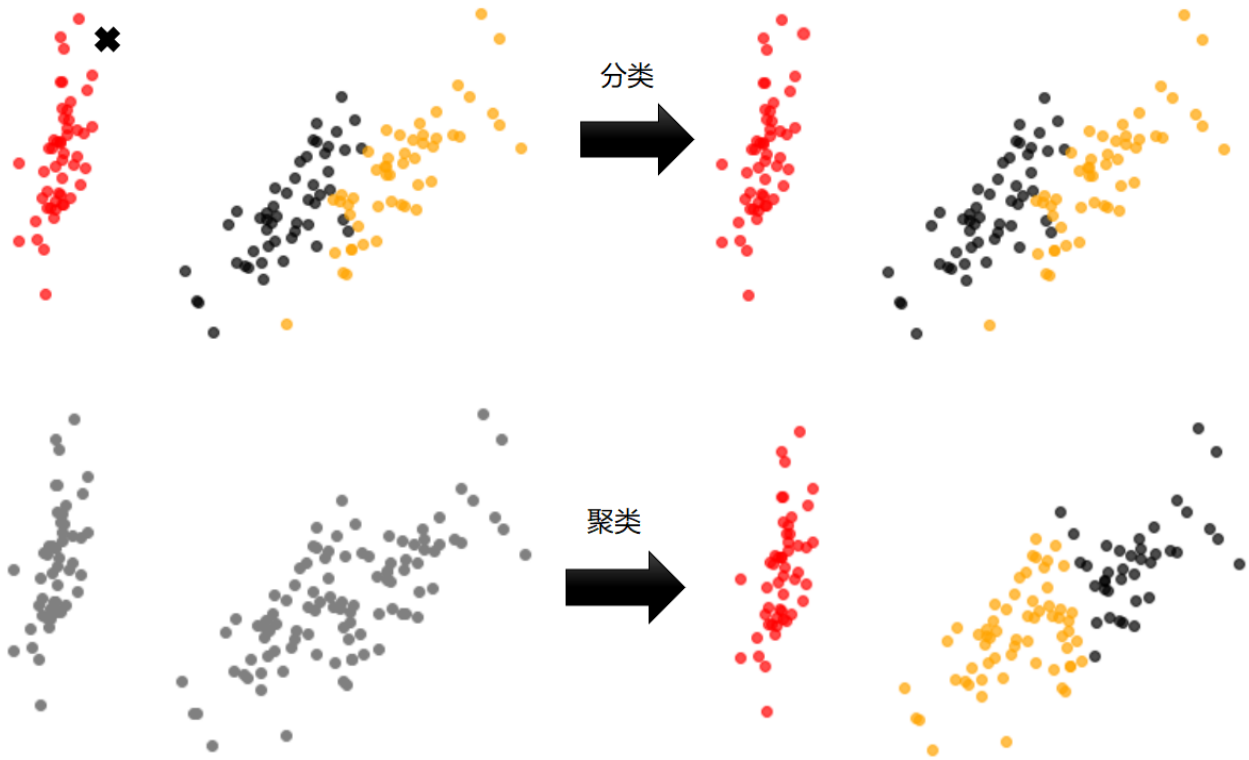
聚类算法

一、无监督学习与聚类算法

决策树、线性和逻辑回归都是比较常用的机器学习算法，他们虽然有着不同的功能，但却都属于“有监督学习”的一部分，即是说，模型在训练的时候，即需要特征矩阵X，也需要真实标签y。机器学习当中，还有相当一部分算法属于“无监督学习”，无监督的算法在训练的时候只需要特征矩阵X，不需要标签。无监督学习的代表算法有聚类算法、降维算法。

- **聚类vs分类**



	聚类	分类
核心	将数据分成多个组 探索每个组的数据是否有联系	从已经分组的数据中去学习 把新数据放到已经分好的组中去
学习类型	无监督，无需标签进行训练	有监督，需要标签进行训练
典型算法	K-Means，DBSCAN，层次聚类，光谱聚类	决策树，贝叶斯，逻辑回归
算法输出	聚类结果是不确定的 不一定总是能够反映数据的真实分类 同样的聚类，根据不同的业务需求 可能是一个好结果，也可能是一个坏结果	分类结果是确定的 分类的优劣是客观的 不是根据业务或算法需求决定

聚类算法是无监督类机器学习算法中最常用的一类，其目的是将数据划分成有意义或有用的组（也被称为簇）。这种划分可以基于我们的业务需求或建模需求来完成，也可以单纯地帮助我们探索数据的自然结构和分布。如果目标是划分成有意义的组，则簇应当捕获数据的自然结构。然而，在某种意义下，聚类分析只是解决其他问题（如数据汇总）的起点。无论是旨在理解还是应用，聚类分析都在广泛的领域扮演着重要角色。这些领域包括：心理学和其他社会科学、生物学、统计学、模式识别、信息检索、机器学习和数据挖掘。

聚类分析在许多实际问题上都有应用，下面是一些具体的例子，按聚类目的是为了理解数据自然结构还是用于数据处理来组织。

1. 旨在理解数据自然结构的聚类

在对世界的分析和描述中，类，或在概念上有意义的具有公共特性的对象组，扮演着重要的角色。的确，人类擅长将对象划分成组（聚类），并将特定的对象指派到这些组（分类）。例如，即使很小的孩子也能很快地将图片上的对象标记为建筑物、车辆、人、动物、植物等。就理解数据而言，簇是潜在的类，而聚类分析是研究自动发现这些类的的技术。下面是一些例子：

- **生物学**

  生物学家花了许多年来创建所有生物体的系统分类学（层次结构的分类）：界（kingdom）、门(phylum)、纲（class）、目（order）、科（family）、属（genus）和种（species）这样，或许并不奇怪，聚类分析早期的大部分工作都是在寻求创建可以自动发现分类结构的数学分类方法。最近，生物学家使用聚类分析大量的遗传信息。例如，聚类已经用来发现具有类似功能的基因组。

- **信息检索**

  万维网包含数以亿计的Web页面，网络搜索引擎可能返回数以千计的页面。可以使用聚类将搜索结果分成若干簇，每个簇捕获查询的某个特定方面。例如，查询 ”电影“ 返回的网页可以分成诸如评论、电影预告片、影星和电影院等类别。每一个类别（簇）又可以划分成若干子类别（子簇），从而产生一个层次结构，帮助用户进一步探索查询结果。

