第3章

聚类与降维

人们在面对大量未知事物时，往往会采取分而治之的策略，即先将事物按照相似程度分成多个组，然后按组对事物进行处理。机器学习里的聚类就是用来完成对事物进行分组的任务。Cluster 常翻译为簇或簇类，聚类算法是对样本进行分簇(组)的算法。本章将讨论 k-means、DBSCAN、OPTICS、Mean Shift 和 GaussianMixture 聚类算法，其中 GaussianMixture 聚类算法属于概率模型，其他算法属于决策函数模型。

降维(Dimensionality Reduction)是处理维数灾难的一种方法，它将高维空间中的样本点映射到低维空间中，以减少样本的特征数量。通过将高维降到二维或者三维，可以直观地看到样本点在空间中的分布，有助于人们对样本数据的理解并选择适用的机器学习算法。本章将讨论 PCA 降维算法。

聚类和降维是无监督学习的两个重要内容。在无监督学习中，训练样本没有标签，等同于实例，因此，在本章不对样本和实例进行特别区分。


视频讲解

3.1 k 均值聚类算法

k 均值聚类算法(k-means Clustering Algorithm)是一种迭代求解算法。迭代求解算法的讨论见 1.4 节。

聚类算法对样本集按相似性进行分簇，因此，聚类算法能够运行的前提是要有样本集以及能对样本之间的相似性进行比较的方法。

样本的相似性差异称为样本距离，相似性比较称为距离度量。

设样本特征维数为 n，第 i 个样本表示为 $\boldsymbol{x}_i=\{x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)}\}$。因此，样本可以

看成 n 维空间中的点。当 $n=2$ 时，样本可以看成是二维平面上的点。二维平面上两点 $\boldsymbol{x}_i$ 和 $\boldsymbol{x}_j$ 之间的距离常用式(3-1)来计算：

$$L_2(\boldsymbol{x}_i,\boldsymbol{x}_j)=\sqrt[2]{(x_i^{(1)}-x_j^{(1)})^2+(x_i^{(2)}-x_j^{(2)})^2}$$
$$=\sqrt[2]{\sum_{l=1}^{2}(x_i^{(l)}-x_j^{(l)})^2} \tag{3-1}$$

该距离度量方法称为欧氏距离(Euclidean Distance)。k 均值聚类算法常采用欧氏距离作为样本距离度量准则。

将式(3-1)表示的二维平面上两点间欧氏距离的计算公式推广到 n 维空间中两点 $\boldsymbol{x}_i$ 和 $\boldsymbol{x}_j$ 的欧氏距离计算公式：

$$L_2(\boldsymbol{x}_i,\boldsymbol{x}_j)=\sqrt[2]{\sum_{l=1}^{n}(x_i^{(l)}-x_j^{(l)})^2} \tag{3-2}$$

设样本总数为 m，样本集为 $\boldsymbol{S}=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_m\}$。$k$ 均值聚类算法对样本集分簇的个数是事先指定的，即 k。设分簇后的集合表示为 $\boldsymbol{C}=\{\boldsymbol{C}_1,\boldsymbol{C}_2,\cdots,\boldsymbol{C}_k\}$，其中每个簇都是样本的集合。

k 均值聚类算法的基本思想是让簇内的样本点更“紧密”一些，也就是说，让每个样本点到本簇中心的距离更近一些。该算法常采用该距离的平方之和作为“紧密”程度的度量标准，因此，使每个样本点到本簇中心的距离的平方和尽量小是 k-means 算法的优化目标。

每个样本点到本簇中心的距离的平方和也称为误差平方和(Sum of Squared Error，SSE)。从机器学习算法的实施过程来说，这类优化目标一般统称为损失函数(Loss Function)或代价函数(Cost Function)。

当采用欧氏距离，并以误差平方和 SSE 作为损失函数时，一个簇的簇中心按如下方法计算。

对于第 i 个簇 $\boldsymbol{C}_i$，簇中心 $\boldsymbol{u}_i=(u_i^{(1)},u_i^{(2)},\cdots,u_i^{(n)})$ 为簇 $\boldsymbol{C}_i$ 内所有点的均值，簇中心 $\boldsymbol{u}_i$ 第 j 个特征为：

$$u_i^{(j)}=\frac{1}{|\boldsymbol{C}_i|}\sum_{\boldsymbol{x}\in\boldsymbol{C}_i}x^{(j)} \tag{3-3}$$

其中，$|\boldsymbol{C}_i|$ 表示簇 $\boldsymbol{C}_i$ 中样本的总数。

SSE 的计算方法为：

$$\mathrm{SSE}=\sum_{i=1}^{m}[\mathrm{dist}(\boldsymbol{x}_i,\boldsymbol{u}_{C_{(i)}})]^2 \tag{3-4}$$

其中，dist(•)是距离计算函数，常用欧氏距离 L_2；$\boldsymbol{u}_{C_{(i)}}$ 表示样本 $\boldsymbol{x}_i$ 所在簇的中心。

k 均值聚类算法基本流程如图 3-1 所示。

k 均值聚类算法以计算簇中心并重新分簇为一个周期进行迭代，直到簇稳定(分配结果不再发生变化)为止。下面来看一个对二维平面上的点进行聚类的例子。本书附属资源文件 kmeansSamples.txt 存放了 30 个点的坐标。代码 3-1 查看各点坐标。

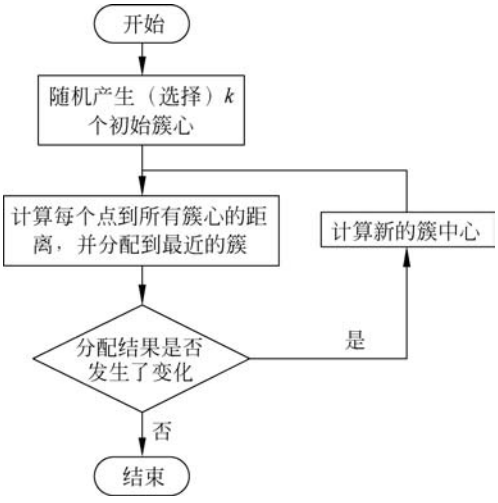


图 3-1 *k*-means 算法流程

代码 3-1 kmeansSamples.txt 文件中的点坐标(kmeans 算法及示例.ipynb)

```
1. >>> import numpy as np
2. >>> samples = np.loadtxt("kmeansSamples.txt")
3. >>> print(samples)
4. [[ 8.76474369 14.97536963]
5.  [ 4.54577845 7.39433243]
6.  [ 5.66184177 10.45327224]
7.  [ 6.02005553 18.60759073]
8.  [12.56729723 5.50656992]
9.  [ 4.18694228 14.02615036]
10. [ 5.72670608 8.37561397]
11. [ 4.09989928 14.44273323]
12. ...
```

第 2 行用 NumPy 库的 loadtxt()函数将 kmeansSamples.txt 文件中的数据加载到列表 samples 中。该函数可以对文本格式的文件(包括 TXT、CSV 等后缀的文件)进行去注释、指定分隔符、选择指定行和列等操作,并将数据加载到指定的列表中。

对以上 30 个点进行 *k*-means 聚类。第一次迭代先随机产生 3 个簇中心:[[-1.93964824 2.33260803] [7.79822795 6.72621783] [10.64183154 0.20088133]],然后进行分簇,如图 3-2 所示,图中三角形表示簇中心所在位置,三种不同大小的圆点表示不同的三个簇。第一次迭代后,计算得到 SSE 值为 1674.1944460020268。

第二次迭代计算得到簇中心:[[-1.37291143 3.62583718] [6.49809152 12.82443961] [5.55255572 -0.06114142]],分簇如图 3-3 所示。可以看到,经过一次迭代之后,分簇更为合理。第二次迭代后,计算得到 SSE 值为 641.6091611948824。

第三次迭代计算得到簇中心:[[-2.0989295 3.9554255] [6.25766711 13.77999631] [6.07984882 1.54796222]],分簇如图 3-4 所示。第三次迭代后,计算得到 SSE 值为 595.6061857081733。

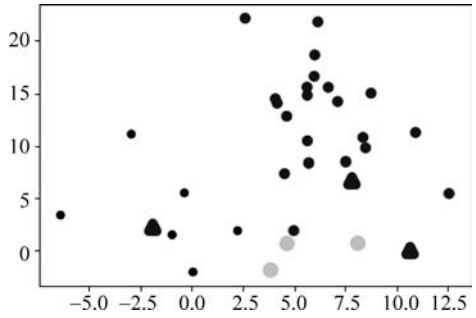


图 3-2　*k*-means 算法举例第 1 次迭代

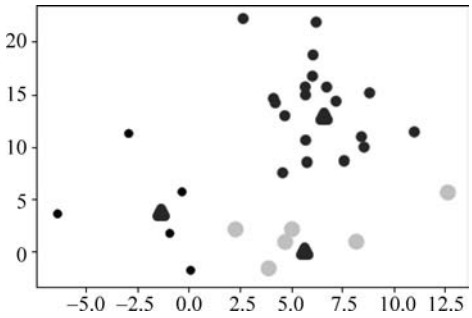


图 3-3　*k*-means 算法举例第 2 次迭代

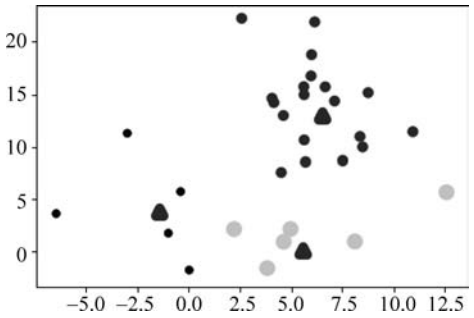


图 3-4　*k*-means 算法举例第 3 次迭代

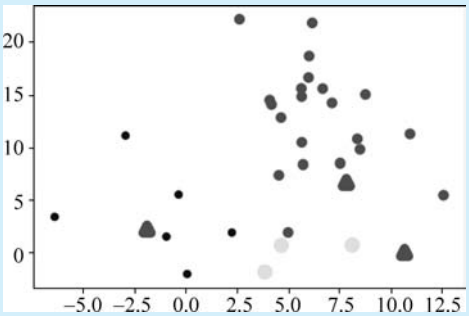
实现上述算法的代码见代码 3-2。

代码 3-2　*k*-means 示例代码（*k*-means 算法及示例. ipynb）

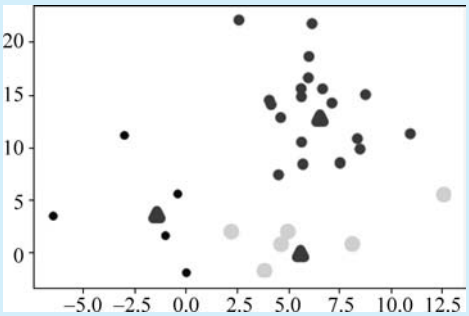
```
def L2(vecXi, vecXj):
    '''
    计算欧氏距离
    para vecXi: 点坐标,向量
    para vecXj: 点坐标,向量
    retrurn: 两点之间的欧氏距离
    '''
    return np.sqrt(np.sum(np.power(vecXi - vecXj, 2)))

from sklearn.metrics import silhouette_score, davies_bouldin_score
def kMeans(S, k, distMeas = L2):
    '''
    k 均值聚类算法
    para S: 样本集,多维数组
    para k: 簇个数
    para distMeas: 距离度量函数,默认为欧氏距离计算函数
    return sampleTag: 一维数组,存储样本对应的簇标记
    return clusterCents: 一维数组,各簇中心
    retrun SSE:误差平方和
```

```
    '''
    m = np.shape(S)[0]                    # 样本总数
    sampleTag = np.zeros(m)

    # 随机产生 k 个初始簇中心
    n = np.shape(S)[1]                    # 样本向量的特征数
    clusterCents = np.mat([[ - 1.93964824, 2.33260803], [7.79822795, 6.72621783],
    [10.64183154, 0.20088133]])  # 为可重复实验,注释掉随机产生簇中心的代码,改为
                                          # 指定三个簇中心
    #clusterCents = np.mat(np.zeros((k,n)))
    #for j in range(n):
    #    minJ = min(S[:,j])
    #    rangeJ = float(max(S[:,j]) - minJ)
    #    clusterCents[:,j] = np.mat(minJ + rangeJ * np.random.rand(k,1))

    sampleTagChanged = True
    SSE = 0.0
    while sampleTagChanged:               # 如果没有点的分配结果改变,则结束
        sampleTagChanged = False
        SSE = 0.0

        # 计算每个样本点到各簇中心的距离
        for i in range(m):
            minD = np.inf
            minIndex = -1
            for j in range(k):
                d = distMeas(clusterCents[j,:],S[i,:])
                if d < minD:
                    minD = d
                    minIndex = j
            if sampleTag[i] != minIndex:
                sampleTagChanged = True
            sampleTag[i] = minIndex
            SSE += minD**2
        print(clusterCents)
        # 为了演示分簇过程,在每次迭代中都画出簇心和簇成员
    plt.scatter(clusterCents[:,0].tolist(),clusterCents[:,1].tolist(),c = 'r',marker = '^',
    linewidths = 7)
    plt.scatter(S[:,0],S[:,1],c = sampleTag,linewidths = np.power(sampleTag + 0.5, 2))
                                          # 用不同大小的点来表示不同簇的点
        plt.show()
        print("SSE:" + str(SSE))
        print("SC:" + str(silhouette_score(S, sampleTag, metric = 'euclidean')))
        # 聚类算法评价指标(在后文讨论)
        print("DBI:" + str(davies_bouldin_score(S, sampleTag)))     # 聚类算法评价指
        # 标(在后文讨论)

        print("--------------------------")
```

```
62.
63.         # 重新计算簇中心
64.         for i in range(k):
65.             ClustI = S[np.nonzero(sampleTag[:] == i)[0]]
66.             clusterCents[i,:] = np.mean(ClustI, axis = 0)
67.     return clusterCents, sampleTag, SSE
68.
69. import matplotlib.pyplot as plt
70. samples = np.loadtxt("kmeansSamples.txt")
71. clusterCents, sampleTag, SSE = kMeans(samples, 3)
72. plt.show()
73. print(clusterCents)
74. print(SSE)
75. >>>
76. [[ -1.93964824 2.33260803]
77.  [ 7.79822795 6.72621783]
78.  [10.64183154 0.20088133]]
```

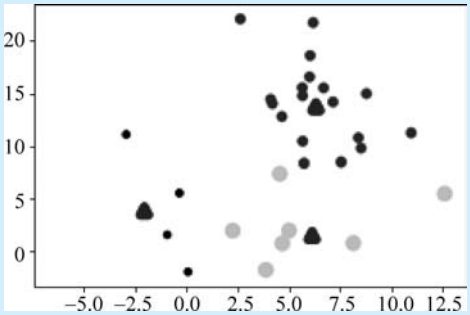



```
79.
80. SSE:1674.1944460020268
81. SC:0.3633082354377029
82. DBI:0.8056369072268063
83.  -------------------------
84. [[ -1.37291143 3.62583718]
85.  [ 6.49809152 12.82443961]
86.  [ 5.55255572  -0.06114142]]
```

