南昌大学

NANCHANG UNIVERSITY

期中报告

题　　目：鸢尾花大数据分析

学　　院：软件学院

班　　级：信息安全 193 班

小组成员：黄昊，李璇，丁俊，邹昆

起讫日期：2022.5.4

任课教师：赵志宾

完成时间：2022.5.10

目录

一、 聚类问题描述和数据集特征提取

1.1 聚类问题描述

- **聚类概念**

迄今为止，聚类还没有一个学术界公认的定义。这里给出 Everitt 在 1974 年关于聚类所下的定义：一个类簇内的实体是相似的，不同类簇的实体是不相似的；一个类簇是测试空间中点的会聚，同一类簇的任意两个点间的距离小于不同类簇的任意两个点间的距离；类簇可以描述为一个包含密度相对较高的点集的多维空间中的连通区域，它们借助包含密度相对较低的点集的区域与其他区域(类簇)相分离。

事实上，聚类是一个无监督的分类，它没有任何先验知识可用，聚类的形式描述如下：

令 U=$\{p_1, p_2\ldots, p_n\}$表示一个模式(实体)集合，p_i表示第 i 个模式 i={1,2,…,n}；C_t⊆U，t=1,2,…,k，$C_t=\{p_{t1}, p_{t2}\ldots, p_{tw}\}$；proximity($p_{ms}$，$p_{ir}$)，其中，第 1 个下标表示模式所属的类，第 2 个下标表示某类中某一模式，函数 proximity 用来刻画模式的相似性距离。若诸类C_t为聚类之结果，则诸C_t需满足如下条件：

1) $\bigcup_{t=1}^{k} C_t = U$.

2) 对于$\forall C_m, C_r \subseteq U, C_m \neq C_r$,有 $C_m \cap C_r = \varnothing$(仅限于刚性聚类);

$$\mathrm{MIN}_{\forall p_{mu}\in C_m, \forall p_{rv}\in C_r, \forall C_m, C_r \subseteq U \& C_m \neq C_r} (proximity(p_{mu}, p_{rv})) > \mathrm{MAX}_{\forall p_{mx}, p_{my}\in C_m, \forall C_m \subseteq U} (proximity(p_{mx}, p_{my})).$$

- **数据集选取**

我们选取的 iris 数据集是常用的分类实验数据集，由 Fisher 于 1936 收集整理。iris 也称鸢尾花卉数据集，是一类多重变量分析的数据集，包含 150 个数据样本，分为 3 类，每类 50 个数据，每个数据包含 4 个属性。其中每个类别指的是一种鸢尾植物。一类与另一类线性可分；后者不能彼此线性分离。

我们可通过数据集中花萼长度（sepallength），花萼宽度（sepalwidth），花瓣长度（petallength），花瓣宽度（petalwidth）4 个属性，预测鸢尾花卉属于山鸢尾（Iris-setosa）、变色鸢尾（Iris-versicolor）和维吉尼亚鸢尾（Iris-virginica）三个种类中的哪一类。

注：以上特征单位均为厘米（cm）

Iris Data Set

Download: Data Folder, Data Set Description

Abstract: Famous database; from Fisher, 1936

Data Set Characteristics:	Multivariate	**Number of Instances:**	150	**Area:**	Life
Attribute Characteristics:	Real	**Number of Attributes:**	4	**Date Donated**	1988-07-01
Associated Tasks:	Classification	**Missing Values?**	No	**Number of Web Hits:**	4657282

图 1-1 UCI 数据集-iris 数据集

1.2 数据集特征提取

1. 数据集导入

导入数据文件 iris.csv，并通过.head()函数来查看其前五行数据。

```
import numpy as np
import pandas as pd
import warnings
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='darkgrid') #设置画图风格
warnings.filterwarnings('ignore') #忽略管告
iris =pd.read_csv('iris.csv')
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

2. 数据处理

查看数据是否有缺失，发现数据很完整没有缺失。

```
iris.isnull().sum()
```

```
sepal_length    0
sepal_width     0
petal_length    0
petal_width     0
class           0
dtype: int64
```

接着查看其各个特征的统计信息，观察到四个特征的方差均不接近 0，说明这些特征在数据中分布广，可以用这四个特征来预测数据。

```
iris.describe()
```

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054667	3.758667	1.198667
std	0.828066	0.433588	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

3. 探索性数据分析

- 散点图

导入 seaborn 包中 pairplot 查看不同特征之间的关系以及关系的分布情况。

```
#Seaborn库是基于matplotlib的高阶绘图库，可以简洁而优美的绘制图形
sns.pairplot(iris,hue = 'class')
```

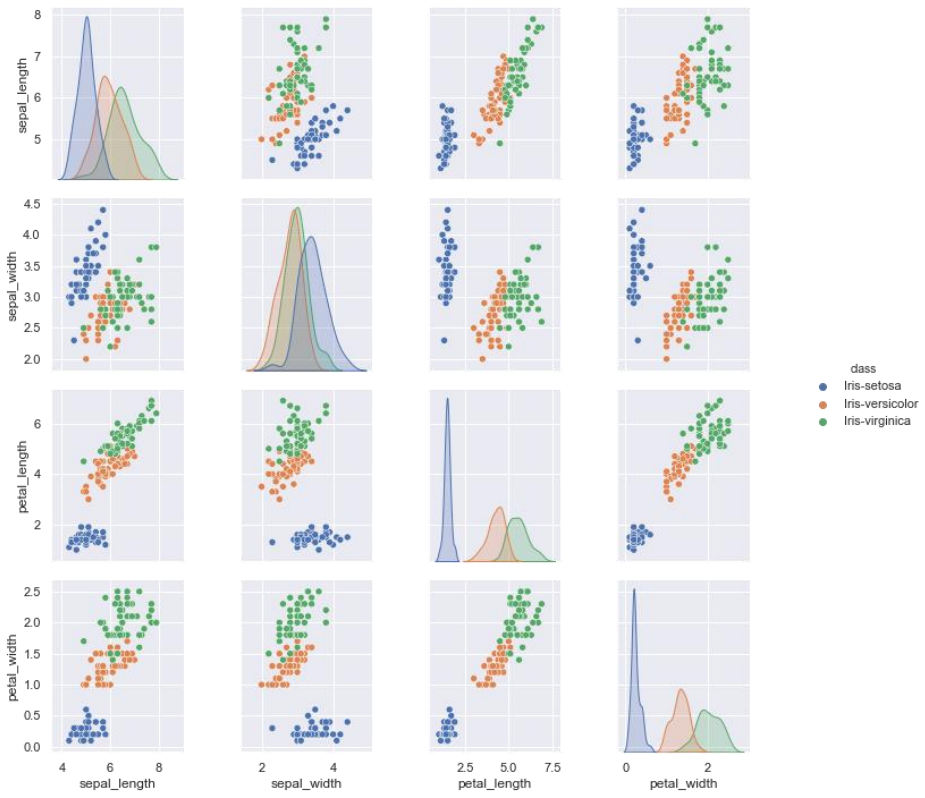


图 1-2 特征关系分布散点图

从上图中可以观察到，petal length 以及 petal width 两特征可以较好的对鸢尾花进行分类。

- 小提琴图

导入 seaborn 包中 violinplot，展示这四个特征对三类鸢尾花的分离效果，得到图 1-3 所示输出。

```
#绘制小提琴图，展示各个特征分类数据分布
fig,ax = plt.subplots(2,2,figsize =(8,8))
sns.set(style='white',palette='muted')
sns.violinplot(x = iris['class'],y=iris['sepal_length'],ax =ax[0,0])
sns.violinplot(x = iris['class'],y=iris['sepal_width'],ax =ax[0,1])
sns.violinplot(x = iris['class'],y=iris['petal_length'],ax =ax[1,0])
sns.violinplot(x = iris['class'],y=iris['petal_width'],ax =ax[1,1])
plt.tight_layout
```

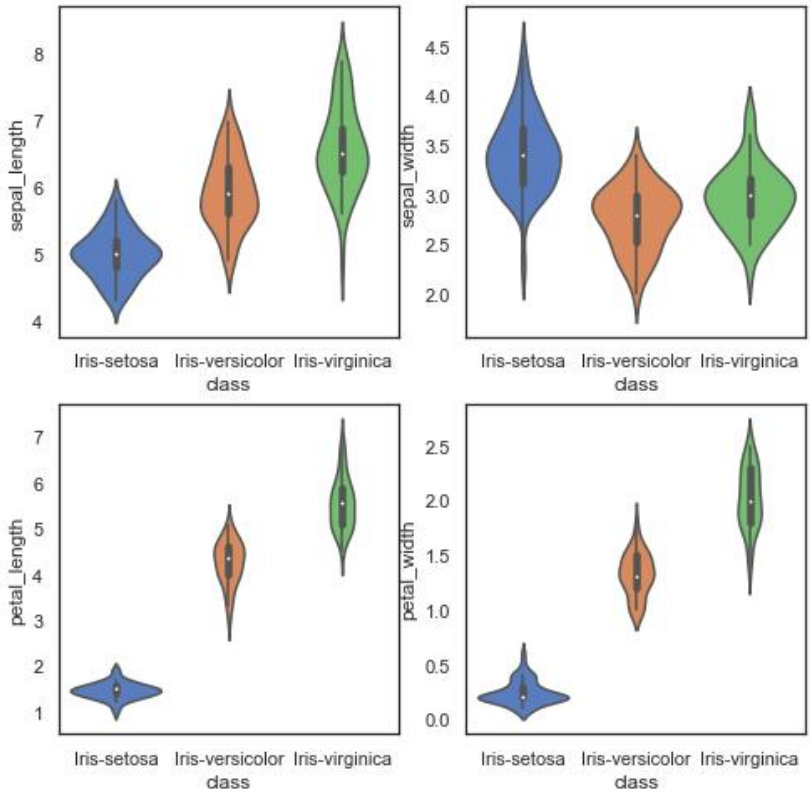


图 1-3 特征分类数据分布图

可以直观看到，花萼长度和花萼宽度在三类中有很大一部分的重叠，而花瓣长度和花瓣宽度这两个特征则能很好的来区分这三类鸢尾花。

- 直方图

绘制直方图查看 sepal width 的分布，发现数据大部分集中在 3.0 到 3.5 之间。