- **气候**

  理解地球气候需要发现大气层和海洋的模式。聚类分析已经用来发现对陆地气候具有显著影响的极地和海洋压力模式。

- **心理学和医学**

  一种疾病或健康状况通常有多种变种，聚类分析可以用来发现这些子类别。例如，聚类已经用来识别不同类型的抑郁症。聚类分析也可以用来检测疾病的时间和空间分布模式。

- **商业**

  在商业中，如果我们手头有大量的当前和潜在客户的信息，我们可以使用聚类将客户划分为若干组，以便进一步分析和开展营销活动，最有名的客户价值判断模型RFM（Recency Frequency Monetary），就常常和聚类分析共同使用。

2. 用于数据处理的聚类

聚类分析提供由个别数据对象到数据对象所指派的簇的抽象。此外，一些聚类技术使用簇原型（即同一个簇中用于代表其他对象的数据对象，例如质心等）来刻画簇特征。这些簇原型可以用作大量数据分析和数据处理技术的基础。因此，就聚类用于数据处理层面而言，聚类分析是研究发现最有代表性的簇原型的技术。

- **数据降维**

  许多数据分析技术，如回归和PCA，都具有$O(m^2)$ 或更高的时间或空间复杂度（其中m 是对象的个数）。因此对于大型数据集，这些技术不切实际。然而，可以将算法用于仅包含簇原型的数据集，即每一列只保留其原型，而不是整个数据集。依赖分析类型、原型个数和原型代表数据的精度，行汇总结果可以与使用所有数据得到的结果相媲美。

- **数据离散压缩**

  簇原型可以用于数据列压缩。例如，创建一个包含所有簇原型的表，即每个原型赋予一个整数值，作为它在表中的位置（索引）。每个对象用于它所在的簇相关联的原型的索引表示。这类压缩称作向量量化（vector quantization），并常常用于图像、声音和视频数据。

  无论是用于数据降维还是离散压缩，都将损失一定的数据信息量，而能够使用聚类进行信息浓缩的数据的特点是：

  （1）许多数据对象之间高度相似

  （2）某些信息丢失是可以接受的

  （3）希望大幅度压缩数据量

- **有效地发现最近邻**

  很多机器学习算法都是基于距离的模型，即找出最近邻可能需要计算所有点对点之间的距离，如之前介绍的KNN。通常我们可以使用簇原型减少发现对象最近邻所需要计算的距离的数目。直观地说，如果两个簇原型相距很远，则对应簇中的对象不可能互为近邻。这样，为了找出一个对象的最近邻，只需要计算到邻近簇中对象的距离，其中两个簇的邻近性用其原型之间的距离度量，从而大幅降低KNN类算法的计算量。

二、核心概念

在讨论具体的聚类技术之前，还需要讨论必要的背景知识。首先，我们就聚类分析中的核心概念展开讨论。

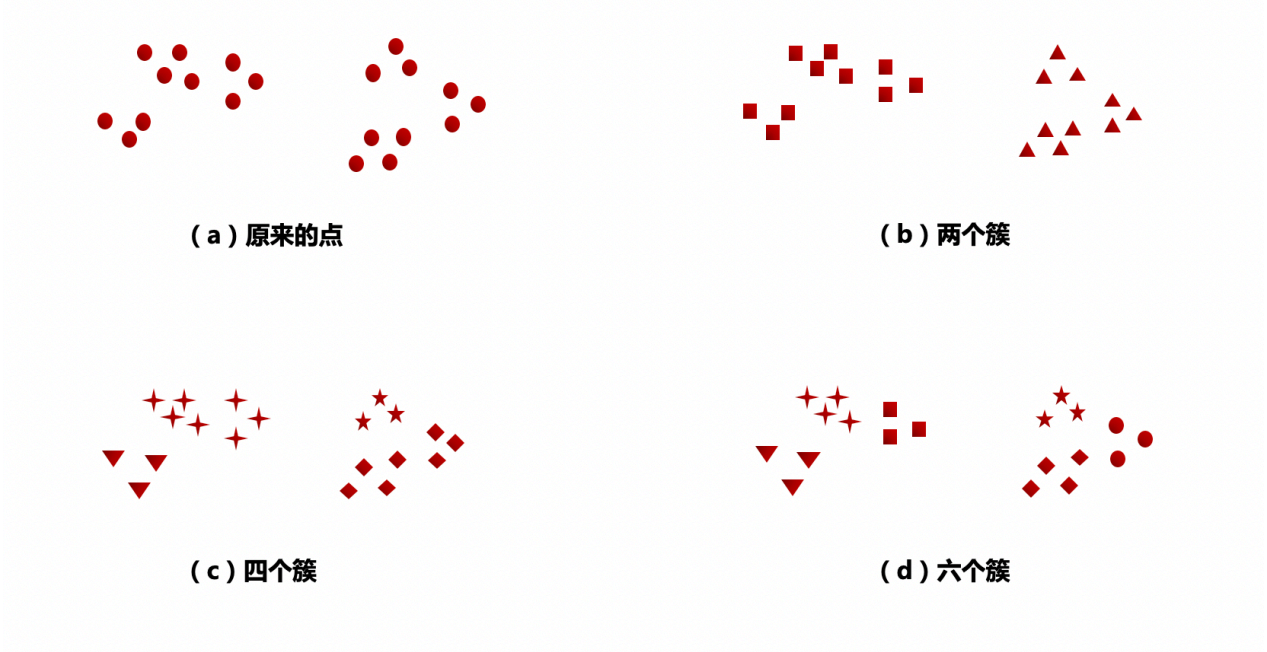
1. 聚类分析

聚类分析仅根据在数据中发现的描述对象及其关系的信息，将数据对象分组。其目标是，组内的对象相互之间是相似的（相关的），而不同组中的对象是不同的（不相关的）。组内的相似性（同质性）越大，组间差别越大，聚类就越好。

2. 簇

简单来说，簇就是分类结果的类，但实际上簇并没有明确的定义，并且簇的划分没有客观标准，我们可以利用下图来理解什么是簇。该图显示了20个点和将它们划分成簇的3种不同方法。标记的形状指示簇的隶属关系。下图分别将数据划分成两部分和六部分。然而，将2个较大的簇都划分成3个子簇可能是人的视觉系统造成的假象。此外，说这些点形成4个簇可能也不无道理。该图表明簇的定义是不精确的，而最好的定义依赖于数据的特性和期望的结果。