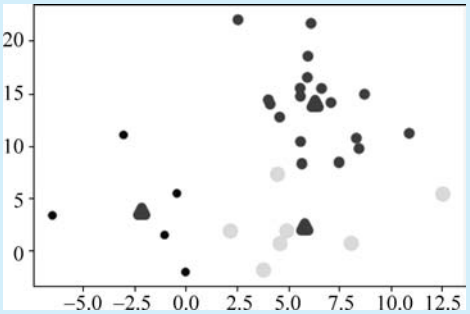



```
87.
88. SSE:641.6091611948824
```

```
SC:0.4332385538550796
DBI:0.8403130057712209
-------------------------
[[ -2.0989295 3.9554255 ]
 [ 6.25766711 13.77999631]
 [ 6.07984882 1.54796222]]
```




```

SSE:595.6061857081733
SC:0.41650941198623903
DBI:0.8981951114181145
-------------------------
[[ -2.0989295 3.9554255 ]
 [ 6.35277204 14.13475541]
 [ 5.86069591 2.38315796]]
```




```

SSE:587.9589447573272
SC:0.41650941198623903
DBI:0.8981951114181145
-------------------------
[[ -2.0989295 3.9554255 ]
 [ 6.35277204 14.13475541]
 [ 5.86069591 2.38315796]]
587.9589447573272
```

经过4个周期的迭代,簇结构不再发生变化,算法结束。最后一个周期产生的簇为最终聚类结果。每个迭代得到的SSE值分别约为:1674,641,595,587。可见随着簇结构的

优化，损失函数值一直下降。

在 sklearn 的 cluster 模块中提供了实现 k-means 算法的 KMeans 类，在 Scikit-Learn 官方网站可直接阅读源代码[①]。KMeans 类及常用方法原型见代码 3-3。

代码 3-3　sklearn 中的 KMeans 类及常用方法

```
class sklearn.cluster.KMeans(n_clusters = 8, init = 'k - means++', n_init = 10, max_iter =
300, tol = 0.0001, precompute_distances = 'auto', verbose = 0, random_state = None, copy_x
= True, n_jobs = None, algorithm = 'auto')

fit(X[, y, sample_weight])                  # 分簇训练
fit_predict(X[, y, sample_weight])          # 分簇训练并给出每个样本的簇号
predict(X[, sample_weight])                 # 在训练之后，对输入的样本进行预测
transform(X)                                # 计算样本点 X 与各簇中心的距离
```

n_clusters 是指定超参数 k 的值。其他输入参数和返回值，在网站上有详细介绍，建议直接看原版文档，这里仅讨论几个实例化时常用的重要参数。

1）init 参数

在上文 k-means 算法实现示例中，初始簇中心采用随机产生的办法（代码 3-2 为了实验的可复现，指定了初始簇中心）。采用随机产生初始簇中心的方法，可能会出现运行结果不一致的情况。这是由于最优化计算中的局部最优问题而产生的。

从示例可以看出，在确定了样本集和分簇数后，k-means 算法的任务就成了使 SSE 最小的优化计算。最优化计算是机器学习中极为重要的基础，各类算法大都可归结为一个最优化问题。最优化问题的基本内容将在后续章节中进行必要的讨论。

在最优化问题中，常常会出现所谓的局部最优解，如图 3-5 所示。局部最优解是在小范围内的最优解。全局最优解是在问题域内的最优解。

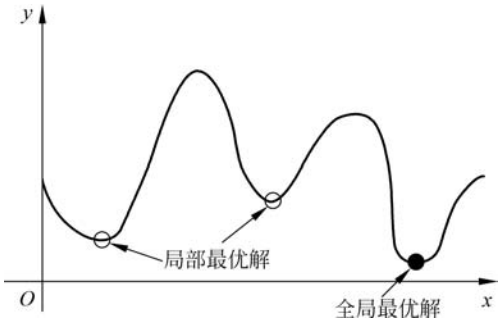


图 3-5　全局最优解和局部最优解

在 k-means 算法中，如果初始点选取不好，就会陷入局部最优解，而无法得到全局最优解。如果多次运行代码 3-2，可以发现每次的分簇结果和 SSE 值未必相同。这是因为不同的初始簇中心使得算法可能收敛到不同的局部极小值。

① https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

不能收敛到全局最小值,是最优化计算中常常遇到的问题。有一类称为凸优化的优化计算,不存在局部最优问题。凸优化是指损失函数为凸函数的最优化计算。凸函数没有局部极小值这样的小"洼地",因此是最理想的损失函数。如果能将优化目标转化为凸函数,就可以解决局部最优问题。有关机器学习中的最优化计算和凸优化的详细讨论,在需要时,读者可以参考原版书[①]中有关内容。

不幸的是,SSE一般不是凸函数,所以人们采用了许多尽量避免陷入局部极小值的方法。方法之一就是设置初始簇中心。

KMeans类通过init参数提供了三种设置初始簇中心的方法,分别为k-means++、random和用户指定。其中k-means++方法就是一种尽量避免陷入局部极小值的方法。

k-means++通过一个算法来产生初始簇中心,其基本思想是使初始簇中心尽量分散开,从而尽可能使算法取得全局最优解。

random由算法随机产生簇中心。

用户指定通过一个ndarray数组将用户设置好的初始簇中心传入算法。

2) n_init参数

n_init参数提供了另一种使算法尽量取得全局最优解的方法,它指定算法重复运行次数。它在不指定初始簇中心时,通过多次重复运行算法,最终选择最好的结果作为输出。

3) max_iter参数和tol参数

max_iter参数和tol参数是迭代的退出条件(见1.4节)。max_iter参数指定一次运行中的最大迭代次数,当达到最大次数时结束迭代。在大规模数据集中,算法往往要耗费大量的时间,可通过指定迭代次数来折衷耗时和效果。tol参数指定连续两次迭代变化的阈值,如果损失函数的变化小于阈值,则结束迭代。在大规模数据集中,算法往往难以完全收敛,即达到连续两次相同的分簇需要耗费很长时间,因此可以通过指定阈值来折衷耗时和效果。

下面继续讨论有关k-means算法在具体应用中的两个问题。

1) 超参数k值的确定

k-means算法需要事先指定簇数量k值,它是超参数。在很多应用场合,该值是明确的;在另一些应用场合,该值并不能事先确定,使得该算法的应用受到一定限制。

可以对不同的k值逐次运行算法,取"最好结果"。要注意的是,这个"最好结果"并非SSE的算法指标,而是要根据具体应用来确定。这是因为,当k值增大时,一般来说,每个簇内的平均样本数会减少,各簇更加紧密,SSE值将会减少。当k值增加到与样本数量相同时,SSE将减少为0,但此时并没有意义。

不考虑应用场景,就算法本身的一些评价指标而言,人们提出了一些通过"拐点"确定k值的方法。如图3-6所示,横坐标是k值,纵坐标为SSE值。SSE值在k小于4时下降显著;而在大于4时,下降缓慢。因此,认为在分簇数为4时,簇结构已经相对稳定,于是确定k值为4。

① 指《机器学习(Python+sklearn+TensorFlow 2.0)微课视频版》,后同。

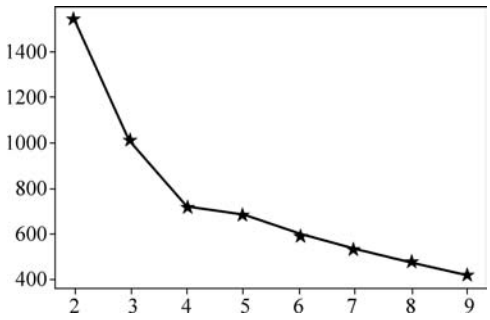


图 3-6　通过拐点确定 k 值

2）特征归一化

k-means 算法对样本不同特征的分布范围非常敏感。例如，在样本的特征数量为 2 时，第 0 个特征的变化范围是[0,1]，第 1 个特征的变化范围是[0,1000]，如果两个特征发生相同比例的变化，那么在计算欧氏距离时，显然第 1 个特征带来的影响要远远大于第 0 个特征带来的影响。如果以厘米(cm)为单位来测量人的身高，以克(g)为单位测量人的体重，每个人表示为一个二维向量。已知小明(160,60000)，小王(160,59000)，小李(170,60000)。根据常识可以知道小明和小王体型相似；但是如果根据欧氏距离来判断，小明和小王的“距离”要远远大于小明和小李之间的“距离”，即小明和小李体型相似。这是因为不同特征的度量标准之间存在差异而导致判断出错。

为了使不同变化范围的特征能起到相同的影响力，可以对特征进行归一化(Standardize)的预处理，使之变化范围保持一致。常用的归一化处理方法是将取值范围内的值线性缩放到[0,1]或[−1,1]。对第 j 个特征 $x^{(j)}$ 来说，如果它的最大值和最小值分别是 $\max x^{(j)}$ 和 $\min x^{(j)}$，则对于某值 $x_i^{(j)}$ 来说，其[0,1]归一化结果为：

$$\text{Standard}(x_i^{(j)}) = \frac{x_i^{(j)} - \min x^{(j)}}{\max x^{(j)} - \min x^{(j)}} \tag{3-5}$$

实现式(3-5)的代码并不复杂，推荐直接调用 sklearn. preprocessing. MinMaxScaler 类来实现，示例代码及运行结果见代码 3-4。

代码 3-4　特征归一化示例(Standardize. ipynb)

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np
#对数据进行归一化
X = np.array([[ 0., 1000.],
              [ 0.5, 1500.],
              [ 1., 2000.]])
min_max_scaler = MinMaxScaler()
X_minmax = min_max_scaler.fit_transform(X)
X_minmax
>>>array([[0. , 0. ],
          [0.5, 0.5],
          [1. , 1. ]])
```