```
plt.style.use('ggplot')
fig,ax = plt.subplots(1,1,figsize=(4,4))
ax.hist(iris['sepal_width'],color = 'black')
ax.set_xlabel('sepal_width')
plt.tight_layout()
```

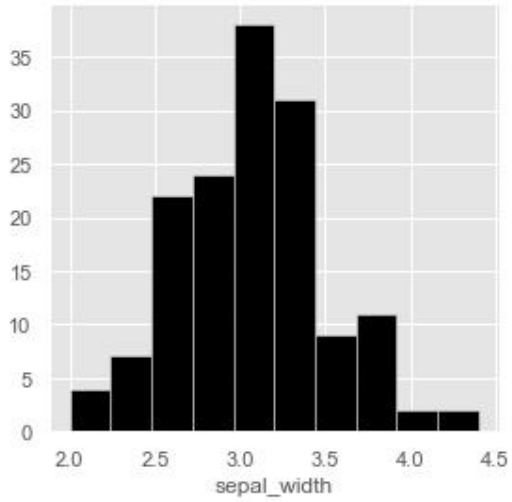


图 1-4 sepal width 分布图

- 热力图

通过作出热力图来观察这四个特征之间的相关性。

```
fig=plt.gcf()
fig.set_size_inches(12,8)
fig=sns.heatmap(iris.corr(),annot=True,cmap='GnBu',linewidths=1,linecolor='k',square=True,\
                mask=False,vmin=-1,vmax=1,cbar_kws={"orientation":"vertical"},cbar=True)
```

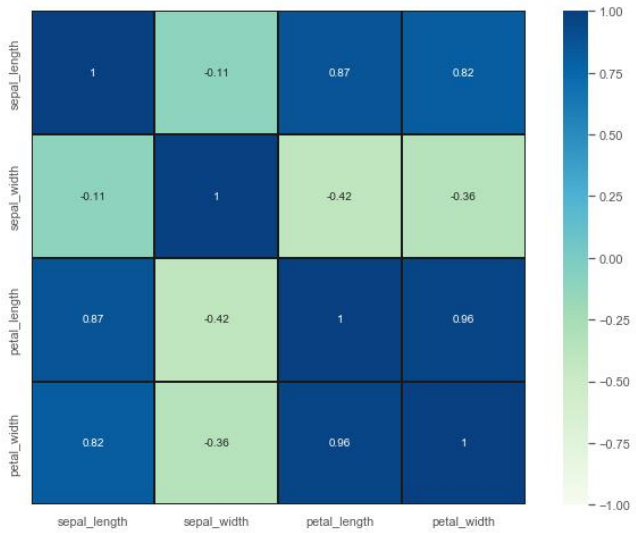


图 1-5 特征热力图

观察上图，发现萼片宽度和长度不相关，而花瓣宽度和长度高度具有一定相关性。

二、 Kmeans 算法

2.1 算法简介

- **算法原理**

K 均值聚类（Kmeans）算法，该算法的主要作用是将相似的样本自动归到一个类别中，其中 K 指的是它可以发现 K 个簇，Means 指的是簇中心采用簇所含的值的均值来计算。该算法也被称为 k-平均或 k-均值算法。

Kmeans 算法主要思想是：在给定 K 值和 K 个初始类簇中心点的情况下，把每个点(亦即数据记录)分到离其最近的类簇中心点所代表的类簇中，所有点分配完毕之后，根据一个类簇内的所有点重新计算该类簇的中心点(取平均值)，然后再迭代的进行分配点和更新类簇中心点的步骤，直至类簇中心点的变化很小，或者达到指定的迭代次数。

- **算法步骤**

1. 选择一个适当的 K；
2. 初始化质心，从数据样本随机选取 K 个数据样本作为聚类中心；
3. 计算每个样本到 K 个聚类中心的距离；
4. 将每个样本分配到离该样本距离最近的这个聚类中心所在的簇；
5. 迭代以下步骤：

（1）重新计算聚类中心：将每个簇中的所有样本的特征值求平均值，得到的数据样本作为新的聚类中心，最后会得到 K 个新的聚类中心；

（2）重新计算每个样本到上一步得到的 K 个聚类中心的距离，按照最小距离，将样本分配到离该样本距离最近的这个聚类中心所在的簇；

（3）如果样本分类不再变化，那么迭代结束。

- **伪代码**

```
创建 k 个点作为起始质心（随机选择）：
    当任意一个点的簇分配结果发生改变的时候：
        对数据集中的每个数据点：
            对每个质心：
                计算质心与数据点之间的距离
            将数据点分配到距其最近的簇
```

对每一个簇： 求出均值并将其更新为质心

Kmeans 算法流程图如下所示：

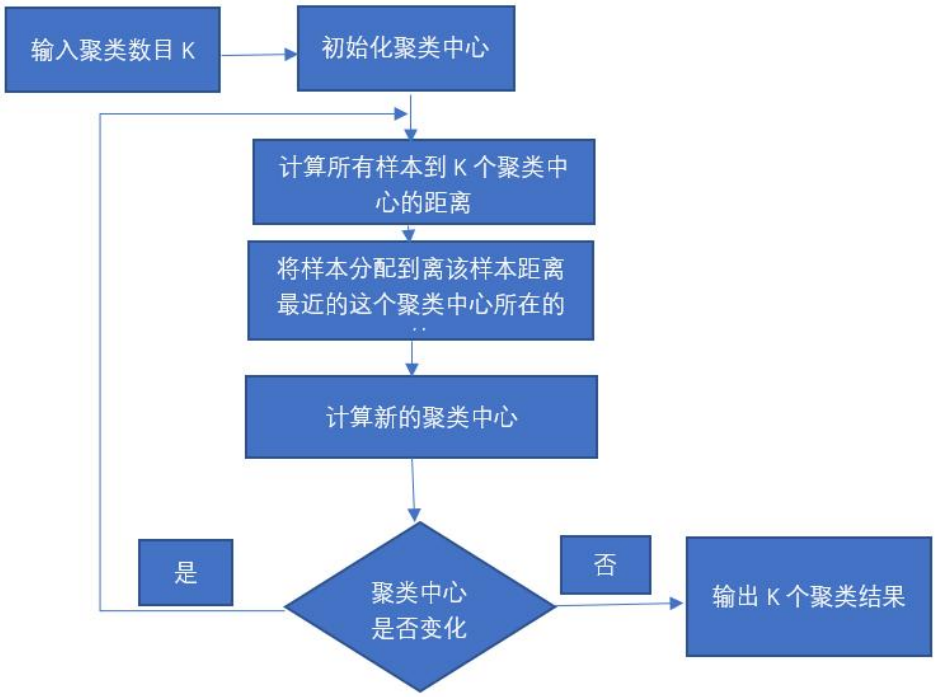


图 2-1 Kmeans 算法流程图

- **算法关键步骤**

由上述算法原理及步骤可知，实现 Kmeans 算法需要解决以下几个步骤。

1. K 值的选取

Kmeans 算法需要手动输入 K 来确定聚类算法最后生成的 k 个不同的簇，这里由数据特征值分析可知，这里直接手动令 K=3。

2. 距离函数的定义

K-means 算法对初始聚类中心较敏感，相似度计算方式会影响聚类的划分。常见的相似度计算方法有欧氏距离、曼哈顿距离、闵可夫斯基距离三种，本次实验使用欧氏距离。

（1）欧式距离

欧式距离就是指在 m 维空间中两个点之间的真实距离，或者向量的自然长度（即该点到原点的距离）。在二维和三维空间中的欧氏距离就是两点之间的实际距离。N 维向量的欧式距离公式如下：

$$d(x,y) := \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}$$

（2）曼哈顿距离

城市街区距离(City Block distance)，从一个十字路口到另外一个十字路口的街区距离，即两个点在标准坐标系上的绝对轴距总和。计算公式为：

$$d_{12} = \sum_{k=1}^{n} |x_{1k} - x_{2k}|$$

（3）闵可夫斯基距离

闵氏距离不是一种距离，而是一组距离的定义，两个 n 维变量 a(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的闵可夫斯基距离定义如下：

$$d_{12} = \sqrt[p]{\sum_{k=1}^{n} |x_{1k} - x_{2k}|^p}$$

3. 初始聚类中心点的确定

K-means 算法对初始聚类中心较敏感，中心点的选择会影响 KMeans 的聚类效果。 于是本实验使用 K-means++ 的初始点选取的规则，选取距离尽量远的 K 个样本点作为中心点：随机选取第一个样本 C1 作为第一个中心点，遍历所有样本选取离 C1 最远的样本 C2 为第二个中心点，以此类推，选出 K 个初始中心点具体步骤如下：

（1）随机选取一个点作为第一个聚类中心。

（2）计算所有样本与第一个聚类中心的距离。

（3）选择出上一步中距离最大的点作为第二个聚类中心。

（4）迭代：计算所有点到与之最近的聚类中心的距离，选取最大距离的点作为新的聚类中心。

（5）终止条件：直到选出了这 k 个中心。

2.2 算法实现

4. 初始化数据

使用 pandas 库的 read_csv() 函数对数据文件进行读取，定义相关的列索引特征属性，并在最后一列添加一列记录每个样本所属的簇类。

```
path = r'D:\College\2022-2\数据仓库\期中\dataset\iris.data'
names=['sepal_length','sepal_width','petal_length','petal_width','class']
```

```
# 读取原始数据
data =  pd.read_csv(path,sep=',',names=names,header=None)
# 处理数据
df_data =data.loc[:,names[0:4]]
df_data['cluster'] = np.zeros(len(df_data),int)
df_data
```

	sepal_length	sepal_width	petal_length	petal_width	cluster
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	0
146	6.3	2.5	5.0	1.9	0
147	6.5	3.0	5.2	2.0	0
148	6.2	3.4	5.4	2.3	0
149	5.9	3.0	5.1	1.8	0

150 rows × 5 columns

5. 定义距离函数

注：本实验使用的距离定义是欧式距离。

```
# 计算样本的距离
def distEclud(A, B):
    '''
    定义距离函数
    :param A:  numpy 数组A
    :param B:  numpy 数组B
    :return: 返回 数组A和数组B的欧式距离
    '''
    return np.sqrt(np.sum(np.power(A - B, 2)))
```

6. 初始化 K 个质心

初始化质心过程中，关键点在于，第一个质心的随机生成，然后遍历所有的样本点，计算所有样本点到这个质心的距离，并将他们记录下来，然后再里面找到距离该质心最远的那个点，并将该点作为 第二个执行，以此类推，再去找第三个质心。