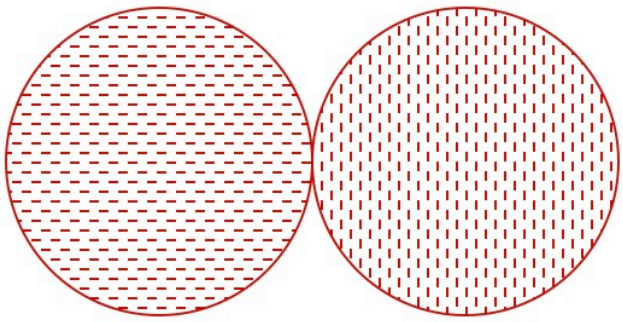
聚类分析与其他将数据对象分组的技术相关。例如，聚类可以看作一种分类，它用类（簇）标号创建对象的标记。然而，只能从数据导出这些标号。相比之下，KNN则是监督分类（supervised classfication），即使用由类标号已知的对象开发的模型，对新的、无标记的对象赋予类标号。为此，有时称聚类分析为非监督分类（unsupervised classification）。在数据挖掘中，不附加任何条件使用术语分类时，通常是指监督分类。



三、基于原型的聚类技术：K-Means

1. 基于原型的簇

此时簇是对象的集合，并且其中每个对象到定义该簇的原型的距离比到其他簇的原型的距离更近（或更加相似）。**对于具有连续属性的数据，簇的原型通常是质心，即簇中所有点的平均值。**当质心没有意义时（例如当数据具有分类属性时），原型通常是中心点，即簇中最有代表性的点。对于许多数据类型，原型可以视为最靠近中心的点；在这种情况下，通常把基于原型的簇看作**基于中心的簇**（center-based cluster）。毫无疑问，这种簇趋向于呈球状。下图是一个基于中心簇的例子。



基于中心的簇。每个点到其簇中心的距离比到任何其他簇中心的距离更近。

2. K-Means基本定义

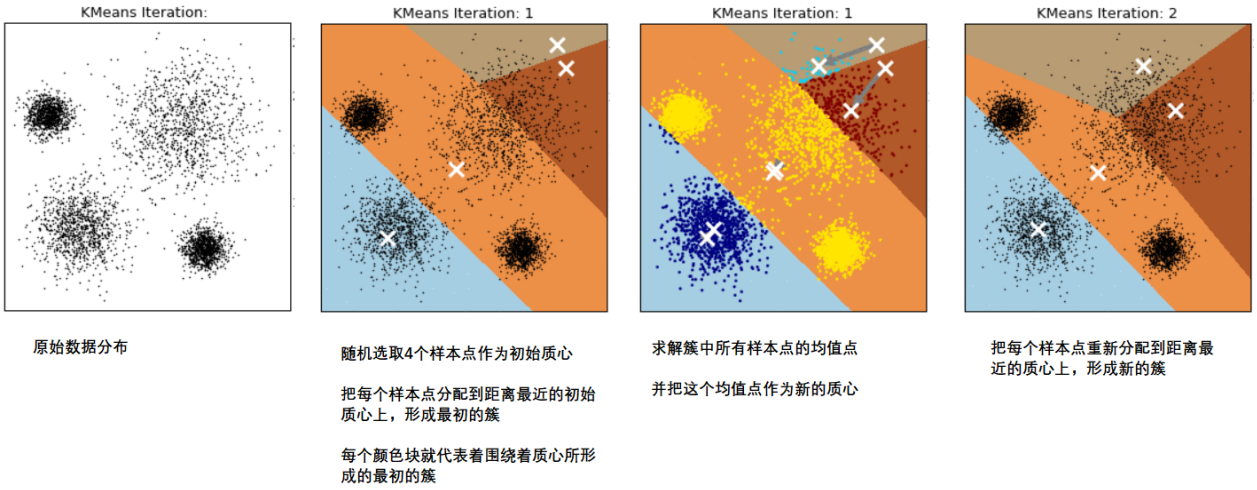
K均值算法比较简单，我们从介绍它的基本算法开始。首先，选择K个初始质心，其中K是我们指定的参数，即所期望的簇的个数。每个点指派到最近的质心，而指派到一个质心的点集为一个簇。然后，根据指派到簇的点，更新每个簇的质心。重复指派和更新步骤，直到簇不发生变化，或等价地，直到质心不发生变化。

K-Means的核心任务就是根据我们设定好的K，找出K个最优的质心，并将离这些质心最近的数据分别分配到这些质心代表的簇中去。具体过程可以总结如下：

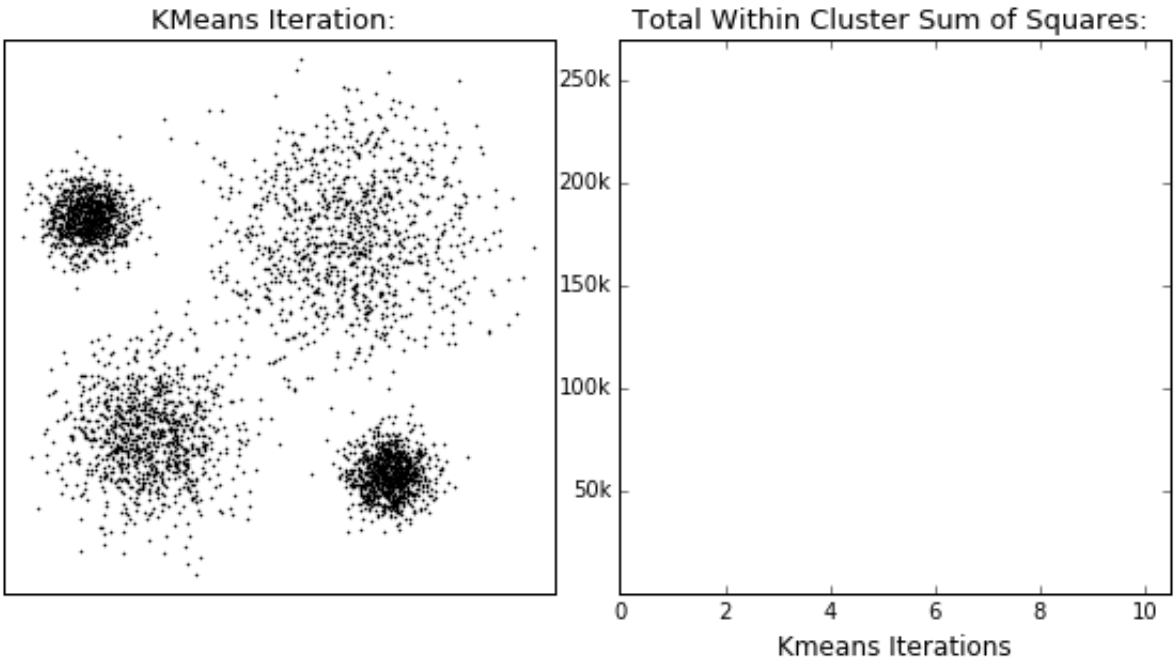
```
创建k个点作为初始质心（通常是随机选择）
当任意一个点的簇分配结果发生改变时：
    对数据集中的每个点：
        对每个质心：
            计算质心与数据点之间的距离
        将数据点分配到据其最近的簇
    对每个簇，计算簇中所有点的均值并将均值作为新的质心
直到簇不再发生变化或者达到最大迭代次数
```