```
13. # 将相同的缩放应用到其他数据
14. X_test = np.array([[ 0.8, 1800.]])
15. X_test_minmax = min_max_scaler.transform(X_test)
16. X_test_minmax
17. >>> array([[0.8, 0.8]])
18. # 缩放因子
19. min_max_scaler.scale_
20. >>> array([1. , 0.001])
21. # 最小值
22. min_max_scaler.min_
23. >>> array([ 0., -1.])
```

sklearn 的 preprocessing 模块提供了一些通用的对原始数据进行特征处理的工具。第 7 行实例化该模块的 MinMaxScaler 类创建它的一个对象 min_max_scaler。第 8 行调用它的方法 fit_transform 来实现输入特征数据的归一化。

3.2 聚类算法基础

前文对 k-means 算法的讨论初步介绍了聚类算法，本节进一步介绍聚类的基础知识。

3.2.1 聚类任务

聚类是将样本集划分为若干个子集，每个子集称为“簇”，同簇内的样本具有某些相同的特点。具体来讲，聚类任务分为分簇过程和分配过程，如图 3-7 所示。

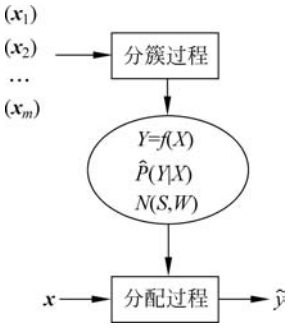


图 3-7 聚类任务的模型

设样本集 $\boldsymbol{S}=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_m\}$ 包含 m 个未标记样本，样本 $\boldsymbol{x}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})$ 是一个 n 维特征向量。

聚类在分簇过程的任务是建立簇结构，即要将 $\boldsymbol{S}$ 划分为 k（有的聚类算法将 k 作为需事先指定的超参数，有的聚类算法可以自动确定 k 的值）个不相交的簇 $\boldsymbol{C}_1,\boldsymbol{C}_2,\cdots,\boldsymbol{C}_k,\boldsymbol{C}_l\cap\boldsymbol{C}_{l'}=\varnothing$ 且 $\bigcup_{l=1}^{k}\boldsymbol{C}_l=\boldsymbol{S}$，其中 $1\leqslant l,l'\leqslant k,l\neq l'$。记簇 $\boldsymbol{C}_l$ 的标签为 y_l，簇标签共有 k 个，且互不相同。

记测试样本为 $\boldsymbol{x}=(x^{(1)},x^{(2)},\cdots,x^{(n)})$。聚类在分配阶段的任务是根据簇结构将测试样本 $\boldsymbol{x}$ 分配到一个合适的簇（簇标签为 $\hat{y}$）中。

可以从决策函数、概率和神经网络三类模型来描述分簇过程和分配过程。

在分簇过程，决策函数聚类模型要建立起合适的从样本到簇标签的映射函数 $Y=f(X)$，X 是定义域，它是所有样本特征向量的集合；Y 是值域，它是所有簇标签的集合（在聚类算法里，簇标签没有实际含义，一般只是算法自动产生的簇的编号）。概率聚类模型要建立起正确的条件概率 $\hat{P}(Y|X)$。神经网络聚类模型要利用一定的网络结构 N，生

成能够反映分簇结构的网络参数 W，即得到合适的网络模型 $N(S,W)$。

在分配过程，决策函数聚类模型依据决策函数 $Y=f(X)$ 给予测试样本 $\boldsymbol{x}$ 一个簇标签 $\hat{y}$；概率聚类模型依据条件概率 $\hat{P}(Y|X)$ 计算在给定 $\boldsymbol{x}$ 时取每一个 $\hat{y}$ 的条件概率值，取最大值对应的 $\hat{y}$ 作为输出；神经网络聚类模型将 $\boldsymbol{x}$ 馈入已经训练好的网络 $N(S,W)$，从输出得到标签 $\hat{y}$。

决策函数聚类模型有 k-means、DBSCAN、OPTICS、Mean Shift 等。概率聚类模型有高斯聚类模型等。用于聚类的神经网络有自组织特征映射（Self-Organizing Feature Map，SOFM）网络，因为并不常用，因此本书不对其进行讨论，感兴趣的读者可参考原版书。

聚类不仅可以是单独的任务，也可以对数据进行预处理，作为其他机器学习任务的前驱任务。

3.2.2　聚类算法评价指标


视频讲解

聚类算法的评价有两类指标：外部指标和内部指标。

外部指标是根据参照物给出的指标，这个参照物是预先给出的样本分组，也就是说外部指标是拿分簇算法运行的结果去跟预先确定的分组情况进行比较，目标是衡量分簇结果与预先分组情况的差异。预先知道样本分组的情况很少见，此处不过多讨论外部指标，如有需要可参考原版书的相关内容。

内部指标关注分簇后的内部结构，目标是衡量簇内结构是否紧密、簇间距离是否拉开等。内部指标应用广泛，是用来评估聚类算法是否合适的常用标准。

先讨论三个常见的内部指标。

设样本集为 $\boldsymbol{S}=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_m\}$。若某聚类算法给出的分簇为 $\boldsymbol{C}=\{\boldsymbol{C}_1,\boldsymbol{C}_2,\cdots,\boldsymbol{C}_k\}$，定义：

（1）样本 $\boldsymbol{x}_m$ 与同簇 $\boldsymbol{C}_i$ 其他样本的平均距离：

$$a(\boldsymbol{x}_m)=\frac{\sum\limits_{1\leqslant n\leqslant|C_i|}\operatorname{dist}(\boldsymbol{x}_m,\boldsymbol{x}_n)}{|\boldsymbol{C}_i|-1},\quad \boldsymbol{x}_m,\boldsymbol{x}_n\in\boldsymbol{C}_i \tag{3-6}$$

该距离也称为 $\boldsymbol{x}_m$ 的簇内平均不相似度（Average Dissimilarity）[2]。

（2）样本 $\boldsymbol{x}_m$ 与不同簇 $\boldsymbol{C}_j$ 内样本的平均距离：

$$d(\boldsymbol{x}_m,\boldsymbol{C}_j)=\frac{\sum\limits_{1\leqslant n\leqslant|\boldsymbol{C}_j|}\operatorname{dist}(\boldsymbol{x}_m,\boldsymbol{x}_n)}{|\boldsymbol{C}_j|},\quad x_m\notin\boldsymbol{C}_j,x_n\in\boldsymbol{C}_j \tag{3-7}$$

该距离也称为 $\boldsymbol{x}_m$ 与簇 $\boldsymbol{C}_j$ 的平均不相似度。

（3）样本 $\boldsymbol{x}_m$ 与簇的最小平均距离：

$$b(\boldsymbol{x}_m)=\min_{\boldsymbol{C}_j}d(\boldsymbol{x}_m,\boldsymbol{C}_j),\quad \boldsymbol{x}_m\in\boldsymbol{C}_i,\boldsymbol{C}_j\neq\boldsymbol{C}_i \tag{3-8}$$

该距离取 $\boldsymbol{x}_m$ 与所有其他不同簇的平均距离中的最小值。

(4) 簇内样本平均距离：

$$\mathrm{avg}(\boldsymbol{C}_i)=\frac{\sum\limits_{1\leqslant m<n\leqslant|\boldsymbol{C}_i|}\mathrm{dist}(\boldsymbol{x}_m,\boldsymbol{x}_n)}{\binom{|\boldsymbol{C}_i|}{2}}=\frac{2}{|\boldsymbol{C}_i|(|\boldsymbol{C}_i|-1)}\sum_{1\leqslant m<n\leqslant|\boldsymbol{C}_i|}\mathrm{dist}(\boldsymbol{x}_m,\boldsymbol{x}_n) \tag{3-9}$$

(5) 簇中心距离：

$$d_{\mathrm{cen}}(\boldsymbol{C}_i,\boldsymbol{C}_j)=\mathrm{dist}(\boldsymbol{u}_i,\boldsymbol{u}_j) \tag{3-10}$$

其中，$\boldsymbol{u}_i$ 和 $\boldsymbol{u}_j$ 是 $\boldsymbol{C}_i$ 和 $\boldsymbol{C}_j$ 的中心。

基于式(3-6)～式(3-10)的距离，可定义以下两个常用的较容易理解的聚类算法的内部评价指标。

1. 轮廓系数(Silhouette Coefficient，SC)

单一样本 $\boldsymbol{x}_m$ 的轮廓系数为：

$$s(\boldsymbol{x}_m)=\frac{b(\boldsymbol{x}_m)-a(\boldsymbol{x}_m)}{\max\{a(\boldsymbol{x}_m),b(\boldsymbol{x}_m)\}} \tag{3-11}$$

一般使用的轮廓系数是对所有样本的轮廓系数取均值。SC 值高表示簇内密集、簇间疏散。该指标在 slkearn. metrics 包中有实现，函数原型为：silhouette_score()。

2. DB 指数(Davies-Bouldin Index，DBI)

$$R_{ij}=\frac{\mathrm{avg}(\boldsymbol{C}_i)+\mathrm{avg}(\boldsymbol{C}_j)}{d_{\mathrm{cen}}(\boldsymbol{C}_i,\boldsymbol{C}_j)} \tag{3-12}$$

$$\mathrm{DBI}=\frac{1}{k}\sum_{i=1}^{k}\max_{j\neq i}R_{ij} \tag{3-13}$$

式(3-12)的分子是两个簇内样本平均距离之和，分母是两簇的中心距离。该指数越小说明簇内样本点越紧密，簇的间隔越远。该指标在 slkearn. metrics 包中也有实现，函数原型为：davies_bouldin_score()。

在代码 3-2 所示的 k-means 示例中，每次循环除了优化目标 SSE 值，还计算并输出了轮廓系数和 DBI，如第 58～59 行代码。从输出可以看到，SC 和 DBI 的值并不都是随着迭代次数的增加而改善。

在 slkearn. metrics 模块中还提供了一个称为 CH 系数的内部评价指标，它是用簇内样本和簇间样本的协方差来评估分簇效果，它的值越高说明簇内样本点越紧密、簇间隔越大，它的函数原型为：sklearn. metrics. calinski_harabasz_score()。

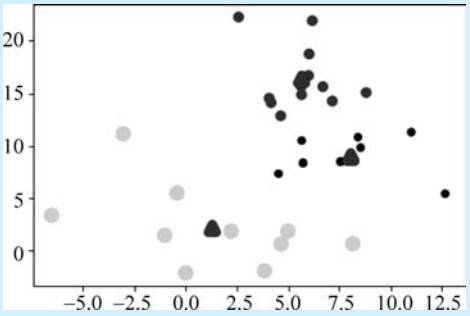
下面的示例(见代码 3-5)用代码 3-2 所用的数据集来讨论内部评价指标 SSE、SC、DBI 和 CH，以及 KMeans 类实例化的 init 参数和 n_init 参数。在示例中，分别通过设置 init 参数为 k-means++和“随机”两种初始簇中心方式，设置 n_init 参数控制重复运行次

数，进行多次试验，并以运行时间和 SSE、SC、DBI、CH 指标来分析结论。

代码 3-5　SC、DBI 和 CH 评价指标示例（聚类算法内部评价指标示例.ipynb）

```
import numpy as np
from time import time
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.cluster import KMeans

#np.random.seed(719)                          # 指定随机数种子，确保每次运行可重复观察

samples = np.loadtxt("kmeansSamples.txt") # 加载数据集

print(54 * '_')
print('init\t\ttime\tinertia\tSC\tDB\tCH') # 打印表头

n_init = 1                                   # 指定 kmeans 算法重复运行的次数

estimator = KMeans(init = 'k-means++', n_clusters = 3, n_init = n_init)
                                             # 以 k-means++方式指定初始簇中心
t0 = time()                                  # 开始计时
estimator.fit(samples)
print('%-9s\t%.2fs\t%i\t%.3f\t%.3f\t%.3f'
        % ('k-means++', (time() - t0), estimator.inertia_,
            metrics.silhouette_score(samples, estimator.labels_, metric = 'euclidean'),
            metrics.davies_bouldin_score(samples, estimator.labels_),
            metrics.calinski_harabasz_score(samples, estimator.labels_)))

estimator = KMeans(init = 'random', n_clusters = 3, n_init = n_init)
                                             # 以随机方式指定初始簇中心
t0 = time()
estimator.fit(samples)
print('%-9s\t%.2fs\t%i\t%.3f\t%.3f\t%.3f'
        % ('random', (time() - t0), estimator.inertia_,
            metrics.silhouette_score(samples, estimator.labels_, metric = 'euclidean'),
            metrics.davies_bouldin_score(samples, estimator.labels_),
            metrics.calinski_harabasz_score(samples, estimator.labels_)))

plt.scatter(samples[:,0],samples[:,1],c = estimator.labels_,linewidths = np.power
(estimator.labels_ + 0.5, 2))                # 用不同大小的点来表示不同簇的点
plt.scatter(estimator.cluster_centers_[:,0],estimator.cluster_centers_[:,1],c = 'r',
marker = '^',linewidths = 7)                 # 打印簇中心
plt.show()
>>>
```

```
38. init        time       inertia       SC       DBI       CH
39. k-means++0.00s         487           0.421    0.769     35.507
40. random      0.01s      500           0.421    0.774     34.284
```




```
41.
```

第 20 行和第 29 行中的 estimator.inertia_为 KMeans 计算得到的 SSE 值。

经过多次试验,可以发现,k-means++初始簇中心的方式一般要优于随机方式,各评价指标要好一些,但运行时间一般略长一些。也就是说,k-means++初始簇中心的方式在避免陷入局部极值的问题上有一定作用。

因为数据量较少,如果设置为多次运行,两种初始簇中心的方式一般都能使算法找到最优解。

SC、DBI 和 CH 评价指标是经常讨论的聚类算法内部评价指标,实际上,它们只适合应用于凸簇,而不适合用来衡量非凸簇的聚集程度。

凸集和非凸集的示意如图 3-8 所示。

在欧氏空间中,凸集在直观上就是一个向四周凸起的图形。在一维空间中,凸集是一个点,或者一条连续的非曲线(线段、射线和直线);在二维空间中,就是上凸的图形,如锥形扇面、圆、椭圆、凸多边形等;在三维空间中,凸集可以是一个实心的球体等。总之,凸集就是由向周边凸起的点构成的集合。

簇的成员的集合为凸集的簇,称为凸簇。以图 3-9 所示的非凸簇来简要讨论下 SC 和 DBI 的适用性。

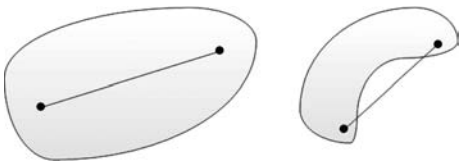


图 3-8　凸集和非凸集的示意

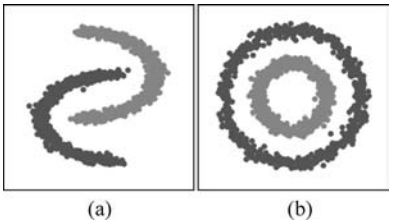


图 3-9　非凸簇示例①

① 图片来自:https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html

对于图 3-9 所示的非凸簇，同簇内的样本间的距离很大，也就是说，簇内平均不相似度 $a(\boldsymbol{x}_m)$可能比 $b(\boldsymbol{x}_m)$还大，所以，由式(3-11)定义的 SC 甚至可能取负值。

式(3-12)定义了 DBI，该式的分母为两个簇中心的距离，对图 3-9(b)的两个圆环簇来说，它们的中心可能很近，并不能真实反映簇的间距。

下面讨论一个适合用来对非凸簇进行评价的指标。

实际上，衡量聚类效果的有两个因素，分别是簇内密集程度和簇间隔程度。由于非凸簇的分布特点导致簇内密集程度不再适合用簇内所有样本间的距离来衡量，簇间隔程度也不再适合用簇中心的距离来衡量。

(1) 定义众距离 Z 来衡量簇内密集程度。记 MinPts_distance($\boldsymbol{x}_m$)为样本点 $\boldsymbol{x}_m$ 到它的第 MinPts 近邻居样本点的距离，则簇 $\boldsymbol{C}_i$ 的众距离 Z_i 为：

$$Z_i=\frac{\sum \text{MinPts\_distance}(\boldsymbol{x}_m)}{|\boldsymbol{C}_i|},\quad \boldsymbol{x}_m\in\boldsymbol{C}_i \tag{3-14}$$

MinPts 可以根据样本密集程度取 1,2,3 等值。

(2) 定义群距离 Q 来衡量簇的间隔程度。下面给出两种群距离的定义。

① 两个簇的群距离 Q 是它们的样本点之间的距离的最小值：

$$Q(\boldsymbol{C}_i,\boldsymbol{C}_j)=\min_{\boldsymbol{x}_m\in\boldsymbol{C}_i,\boldsymbol{x}_n\in\boldsymbol{C}_j,\boldsymbol{C}_i\neq\boldsymbol{C}_j} d(\boldsymbol{x}_m,\boldsymbol{x}_n) \tag{3-15}$$

② 也可以用点到簇的距离来定义群距离，记样本 $\boldsymbol{x}_m$ 到不同簇 $\boldsymbol{C}_j$ 的距离为：

$$q(\boldsymbol{x}_m,\boldsymbol{C}_j)=\min_{\boldsymbol{x}_n\in\boldsymbol{C}_j} d(\boldsymbol{x}_m,\boldsymbol{x}_n),\quad \boldsymbol{x}_m\in\boldsymbol{C}_i,\boldsymbol{C}_j\neq\boldsymbol{C}_i \tag{3-16}$$

即样本点到不同簇内点的最小值。簇 $\boldsymbol{C}_i$ 到 $\boldsymbol{C}_j$ 的群距离 $Q(\boldsymbol{C}_i,\boldsymbol{C}_j)$为它们的均值：

$$Q(\boldsymbol{C}_i,\boldsymbol{C}_j)=\frac{1}{|\boldsymbol{C}_i|}\sum_{\boldsymbol{x}_m\in\boldsymbol{C}_i} q(\boldsymbol{x}_m,\boldsymbol{C}_j) \tag{3-17}$$

要注意的是，在此定义下，簇之间的群距离并不都是对称的，即 $Q(\boldsymbol{C}_i,\boldsymbol{C}_j)\neq Q(\boldsymbol{C}_j,\boldsymbol{C}_i)$。

把所有簇的众距离的均值除以所有簇间群距离的均值的结果作为评价聚类效果的指标，称为 ZQ 系数：

$$\text{ZQ}=\frac{\frac{1}{k}\sum_{i=1}^{k}Z_i}{\frac{1}{k(k-1)}\sum_{i\neq j}Q(\boldsymbol{C}_i,\boldsymbol{C}_j)} \tag{3-18}$$

ZQ 系数小，表示簇内密集、簇间疏散。

实现 ZQ 系数的代码见代码 3-6，其中，MinPts 取值 1，群距离采用式(3-15)的定义。代码中的向量平方、开方、求和、求最小值、求均值等计算采用 NumPy 模块中的 square、sqrt、sum、min、mean 等函数来完成。第 32 行和第 39 行的 np.inf 表示无穷大值。

代码 3-6　ZQ 系数(zqscore.py)

```
1. import numpy as np
2.
3. def ZQ_score(X, labels):
4.     '''
5.     计算 ZQ 系数。
```

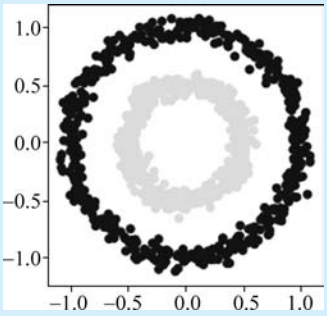
```
    para X: 数组形式的样本点集,每一行是一个样本点。
    para labels: 数组形式的测试标签集。
    retrurn: ZQ 系数。
    '''
    n_samples = len(X)                          # 标本总数
    label = list(set(labels))                   # 标签列表
    n_labels = len(label)                       # 标签数

    # 把样本及标签分簇存放
    X_i = []
    y_i = []
    for i in label:
        X_i.append([])
        y_i.append([])

    for i in range(n_samples):
        j = label.index(labels[i])              # 该样本在 label 标签列表中的下标
        X_i[j].append(X[i])
        y_i[j].append(labels[i])