```python
def randCentKmeanspp(dataSet, k):
    '''
    构建一个包含k个随机质心的集合（k行n列，n表示数据的维度/特征个数），
    :param dataSet: 输入数据集
    :param k: 质心个数
    :return: K 个初始质心
    '''
    # 得到数据样本的维度,除最后面的cluster
    n = np.shape(dataSet)[1] - 1
    # 初始化为一个(k,n)的全零 numpy 数组
    centroids = pd.DataFrame(np.zeros((k, n),dtype=np.float64),columns=names[:4])
    # 随机选取样本中的一个点
    centroids.iloc[0,:] = dataSet.iloc[np.random.randint(dataSet.shape[0]), :4]
    # 迭代 生成 聚类中心点
    for k_id in range(k - 1):
        dist = []
        # 遍历所有点
        for i in range(dataSet.shape[0]):
            point = dataSet.iloc[i, :4]
            d = float('inf')
            # 扫描所有质心，选出该样本点与最近的类中心的距离
            for j in range(centroids.shape[0]):
                temp_dist = distEclud(np.array(point), np.array(centroids.iloc[j, :]))
                d = min(d, temp_dist)
            dist.append(d)
        dist = np.array(dist)
        # 返回dist里面最大值的下标，对应的是上面循环中的i
        next_centroid = dataSet.iloc[np.argmax(dist), :4]
        # 选出了下一次的聚类中心，开始第k+1循环
        centroids.iloc[k_id+1, :] = next_centroid
        dist = []
    return centroids
```

7. Kmeans 算法主体部分

主体部分首先便是先初始化 K 个质心，然后需要定义个变量来记录 样本中心是否发生变化，如果发生变化，那么就计算每个样本到 K 个聚类中心的距离，并将每个样本分配到离该样本距离最近的这个聚类中心所在的簇，同时将每个簇中的所有样本的特征值求平均值，得到的数据样本作为新的聚类中心。如果不再变化，那么聚类结束，返回每个聚类结果。

```python
def kMeans(df_data, k, distMeas, createCent):
    '''
    :param dataSet: 输入的数据集
    :param k: 聚类的个数，可调
    :param distMeas: 计算距离的方法，可调
    :param createCent: 初始化质心的位置的方法，可调
    '''
    dataSet = df_data.copy(deep=True)
    # 获取数据集样本数
    m = np.shape(dataSet)[0]
    # 创建初始的k个质心向量
    centroids = createCent(dataSet, k)
    # 聚类结果是否发生变化的布尔类型
    clusterChanged = True
    # 终止条件：所有数据点聚类结果不发生变化
    t = 0
```

```
    while clusterChanged:
        # 聚类结果变化布尔类型置为False
        t+=1
        print("第{}次迭代".format(t))
        clusterChanged = False
        # 遍历数据集每一个样本向量
        for i in range(m):
            # 初始化最小距离为正无穷，最小距离对应的索引为-1
            minDist = float('inf')
            minIndex = -1
            # 循环k个类的质心
            for j in range(k):
                # 计算数据点到质心的欧氏距离
                distJI = distMeas(np.array(dataSet.iloc[i,:4]),np.array(centroids.iloc[j,:4]))
                # 如果距离小于当前最小距离
                if distJI < minDist:
                    # 当前距离为最小距离，最小距离对应索引应为j(第j个类)
                    minDist = distJI
                    minIndex = j
            # 当前聚类结果中第i个样本的聚类结果发生变化：布尔值置为True，继续聚类算法
            if dataSet.cluster[i] != minIndex:
                clusterChanged = True
                dataSet.loc[i,'cluster'] = minIndex
        # 打印k-means聚类的质心
        # 遍历每一个质心
        for cent in range(k):
            # 将数据集中所有属于当前质心类的样本通过条件过滤筛选出来
            ptsInClust = dataSet[dataSet.cluster==cent]
            # 计算这些数据的均值(axis=0:求列均值)，作为该类质心向量
            centroids.iloc[cent,0:4] = np.array(ptsInClust.iloc[:,0:4].mean(axis=0))
        print(centroids)
    # 返回k个聚类中心，聚类结果
    return centroids,dataSet
```

2.3 结果与展示

- **散点图**

由于数据有 4 维，这里就分别对'sepal_length'，'sepal_width'以及'petal_length'，'petal_width'进行二维展示；

```
import matplotlib.pyplot as plt
color = ["red","green","black"]
plt.xlabel(names[0])
plt.ylabel(names[1])
for cent in range(3):
    ps = ds[ds.cluster==cent]
    plt.scatter(np.array(ps[names[0]]),np.array(ps[names[1]]),c = color[cent])
plt.show()
```

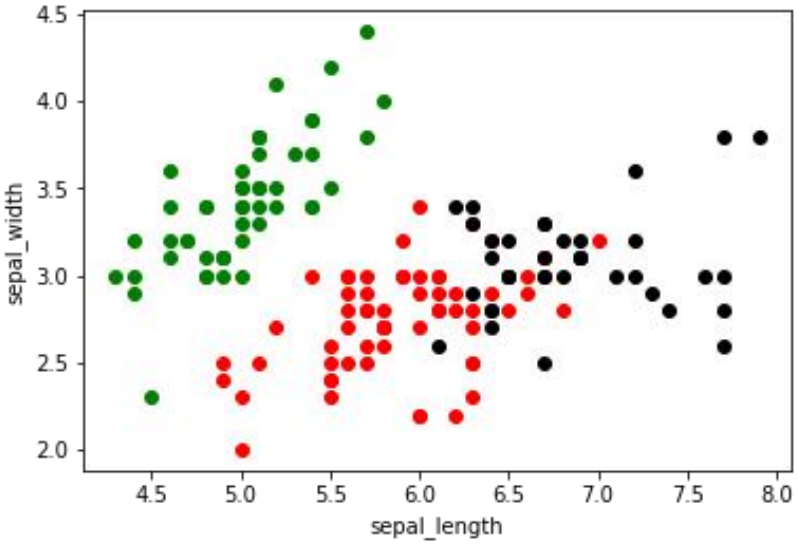


图 2-2 'sepal_length'-'sepal_width'散点图

```
plt.xlabel(names[2])
plt.ylabel(names[3])
for cent in range(3):
    ps = ds[ds.cluster==cent]
    plt.scatter(np.array(ps[names[2]]),np.array(ps[names[3]]),c = color[cent])
plt.show()
```

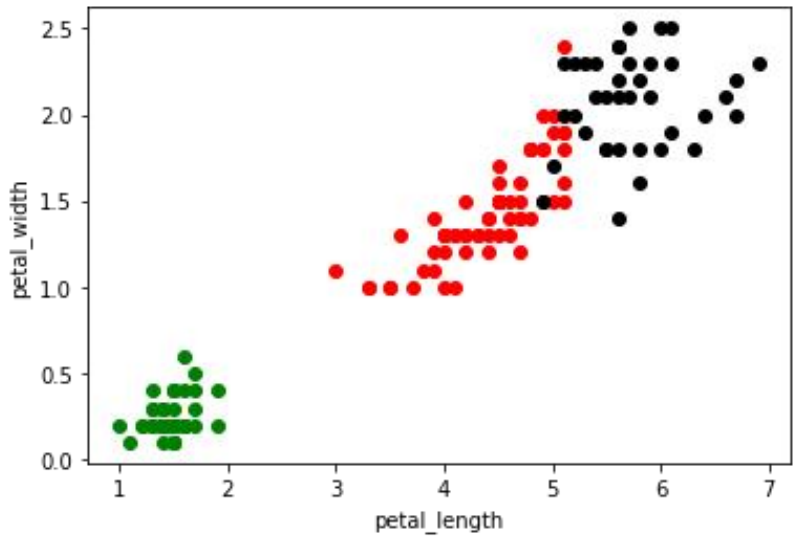


图 2-3 'petal_length'-'petal_width'散点图

在此基础上，使用平行坐标图以及矩阵图对四维数据进行多维展示如下。

- **平行坐标图**

```
from pandas.plotting import parallel_coordinates
parallel_coordinates(ds,'cluster',color=['blue', 'green', 'red'])
plt.show()
```

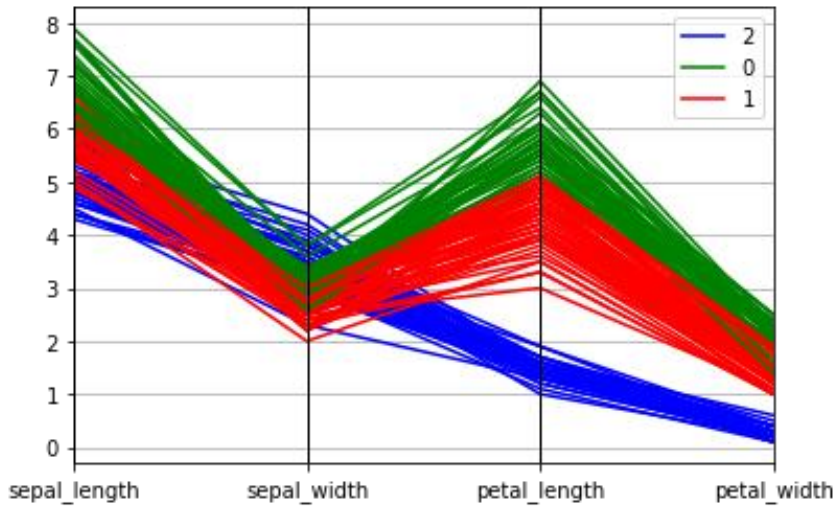


图 2-4 平行坐标图

平行坐标图每个特征变量都用一个垂直坐标的竖线表示，变量值对应 y 轴上位置，然后将每个变量所对应的值通过折线连接起来，每条折线就是一个样本数据。

- **矩阵图**