那什么情况下，质心的位置会不再变化呢？当我们找到一个质心，在每次迭代中被分配到这个质心上的样本都是一致的，即每次新生成的簇都是一致的，所有的样本点都不会再从一个簇转移到另一个簇，质心就不会变化了。

这个过程在可以由下图来显示，我们规定，将数据分为4簇（K=4），其中白色X代表质心的位置：



在数据集下多次迭代(iteration)，就会有：



可以看见，第六次迭代之后，基本上质心的位置就不再改变了，生成的簇也变得稳定。此时我们的聚类就完成了，我们可以明显看出，K-Means按照数据的分布，将数据聚集成了我们规定的4类，接下来我们就可以按照我们的业务需求或者算法需求，对这四类数据进行不同的处理。

2. 算法执行细节

接下来，我们详细讨论上述算法执行过程的各细节。

聚类算法聚出的类有什么含义呢？这些类有什么样的性质？我们认为，**被分在同一个簇中的数据是有相似性的，而不同簇中的数据是不同的**，当聚类完毕之后，我们就要分别去研究每个簇中的样本都有什么样的性质，从而根据业务需求制定不同的商业或者科技策略。聚类算法的目的就是追求“簇内差异小，簇外差异大”。而这个“差异“，由**样本点到其所在簇的质心的距离**来衡量。

对于一个簇来说，所有样本点到质心的距离之和越小，我们就认为这个簇中的样本越相似，簇内差异就越小。而距离的衡量方法有多种，令x表示簇中的一个样本点，μ表示该簇中的质心，n表示每个样本点中的特征数目，i表示组成点x的每个特征，则该样本点到质心的距离可以由以下距离来度量。

2.1 距离衡量方法

常见距离衡量方法即对应公式如下所示：

- **数据距离**

  对于数据之间的距离而言，在欧式空间中我们常常使用欧几里得距离，也就是我们常说的距离平方和开平方，其基本计算公式如下：

$$d(x,\mu)=\sqrt{\sum_{i=1}^{n}(x_i-\mu_i)^2}$$

  除此之外，还有曼哈顿距离，也被称作街道距离，该距离的计算方法相当于是欧式距离的1次方表示形式，其基本计算公式如下：

$$d(x,\mu)=\sum_{i=1}^{n}(|x_i-\mu|)$$

当然，不管是欧式距离还是曼哈顿距离，都可视为闵可夫斯基距离的一种特例，该距离计　算公式如下：

$$d(x,\mu)=\sqrt[n]{\sum_{i=1}^{n}(|x_i-\mu_i|)^n}$$

  当n为1时，就是曼哈顿距离，当n为2时即为欧式距离，当n趋于无穷时，即为切比雪夫距　离。

- **文本距离**

  而除了数据距离之外，距离的衡量还常见于文本数据之间，此时我们常用余弦相似性或杰卡德相似度来进行衡量，余弦相似性计算本质上就是计算余弦夹角，其基本计算公式如下：

$$cos\theta = \frac{\sum_1^n (x_i * \mu)}{\sqrt{\sum_1^n (x_i)^2} * \sqrt{\sum_1^n (\mu)^2}}$$

而杰卡德相似度则主要以集合运算为主：

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

2.2 误差平方和SSE（Sum of the Squared Error，SSE）

此处引入机器学习算法中非常重要的概念，误差平方和，也被称为组内误差平方和。该概念在聚类和回归算法中均有广泛应用。在聚类算法中所谓误差平方和是指每个数据点的误差，即它到最近所属类别质心的欧几里得距离，然后求和汇总既得误差平方和。在聚类算法中，SSE是我们判断模型是否最优的重要指标，我们希望求得的模型是在给定K值得情况下SSE最小的模型，即在相同的K值情况下聚类模型SSE越小越好，这也是聚类算法最核心的优化条件。

2.3 聚类目标函数和质心计算方法

聚类算法中的目标函数是实现聚类这一目标所使用的函数，如最小化对象到其所属类的质心距离等，这里需要注意，一般如果采用距离作为邻近度衡量标准，则目标函数往往是利用最小距离来构建聚类类别，而若采用相似度作为邻近度衡量标准，则目标函数就是利用最大化相似度和来构建聚类类别。

而质心是聚类类别的原型，一般可用均值或中位数来表示。但无论如何，在一旦确定距离衡量方法或邻近度函数（如欧式距离等），并在模型优化条件（SSE最小）的指导下，目标函数和质心选取方法都会随之确定，常用邻近度函数、质心和目标函数组合如下：

邻近度函数	质心计算方法	目标函数
欧几里得距离	均值	最小化每个样本点到质心的欧式距离之和
曼哈顿距离	中位数	最小化每个样本点到质心的曼哈顿距离之和
余弦距离	均值	最小化每个样本点到质心的余弦距离之和

而这一切实际上是建立在严格的数学理论推导的基础之上的，推导方法之一就是利用EM算法进行计算。首先，我们需要定义聚类算法中的符号集：

符号	指代对象
x	数据集对象/样本点
C_i	第 i 个簇
c_i	簇 C_i 的质心
c	所有点的质心
m_i	第 i 个簇中对象的个数
m	数据集中对象的个数
k	簇的个数

然后，误差平方和可由下列计算公式表示：

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} (c_i - x)^2$$

其中，C_i 是第 i 个簇，x 是 C_i 中的点，c_i 是第 i 个簇的质心。我们这里验证最常用的，当采用欧式距离时当且仅当质心为均值的时候才能够保证在目标函数的聚类过程中SSE保持最小。

