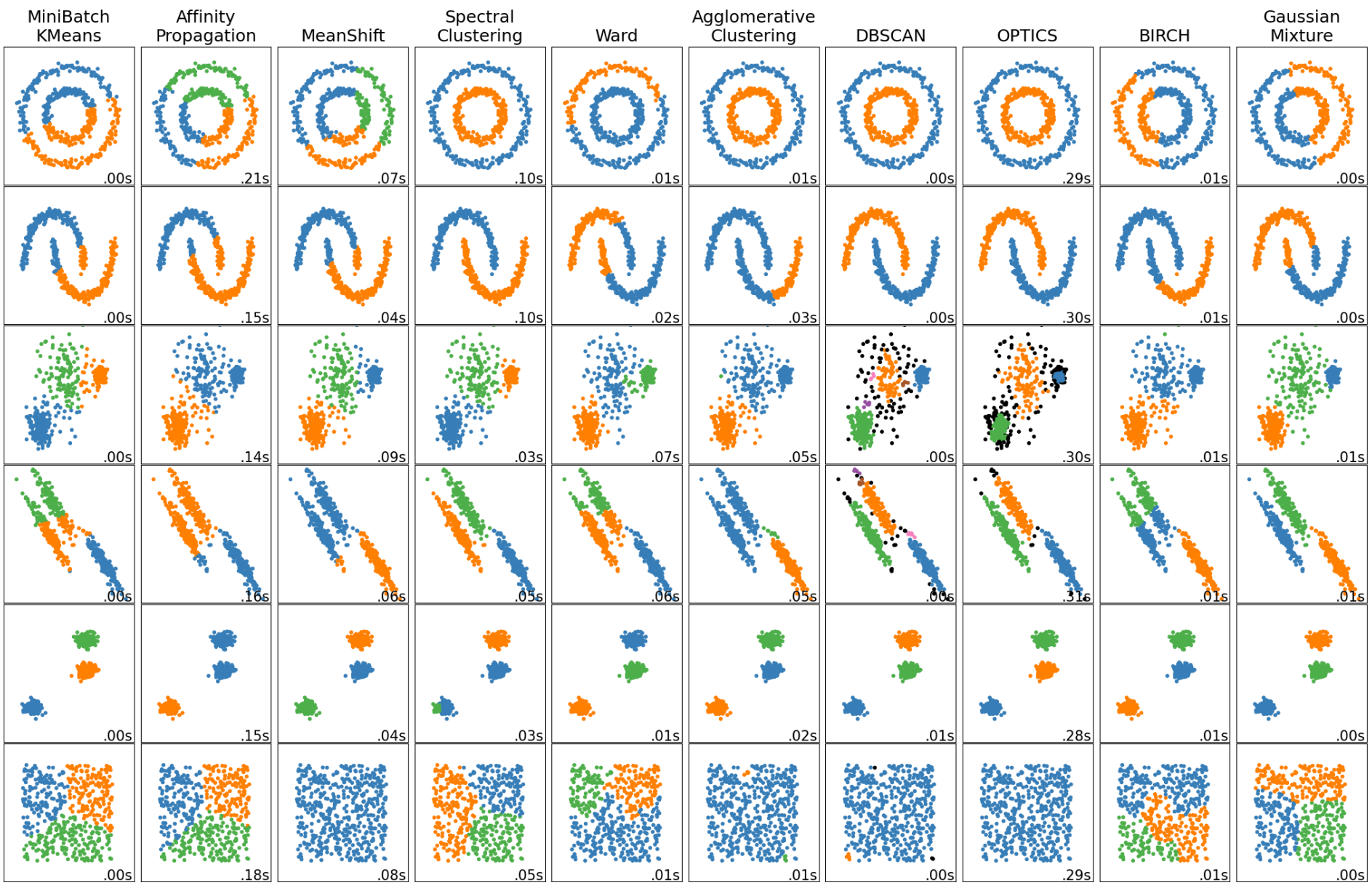
从等高线图不难看出，这4堆点刚好凑成4座高山，那么有没有什么办法把这4个高山上面的树木（就是数据点）分配到这棵树所在的山上呢，这就是mean shift算法，随便从一棵树出发，沿着上山方向走，把这棵树半径r（超参）以内的树都归为一类，当走到end point（山顶）的时候，也就是点最密集的地方，就会停下来，怎么停下来可以参考梯度下降法.
然后这些树已经归好帮派了，后面不再参与，接着从其他树出发，继续上山，就这么把所有树都归好类了。
最后对比每个类的end point，都在同一个山顶的类进行合并。



sklearn中的更多例子

数据集下载

实际上这种例子很多，基于局部密度考虑的算法可以轻易聚类，但是参数比较难调


MiniBatch KMeans
Affinity Propagation
MeanShift
Spectral Clustering
Ward
Agglomerative Clustering
DBSCAN
OPTICS
BIRCH
Gaussian Mixture
.00s .21s .07s .10s .01s .01s .00s .29s .01s .00s
.00s .15s .04s .10s .02s .03s .00s .30s .01s .00s
.00s .14s .09s .03s .07s .05s .00s .30s .01s .01s
.00s .16s .06s .05s .06s .05s .06s .31s .01s .01s
.00s .15s .04s .03s .01s .02s .01s .28s .01s .00s
.00s .18s .08s .05s .01s .01s .00s .29s .01s .00s