```
import seaborn as sns
sns.pairplot(ds,hue = 'cluster')
plt.show()
```

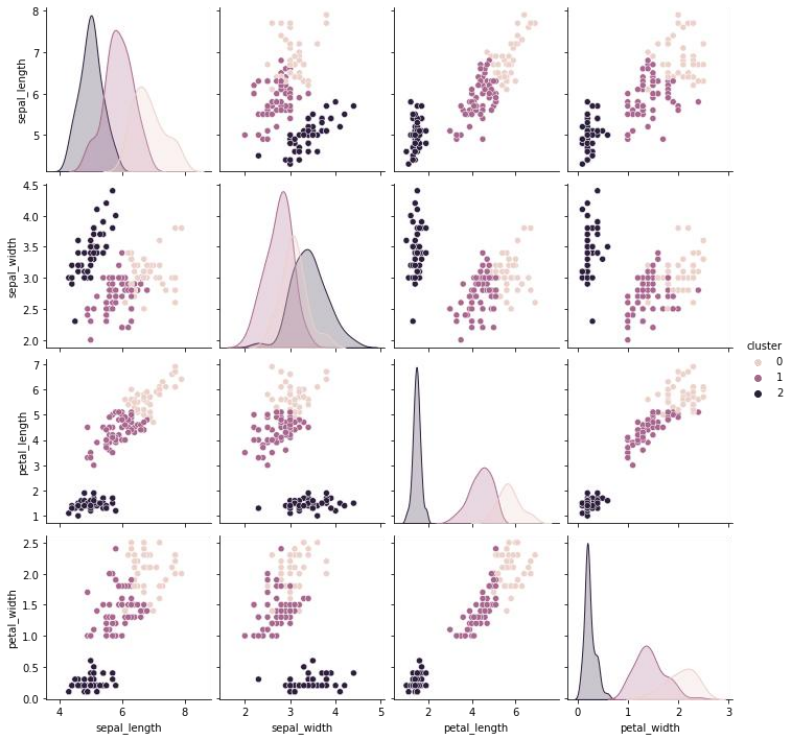


图 2-5 矩阵图

矩阵图将变量两两之间的关系都以散点图以及曲线图展示出来，探索多个变量之间的关系。

三、 DBSCAN 算法

前文中我们使用了 K-Means 算法对数据集进行聚类分析，它是基于距离的聚类算法，即根据距离不断迭代更新最小距离和直到最后每个簇的中心点不再发生改变为止。但是当数据集的聚类结果是非球状结构时，基于距离的聚类算法的聚类效果并不好。此时基于密度的聚类算法应运而生，它可以发现任意形状的聚类。在基于密度的聚类算法中，通过在数据集中寻找被低密度区域分离的高密度区域，将分离出的高密度区域作为一个独立的类别。

图 3-1 非球状聚类效果图

3.1 算法简介

- **算法原理**

DBSCAN（Density-Based Spatial Clustering of Applications with Noise，具有噪声的基于密度的聚类方法）是一种基于密度的空间聚类算法。与划分和层次聚类方法不同，它将簇定义为密度相连的点的最大集合，能够把具有足够高密度的区域划分为簇，并可在噪声的空间数据库中发现任意形状的聚类。它将簇定义为密度相连的点的最大集合。

DBSCAN 算法相关概念如下。

（1）核心对象：若某个点的密度达到算法设定的阈值则其为核心点，即 r 领域内点的数量≫minPts(minPts 是我们人为设定的值)。

（2） ε -领域的距离阈值：设定的半径 r

（3）直接密度可达：若某点 p 在点 q 的 r 领域内，且 q 是核心点则 p-q 直接密度可达。

（4）密度可达：若有一个点的序列q_0、q_1、$....q_k$，对任意$q_i - q_{i-1}$是直接密度

可达的，则称q_0到q_k密度可达，这实际上是直接密度可达的“传播”。

（5）密度相连：若从某核心点 p 出发，点 q 和点 k 都是密度可达的，则点 q 和点 k 是密度相连的。

（6）边界点：属于某一个类的非核心点，不能继续发展新的下线(点)。

（7）噪声点：也称为离群点，不属于任何一个簇的点，从任何一个核心点出发都是密度不可达的。

在这里有两个量需要我们人为设定，一个是半径 r/eps，另一个是指定的数目 MinPts。

我们可以将 DSCAN 算法理解为“不断拉拢成员”的过程，一个点不断去寻找成员，而这些新的成员也不断地去寻找下一层新的成员，直到某一个不再是核心点为止。

例如，从下图很容易理解上述定义，图中 MinPts=5，红色的点都是核心对象，因为其 ε -领域至少有 5 个样本。黑色样本是非核心对象。所有核心对象密度直达的样本在以红色核心对象为中心的超球体内，如果不在超球体内，则不能密度直达。图中红色箭头连起来的核心对象组成了密度可达的样本序列。在这些密度可达的样本序列的 ε -领域内所有的样本相互都是密度相连的。

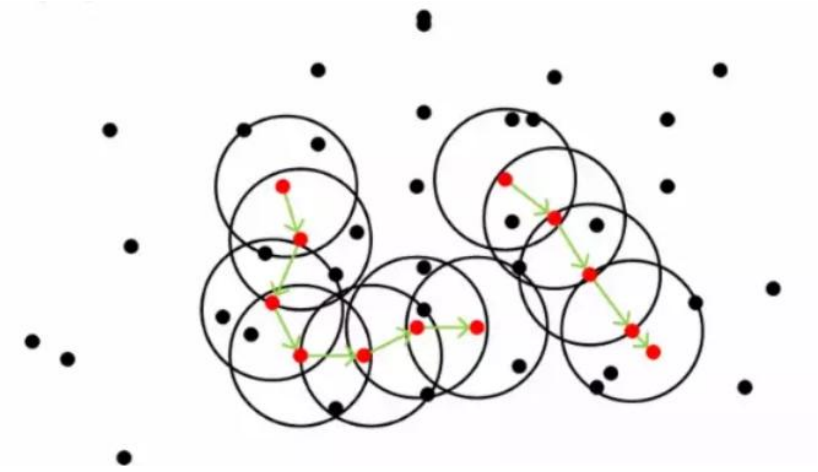


图 3-2 “拉拢成员”示意图

- **算法流程**

1. 首先选择任意一个点，然后这个点半径为 eps 的圆圈内所有的点。如果这些点的个数小于 min_pts，那么这个点被标记为噪声。如果距离在 eps 之内的数据点个数大于 min_pts，则这个点被标记为核心样本，并被分配一个新的簇标签。
2. 然后访问以该点为中心半径 eps 圆范围内的点。如果它们还没有被分配一个

簇，那么就将刚刚创建的新的簇标签分配给它们。如果它们是核心样本，那么就依次访问其邻居，以此类推，簇逐渐增大，直到在簇的 eps 距离内没有更多的核心样本为止。

3. 选取另一个还未被访问过的点，并重复相同的过程。

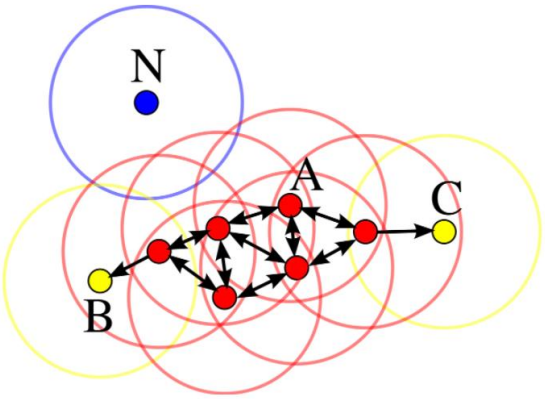


图 3-2 DBSCAN 算法流程示意图

- **伪代码**

```
标记所有对象为 unvisted;
Do
随机选择一个 unvisted 对象 p;
标记 p 为 visted;

If p 的 ε-领域至少有 Minpts 个对象
    创建一个新簇 C,并把 p 添加到 C;
    令 N 为 p 的 ε-领域中的对象集合
    For N 中的每一点
        If p 是 unvisted;
            标记 p 为 visted;
            If p 的 ε-领域至少有 MinPts 个对象，把这些对象添加到 N;
            如果 p 还不是任何簇的成员，把 p 添加到 C;
    End For
Else 标记 p 为噪声;
Until
```

3.2 相关函数实现

1. getDistanceMatrix(datas)

初始化数据点两两之间的距离。N，D 表示数据列表 datas 的维度，N 表示数据条数，D 表示每条数据的维度，再利用 numpy 的 sqrt 和 dot 函数求任意两点的欧氏距离，因为我们这里的数据有 4 维，设 4 维是 x,y,z,w,那么对应的距离计算公式为：

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 + (w_1 - w_2)^2}$$

```
# 计算初始化数据点两两之间的距离
def getDistanceMatrix(datas):
    N,D = np.shape(datas)
    dist = np.zeros([N,N]) # np数组

    for i in range(N):
        for j in range(N):
            vi = datas[i, :]
            vj = datas[j, :]
            dist[i, j] = np.sqrt(np.dot((vi - vj), (vi - vj)))
    return dist
```

2. find_points_in_eps(point_id, eps, dists)

寻找以点 point_id 为中心，eps 为半径的圆内的所有点的 id。

```
# 寻找以点cluster_id为中心，eps为半径的圆内的所有点的id
def find_points_in_eps(point_id, eps, dists):
    ind = (dists[point_id] <= eps)
    return np.where(ind == True)[0].tolist()
```

3. dbscan(datas, eps, min_points)

DBSCAN 算法实现主函数，seeds 是一个列表保存的是当前点为中心，eps 为半径的圆内的所有点的 id，当这些点的个数大于或等于 min_points 时，那么这个点为核心对象，我们开一个新的簇把这个点加入簇中，然后调用 expand_cluster 函数进行扩展，如果 seeds 中的点有核心对象，那我们再利用 find_points_in_eps 函数把这个核心对象的领域内的点找出来并加入到 seeds 中，如果这些点没有被处理过，就属于这个簇了。直到扩展过程中 seeds 列表遍历完了，已经不存在核心对象了，那么最终就找完了一个簇。dbscan 函数继续找下一个未处理过的点。

注：这里用 labels 数组表示一个点所在簇的编号，UNCLASSIFIED 初始化为 0，表示未分类；NOISE 初始化为-1，表示为噪声点。变量 cluster_id 从 0 开始，但簇的编号从 1 开始，即新开一个簇编号加 1，最终 cluster_id 表示簇的总

个数。

```
def dbscan(datas, eps, min_points):
    # 初始化所有点的距离
    dist = getDistanceMatrix(datas)

    # 把所有点的标签初始化为UNCLASSIFIED
    n_points = datas.shape[0] # datas.shape 第一维表示的是数据的个数，第二维是每组数据的维度
    lables = [UNCLASSIFIED] * n_points

    cluster_id = 0
    # 遍历所有点
    for point_id in range(0, n_points):
        # 如果当前点已经处理过了
        if not(lables[point_id] == UNCLASSIFIED):
            continue
        # 没有处理过则计算临近点
        seeds = find_points_in_eps(point_id, eps, dist)

        # 如果临近点数量过少则标记为 NOISE
        if len(seeds) < min_points:
            lables[point_id] = NOISE
        else:
            # 否则就开启一轮簇的扩张
            cluster_id = cluster_id + 1
            # 标记当前点
            lables[point_id] = cluster_id
            # 以当前点继续扩展
            expand_cluster(dist, lables, cluster_id, seeds, eps, min_points)
    return lables, cluster_id
```

4. expand_cluster(dist, lab, cluster_id, seeds, eps, min_points)

参数说明：

#dist: 所有数据两两之间的距离 N * N

labs: 所有数据的标签 labels, N

cluster_id: 一个簇的编号

eps: 密度评估半径

seeds: 用来进行簇扩展的点集

min_points: 半径内最小的点数，初始参数

继续扩充 seeds 列表，加入到当前簇中。

```
def expand_cluster(dist, labs, cluster_id, seeds, eps, min_points):

    i = 0
    while i < len(seeds):
        # 获取一个临近点
        Pn = seeds[i]
        # 如果该点被标记为NOISE,则重新标记
        if labs[Pn] == NOISE:
            labs[Pn] = cluster_id
        # 如果该店没有被标记过
        elif labs[Pn] == UNCLASSIFIED:
            # 标记，计算它的临近点 new_seeds
            labs[Pn] = cluster_id
            new_seeds = find_points_in_eps(Pn, eps, dist)