    # 计算簇内众 Z 距离
    Z_dist = np.zeros(shape = (n_labels))  # 存放每个簇的 Z 距离
    for i in range(n_labels):
        n_cluster = len(X_i[i])
        sample_z_dist = []                      # 用来记录簇内每个样本的最近邻距离
        for j in range(n_cluster):
            min_dist = np.inf
            for k in range(n_cluster):
                if j == k:
                    continue
                dist = np.sqrt(np.sum(np.square(X_i[i][j] - X_i[i][k])))  # 两个
                # 样本间的欧氏距离
                if dist < min_dist:
                    min_dist = dist
            if min_dist == np.inf:
                sample_z_dist.append(0)     # 簇内只有一个元素时
            else:
                sample_z_dist.append(min_dist)
        Z_dist[i] = np.mean(sample_z_dist)

    # 计算簇间群 Q 距离
    Q_dist = np.zeros(shape = (n_labels, n_labels))  # 二维数组,用来存放簇之间的
Q 距离
    for i in range(n_labels):
        for j in range(n_labels):
            if i == j:
                continue
            i2j_min_dist = []      # 用来记录第 i 个簇内样本点到第 j 个簇的最小距离
            for sample1 in X_i[i]:
```

```
                min_dist = np.inf
                for sample2 in X_i[j]:
                    dist = np.sqrt(np.sum(np.square(sample1 - sample2)))  # 两个
# 样本间的欧氏距离
                    if dist < min_dist:
                        min_dist = dist
                if min_dist < np.inf:
                    i2j_min_dist.append(min_dist)
            Q_dist[i,j] = np.min(i2j_min_dist)  # 群距离是样本点之间距离的最小值
            # Q_dist[i,j] = np.min(i2j_min_dist)  # 群距离用点到簇的距离来定义

    return np.mean(Z_dist) / ( np.sum(Q_dist) / ( n_labels * (n_labels - 1) ) )
```

下面示例 ZQ 系数的应用，见代码 3-7。

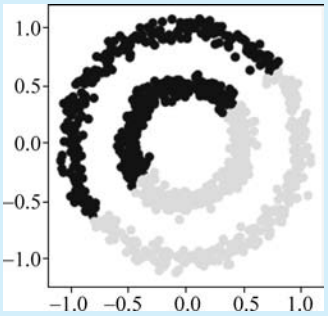
代码 3-7　ZQ 评价指标示例（聚类算法内部评价指标示例.ipynb）

```
from zqscore import ZQ_score
from sklearn.datasets import make_circles
noisy_circles = make_circles(n_samples=1000, factor=.5, noise=.05, random_state=
15)  # 生成圆环型的实验数据
X = noisy_circles[0]
plt.axes(aspect='equal')
plt.scatter(X[:, 0], X[:, 1], marker='o', c=noisy_circles[1])
plt.show()
print("SC:\t" + str(metrics.silhouette_score(X, noisy_circles[1], metric='euclidean')))
print("DBI:\t" + str(metrics.davies_bouldin_score(X, noisy_circles[1])))
print("CH:\t" + str(metrics.calinski_harabasz_score(X, noisy_circles[1])))
print("ZQ:\t" + str(ZQ_score(X, noisy_circles[1])))
```




```
>>>
>>> SC:0.1132345379746796
>>> DBI:300.6033581127579
>>> CH:0.009910351299218752
>>> ZQ:0.09342900522589306

clus = KMeans(n_clusters=2, random_state=0).fit(X)
plt.axes(aspect='equal')
```

```
20. plt.scatter(X[:, 0], X[:, 1], marker = 'o', c = clus.labels_)
21. plt.show()
22. print("SC:\t" + str(metrics.silhouette_score(X, clus.labels_, metric = 'euclidean')))
23. print("DBI:\t" + str(metrics.davies_bouldin_score(X, clus.labels_)))
24. print("CH:\t" + str(metrics.calinski_harabasz_score(X, clus.labels_)))
25. print("ZQ:\t" + str(ZQ_score(X, clus.labels_)))
```




```
26. >>>
27. >>> SC:0.35418150800364173
28. >>> DBI:1.1828798822247686
29. >>> CH:576.2343653533658
30. >>> ZQ:1.1687170232174913
```

第1行从同目录下的zqscore.py文件中引入计算ZQ系数的ZQ_score函数。第3行产生二维平面上的圆环型的实验数据。sklearn扩展库的datasets模块提供了很多加载既有实验数据和产生新实验数据的函数[①]。make_circles()函数产生的实验数据包括样本点以及它们的簇标签(图形如第12行的输出所示)。

该示例先用实验数据及其原始簇标签来计算SC、DBI、CH和ZQ四个指标(第13行到第16行),然后用k-means算法对它们进行簇数为2的聚类(第18行),用得到的结果(图形如第26行输出所示)来计算上述四个指标(第27行到第30行),最后进行对比分析。

一般认为,将整个圆环上的样本点作为一个簇,比从半环处切开成簇更加合理,ZQ系数较好地反映了这一观点,而其他三个指标的评价结果与此相反。

对聚类算法的评价需要综合考量,涉及因素比较多,尤其是在数据量大、特征维数多的情况下,不能仅依靠单一指标就给出结论。

聚类算法评价指标还有助于探索样本数据在空间中的分布,帮助选择合适的聚类算法。


视频讲解

3.3 PCA降维算法

为了方便演示,本书中示例的数据量一般都非常小。实际生产中的数据量往往非常大,有的样本的特征数量甚至达到了上万维,可能带来维数灾难(Curse of Dimensionality)问

① https://scikit-learn.org/stable/datasets.html

题。维数灾难是指在涉及向量计算的问题中，当维数增加时，空间的体积增长得很快，使得可用的数据在空间中的分布变得稀疏，向量的计算量呈指数增长的一种现象。维数灾难涉及数值分析、抽样、组合、机器学习、数据挖掘和数据库等诸多领域。

降维不仅可以减少样本的特征数量，还可以用来解决特征冗余（不同特征有高度相关性）等其他数据预处理问题。可视化并探索高维数据集也是它的一个重要应用，这对于后文将要讨论的不同聚类算法的适用性问题十分重要。

降维算法是专门用于降维的算法，可以分为线性和非线性的。线性的降维算法是基于线性变换来降维，主要有奇异值分解（Singular Value Decomposition，SVD）、主成分分析（Principal Components Analysis，PCA）等算法。主成分分析是最常用的降维算法，本节先简要讨论它的含义，然后示例它的应用。对原理感兴趣的读者，可参考原版书。

顾名思义，主成分分析是指找出主要成分来代替原有数据。用二维平面上的例子来简要说明其过程，如图 3-10 所示。在二维平面上有 $\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{x}_3,\boldsymbol{x}_4$ 四个点，坐标分别是(4,2)、(0,2)、(−2,0) 和(−2,−4)，它们满足中心化要求，即 $\sum_{i=1}^{4}\boldsymbol{x}_i=0$。对于不满足中心化要求的点，可通过减所有点的均值来满足该要求。

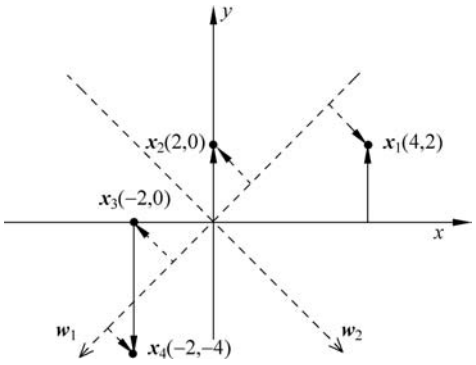


图 3-10　二维平面上的主成分降维示例

现在要将这四个点从二维降为一维，怎么降呢？一个很自然的想法是直接去掉每个点的一个坐标。比如，去掉 y 轴上的坐标，只保留 x 轴上的坐标。这实际上是用各点在 x 轴上的投影来代替原来的点，带来的误差为它们在 y 轴上的投影向量，如图中从 x 轴指向各点的带箭头实线所示，其中 $\boldsymbol{x}_3$ 点在 x 轴上，因此没有误差向量。

降维必定会带来误差，如何使总体误差最小是降维算法追求的目标。用所有误差向量的模的平方之和作为损失函数来衡量降维带来的误差（类似于误差平方和损失函数 SSE）。

试着同步旋转 x 和 y 轴，使得去掉 y 轴上的坐标带来的损失函数最小。比如 x 和 y 坐标轴保持正交旋转到图中的 w_1 和 w_2 坐标轴，降维的结果是只保留各点在 w_1 上的投影，放弃在 w_2 上的投影，所带来的误差向量如图中带箭头的虚线所示。

此例从二维降到一维，即用点到线的投影来代替平面上的点。如果在三维立体空间中，可将空间中的点投影到一个平面上或者一条线上。进一步推广，可以将多维空间中的点投影到一个低维的超平面上。

在 sklearn 扩展库的 decomposition 模块实现了 PCA 算法。先用它来印证上述分析过程,见代码 3-8。

代码 3-8　二维平面上的主成分降维示例(二维平面上的主成分降维示例.ipynb)

```
x = [[4,2], [0,2], [-2,0], [-2,-4]]                    # 平面上四个点的坐标

from sklearn.decomposition import PCA

pca = PCA(n_components=2)                              # 只旋转,不降维
pca.fit(x)
print("新的轴向量:")
print(pca.components_)
print("各维度投影方差占比分布:")
print(pca.explained_variance_ratio_)
print("各点在新轴上的投影:")
print(pca.transform(x))
>>>
新的轴向量:
[[-0.70710678 -0.70710678]
 [ 0.70710678 -0.70710678]]
各维度投影方差占比分布:
[0.83333333 0.16666667]
各点在新轴上的投影:
[[-4.24264069 1.41421356]
 [-1.41421356 -1.41421356]
 [ 1.41421356 -1.41421356]
 [ 4.24264069 1.41421356]]

pca = PCA(n_components=1)                              # 降到一维
pca.fit(x)
print("新的轴向量:")
print(pca.components_)
print("各维度投影方差占比分布:")
print(pca.explained_variance_ratio_)
print("各点在新轴上的投影:")
print(pca.transform(x))
>>>
新的轴向量:
[[-0.70710678 -0.70710678]]
各维度投影方差占比分布:
[0.83333333]
各点在新轴上的投影:
[[-4.24264069]
 [-1.41421356]
 [ 1.41421356]
 [ 4.24264069]]
```

第 5 行实例化 PCA 类,当把参数 n_components 设为与原特征数相同时,它只旋转,

不降维。

第 6 行用四个点的坐标来训练算法。

第 8 行通过属性 components_输出旋转后的轴向量，如第 15、16 行所示，第一个轴向量可以写成分数形式$\left(-\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}}\right)$，可见它的方向与 x，y 轴夹角为 45 度。

第 12 行用 transform 方法对四个点计算新的投影，如第 20 行到第 23 行所示。读者可以用几何知识验证一下。

第 10 行通过属性 explained_variance_ratio_观察各维度投影方差的占比分布，可以理解为各维度的成分比例，它是按从大到小的排列输出，属性 components_输出的新轴向量排列顺序要与它对应。输出如第 18 行所示。

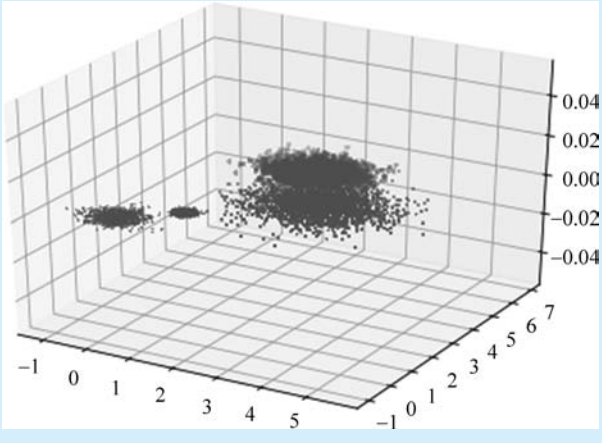
第 25 行将参数 n_components 设为 1，即降到一维。对比第 14 行的输出和第 34 行的输出，可以看到，它将第二个轴上的新投影直接丢弃了，保留了主要成分。

下面来看一个可视化降维高维特征的示例。为了方便对比观察，不采用过高维的数据，而只用三维的数据示例，具体流程是先生成三维空间中分布的点，然后降到二维，观察并分析其过程。

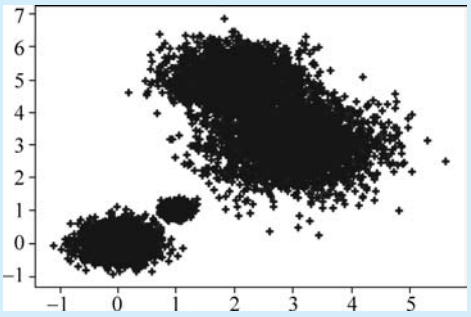
在三维空间中生成四个簇，并查看它们的分布，见代码 3-9。

代码 3-9　在三维空间中产生簇（高维数据可视化降维示例.ipynb）

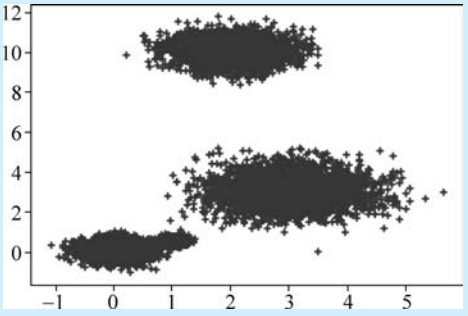
```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. from mpl_toolkits.mplot3d import Axes3D
4. from sklearn.datasets import make_blobs
5.
6. # 在三维空间中产生四个簇，共 1000 个样本
7. X, _ = make_blobs(n_samples = 10000, n_features = 3, centers = [[0,0,0], [1,1,0.5], [3,
   3,3], [2,5,10]], cluster_std = [0.3, 0.1, 0.7, 0.5])
8. fig = plt.figure()
9. ax = Axes3D(fig)
10. plt.scatter(X[:, 0], X[:, 1], X[:, 2], marker = '+')
```



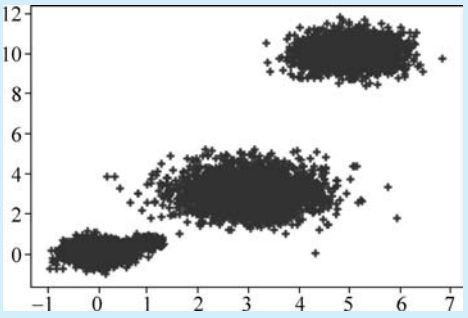
```
11. >>>
12. # 看一下它们在三个面上的投影
13. plt.scatter(X[:, 0], X[:, 1], marker = '+')
14. plt.show()
```