要保证在给定K值情况下总SSE最小，则用梯度下降理论工具可将问题转化为为了保证总SSE最小，K的每一步取值，即K值每增加1，都要能够最大程度降低SSE，即在给定K值步长为1的情况下去找SSE下降速度最快的坡度（详见逻辑回归）。即簇的最小化SSE的最佳质心是簇中各点的均值。同样可类似证明采用曼哈顿距离时最佳质心选取方案是选取中位数，而当采用余弦夹角衡量相似度时，均值是最佳的质心计算方法。

接下来，我们在sklearn中实现K-Means快速聚类，并探索模型进一步优化方案及评估指标。

四、手写一个简单的Kmeans算法

这里我们不使用for循环来完成一个Kmeans的快速编写。网上和很多机器学习算法书中都给出了如何手写一个Kmeans，但是代码实现过于复杂，效率不高，不方便阅读。在此课程当中，会展示如何最大限度的在不用for循环的前提下，利用numpy, pandas的高效的功能来完成Kmeans算法。

一般在python数据清洗中，数据量大的情况下，for循环的方法会使的数据处理的过程特别慢，效率特别低。一个很好的解决方法就是使用numpy，pandas自带的高级功能，不仅可以使得代码效率大大提高，还可以使得代码方便理解阅读。这里在介绍用numpy，pandas来进行Kmeans算法的同时，也是带大家复习一遍numpy，pandas用法。

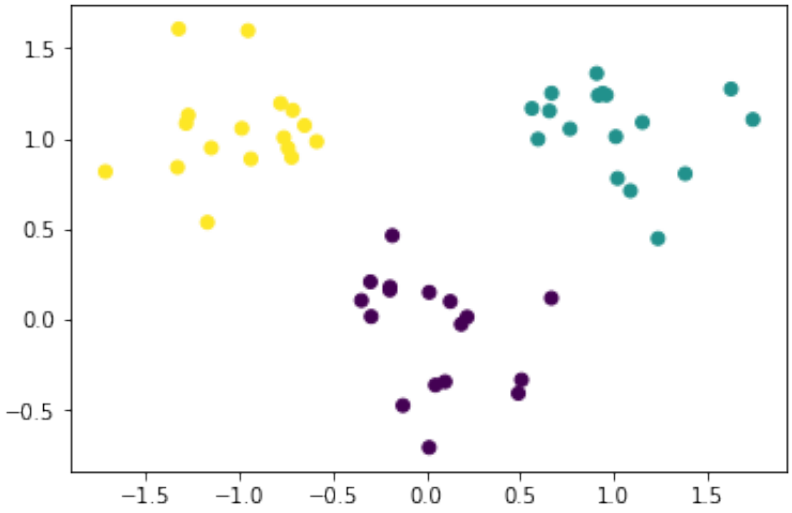
首先再来查看一下Kmeans的迭代步骤

创建k个点作为初始质心心（通常是随机选择）
当任意一个点的簇分配结果发生改变时：
    对数据集中的每个点：
      对每个质心：
          计算质心与数据点之间的距离
          将数据点分配到据其最近的簇
    对每个簇，计算簇中所有点的均值并将均值作为新的质心点
直到簇不再发生变化或者达到最大迭代次数

1. 生成一些数据

```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
import seaborn as sns
```

```
#r = np.random.randint(1,100)
r = 4
#print(r)
k = 3
x , y = make_blobs(n_samples = 51,
                   cluster_std = [0.3, 0.3, 0.3],
                   centers = [[0,0],[1,1],[-1,1]]
                   ,random_state = r
                  )
sim_data = pd.DataFrame(x, columns = ['x', 'y'])
sim_data['label'] = y
sim_data.head(5)

datasets = sim_data.copy()

plt.scatter(sim_data['x'], sim_data['y'], c = y)
```



上图是一个随机生成的2维的数据，可以用来尝试完成Kmeans的代码。

实际过程中，Kmeans需要能运行在多维的数据上，所以下面的代码部分，会考虑多维的数据集，而不是仅仅2维的数据。

3 随机生成数据点

这里的严格意义上不是随机的生成k个质心点，而是取出每个特征的最大值最小值，在最大值和最小值中取出一个随机数作为质心点的一个维度

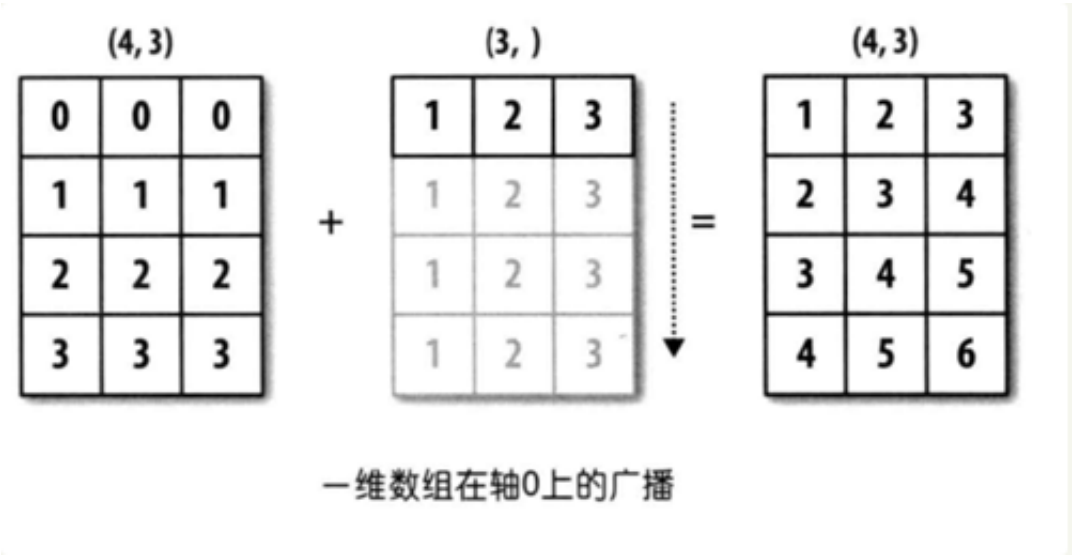
```
def initial_centers(datasets, k = 3):
    #首先将datasets的特征名取出来，这里需要除去label那一列
    cols = datasets.columns
    data_content = datasets.loc[:, cols != 'label']

    #直接用describe的方法将每一列的最小值最大值取出来
    range_info = data_content.describe().loc[['min','max']]

    #用列表解析的方法和np.random.uniform的方法生成k个随机的质心点
    #np.random.uniform(a, b, c) 随机生成在[a,b)区间里的3个数
    #对每个特征都做此操作
    k_randoms = [np.random.uniform(range_info[i]['min'],
                                   range_info[i]['max'], k)
                 for i in range_info.columns]
    centers = pd.DataFrame(k_randoms, index = range_info.columns)
    return centers.T
```

```
centers = initial_centers(data, k = 3)
centers
```