            # 如果 new_seeds 足够长则把它加入到seed队列中
            if len(new_seeds) >= min_points:
                seeds = seeds + new_seeds
        i = i + 1
```

遍历 seeds 列表中点，如果该点被标记为 NOISE，要重新标记为当前簇的编号，如果还没有被标记过，继续找出这个点半径为 eps 领域内的点，如果个数不小于 min_points，说明这个点是核心对象，那么要把这个核心对象领域内的点加入 seeds 中，这样不断扩展 seeds 簇中的点集，直到找不到核心对象为止。

5. draw_cluster(datas, labs, n_cluster)

聚类绘图函数，参数为数据集 datas，簇的标签 labs，簇的个数 n_cluster，用于展示聚类的结果，这里选取的是两个维度进行散点图显示。

```
# 聚类绘图函数
def draw_cluster(datas, labs, n_cluster):
    plt.cla()
    # 随机生成N种颜色
    colors = [plt.cm.Spectral(k) for k in np.linspace(0, 1, n_cluster)]

    for i, lab in enumerate(labs):
        if lab == NOISE:
            plt.scatter(datas[i,2], datas[i,3], s=16, color = (0,0,0))
        else:
            plt.scatter(datas[i,2], datas[i,3], s=16, color = colors[lab-1])
    plt.show()
```

下面是上述函数对应的流程图：

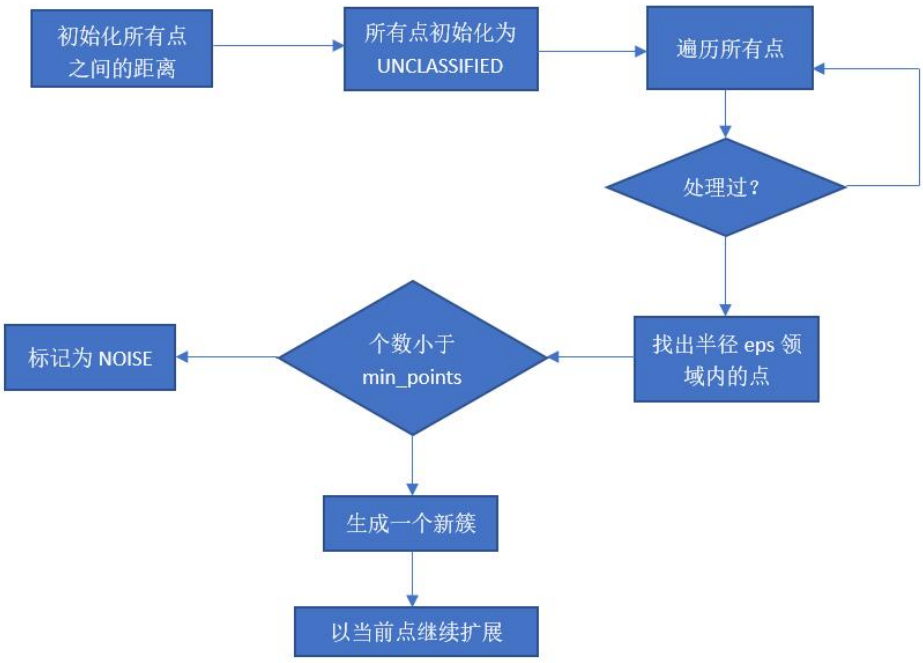


图 3-3 函数流程图

3.3 聚类过程与结果分析

1. 读入数据并查看数据

```
ds_file = os.path.join(os.path.curdir, 'iris.csv')
data = pd.read_csv(ds_file)
```

```
data
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica
150	NaN	NaN	NaN	NaN	NaN

151 rows × 5 columns

2. 查看和对比初始数据分类

由于数据集是四维的，我们不好直接观察，所以我们选取两个维度用散点图的形式展现，并用不同的颜色标记，左图是分成 3 类的一个散点图，右图是不分类情况下的图。

```
iris_types = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
# 用数据中后面两个维度的数据进行数据初始展示
x_axis = 'petal_length'
y_axis =  'petal_width'
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
for iris_type in iris_types:
    plt.scatter(data[x_axis][data['class']==iris_type], data[y_axis][data['class']==iris_type], label = iris_type)
plt.title('label known')
plt.legend()

plt.subplot(1,2,2)
plt.scatter(data[x_axis][:], data[y_axis][:])
plt.title('label unkonwn')
plt.show()
```

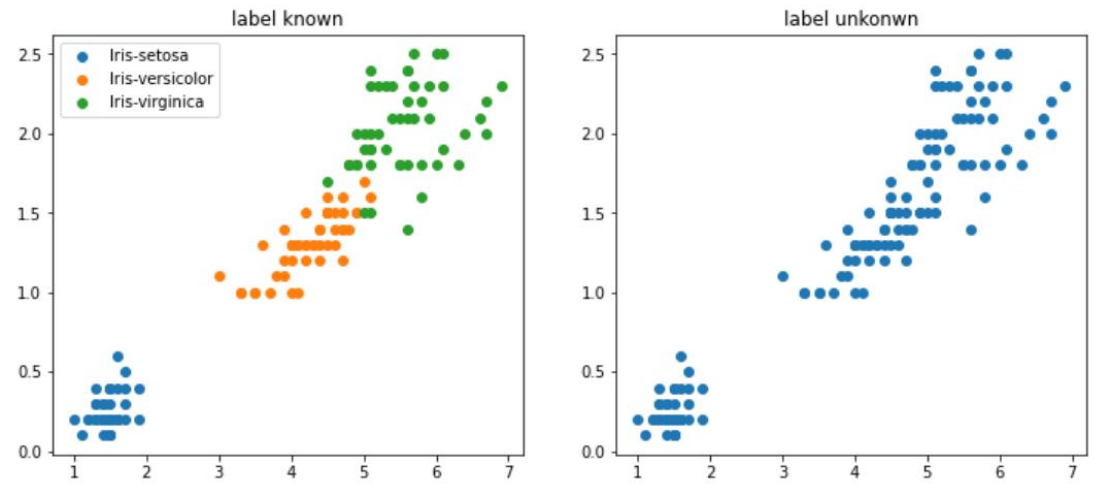
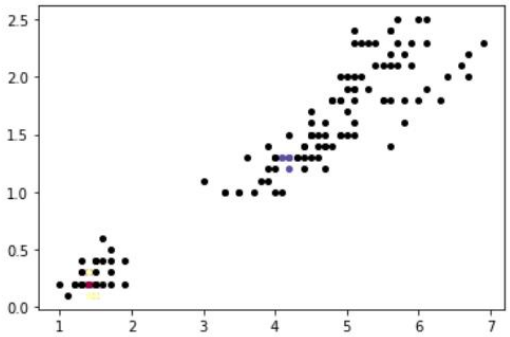


图 3-4 分为 3 类以及不分类情况下的散点图

3. 聚类结果分析

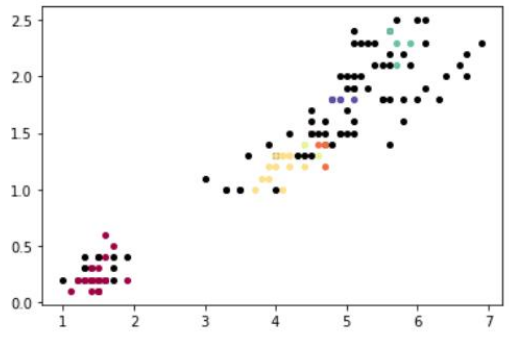
同样的，聚类之前需要确定两个参数，半径 eps 和领域内点的个数阈值 min_points，对于不同的参数会产生不同的效果，实验准确度也差别很大。下面是采用不同参数形成的散点图，为了和初始分类图一致。这里聚类的时候是采用 4 个维度进行的，为了更直观地展示聚类效果，依旧是选取两个维度进行展示。下面是选取不同参数所输出的结果。

1的个数： 9
2的个数： 9
3的个数： 4
聚类个数:3



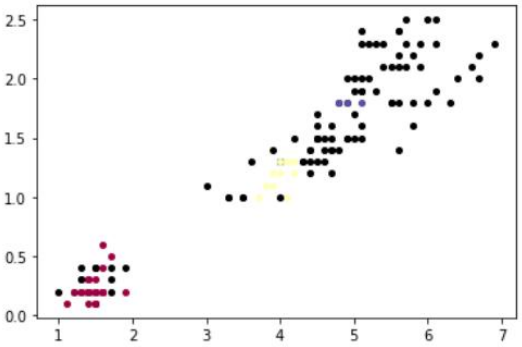
eps=0.2， min_points=4

1的个数： 38
2的个数： 4
3的个数： 13
4的个数： 4
5的个数： 4
6的个数： 6
聚类个数:6



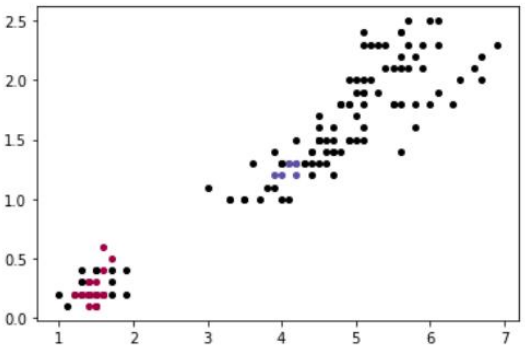
eps=0.3,min_points=4

1的个数： 37
2的个数： 12
3的个数： 5
聚类个数:3



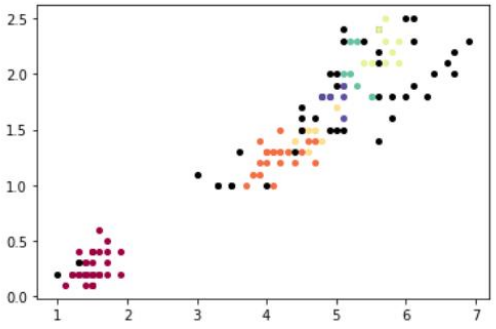
eps=0.3,min_points=5

1的个数： 35
2的个数： 7
聚类个数:2



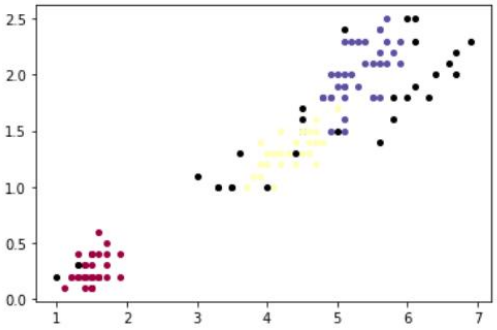
eps=0.3,min_points=7

1的个数： 45
2的个数： 24
3的个数： 11
4的个数： 9
5的个数： 7
6的个数： 7
聚类个数:6



eps=0.4,min_points=7

1的个数： 48
2的个数： 37
3的个数： 36
聚类个数:3



eps=0.424,min_points=5

图 3-5 不同参数情况下的散点图

从上图中可以发现当 min_points 增大时，聚类的个数可能会不断减小，这要求核心对象领域内的点要足够的多，这对于某些点很难满足，所以核心对象的数量减少了，聚类的数量也可能随之减少；当 eps 的值取得过小时，会产生很多的噪声点，因为当 eps 过小时，很多点并不能在核心对象的 eps 领域内，从而导致很多点不属于任意一个核心对象的领域，即噪声点。同理，当 eps 取得过大时，反而会导致聚类数量变多。最终我们发现在参数 eps=0.424，min_points=5 的情况下，聚类结果较为贴近原始数据。即分成 3 个聚类，且每类数据平均接近 50。