```
15. >>>
16. plt.scatter(X[:, 0], X[:, 2], marker = '+')
17. plt.show()
```



```
18. >>>
19. plt.scatter(X[:, 1], X[:, 2], marker = '+')
20. plt.show()
```



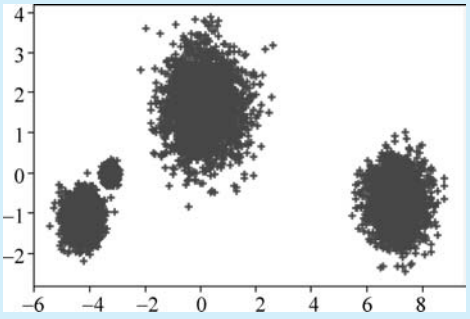
```
21. >>>
```

sklearn扩展库的 datasets 模块中的 Make_blobs()函数产生各向同性的高斯分布(即常说的正态分布)的样本数据。本例中,用它在三维空间中以指定标准差在指定的中心产生了四个簇,如第 7 行所示。第 8 行到第 10 行画出三维的分布图。然后分别看一下它们在三个面上的投影,可见每个面上的投影都有两个簇重叠的情况,不好区分。

用 PCA 对它们进行降维，共进行了三次，见代码 3-10。第一次降到一个二维的平面上，可见可以较好地分开为四个簇。第二次和第三次通过指定 n_components 为一个小数来要求降维后保留的主成分占比。第二次要求保留 90%（第 11 行），此时，降到一维就可以达到要求了。第三次要求保留 99%（第 18 行），此时，不能降低维数，否则就达不到该要求。

代码 3-10　PCA 降维三维空间中的点（高维数据可视化降维示例.ipynb）

```
1. pca = PCA(n_components = 2)
2. pca.fit(X)
3. print(pca.explained_variance_ratio_)
4. >>> [0.92755398 0.06230942]
5. X_new = pca.transform(X)
6. plt.scatter(X_new[:, 0], X_new[:, 1],marker = '+')
7. plt.show()
8. >>>
```




```
9.
10.
11. pca = PCA(n_components = 0.9)            # 指定保留的主成分占比
12. pca.fit(X)
13. print(pca.explained_variance_ratio_)
14. print("降维后的特征数: " + str(pca.n_components_))
15. >>> [0.92755398]
16. 降维后的特征数: 1
17.
18. pca = PCA(n_components = 0.99)           # 指定保留的主成分占比
19. pca.fit(X)
20. print(pca.explained_variance_ratio_)
21. print("降维后的特征数: " + str(pca.n_components_))
22. >>> [0.92755398 0.06230942 0.0101366 ]
23. 降维后的特征数: 3
```

该示例中，通过保留主要成分，将数据降至二维，可以直观地观察到数据的分布情况。在进行聚类和分类时，如果能提前观察到样本在空间的大概分布，就更容易选择合适的算法。

3.4 划分聚类、密度聚类和模型聚类算法

前文讨论了 k-means 算法的分簇思路，人们从不同的角度提出了更多的聚类算法。聚类算法可以分为不同的类别，各有不同的应用场合和优势。本节讨论划分聚类、密度聚类和模型聚类。

划分聚类、密度聚类和模型聚类是比较有代表性的三种聚类思路。

1. 划分聚类

划分(Partitioning)聚类是基于距离的，它的基本思想是使簇内的点距离尽量近、簇间的点距离尽量远。k-means 算法就属于划分聚类。划分聚类适合如图 3-8 所示的凸样本点集合的分簇。

2. 密度聚类

密度(Density)聚类是基于密度进行分簇。这里的密度是指某样本点给定邻域内的其他样本点的数量。密度聚类的思想是当邻域的密度达到指定阈值时，就将邻域内的样本点合并到本簇内，如果本簇内所有样本点的邻域密度都达不到指定阈值，则本簇划分完毕，进行下一个簇的划分。密度聚类对图 3-9 所示的非凸簇很有效，像 k-means 等基于距离划分聚类的方法则难以正确划分此类簇(如代码 3-7 所示的示例)。密度聚类还可以用来对离群点进行检测。

密度聚类的经典算法有 DBSCAN(Density-Based Spatial Clustering of Applications with Noise)和 OPTICS(Ordering Points to Identify the Clustering Structure)。

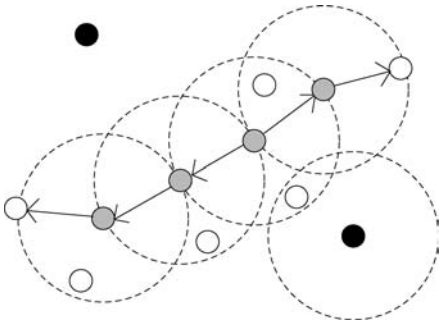


图 3-11 DBSCAN 算法中的核心点、边界点和噪声点示例

1) DBSCAN 算法

DBSCAN 算法将所有样本点分为核心点、边界点和噪声点，分别如图 3-11 中灰色点、白色点和黑色点所示。

核心点是这样的点：在指定大小的邻域内有不少于指定数量的点。指定大小的邻域，一般用邻域半径 eps 来确定。指定数量用 min_samples 来表示。图 3-11 中，用相同半径 eps 的圆来表示领域，将 min_samples 确定为 4，那么图中 4 个灰色的点为核心点。边界点是处于核心点的邻域内的非核心点，如图中的白色点所示。噪声点是邻域内没有核心点的点，如图中黑色的点。

DBSCAN 算法需要预先指定 eps 和 min_samples 两个参数，即超参数。该算法寻找一个簇的过程是先对样本点按顺序排查，如果能找到一个核心点，就从该核心点出发，找出所有直接和间接与之相邻的核心点，以及这些核心点的所有边界点，这些核心点和边界点就形成一个簇。接着，从剩下的点中再找另一个簇，直到没有核心点为止。余下的点为

噪声点。

sklearn 的 cluster 模块实现了该算法，类原型见代码 3-11。应用该算法时，一般要指定 eps 和 min_samples 两个参数(默认为 0.5 和 5)。

代码 3-11　sklearn 中的 DBSCAN 类

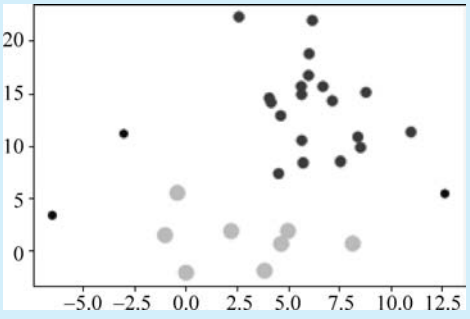
```
class sklearn.cluster.DBSCAN(eps = 0.5, min_samples = 5, metric = 'euclidean', metric_
   params = None, algorithm = 'auto', leaf_size = 30, p = None, n_jobs = None)

fit(X[, y, sample_weight])
fit_predict(X[, y, sample_weight])
get_params([deep])
set_params( ** params)
```

metric 为距离度量方法，默认为欧氏距离。fit()方法用来完成分簇。其应用方法可参考 sklearn 的 Demo[①]。

代码 3-12 示例了用 DBSCAN 对本书附属资源的文件 kmeansSamples.txt 中 30 个点的聚类。

代码 3-12　DBSCAN 应用示例(DBSCAN.ipynb)

```
from sklearn.cluster import DBSCAN
import numpy as np
samples = np.loadtxt("kmeansSamples.txt")
clustering = DBSCAN(eps = 5, min_samples = 5).fit(samples)
clustering.labels_
>>> array([ 0, 0, 0, 0, -1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, -1, 1, 1, 0, 0, 1, 0, 0, 0, 0,
   0, 1, -1, 0], dtype = int64)
import matplotlib.pyplot as plt
plt.scatter(samples[:,0],samples[:,1],c = clustering.labels_ + 1.5,linewidths = np.
   power(clustering.labels_ + 1.5, 2))
plt.show()
>>>
```




```

```

① https://scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html#sphx-glr-auto-examples-cluster-plot-dbscan-py

第 6 行是各样本点的标签值，−1 表示噪声标签。

DBSCAN 算法善于发现任意形状的稠密分布数据集，但它的结果对邻域参数 eps 和 min_samples 敏感。不像 k-means 算法只需要调整一个参数，DBSCAN 算法需要对两个参数进行联合调参，复杂度要高得多。

参数 eps 和 min_samples 分别取(2,2)、(5,4)、(6,3)时示例的聚类结果如图 3-12 所示。不同大小的圆点代表不同的簇。最小的点代表噪声点。

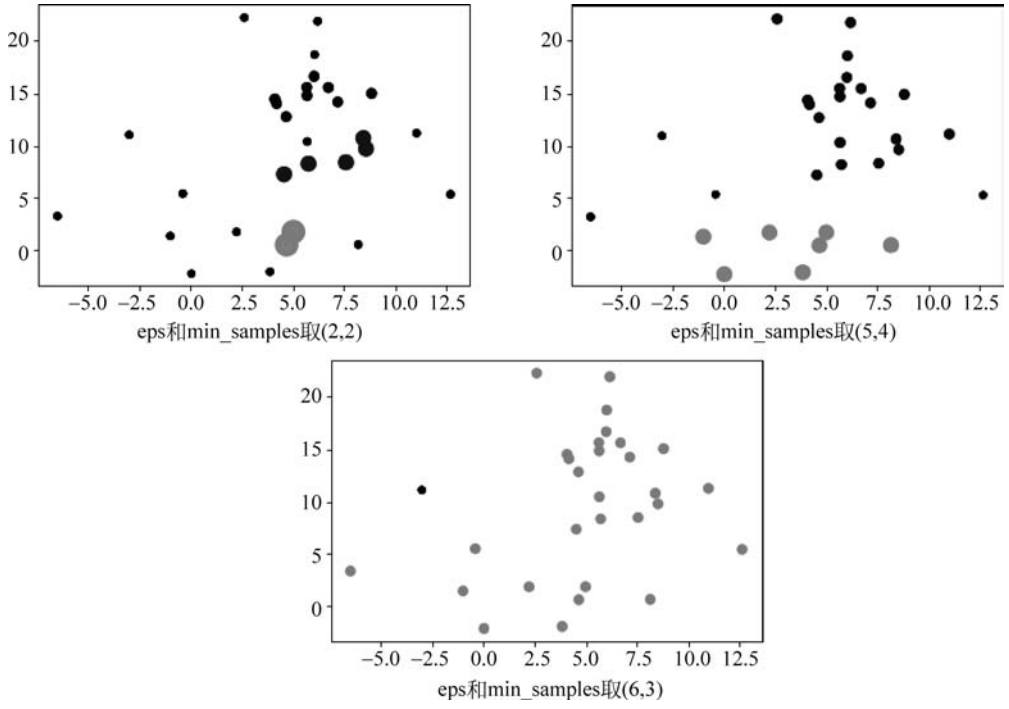


图 3-12　DBSCAN 算法示例取不同参数值时的对比图(见彩插)

在固定参数 eps 时，参数 min_samples 过大会导致核心点过少，使得一些小簇被放弃，噪声点增多；而参数 min_samples 过小会导致某些噪声进入簇中。在固定参数 min_samples 时，参数 eps 过大会将噪声点纳入簇中；过小会使核心点减少，噪声点增多。

在实际应用中，如果能确定聚类的具体评价指标，如簇数、噪声点数限制和 SC、DBI、CH 和 ZQ 等，则可以对参数 eps 和 min_samples 的合理取值依次运行 DBSCAN 算法，取最好的评价结果。如果数据量特别大，则可以将参数空间划分为若干网格，每个网格取一个代表值进行聚类。这种超参数调优方法称为网格调参，sklearn 对此方法提供了支持，有关内容将在 7.2 节中讨论。

2) OPTICS 算法

OPTICS 算法的基本思想是在 DBSCAN 算法的基础上，将每个点离最近的核心点密集区的可达距离都计算出来，然后根据预先指定的距离阈值把每个点分到与密集区对应的簇中，可达距离超过阈值的点是噪声点。点到核心点密集区的可达距离是它到该区内

所有核心点的距离的最小值。

图 3-11 中，4 个灰色的核心点组成一个核心点密集区。显然，这 4 个核心点到该核心点密集区的可达距离为 0。OPTICS 算法要计算所有点到每一个核心点密集区的可达距离，并记录最小的那个可达距离。

可达距离值示例如图 3-13 所示。可以看到，可达距离形成两个凹陷，分别对应两个簇。如果以图中的虚线距离作为距离阈值来划分，则虚线以上的点为噪声点，它们离密集区过远，而虚线以下的点聚集在两个区域，分别对应两个簇。

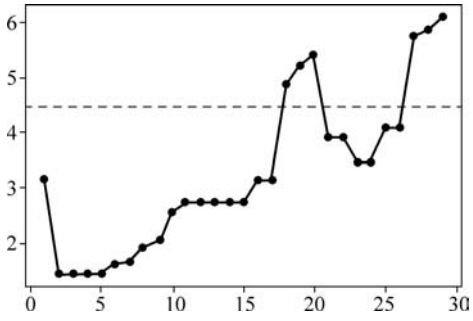


图 3-13　OPTICS 算法计算的可达距离示例

引入可达距离可以直观地看到样本点的聚集情况。OPTICS 算法巧妙地解决了确定 eps 参数值的问题。

sklearn 增加了对 OPTICS 算法的支持，同样位于 cluster 包中，见代码 3-13。

代码 3-13　sklearn 中的 OPTICS 类

```
1. class sklearn.cluster.OPTICS(min_samples = 5, max_eps = inf, metric = 'minkowski', p = 2,
   metric_params = None, cluster_method = 'xi', eps = None, xi = 0.05, predecessor_correction
   = True, min_cluster_size = None, algorithm = 'auto', leaf_size = 30, n_jobs = None)
2.
3. fit(X, y = None)
4. fit_predict(X, y = None)
5. get_params(deep = True)
6. set_params( ** params)
```

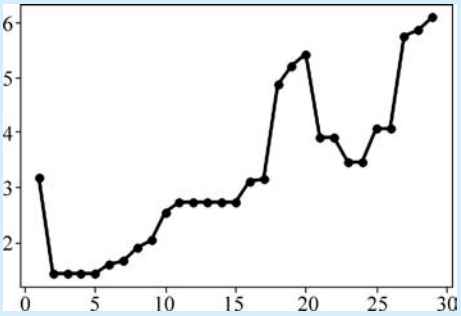
其中，max_eps 参数为邻域半径，min_samples 参数为核心点最小邻域点数，eps 参数为分簇的距离阈值（图 3-13 中的虚线代表的距离）。应用该算法时，一般将邻域半径 max_eps 参数默认为无穷大值，这样可以将所有点都包含在计算范围内。

对文件 kmeansSamples.txt 中 30 个点用 OPTICS 算法进行聚类分析，见代码 3-14。

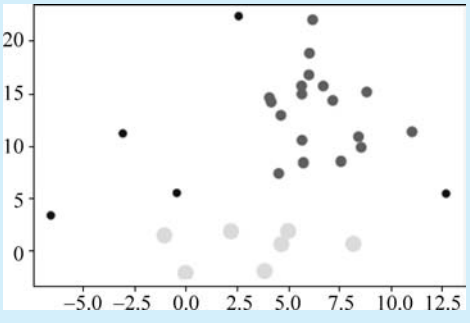
代码 3-14　OPTICS 算法应用示例（OPTICS.ipynb）