	x	y
0	0.122575	0.021762
1	-0.922596	1.367504
2	-0.677202	-0.411821

4 计算所有的点到所有中心点的距离

将每一个中心点取出来，然后使用pandas的广播的功能，可以直接将所有的实例和其中一个质心点相减。如下图，下图中是给出相加的例子，而我们的例子是减法。



所以对于一个DataFrame来说，比如说这里的只包含x和y的data，假设我们的质心是c = [1,1]，可以用以下的方式来给出所有的实例点的x和y和点(1,1)之间的差值。注意，这里的c可以是list，也可以是numpy array，甚至可以是元组。

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix} - [1,1] = \begin{bmatrix} x_1 - 1 & y_1 - 1 \\ x_2 - 1 & y_2 - 1 \\ x_3 - 1 & y_3 - 1 \end{bmatrix}$$

算出每个实例的每个特征和质心点的差距之后，则需要将所有的数平方一下，然后按每一行加起来则给出了每一个实例点到质心的距离了

$$\begin{bmatrix} (x_1 - 1)^2 + (y_1 - 1)^2 \\ (x_2 - 1)^2 + (y_2 - 1)^2 \\ (x_3 - 1)^2 + (y_3 - 1)^2 \end{bmatrix}$$

用的方法就是使用np.power(data - c, 2).sum(axis = 1)

```
def cal_distant(dataset, centers):
    #选出不是label的那些特征列
    data = dataset.loc[:, dataset.columns != 'label']

    #使用列表解析式的格式，对centers表里的每一行也就是每一个随机的质心点，都算一遍所有的点到该质心点的距离，并且存入一个list中
    d_to_centers = [np.power(data - centers.loc[i], 2).sum(axis = 1)
                    for i in centers.index]

    #所有的实例点到质心点的距离都已经存在了list中，则可以直接带入pd.concat里面将数据拼起来
    return pd.concat(d_to_centers, axis = 1)
```

```
d_to_centers = cal_distant(data, centers)
d_to_centers.head(5)
```

	0	1	2
0	0.153365	3.935546	0.528286
1	1.987879	0.088006	2.462444
2	0.027977	2.361753	0.795004
3	0.543410	5.183283	0.565696
4	1.505514	2.248264	4.031165

5 找出最近的质心点

当每个实例点都和中心点计算好距离后，对于每个实例点找出最近的那个中心点,可以用np.where的方法，但是pandas已经提供更加方便的方法，用idxmin和idxmax,这2个函数可以直接给出DataFrame每行或者每列的最小值和最大值的索引，设置axis = 1则是想找出对每个实例点来说，哪个质心点离得最近。

```
curr_group = d_to_centers.idxmin(axis=1)
```

这个时候，每个点都有了新的group，这里我们则需要开始更新我们的3个中心点了。对每一个临时的簇来说，算出X的平均， 和Y的平均，就是这个临时的簇的中心点。

6 重新计算新的质心点

```
centers = data.loc[:, data.columns != 'label'].groupby(curr_group).mean()
centers
```

	x	y
0	0.548468	0.523474
1	-1.003680	1.044955
2	-0.125490	-0.475373

7 迭代

这样我们新的质心点就得到了，只是这个时候的算法还是没有收敛的，需要将上面的步骤重复多次。

Kmeans代码迭代部分就完成了，将上面的步骤做成一个函数，做成函数后，方便展示Kmeans的中间过程。

```
def iterate(dataset, centers):
    #计算所有的实例点到所有的质心点之间的距离
    d_to_centers = cal_distant(dataset, centers)

    #得出每个实例点新的类别
    curr_group = d_to_centers.idxmin(axis=1)

    #算出当前新的类别下每个簇的组内误差
    SSE = d_to_centers.min(axis = 1).sum()

    #给出在新的实例点类别下，新的质心点的位置
    centers = dataset.loc[:, dataset.columns !=
'label'].groupby(curr_group).mean()
    return curr_group, SSE, centers
```

```
curr_group, SSE, centers = iterate(data,centers)
```

```
centers, SSE
```

```
(          x         y
 0  0.892579  0.931085
 1 -1.003680  1.044955
 2  0.008740 -0.130172, 19.041432436034352)
```

最后需要判断什么时候迭代停止，可以判断SSE差值不变的时候，算法停止

```
#创建一个空的SSE_list,用来存SSE的，第一个位置的数为0，无意义，只是方便收敛时最后一个SSE和上一个SSE的对比
SSE_list = [0]
```

```
#初始化质心点
centers = initial_centers(data, k = 3)

#开始迭代
while True:
    #每次迭代中得出新的组，组内误差，和新的质心点，当前的新的质心点会被用于下一次迭代
    curr_group, SSE, centers = iterate(data,centers)

    #检查这一次算出的SSE和上一次迭代的SSE是否相同，如果相同，则收敛结束
    if SSE_list[-1] == SSE:
        break

    #如果不相同，则记录SSE，进入下一次迭代
    SSE_list.append(SSE)
```

```
SSE_list
```

```
[0, 37.86874675507244, 11.231524142566894, 8.419267088238051]
```

8 代码整合

算法完成了，将所有的代码整合在一起

```
def initial_centers(datasets, k = 3):
    cols = datasets.columns
    data_content = datasets.loc[:, cols != 'label']
    range_info = data_content.describe().loc[['min','max']]
    k_randoms = [np.random.uniform(range_info[i]['min'],
                                   range_info[i]['max'], k)
                 for i in range_info.columns]
    centers = pd.DataFrame(k_randoms, index = range_info.columns)
    return centers.T

def cal_distant(dataset, centers):
    data = dataset.loc[:, dataset.columns != 'label']
    d_to_centers = [np.power(data - centers.loc[i], 2).sum(axis = 1)
                    for i in centers.index]
    return pd.concat(d_to_centers, axis = 1)

def iterate(dataset, centers):
    d_to_centers = cal_distant(dataset, centers)
    curr_group = d_to_centers.idxmin(axis=1)
    SSE = d_to_centers.min(axis = 1).sum()
    centers = dataset.loc[:, dataset.columns !=
'label'].groupby(curr_group).mean()
    return curr_group, SSE, centers
```

```
def Kmeans_regular(data, k = 3):
    SSE_list = [0]
    centers = initial_centers(data, k = k)

    while True:
        curr_group, SSE, centers = iterate(data,centers)
        if SSE_list[-1] == SSE:
            break
        SSE_list.append(SSE)
    return curr_group, SSE_list, centers
```