四、 Diana 算法

4.1 算法简介

- **算法原理**

Diana（Divisive Analysis）算法属于分裂的层次聚类，首先将所有的对象初始化到一个簇中，然后根据一些原则（比如最邻近的最大欧式距离），将该簇分类。直到到达用户指定的簇数目或者两个簇之间的距离超过了某个阈值。

相关概念：

（1）簇的直径：在一个簇中的任意两个数据点都有一个欧氏距离，这些距离中的最大值是簇的直径

（2）平均相异度（平均距离）

- **算法流程**

数据：

序号	属性 1	属性 2
1	1	1
2	1	2
3	2	1
4	2	2
5	3	4
6	3	5
7	4	4
8	4	5

实例分析：

对于所给的数据进行 DIANA 算法（设 n=8,用户输入的终止条件为两个簇），初始簇{1,2,3,4,5,6,7,8}。

1. 找到具有最大直径的簇，对簇中的每个点计算平均相异度

   序号 1 的平均距离（就是 1 距离其他各个点的距离长度之和除以 7）

   s1=(1+1+1.414+3.6+4.24+4.47+5)/7=2.96;

   序列 2 的平均距离 s2=(1+1.414+1+2.828+3.6+3.6+4.24)/7=2.526;

   序列 3 的平均距离 s3=(1+1.414+1+3.16+4.12+3.6+4.27)/7=2.68;

   序列 4 的平均距离 s4=(1.414+1+1+2.24+3.16+2.828+3.6)/7=2.18

   序列 5 的平均距离 s5=2.18;

   序列 6 的平均距离 s6=2.68;

   序列 7 的平均距离 s7=2.526;

   序列 8 的平均距离 s8=2.96;

   将 8 个点的平均距离中:

   挑选一个最大平均相异度的点 1 放入 splinter group 中，剩余点放在 old party 中 splinter group={1}，old party={2,3,4,5,6,7,8}

2. 在 old party 里找出到 splinter group 的距离小于到 old party 的距离的点，将该点放入 splinter group 中，该点就是 2；

   splinter group={1,2}, old party={3,4,5,6,7,8}

3. 在 old party 里找出到 splinter group 的距离小于到 old party 的距离的点，将该点放在 splinter group 中，该点就是 3；

   spliner group={1,2,3}, old party={4,5,6,7,8}

4. 在 old party 里找出到 splinter group 的距离小于到 old party 的距离的点，将该点放在 splinter group 中，该点就是 4；

   spliner group={1,2,3,4}, old party={5,6,7,8}

5. 在 old party 里找出到 splinter group 的距离小于到 old party 的距离的点。

   此时的分裂的簇数为 2，已经达到了终止条件。

如果还没达到终止条件，下一阶段还会在分裂好的簇中选一个直径最大的簇按刚才的分类方法分裂。

- 伪代码

```
- 输入：包含n个对象的数据库，终止条件簇的数目k
- 输出：k个簇，达到终止条件规定簇数目
        1. 将所有对象整个当成一个初始簇[将每个点当成一个初始簇]
        2. For (i=1;i!=k;i++) Do Begin
        3.      在所有簇中挑选出具有最大直径的簇C
        4.      找出C与其他点**平均相异度最大**的一个点P放入splinter group，
        5.      剩余的放入old party中。
        5.    Repeat
        6.      在old party里找出**到splinter group中点的最近距离** <= **到old party中点的最近距离的点**，并
        7.    Until 没有新的old party的点被分配给splinter group；
        8. Splinter group 和old party为被选中的簇分裂成的两个簇，与其他簇一起组成新的簇集合
        9. END
- 算法性能：缺点是已做的分裂操作不能撤销，类之间不能交换对象。如果在某步没有选择好分裂点，可能会导致低质量的聚类结果。
```

4.2 相关函数实现

1. class Point

这个类会初始化数据点两两之间的距离。x,y,z,w 分别表示这个四位数据的每个维度特征，name 表示名字，id 表示标号，再利用 numpy 的 sqrt 和 dot 函数求任意两点的欧氏距离，因为我们这里的数据有 4 维，设 4 维是 x,y,z,w,那么对应的距离计算公式为：

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 + (w_1 - w_2)^2}$$

```python
class Point:
    '''
    初始化函数
    '''

    def __init__(self, x, y, z, w, name, id):
        self.x = x  # 横坐标
        self.y = y  # 纵坐标
        self.z = 0
        self.w = 0
        self.name = name  # 名字
        self.id = id  # 编号

    '''
    计算两点之间的欧几里得距离
    '''

    def calc_Euclidean_distance(self, p2):
        return math.sqrt((self.x - p2.x) * (self.x - p2.x) + (self.y - p2.y) * (self.y - p2.y) + (self.z - p2.z
                    self.z - p2.z) + (self.w - p2.w) * (self.w - p2.w))
```

2. def get_dataset

获取数据集，将原始数据集以元组形式存放，并且将每个 point 进行初始化并存入 dataset，返回 dataset 点的集合以及 id_point_dict 编号和点的映射。

```python
def get_dataset():
    # 原始数据集以元组形式存放，(横坐标，纵坐标，编号)
    # ds_file = os.path.join()
    with open('./Garden-Iris-master/iris.data') as f:
        datas = [tuple(line) for line in csv.reader(f)]
    print(datas)
    # datas = [(0, 2, 'A'), (0, 0, 'B'), (1.5, 0, 'C'), (5, 0, 'D'), (5, 2, 'E')]
    dataset = []  # 用于计算两点之间的距离，形式 [point1, point2...]
    id_point_dict = {}  # 编号和点的映射
    temp_list = []
    for i in range(len(datas)):  # 遍历原始数据集
        point = Point(datas[i][0], datas[i][1], datas[i][2], datas[1][3], datas[1][4], i)  # 利用(横坐标，纵坐标，编号)实例化
        id_point_dict[str(i)] = point
        dataset.append(point)  # 放入dataset中
        temp_list.append(point)
    return dataset, id_point_dict  # [p1, p2], {id: point}
```

3. def get_dist(dataset)

计算存入的 dataset 点集合中两两之间点的距离，并将 dist 距离的列表返回。

```python
def get_dist(dataset):
    n = len(dataset)  # 点的个数
    dist = []  # 存放任意两点之间的距离
    for i in range(n):
        dist_i = []  # 临时列表
        for j in range(n):  # 遍历数据集
            # 计算距离并放入临时列表中
            dist_i.append(dataset[i].calc_Euclidean_distance(dataset[j]))
        dist.append(dist_i)  # 利用临时列表创建二维列表
    # 打印dist
    print("任意两点之间的距离：")
    for d in dist:
        print(d)
    print()
    return dist
```

4. def dissimilitude(dist, ids)

这个函数用来计算簇内数据点相异度，参数中 dist 代表点与点的距离的二维列表，ids 表示簇中的数据点的集合。

```python
def get_dissimilitude(dist, ids):
    n = len(ids)  # 这个簇的数据点个数
    dissimilitudes = {}  # 存放数据点相异度
    for id1 in ids:
        id1_num = int(id1)
        d = 0  # 点id1的相异度，初始化为0
        for id2 in ids:  # 遍历其它数据点
            id2_num = int(id2)
            d += dist[id1_num][id2_num]  # 加上两点距离
        dissimilitudes[id1] = d / (n - 1)  # 计算相异度
    return dissimilitudes
```

5. def max_diff(dissimilitudes)

寻找最大相异度的点，参数 dissimilitudes 代表之前得到的相异度字典，返回最大相异度值的数据点编号。

```
def get_max_diff(dissimilitudes):
    Max = -1  # 最大相异度值，初始化为一个负值
    Max_id = -1  # 最大相异度值的数据点编号
    for id, diff in dissimilitudes.items():  # 遍历之前得到的相异度字典
        if diff > Max:  # 有更大的，就更新
            Max = diff
            Max_id = id
    return Max_id  # 返回最大相异度值的数据点编号
```

6. def DIANA(dataset, k, id_point_dict)

该函数为 Diana 算法实现主函数，dataset 表示所有点的集合，k 表示将所有点分为几个簇，id_point_dict 则为了操作更简单，而加入的编号和点的映射。

首先获取任意两点之间距离（欧几里得距离），其次将初始簇中包含所有数据点的编号进行存入，再将初始簇存入结果列表。

然后开始一个循环，当簇的个数为 k 个时，退出循环。splinter group 表示新划分出去的簇，old_party 表示旧的簇。在计算 ids 这个簇的相异度之后，得到这个簇里面最大相异度的数据点，将这个点放入 splinter group 中，其余的数据点放入 old party 中。

接着在 old party 中寻找到 splinter group 中的点（E 点）的最近距离小于等于到 old party 中的点的最近距离的点，找出 D 点，把该点加入 splinter group 中。在此数据集中，仅有点 D 到点 E 的距离 2.3<3.5（5.3，5，3.5），所以将点 D 加入到 splinter group 中（D, E 点）。