```
1. from sklearn.cluster import OPTICS, cluster_optics_dbscan
2. import matplotlib.pyplot as plt
3. import numpy as np
4. samples = np.loadtxt("kmeansSamples.txt")
```

```
clust = OPTICS(max_eps = np.inf, min_samples = 5, cluster_method = 'dbscan', eps = 4)
clust.fit(samples)
clust.ordering_
>>> array([ 0, 13, 14, 15, 20, 29, 7, 5, 23, 3, 2, 6, 12, 22, 24, 25, 1, 26, 19, 21, 17, 8,
9, 10, 18, 11, 27, 28, 4, 16])
reachability = clust.reachability_[clust.ordering_]
Reachability
>>> array([ inf, 3.17458968, 1.42768959, 1.42768959, 1.42768959,
        1.42768959, 1.59655377, 1.65018931, 1.89652558, 2.03045666,
        2.54510242, 2.72758242, 2.72758242, 2.72758242, 2.72758242,
        2.72758242, 3.11074555, 3.14659536, 4.86176447, 5.2144061 ,
        5.42638897, 3.90666353, 3.90666353, 3.45290884, 3.45290884,
        4.06306139, 4.06306139, 5.75576757, 5.86039336, 6.09507337])
labels = clust.labels_[clust.ordering_]
labels
>>> array([ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 1, 1, 1, 1, 1,
-1, -1, -1, -1, -1])
plt.plot(list(range(1, 31)), reachability, marker = '.', markeredgewidth = 3, linestyle = '-')
plt.show()
```




```
>>>
clust.labels_
>>> array([ 0, 0, 0, 0, -1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, -1, 1, 1, -1, 0, -1, 0, 0,
0, 0, 0, 1, -1, 0])
plt.scatter(samples[:,0],samples[:,1],c = clust.labels_ + 1.5,linewidths = np.power
(clust.labels_ + 1.5, 2))
plt.show()
```




```
>>>
```

第 5 行配置了初始参数，读者可以设置其他值来看看聚类的效果。第 7 行输出的是按先后次序排列的结果队列。第 10 行输出的是结果队列中各点的可达距离。第 18 行输出的是按可达距离进行分簇的结果，可见分为 0、1 两个簇，－1 表示噪声点。第 23 行输出的是按原始顺序排列的各点的簇号。最后第 27 行是用图显示的最终分簇结果。

3. 模型聚类

模型(Model)聚类假定每个簇符合一个分布模型，找到这个分布模型，就可以对样本点进行分簇。

在机器学习领域，这种先假定模型符合某种概率分布(或决策函数)，然后在学习过程中学习到概率分布参数(或决策函数参数)的最优值的模型，称为参数学习模型。而前文讨论的 k-means、DBSCAN 等模型，不需要在学习之前假定分布(或决策函数)的模型，称为非参数学习模型。

模型聚类主要包括概率模型和神经网络模型两大类，前者以高斯混合模型(Gaussian Mixture Models，GMM)为代表，后者以自组织映射网络(Self Organizing Map，SOM)为代表。下面简要讨论高斯混合模型的基本思想。

用多个简单的模型来拟合一个复杂的模型是常用的机器学习方法。高斯混合模型采用多个分布的混合来拟合不能用单一分布来描述的样本集。

记随机变量 X 服从含有未知变量 $\tau=(\mu,\sigma^2)$ 的高斯分布，其概率密度为：

$$f(x\mid\tau)=\frac{1}{\sqrt{2\pi}\sigma}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3-19}$$

高斯混合模型 $P(x|\boldsymbol{\theta})$ 是多个高斯分布混合的模型：

$$P(x\mid\boldsymbol{\theta})=\sum_{i=1}^{K}\alpha_i f(x\mid\tau_i) \tag{3-20}$$

其中，K 是混合的高斯分布的总数；τ_i 是第 i 个高斯分布的未知变量，记 $\boldsymbol{\tau}=(\tau_1,\tau_2,\cdots,\tau_K)$；$\alpha_i$ 是第 i 个高斯分布的混合系数，$\alpha_i>0$；$\sum\alpha_i=1$，α_i 可看作概率值，记 $\boldsymbol{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_K)$。记 $\boldsymbol{\theta}=(\boldsymbol{\alpha},\boldsymbol{\tau})$。

用于聚类任务时，高斯混合模型认为样本是由 $P(x|\boldsymbol{\theta})$ 产生的，产生的过程是先按概率 $\boldsymbol{\alpha}$ 选择一个高斯分布 $f(x|\tau_j)$，再由该高斯分布生成样本。由同一高斯分布产生的样本属于同一簇，即高斯混合模型中的高斯分布与聚类的簇一一对应。在分簇过程(图 3-7 所示)中，算法的任务是从训练集中学习到模型参数 $\boldsymbol{\theta}=(\boldsymbol{\alpha},\boldsymbol{\tau})$。在分配过程，模型计算测试样本由每个高斯分布产生的概率，取最大概率对应的高斯分布的簇作为分配的簇。

一般采用 EM(Expectation-Maximization，期望最大化)算法来学习模型，需要深入研究 EM 算法的读者可以参考原版书。

Sklearn.mixture.GaussianMixture 类实现了高斯混合模型，其原型见代码 3-15。

代码 3-15　sklearn 中的 GaussianMixture 类

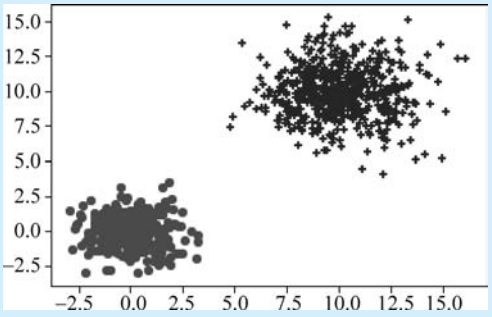
```
class sklearn.mixture.GaussianMixture(n_components = 1, *, covariance_type = 'full', tol = 0.001, reg_covar = 1e-06, max_iter = 100, n_init = 1, init_params = 'kmeans', weights_init = None, means_init = None, precisions_init = None, random_state = None, warm_start = False, verbose = 0, verbose_interval = 10)[source]

fit(X[, y])
predict(X)
```

n_components 参数是分簇的个数,即高斯分布的个数,是超参数。

下面用该实现来验证高斯混合聚类的实现过程,先用 datasets 模块中的 make_blobs 在平面上生成两个高斯分布的簇,再用 mixture 模块中的 GaussianMixture 去学习,见代码 3-16。

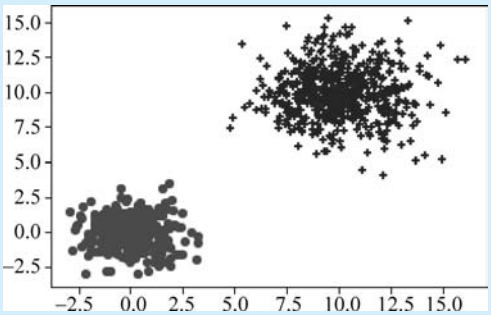
代码 3-16　高斯混合聚类验证(GaussianMixture.ipynb)

```
# 以(0,0)和(10,10)为中心,以 1.2 和 1.8 为标准差,分别生成两个簇
X1, y1 = make_blobs(n_samples = 300, n_features = 2, centers = [[0,0]], cluster_std = [1.2])
X2, y2 = make_blobs(n_samples = 600, n_features = 2, centers = [[10,10]], cluster_std = [1.8])
plt.scatter(X1[:, 0], X1[:, 1], marker = 'o', color = 'r')
plt.scatter(X2[:, 0], X2[:, 1], marker = '+', color = 'b')
plt.show()
```




```
>>>

# 合并样本点,对高斯混合模型进行训练
X = np.vstack((X1, X2))
gm = GaussianMixture(n_components = 2, random_state = 0).fit(X)
print('均值:' + str(gm.means_))
print('协方差:' + str(gm.covariances_))
>>>均值:[[10.06436709 9.92971751]
[-0.01042938 0.0122576 ]]
协方差:[[[ 3.30748342 -0.0189866 ]
[-0.0189866 3.28092042]]
[[ 1.38696931 -0.02175968]
[-0.02175968 1.26746825]]]

# 按预测结果用不同的标记显示各点
```

```
22. y_pred = gm.predict(X)
23. C1 = []
24. C2 = []
25. for i in range(len(X)):
26.     if y_pred[i] == 1:
27.         C1.append(list(X[i]))
28.     else:
29.         C2.append(list(X[i]))
30. C1 = np.array(C1)
31. C2 = np.array(C2)
32. plt.scatter(C1[:, 0], C1[:, 1], marker = 'o', color = 'r')
33. plt.scatter(C2[:, 0], C2[:, 1], marker = '+', color = 'b')
34. plt.show()
```




```
35. >>>
```

本次示例中，生成的两个簇是完全间隔开的，观察模型学习到的均值（第 14 行）和方差（第 16 行），对比生成时设定的均值和标准差（第 2、3 行），可见误差很小。

如果将两个簇的一部分重合，比如将第 3 行的簇中心设为（3，3），重新运行程序，可得原始簇分布图和分簇结果分布图如图 3-14 的（a）和（b）所示。

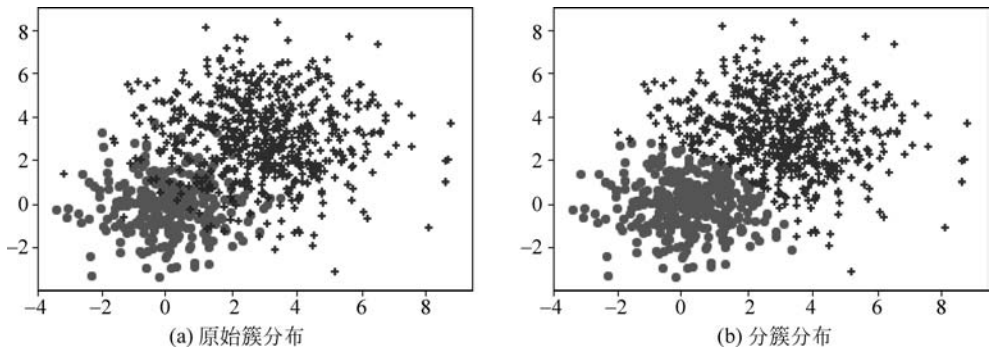


图 3-14　高斯混合聚类示例（见彩插）

可见，高斯混合聚类对重合部分的点并不能很好地进行预测，分簇结果有一条明显的分界线，容易理解，该分界线是两个模型计算概率值相等的地方。

新的均值为：

```
[[3.0551578 3.22641624]
 [0.24980259 0.07975279]]
```

新的协方差矩阵为：

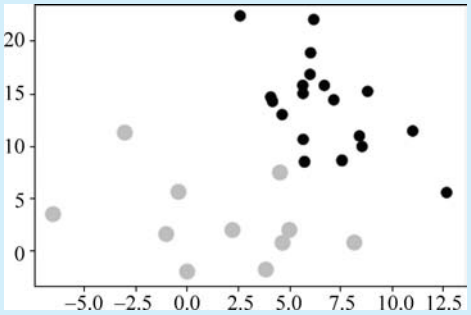
```
[[[ 3.18963888 -0.38221445]
   [-0.38221445 3.18689817]],
  [[ 1.69908699 0.11174126]
   [ 0.11174126 1.6890054 ]]].
```

可见均值和方差都发生了一定的偏移。

对文件 kmeansSamples.txt 中 30 个点进行高斯混合聚类分析，见代码 3-17。

代码 3-17　GaussianMixture 算法应用示例(GaussianMixture.ipynb)

```
36. from sklearn.mixture import GaussianMixture
37. import numpy as np
38. samples = np.loadtxt("kmeansSamples.txt")
39. gm = GaussianMixture(n_components = 2, random_state = 0).fit(samples)
40. labels = gm.predict(samples)
41. import matplotlib.pyplot as plt
42. plt.scatter(samples[:,0],samples[:,1],c = labels + 1.5,linewidths = np.power(labels +
    1.5, 2))
43. plt.show()
```