上面的函数已经完成，当然这里推荐大家尽量写成class的形式更好，这里为了方便观看，则用简单的函数完成。

最后的函数是Kmeans_regular函数，这个函数里面包含了上面所有的函数。现在需要测试Kmeans_regular代码对于多特征的数据集鸢尾花数据集，是否也能进行Kmeans聚类算法。**注意这里数据集是有标签的，我们只是尝试来看下Kmeans给出的结果是否符合标签。这里不是在做有监督的学习，没有去学习X与Y之间的关系，而是只是学了X。**

```
from sklearn.datasets import load_iris
data_dict = load_iris()
iris = pd.DataFrame(data_dict.data, columns = data_dict.feature_names)
iris['label'] = data_dict.target
```

```
curr_group, SSE_list, centers = Kmeans_regular(iris.copy(), k = 3)
```

```
np.array(SSE_list)
```

```
array([  0.        , 589.73485975, 115.8301874 ,  83.29216169,
        79.45325846,  78.91005674,  78.85144143])
```

```
pd.crosstab(iris['label'], curr_group)
```

col_0	0	1	2
label			
0	50	0	0
1	0	48	2
2	0	14	36

```
np.diag(pd.crosstab(iris['label'], curr_group)).sum() /  iris.shape[0]
```

```
0.8933333333333333
```

最后可以看出我们的代码是可以适用于多特征变量的数据集，并且对于鸢尾花数据集来说，对角线上的数是预测正确的个数，准确率大约为90%。

额外内容:查看另外一个Kmeans中间过程可视化的HTML文档来了解，Kmeans的中间过程。

五、使用sklearn实现K-Means

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'*)

1. 重要参数：n_clusters：要分几类才比较好?

n_clusters是K-Means中的k，表示着我们告诉模型我们要分几类。这是K-Means当中唯一一个必填的参数，默认为8类，但通常我们的聚类结果会是一个小于8的结果。通常，在开始聚类之前，我们并不知道n_clusters究竟是多少，因此我们要对它进行探索。

当我们拿到一个数据集，如果可能的话，我们希望能够通过绘图先观察一下这个数据集的数据分布，以此来为我们聚类时输入的n_clusters做一个参考。

首先，我们再来创建一个数据集。这样的数据集是我们自己创建，所以是有标签的。

```
from sklearn.datasets import make_blobs

#自己创建数据集
X, y = make_blobs(n_samples=500,n_features=2,centers=4,random_state=1)

plt.scatter(X[:, 0], X[:, 1]
            ,marker='o' #点的形状
            ,s=8 #点的大小
            ,c=y
           )
data = pd.DataFrame(X)
data['label'] = y
```

基于这个分布，我们来使用K-Means进行聚类。首先，我们要猜测一下，这个数据中有几簇?

```
from sklearn.cluster import KMeans

n_clusters = 3

cluster = KMeans(n_clusters=n_clusters, random_state=0).fit(X)

#重要属性labels_，查看聚好的类别，每个样本所对应的类
y_pred = cluster.labels_
y_pred
```

2. 聚类算法的模型评估指标：轮廓系数

不同于分类模型和回归，聚类算法的模型评估不是一件简单的事。在分类中，有直接结果（标签）的输出，并且分类的结果有正误之分，所以我们使用预测的准确度，混淆矩阵，ROC曲线等等指标来进行评估，但无论如何评估，都是在"模型找到正确答案"的能力。而回归中，由于要拟合数据，我们有MSE均方误差，有损失函数来衡量模型的拟合程度。但这些衡量指标都不能够适用于聚类。

> **面试高危问题：如何衡量聚类算法的效果?**
>
> 聚类模型的结果不是某种标签输出，并且聚类的结果是不确定的，其优劣由业务需求或者算法需求来决定，并且没有永远的正确答案。那我们如何衡量聚类的效果呢?

记得我们说过，K-Means的目标是确保"簇内差异小，簇外差异大"，我们就可以通过**衡量簇内差异来衡量聚类的效果**。我们刚才说过，SSE是用距离来衡量簇内差异的指标，因此，我们是否可以使用SSE来作为聚类的衡量指标呢？SSE越小模型越好嘛。