最后通过判断簇中的数量是否发生变化，如果发生变化了，就更新结果列表 res，即删除旧簇，加入两个新簇。

```python
def DIANA(dataset, k, id_point_dict):
    dist = get_dist(dataset)  # 获取任意两点之间距离（欧几里得距离）
    res = []  # 结果列表，存放每次操作完成后的簇组合
    ids = []  # 初始簇
    for i in range(len(dataset)):
        ids.append(str(i))  # 初始簇中包含所有数据点的编号
    res.append(ids)  # 初始簇入结果列表

    while len(res) < k:  # 簇的个数为k个时，退出循环
        t_res = []  # 结果列表res的复制，只用于遍历
        for t in res:
            t_res.append(t)
        for ids in t_res:  # 遍历复制的结果列表
            splinter_group = []  # splinter group
            old_party = []  # old party
            dissimilitudes = get_dissimilitude(dist, ids)  # 计算ids这个簇的相异度
            Max_id = get_max_diff(dissimilitudes)  # 得到这个簇里最大相异度的数据点
            splinter_group.append(Max_id)  # 放入splinter group
            for id in ids:  # 其余数据点放入old party
                old_party.append(id)
            old_party.remove(Max_id)  # 全放进去，然后把最大点删掉就可以了
            pre_len = -1  # 用于判断old_party列表不再增加时，退出循环
            while pre_len != len(old_party):  # 不相等说明，old_party列表还在变化
                pre_len = len(old_party)  # 更新pre_len
                change_ids = []
                # 在old party中寻找 到splinter group中的点（E点）的最近距离
                # 小于等于到old party中的点的最近距离的点，找出D点，
                # 把该点加入splinter group中。在此数据集中，
                # 仅有点D到点E的距离2.3<3.5（5.3，5，3.5），
                # 所以将点D加入到splinter group 中（D,E点）；
                for id1 in old_party:  # 在old party中寻找，遍历
                    Min = float("INF")
                    flag = True  # 判断该点是否符合要求
                    for id2 in splinter_group:  # splinter_group中若有多个点，需要找到最近距离
                        if dist[int(id1)][int(id2)] < Min:
                            Min = dist[int(id1)][int(id2)]
                    for id3 in old_party:  # 寻找最近距离小于等于到old party中的点的最近距离的点
                        if (Min > dist[int(id1)][int(id3)]) and (id1 != id3):  # 不符合要求的置为Fa
                            flag = False
                            break
                    if flag:  # 该点符合要求
                        change_ids.append(id1)  # 放入change_ids列表中，表示需要变化的数据点
                for id in change_ids:  # 遍历
                    old_party.remove(id)  # 从old_party中删除
                    splinter_group.append(id)  # 放入splinter_group
            if len(splinter_group) != 0 and len(old_party) != 0:  # 当前簇发生变化了，更新结果列表res
                res.remove(ids)  # 删除旧簇
                res.append(splinter_group)  # 加入两个新簇
                res.append(old_party)
```

7. draw_cluster

聚类绘图函数，用于展示聚类的结果，这里选取的是三个维度进行散点图显示。

```
plt.figure()
d0 = irisdata[clustering.labels_ == 0]
plt.plot(d0[:, 0], d0[:, 1], 'r.')
d1 = irisdata[clustering.labels_ == 1]
plt.plot(d1[:, 0], d1[:, 1], 'go')
d2 = irisdata[clustering.labels_ == 2]
plt.plot(d2[:, 0], d2[:, 1], 'b*')
plt.xlabel("Sepal.Length")
plt.ylabel("Sepal.Width")
plt.title("Diana Clustering")
plt.show()
```

下面是上述函数对应的流程图：

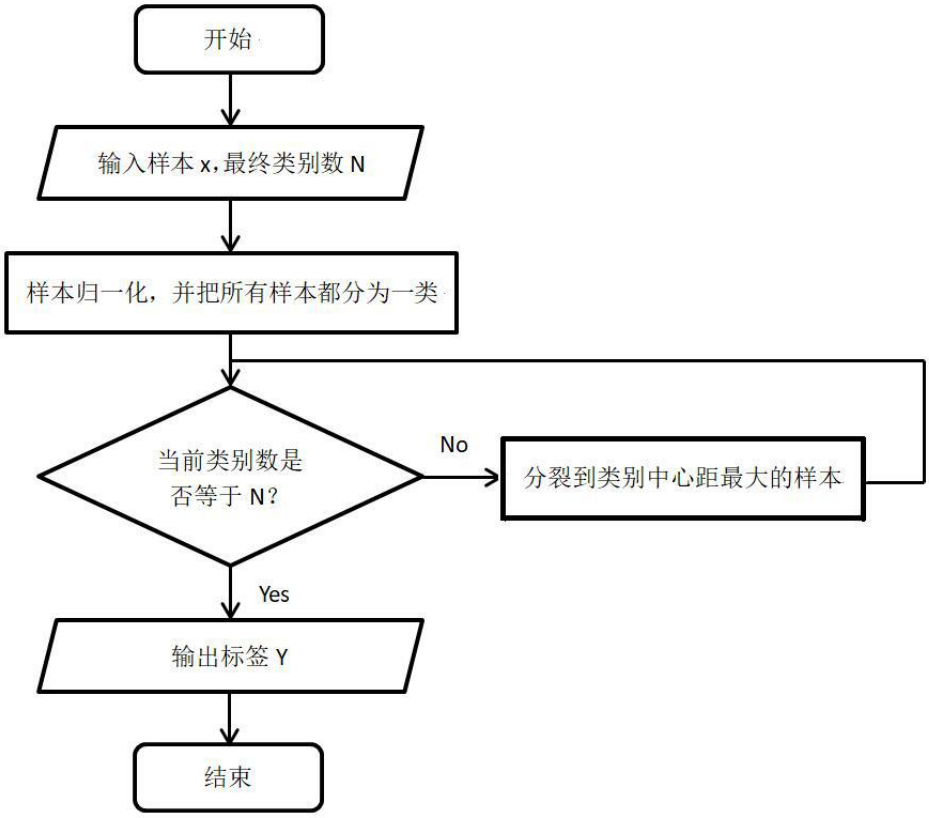


图 4-1 函数流程图

4.3 聚类过程与结果分析

1. 读入数据并查看数据

```
with open('./Garden-Iris-master/iris.data') as f:
    datas = [tuple(line) for line in csv.reader(f)]
print(datas)
```

	sepal_length_cm	sepal_width_cm	petal_length_cm	petal_width_cm	class
0	4.9	3.0	1.4	0.2	Iris-setosa
1	4.7	3.2	1.3	0.2	Iris-setosa
2	4.6	3.1	1.5	0.2	Iris-setosa
3	5.0	3.6	1.4	0.2	Iris-setosa
4	5.4	3.9	1.7	0.4	Iris-setosa
5	4.6	3.4	1.4	0.3	Iris-setosa
6	5.0	3.4	1.5	0.2	Iris-setosa
7	4.4	2.9	1.4	0.2	Iris-setosa
8	4.9	3.1	1.5	0.1	Iris-setosa
9	5.4	3.7	1.5	0.2	Iris-setosa
10	4.8	3.4	1.6	0.2	Iris-setosa
11	4.8	3.0	1.4	0.1	Iris-setosa
12	4.3	3.0	1.1	0.1	Iris-setosa
13	5.8	4.0	1.2	0.2	Iris-setosa
14	5.7	4.4	1.5	0.4	Iris-setosa
15	5.4	3.9	1.3	0.4	Iris-setosa

2. 将数据聚类为 3 个簇

由于数据集是四维的，我们不好直接观察，所以我们选取两个维度用散点图的形式展现，并用不同的颜色标记，下图是分成 3 类的一个散点图。

```
plt.figure()
d0 = irisdata[clustering.labels_ == 0]
plt.plot(d0[:, 0], d0[:, 1], 'r.')
d1 = irisdata[clustering.labels_ == 1]
plt.plot(d1[:, 0], d1[:, 1], 'go')
d2 = irisdata[clustering.labels_ == 2]
plt.plot(d2[:, 0], d2[:, 1], 'b*')
plt.xlabel("Sepal.Length")
plt.ylabel("Sepal.Width")
plt.title("Diana Clustering")
plt.show()
```

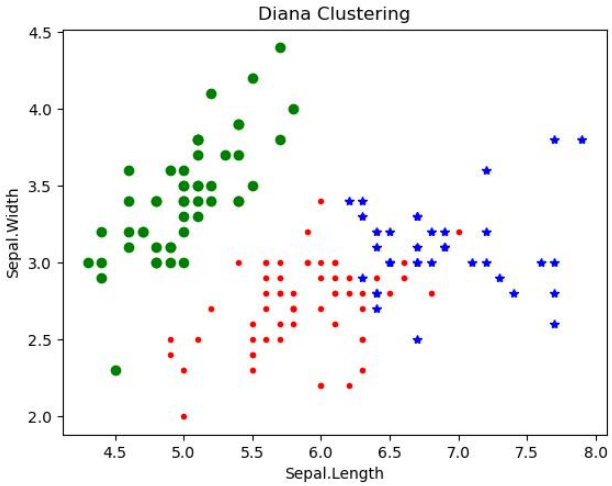


图 4-2 分 3 类情况下的散点图

3. Pairplot 效果展示

pairplot 主要展现的是变量两两之间的关系（线性或非线性，有无较为明显的相关关系）。

```
import matplotlib.pyplot as plt
import seaborn as sb

sb.pairplot(iris_data.dropna(), hue='class')
```

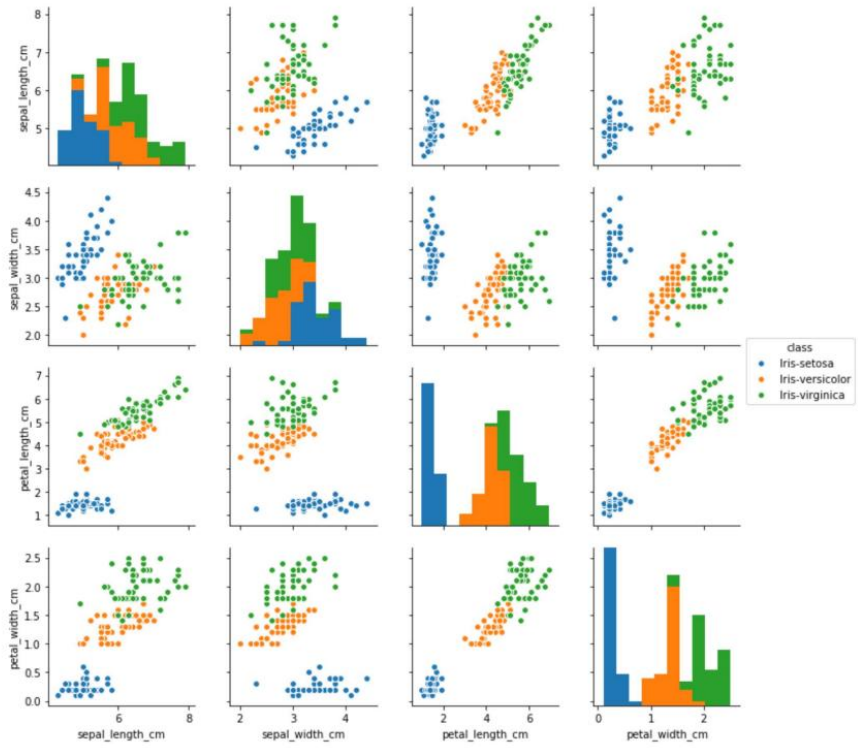


图 4-3 pairplot 图

4. Violin plot

小提琴图(Violin Plot)用于显示数据分布及其概率密度。这种图表结合了箱形图和密度图的特征，主要用来显示数据的分布形状。中间白点为中位数，中间的黑色粗条表示四分位数范围。上下贯穿小提琴图的黑线代表最小非异常值 min 到最大非异常值 max 的区间，线上下端分别代表上限和下限，超出此范围为异常数据（或者，从黑色粗条延伸的细黑线代表 95%置信区间）。

```
plt.figure(figsize=(10, 10))
for column_index, column in enumerate(iris_data.columns):
    if column == 'class':
        continue
    plt.subplot(2, 2, column_index + 1)
    sb.violinplot(x='class', y=column, data=iris_data)
```