```
44. >>>
```

下面给出一个从多方面综合分析划分聚类、密度聚类和模型聚类，以及聚类算法内部评价指标的示例。该示例先生成三种二维平面上的实验数据和一种高维空间中的实验数据，然后分别用 k-means、DBSCAN 和 GaussianMixture 三种算法对它们进行聚类，并计算 SC、DBI、CH 和 ZQ 四个指标。该示例展示了实验样本点的分布与聚类算法适用性、评价指标值有效性的关系。示例的实现代码见附属资源中的文件“聚类算法综合比较示例.ipynb”，该代码不难理解，且与前文的示例有多处重复，故不再占用篇幅展示。

三种二维平面上的实验样本分布如图 3-15 所示，它们分别是圆环、高斯分布和月牙形状的，由 datasets 模块中相应的函数产生。

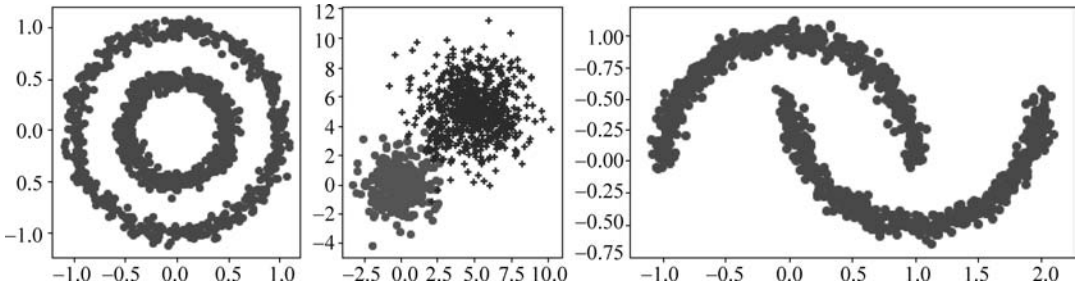


图 3-15　二维平面上的三种实验样本分布

高维空间中的实验样本通过 PCA 降维后，在二维平面上的分布如图 3-16 所示。它是由 datasets 模块中的 make_gaussian_quantiles()函数在四维空间中以原点(0,0,0,0)为中心，按高斯分布随机产生的，由内向外分为 9 层的类球状分布。随后去掉第 1～6 层和第 8 层，只保留内核的第 0 层和外面的第 7 层。可以将此数据想象成一个带核的空心四维类球体。

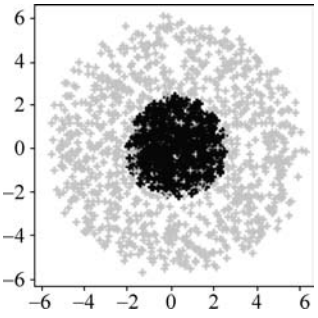
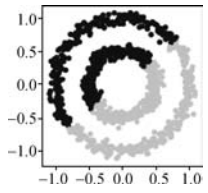
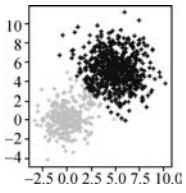
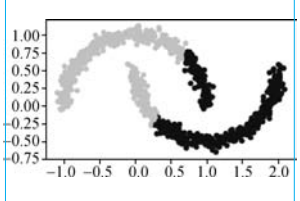
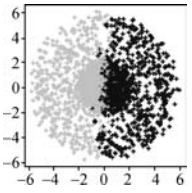
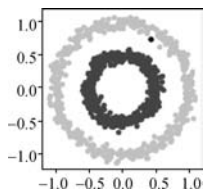
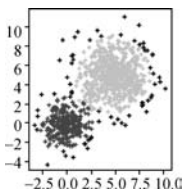
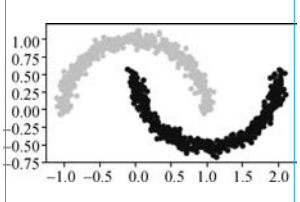
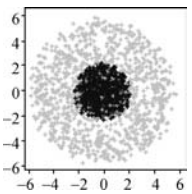


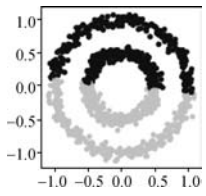
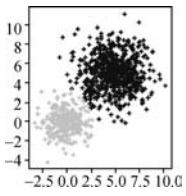
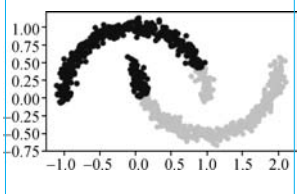
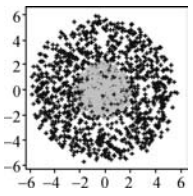
图 3-16　高维空间中实验样本降维后的分布

算法运行后的聚类效果及各评价指标值如表 3-1 所示。

表 3-1　聚类综合示例结果

算法	圆　环	高斯分布	月　牙	四维类球体
k-means				
	SC：0.35418150800 DBI：1.1828798822 CH：576.234365353 ZQ：1.16871702321	SC：0.595507603 DBI：0.54554156 CH：2088.867795 ZQ：0.789103847	SC：0.4894026970788733 DBI：0.7797138414306899 CH：1487.6573120666626 ZQ：1.4870808283885213	SC：0.14673718069 DBI：2.0836980081 CH：411.728662603 ZQ：2.59313957972
DBSCAN				
	SC：−0.0688868183 DBI：150.72658703 CH：0.60426715731 ZQ：0.07193600153	SC：0.497113248 DBI：3.39191627 CH：800.1776282 ZQ：1.228896011	SC：0.33345414657834466 DBI：1.1539812259101607 CH：663.1677674098607 ZQ：0.06232429416022914	SC：0.16606150306 DBI：98.110576302 CH：0.18240617409 ZQ：0.21742577037

续表

算法	圆　环	高斯分布	月　牙	四维类球体
Gaussian-Mixture				
SC：0.35177000142 DBI：1.1896529582 CH：569.108807414 ZQ：1.24921647476	SC：0.596057416 DBI：0.52269590 CH：2020.497457 ZQ：1.032134517	SC：0.4680579424239563 DBI：0.823742120847451 CH：1282.7601223372637 ZQ：2.5003513365473453	SC：0.16606150306 DBI：98.110576302 CH：0.18240617409 ZQ：0.21742577037	

DBSCAN 算法对非凸簇(四维类球体也是非凸簇)有较好的聚类效果。GaussianMixture 算法对高斯分布的簇有较好的聚类效果,四维类球体样本集也是按高斯分布产生的,因此,它可以很好地学习到模型参数。高斯分布的样本集在实际工程中比较常见。

预先探索样本集在空间中的分布对于选择合适的聚类算法很重要。除了通过降维来直观地观察样本集在空间中的分布外,聚类内部评价指标也可以帮助分析。比如在面对大数据量的聚类任务时,可以先随机抽取或者划分网格抽取小部分样本进行试分簇,如果发现运行 DBSCAN 算法后的 ZQ 指标改善较多,而其他指标变差,则样本集可能是非凸的分布。

3.5　层次聚类算法

在聚类算法中,有一类研究执行过程的算法,它们以其他聚类算法为基础,通过不同的运用方式试图达到提高效率、避免局部最优等目的。这类算法主要有网格聚类和层次聚类算法。

网格(Grid)聚类算法强调的是分批统一处理以提高效率,具体的做法是将特征空间划分为若干个网格,网格内的所有样本点看成一个单元进行处理。网格聚类算法要与划分聚类或密度聚类算法结合使用。网格聚类算法处理的单元只与网格数量有关,与样本数量无关,因此在数据量较大时,网格聚类算法可以极大地提高效率。网格聚类算法的思路容易理解,实现简单,不再赘述。

层次(Hierarchical)聚类算法强调的是聚类执行的过程,分为自底向上的凝聚方法和自顶向下的分裂方法。凝聚方法先将每一个样本点当成一个簇,然后根据距离和密度等度量准则进行逐步合并。分裂方法先将所有样本点放在一个簇内,然后再逐步分解。前者的典型算法有 AGNES 算法,后者的典型算法有二分 k-means 算法。

1. 二分 k-means 算法

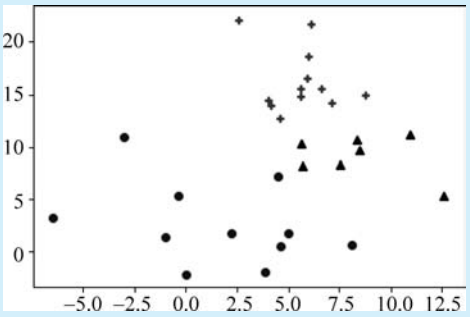
二分 k-means(Bisecting k-means)算法[5]的基本思想是“分裂”,它首先将所有点看成一个簇,然后将该簇一分为二,之后选择其中一个簇继续分裂。选择哪一个簇进行分裂,取决于对其进行的分裂是否可以最大限度地降低 SSE 值。如此分裂下去,直到达到指定的簇数目 k 为止。

用 cluster 模块中的 KMeans 类作为基本分簇算法来实现二分 k-means 算法的代码见代码 3-18。该代码还对二分 k-means 算法进行了简单的应用示例。

代码 3-18　二分 k-means 算法实现及应用示例（二分 k-means 算法.ipynb）

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
samples = np.loadtxt("kmeansSamples.txt")
n_clusters = 3                  # 分簇的数量
n_init = 1                      # 指定 k-means 算法重复运行次数
estimator = KMeans(init = 'k-means++', n_clusters = 1, n_init = n_init)   # 设置 n_
# clusters = 1 是为了计算 SSE 值
estimator.fit(samples)
samples = [samples]
SSE = [estimator.inertia_]   # 记录下簇的 SSE 值
# 分裂主循环
while len(SSE) < n_clusters:
    max_changed_SSE = 0
    tag = -1
    for i in range(len(SSE)): # 对每个簇进行试分簇,计算 SSE 的减少量
        estimator = KMeans(init = 'k-means++', n_clusters = 2, n_init = n_init).fit
(samples[i])                    # 二分簇
        changed_SSE = SSE[i] - estimator.inertia_
        if changed_SSE > max_changed_SSE:   # 比较 SSE 值是不是减少了
            max_changed_SSE = changed_SSE
            tag = i
    # 正式分簇
    estimator = KMeans(init = 'k-means++', n_clusters = 2, n_init = n_init).fit
(samples[tag])
    indexs0 = np.where(estimator.labels_ == 0)  # 标签为 0 的样本在数组中的下标
    cluster0 = samples[tag][indexs0]  # 从簇中分出标签为 0 的新簇
    indexs1 = np.where(estimator.labels_ == 1)
    cluster1 = samples[tag][indexs1]  # 从簇中分出标签为 1 的新簇

    del samples[tag]
    samples.append(cluster0)
    samples.append(cluster1)
    del SSE[tag]
    estimator = KMeans(init = 'k-means++', n_clusters = 1, n_init = n_init).fit(cluster0)
    SSE.append(estimator.inertia_)  # 新簇的 SSE 值
    estimator = KMeans(init = 'k-means++', n_clusters = 1, n_init = n_init).fit(cluster1)
    SSE.append(estimator.inertia_)
# 简单应用示例
markers = [ 'o', '+', '^', 'x', 'D', '*', 'p' ]
colors = [ 'g', 'r', 'b', 'c', 'm', 'y', 'k' ]
linestyle = [ '-', '--', '-.', ':' ]
if len(samples) <= len(markers):
```

```
41.     for i in range(len(samples)):
42.         plt.scatter(samples[i][:, 0], samples[i][:, 1], marker = markers[i], c = colors[i])
43.     plt.show()
```




```
44. >>>
```

2. AGNES 算法

AGNES(AGglomerative NESting)算法[6]先将每个样本点看成一个簇,然后根据簇与簇之间的距离度量,将最近的两个簇合并,一直重复合并到指定的簇数目 k 为止。

该算法的思路很简单,应用的关键在于处理簇与簇之间不同的距离度量方法带来的影响差异问题。

设有两个簇 $\boldsymbol{C}_i$ 和簇 $\boldsymbol{C}_j$,用 $\mathrm{DIST}(\boldsymbol{C}_i,\boldsymbol{C}_j)$表示簇之间的距离,用 $\mathrm{dist}(\boldsymbol{x}_i,\boldsymbol{x}_j)$表示点之间的距离。

1) 簇最小距离

$$\mathrm{DIST}_{\min}(\boldsymbol{C}_i,\boldsymbol{C}_j)=\min_{\substack{\boldsymbol{x}_k\in\boldsymbol{C}_i\\ \boldsymbol{x}_l\in\boldsymbol{C}_j}}\mathrm{dist}(\boldsymbol{x}_k,\boldsymbol{x}_l) \tag{3-21}$$

簇最小距离是两个簇成员之间的最小距离。

2) 簇最大距离

$$\mathrm{DIST}_{\max}(\boldsymbol{C}_i,\boldsymbol{C}_j)=\max_{\substack{\boldsymbol{x}_k\in\boldsymbol{C}_i\\ \boldsymbol{x}_l\in\boldsymbol{C}_j}}\mathrm{dist}(\boldsymbol{x}_k,\boldsymbol{x}_l) \tag{3-22}$$

与簇最小距离相反,簇最大距离是两个簇成员之间的最大距离。

3) 簇平均距离

$$\mathrm{DIST}_{\mathrm{avg}}(\boldsymbol{C}_i,\boldsymbol{C}_j)=\frac{1}{|\boldsymbol{C}_i||\boldsymbol{C}_j|}\sum_{\substack{\boldsymbol{x}_k\in\boldsymbol{C}_i\\ \boldsymbol{x}_l\in\boldsymbol{C}_j}}\mathrm{dist}(\boldsymbol{x}_k,\boldsymbol{x}_l) \tag{3-23}$$

簇平均距离是两个簇成员之间距离的平均值。

sklearn 扩展库的 cluster 模块的 AgglomerativeClustering 类实现了 AGNES 算法,类原型和方法见代码 3-19。

代码 3-19　sklearn 中的 AGNES 算法

```
1. class sklearn.cluster.AgglomerativeClustering(n_clusters = 2, affinity = 'euclidean',
   memory = None, connectivity = None, compute_full_tree = 'auto', linkage = 'ward', pooling_
   func = 'deprecated', distance_threshold = None)
```

```
2.
3. fit(X[, y])
4. fit_predict(X[, y])
5. get_params(deep = True)
6. set_params( ** params)
```

n_clusters 是指定的分簇数。

linkage 是簇距离度量方法，支持 ward、complete、average 和 single 四种方法。complete、average 和 single 分别对应簇最大距离、簇平均距离和簇最小距离。

ward 方法与其他方法不一样，它不是按距离合并簇，而是合并使得偏差（样本点与簇中心的差值）平方和增加最小的两个簇。它先要对所有簇进行两两试合并，并计算偏差平方和的增加值，然后取增加最小的两个簇进行合并。

sklearn 官方网站对 linkage 的不同设值的影响进行了分析[①]，结果如图 3-17 所示，可

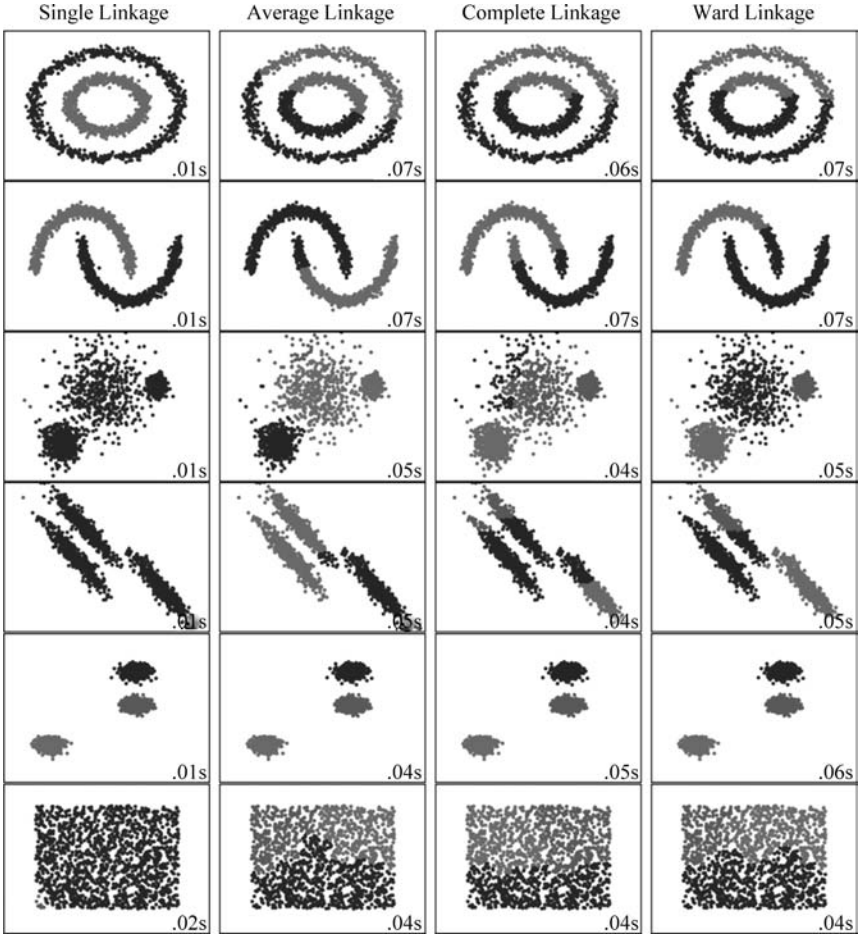


图 3-17　AGNES 算法中 linkage 不同设值的影响

① https://scikit-learn.org/stable/auto_examples/cluster/plot_linkage_comparison.html

见采用簇最小距离时,其分簇效果接近于密度聚类。其示例代码(plot_linkage_comparison.ipynb)的关键部分已经在前面的例子中多次应用过,读者应该不难理解,因此不再分析。