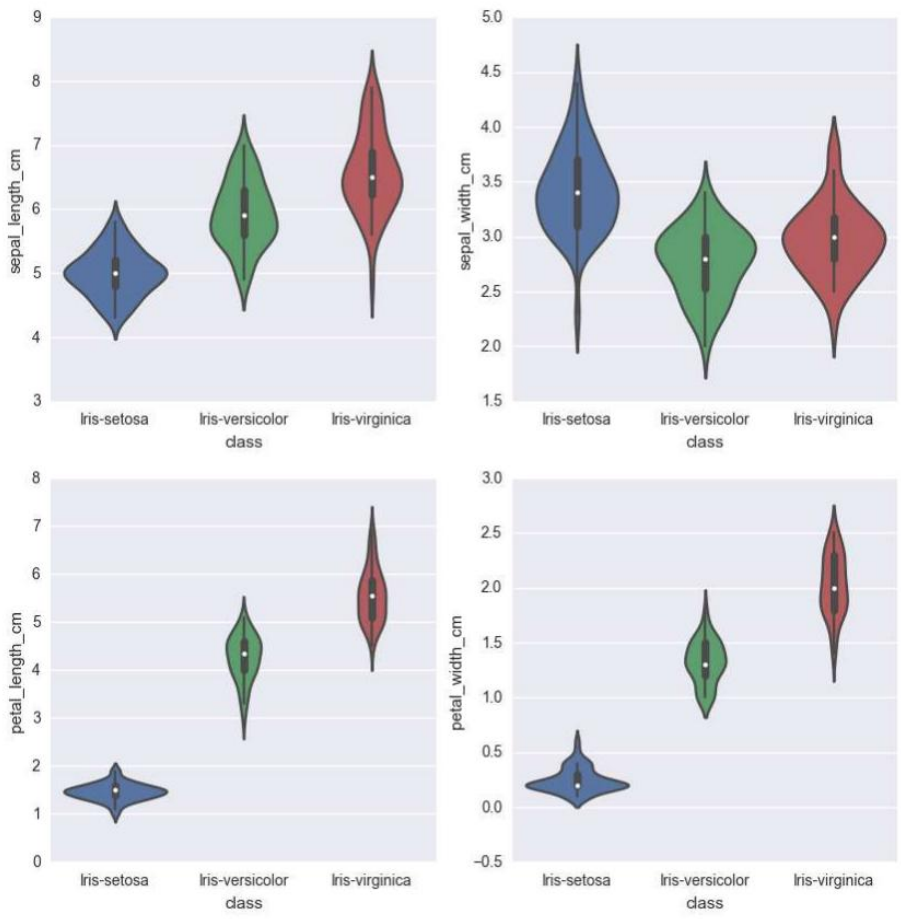
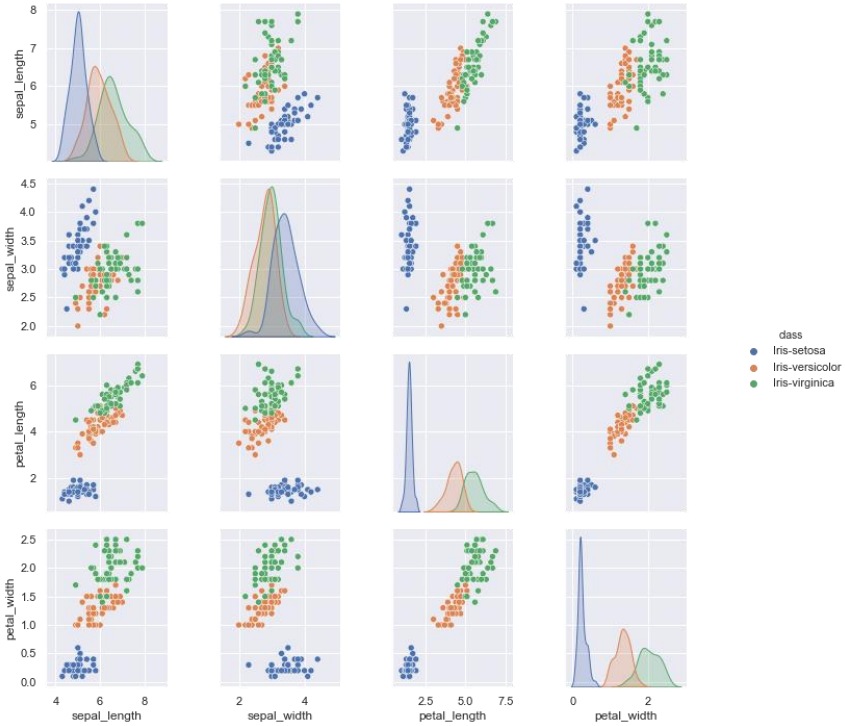


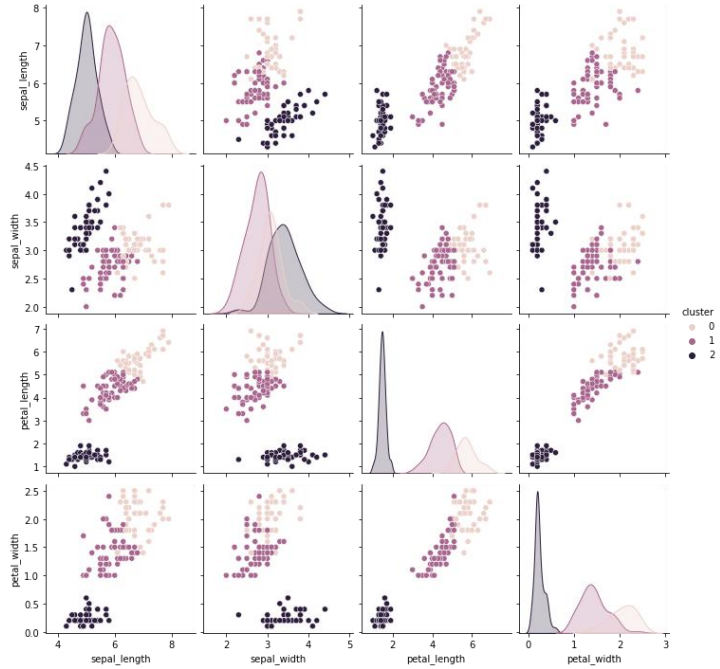
图 4-4 violin plot 图

五、 对比分析

- 原始数据集

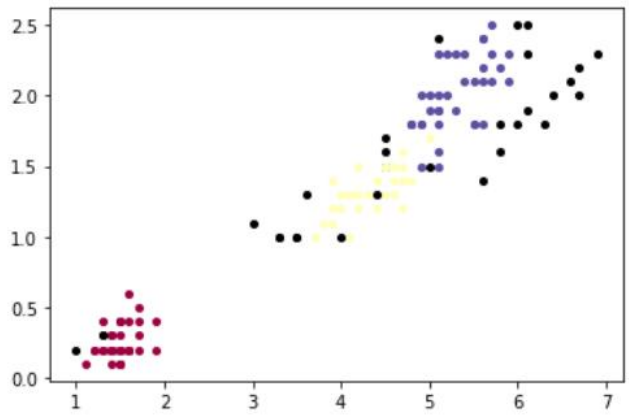


- Kmeans 算法聚类结果图

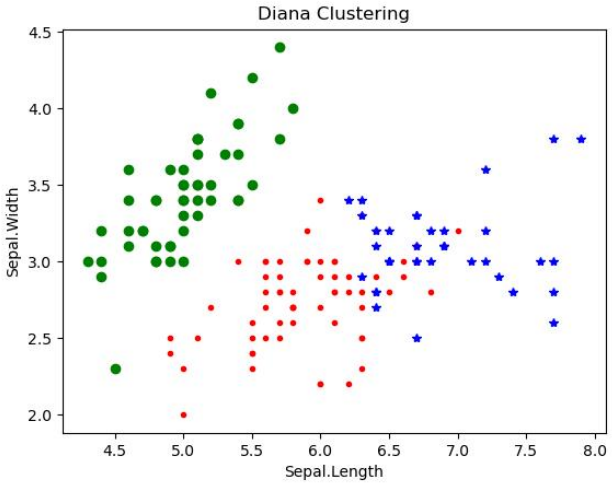


- DBSCAN 算法聚类结果

```
1的个数：  48
2的个数：  37
3的个数：  36
聚类个数:3
```



- Diana 算法聚类结果



以上是三种算法聚类的结果，与原始数据集散点图相比较可以看出，对于鸢尾花数据集来说 Diana 算法聚类效果更好，聚类分布的也更为均匀。

这是因为 Diana 聚类算法对时间和空间需求很大，这种层次聚类算法适合于小型数据集的聚类，而本数据集正好为小型数据集。同时在不清楚数据集不清楚聚成几类的情况下，分层聚类可以在不同粒度水平上对数据进行探测，发现类之间的层次关系，这也是 Diana 层次聚类算法优于其他算法的原因之一。

DBSCAN 算法由于半径 eps 和阈值 min_points 难以确定很容易会产生离群点，甚至有时因为参数的微小变化而导致聚类的数量和结果变化很大，并不适合此数据集。

而 Kmeans 算法的聚类结果对初始聚类中的选择依赖性强，当选取到的 k 个对象不适合作为中点时，不仅会增加聚类的迭代次数，增加聚类的时间复杂性，

甚至有可能造成错误的聚类结果。另外 kmeans 算法偏向于识别球形或类球形结构的簇，对非凸形状的类簇识别效果差，可以看到原数据的分布呈类凸形状，相同类别的点并不是分布集中，而是和其它类别的点有重叠，因此导致 kmeans 对这些样本数据聚类效果并不是很理想。

六、 参考文献

[1] 阿曼多.凡丹戈. Python 数据分析[M]. 北京：人民邮电出版社，2018. 601.

[2] 贺玲,吴玲达,蔡益朝.数据挖掘中的聚类算法综述[J].计算机应用研究,2007,(01):10-13.