视频讲解

3.6 Mean Shift 算法及其在图像分割中的应用示例

除了前文讨论过的几类典型的聚类算法外,人们还提出了很多聚类算法,限于篇幅,不能在本章一一讨论。本章最后介绍在图像处理领域有较多应用的 Mean Shift 算法,并简单示例它在图像分割中的有趣应用。

Mean Shift 算法是根据样本点分布密度进行迭代的聚类算法,它可以发现在空间中聚集的样本簇。簇中心是样本点密度最大的地方。Mean Shift 算法寻找一个簇的过程是先随机选择一个点作为初始簇中心,然后从该点开始,始终向密度大的方向持续迭代前进,直到到达密度最大的位置。然后将剩下的点重复以上过程,找到所有簇中心。

如何找到密度大的方向并确定前进多少呢?设第 i 个簇在第 t 轮迭代时簇中心位于 $\boldsymbol{x}_i^t$,则第 $t+1$ 轮迭代簇中心位置 $\boldsymbol{x}_i^{t+1}$ 为:

$$\boldsymbol{x}_i^{t+1}=\frac{\sum\limits_{\boldsymbol{x}_j\in\boldsymbol{N}(\boldsymbol{x}_i^t)}K(\boldsymbol{x}_j-\boldsymbol{x}_i^t)\boldsymbol{x}_j}{\sum\limits_{\boldsymbol{x}_j\in\boldsymbol{N}(\boldsymbol{x}_i^t)}K(\boldsymbol{x}_j-\boldsymbol{x}_i^t)} \tag{3-24}$$

其中,$\boldsymbol{N}(\boldsymbol{x}_i^t)$是以 $\boldsymbol{x}_i^t$ 为中心、指定长度 bandwidth 为半径的邻域;$\boldsymbol{x}_j$ 是该邻域内的样本点。K 是核函数。

简要讨论一下式(3-24)的含义。先假定核函数 K 的值取常数 1,此时式(3-24)为:

$$\boldsymbol{x}_i^{t+1}=\frac{\sum\limits_{\boldsymbol{x}_j\in\boldsymbol{N}(\boldsymbol{x}_i^t)}\boldsymbol{x}_j}{\sum\limits_{\boldsymbol{x}_j\in\boldsymbol{N}(\boldsymbol{x}_i^t)}1}=\frac{\sum\limits_{\boldsymbol{x}_j\in N(\boldsymbol{x}_i^t)}\boldsymbol{x}_j}{m} \tag{3-25}$$

其中,分母 m 是邻域 $\boldsymbol{N}(\boldsymbol{x}_i^t)$中样本点的个数,分子表示邻域内各点的和。下面用仅包含两个点 $\boldsymbol{x}_1$ 和 $\boldsymbol{x}_2$ 的邻域来说明式(3-25)的含义,如图 3-18 所示。

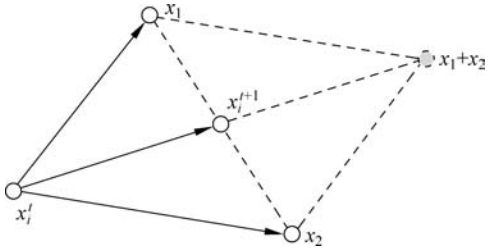


图 3-18 Mean Shift 算法簇中心迭代示意

当邻域中只有两个点 $\boldsymbol{x}_1$ 和 $\boldsymbol{x}_2$ 时,式(3-25)为:

$$\boldsymbol{x}_i^{t+1}=\frac{\boldsymbol{x}_1+\boldsymbol{x}_2}{2} \tag{3-26}$$

如果以 $\boldsymbol{x}_i^t$ 为原点，则式(3-26)可用图 3-18 所示的向量加法来表示。可见从 $\boldsymbol{x}_i^t$ 到 $\boldsymbol{x}_i^{t+1}$ 的方向是邻域内两个样本点向量和的方向，可认为是邻域内样本点密集的方向，前进的距离是两个样本点向量和的一半。如果邻域内有多个点，则按向量加法全部计算。

式(3-24)中，$K(\boldsymbol{x}_j-\boldsymbol{x}_i^t)\boldsymbol{x}_j$ 可看作是对向量 $\boldsymbol{x}_j$ 进行了一次系数为核函数 $K(\boldsymbol{x}_j-\boldsymbol{x}_i^t)$ 的加权运算。核函数 $\boldsymbol{K}$ 是$(\boldsymbol{x}_j-\boldsymbol{x}_i^t)$的函数，比如常用的高斯核函数：

$$G(\boldsymbol{x}_j-\boldsymbol{x}_i^t)=\mathrm{e}^{\left(-\frac{(\boldsymbol{x}_j-\boldsymbol{x}_i^t)^{\mathrm{T}}(\boldsymbol{x}_j-\boldsymbol{x}_i^t)}{2\tau^2}\right)} \tag{3-27}$$

其中，τ 为预设的参数。$\boldsymbol{x}_j$ 到 $\boldsymbol{x}_i^t$ 的距离越大，高斯核函数的值越小。因此，式(3-24)可以看作是对邻域内所有样本点求加权后的均值。通过加权，使得不同距离的样本点对 $\boldsymbol{x}_i^{t+1}$ 有不同的影响。式(3-24)称为均值漂移向量(Mean Shift Vector)。

被簇中心扫过的点计入该簇中心的簇，如果一个点被多个簇中心扫过，则计入被扫过次数最多的簇中心的簇。

sklearn 扩展库的 cluster 模块中实现 Mean Shift 算法的类原型如代码 3-20 所示。

代码 3-20　sklearn 中的 Mean Shift 算法

```
class sklearn.cluster.MeanShift( * , bandwidth = None, seeds = None, bin_seeding = False,
min_bin_freq = 1, cluster_all = True, n_jobs = None, max_iter = 300)

fit(X, y = None)
fit_predict(X, y = None)
predict(X)
get_params(deep = True)
set_params( ** params)
```

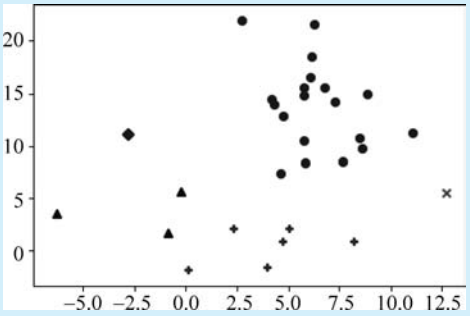
bandwidth 是 Mean Shift 算法的超参数，需要用户指定。bandwidth 决定了邻域的大小，对算法的效果影响很大。如果用户不指定该参数，MeanShift 类在实例化时，会调用 estimate_bandwidth()函数来估计它的值。

对文件 kmeansSamples.txt 中的 30 个点进行 Mean Shift 聚类分析，见代码 3-21。

代码 3-21　Mean Shift 算法应用示例(MeanShift.ipynb)

```
import numpy as np
from sklearn import cluster
import matplotlib.pyplot as plt
samples = np.loadtxt("kmeansSamples.txt")

# 估计 bandwidth
bandwidth = cluster.estimate_bandwidth(samples, quantile = 0.2)
print(bandwidth)
>>> 4.528776571054436

ms = cluster.MeanShift(bandwidth = bandwidth, bin_seeding = True).fit(samples)
print(ms.cluster_centers_)
markers = [ 'o', '+', '^', 'x', 'D', '*', 'p' ]
colors = [ 'g', 'r', 'b', 'c', 'm', 'y', 'k' ]
linestyle = [ '-', '--', '-.', ':' ]
```

```
16. if len(np.unique(ms.labels_)) <= len(markers):
17.     for i in range(len(samples)):
18.         plt.scatter(samples[i, 0], samples[i, 1], marker = markers[ms.labels_[i]], c =
    colors[ms.labels_[i]])
19.     plt.show()
20. >>> [[ 6.278276 13.65518989]
21. [ 2.48936265 0.43984307]
22. [ -2.54974089 3.54244933]
23. [12.56729723 5.50656992]
24. [ -2.92514764 11.0884457 ]]
```




```
25.
```

可见分成了5个簇,其中两个簇只包含一个点,可视为离群点。

在一篇Mean Shift算法的重要论文[7]中,作者Comaniciu成功将它应用到图像平滑和图像分割中。

在计算机中,一幅完整的图像是由像素点组成,像素点包括由高(Height)、宽(Width)组成的位置信息和由红、绿、蓝组成的RGB三通道(Channel)色彩信息。如代码3-22第8行的输出所示,代码表示每个像素点的颜色分别用代表红、绿、蓝3种原色的亮度数据来合成表示。第5行用Matplotlib扩展库中的image函数读入一幅jpg图片,第7行显示它的存放形式、形状和(0,0)像素点的三原色值,可见是用NumPy的数组存储,该图片的高为934像素,宽为734像素。每个像素点的颜色由一个三维的数组来表示,分别表示红、绿、蓝三原色的值,它们的取值范围为0～255。由第8行输出可知,(0,0)位置的像素点的三原色的值为[43 36 26]。

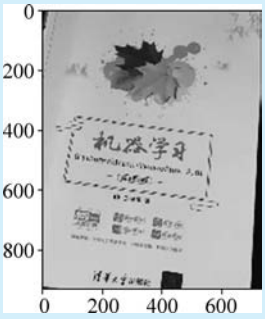
有很多Python扩展库提供了对图像处理的支持,它们存储图像数据的格式不尽相同。为方便读者理解,本示例采用了用NumPy数组来存储图像的Matplotlib扩展库。

用聚类的方法来分割图像,实际上是对图片中出现的颜色进行分簇。它将每一个像素点的由三原色值组成的颜色数组看成是三维空间中的一个点,然后对三维空间中的所有点进行分簇。同一簇内的点被认为颜色相似,因此,图像分割就是把不同簇的像素点分割出来。用Mean Shift算法进行图像分割的示例如代码3-22所示,第10行显示了实验图像。

代码3-22　应用Mean Shift算法进行图像分割(MeanShift.ipynb)

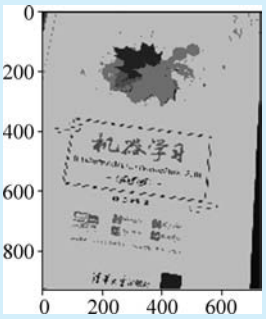
```
1. import matplotlib.image as mpimg
2. from time import time
```

```
3.
4. path = r"原版书.jpg"
5. img = mpimg.imread(path)
6.
7. print(type(img), img.shape, img[0,0])# 图片加载后的数据类型、形状和(0,0)像素点的三
   # 原色值
8. >>><class 'numpy.ndarray'> (934, 734, 3) [43 36 26]
9.
10. plt.imshow(img)
```

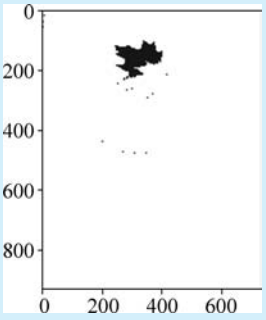



```
11. >>>
12.
13. # 将二维的图像数组改为一维的,以适合聚类算法的要求
14. height = img.shape[0]
15. width = img.shape[1]
16. img1 = img.reshape((height * width, 3))
17.
18. t0 = time()                              # 开始计时
19. ms = cluster.MeanShift(bandwidth = 25, bin_seeding = True).fit(img1)
20. print("time", time() - t0)
21.
22. # 构建一幅新的相同大小的空图片
23. pic_new = np.zeros((height, width, 3), dtype = 'i')
24. # 将分簇后一维标签改为二维的,与图片的形状一致
25. label = ms.labels_.reshape((height, width))
26. print(ms.cluster_centers_)              # 看一下簇中心的 RGB 三通道值
27. >>> time 59.89742588996887
28. [[189.47612188 179.73489904 176.5341613 ]
29. [159.16646716 98.51743676 38.72212912]
30. [ 19.99000406 13.89776514 11.2242178 ]
31. [ 87.88985109 15.029887    18.6358644 ]
32. [150.89538613 51.60313348 47.67554898]
33. [ 45.02707779 35.51087807 36.19893567]]
34.
35. # 将簇中心三通道值改为整型,便于显示
36. center = ms.cluster_centers_
37. center = center.astype(np.int)
38.
39. # 同簇点的颜色用该簇簇中心点的颜色代替
40. for i in range(height):
41.     for j in range(width):
```

```
42.         pic_new[i,j] = center[label[i,j]]
43.
44. plt.imshow(pic_new)
```




```
45. >>>
46.
47. n_labels = len(np.unique(ms.labels_))
48. for i in range(n_labels):          # 看一下每个簇的样本数量
49.     print(len(np.where(ms.labels_ == i)[0]))
50. >>> 597484
51. 38970
52. 11838
53. 10524
54. 15880
55. 10860
56.
57. # 单独显示簇 k,其他簇都用白色代替
58. k = 3
59. center1 = center.copy()
60. for i in range(k):
61.     center1[i] = np.array([255, 255, 255])
62. for i in range(k + 1, n_labels):
63.     center1[i] = np.array([255, 255, 255])
64.
65. for i in range(height):
66.     for j in range(width):
67.         pic_new[i,j] = center1[label[i,j]]
68. plt.imshow(pic_new)
```




```
69. >>>
```

第 13 行将二维的图像数组改为一维的，以适合聚类算法的要求。第 19 行用 Mean Shift 算法进行分簇。第 23 行建立一幅相同大小的空图片，实际上是建立一个相同大小的 0 值数组。第 39 行将一个簇内的点都用簇中心点的颜色来代替，以显示出分割后的形状。

为了便于观察，也可以单独显示出一个簇的形状，如第 57 行到第 69 行，给出了簇 3 分割后的形状，主要是深色枫叶的部分。读者可以修改第 58 行的 k 值看一下其他簇分割后的形状。

当然，其他聚类算法也可以用来进行图像分割，它们有不同的应用特点，不再细述。随书资源中的 MeanShift.ipynb 文件中给出了用 k-means 算法对图像进行分割的示例，供读者参考。

3.7　习题

1. 平面上有以下五个点：(1,2)、(2,4)、(1,−1)、(2,5)、(0,−3)，用 k-means 算法对其进行簇数为 2 的聚类，初始簇中心为(0,0)、(5,5)。请给出经过 1 轮迭代和 2 轮迭代后的簇中心坐标。

2. Scikit-Learn 工具包提供了 7 个试验用的数据集(原文为 toy datasets)，它们经常用于演示各种算法的使用方法。基于其中的鸢尾花数据集 Iris plants dataset 进行 k-means 算法自主试验，试验后可对照官网提供的试验代码[①]。

3. 第 2 题，用本章论及的聚类算法对 Iris plants dataset 进行试验，并观察聚类结果的 SC、DBI、CH 和 ZQ 指标值，分析它们的原因。

4. 代码 3-22 示例了用 Mean Shift 算法来进行图像分割。尝试采用其他聚类算法来实现不同图像的分割应用。

① https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_iris.html#sphx-glr-auto-examples-cluster-plot-cluster-iris-py