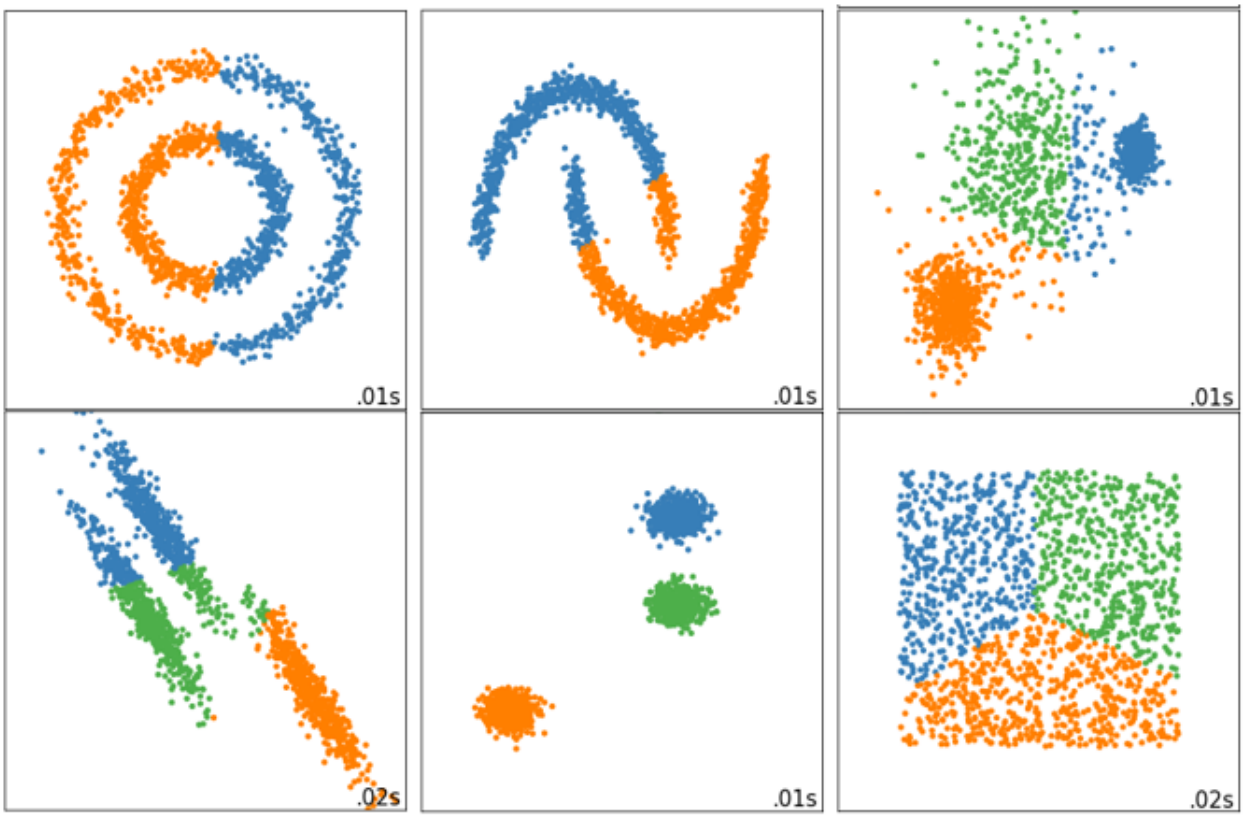
可以，但是这个指标的缺点和极限太大。

首先，它不是有界的。我们只知道，SSE是越小越好，是0最好，但我们不知道，一个较小的SSE究竟有没有达到模型的极限，能否继续提高。

第二，它的计算太容易受到特征数目的影响，数据维度很大的时候，SSE的计算量会陷入维度诅咒之中，计算量会爆炸，不适合用来一次次评估模型。

第三，它会受到超参数K的影响，在我们之前的尝试中其实我们已经发现，随着K越大，SSE注定会越来越小，但这并不代表模型的效果越来越好了。

第四，SSE对数据的分布有假设，它假设数据满足凸分布（即数据在二维平面图像上看起来是一个凸函数的样子），并且它假设数据是各向同性的（isotropic），即是说数据的属性在不同方向上代表着相同的含义。但是现实中的数据往往不是这样。所以使用Inertia作为评估指标，会让聚类算法在一些细长簇，环形簇，或者不规则形状的流形时表现不佳：



那我们可以使用什么指标呢？聚类是没有标签，即不知道真实答案的预测算法，我们必须完全依赖评价簇内的稠密程度（簇内差异小）和簇间的离散程度（簇外差异大）来评估聚类的效果。其中轮廓系数是最常用的聚类算法的评价指标。它是对每个样本来定义的，它能够同时衡量：

1）样本与其自身所在的簇中的其他样本的相似度**a**，等于样本与同一簇中所有其他点之间的平均距离

2）样本与其他簇中的样本的相似度**b**，等于样本与下一个最近的簇中的所有点之间的平均距离

根据聚类的要求 ”簇内差异小，簇外差异大“，我们希望b永远大于a，并且大得越多越好

单个样本的轮廓系数计算为：

$$s = \frac{b-a}{max(a,b)}$$

$$s = \begin{cases} 1-a/b, & if\ a < b \\ 0, & if\ a = b \\ b/a-1, & if\ a > b \end{cases}$$

很容易理解轮廓系数范围是(-1,1)，其中值越接近1表示样本与自己所在的簇中的样本很相似，并且与其他簇中的样本不相似，当样本点与簇外的样本更相似的时候，轮廓系数就为负。当轮廓系数为0时，则代表两个簇中的样本相似度一致，两个簇本应该是一个簇。

如果一个簇中的大多数样本具有比较高的轮廓系数，则簇会有较高的总轮廓系数，则整个数据集的平均轮廓系数越高，则聚类是合适的。如果许多样本点具有低轮廓系数甚至负值，则聚类是不合适的，聚类的超参数K可能设定得太大或者太小。

在sklearn中，我们使用模块metrics中的类silhouette_score来计算轮廓系数，它返回的是一个数据集中，所有样本的轮廓系数的均值。但我们还有同在metrics模块中的silhouette_sample，它的参数与轮廓系数一致，但返回的是数据集中每个样本自己的轮廓系数。

我们来看看轮廓系数在我们自建的数据集上表现如何：

```
from sklearn.metrics import silhouette_score
from sklearn.metrics import silhouette_samples

X
y_pred

silhouette_score(X,y_pred)

silhouette_score(X,cluster_.labels_)

#观察一下不同的k下，轮廓系数发生什么变化?
n_clusters = 5
cluster_ = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
silhouette_score(X,cluster_.labels_)

silhouette_samples(X,y_pred)
```

轮廓系数有很多优点，它在有限空间中取值，使得我们对模型的聚类效果有一个“参考”。并且，轮廓系数对数据的分布没有假设，因此在很多数据集上都表现良好。但它在每个簇的分割比较清洗时表现最好。但轮廓系数也有缺陷，它在凸型的类上表现会虚高，比如基于密度进行的聚类，或通过DBSCAN获得的聚类结果，如果使用轮廓系数来衡量，则会表现出比真实聚类效果更高的分数。

3. 案例：基于轮廓系数来选择n_clusters

我们通常会绘制轮廓系数分布图和聚类后的数据分布图来选择我们的最佳n_clusters。

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

#先设定我们要分成的簇数
n_clusters = 4
```

```
#创建一个画布，画布上共有一行两列两个图
fig, (ax1, ax2) = plt.subplots(1, 2)

#画布尺寸
fig.set_size_inches(18, 7)

# 第一个图是我们的轮廓系数图像，是由各个簇的轮廓系数组成的横向条形图
# 横向条形图的横坐标是我们的轮廓系数取值，纵坐标是我们的每个样本，因为轮廓系数是对于每一个样本进行计算的

# 首先我们来设定横坐标
# 轮廓系数的取值范围在[-1,1]之间，但我们至少是希望轮廓系数要大于0的
# 太长的横坐标不利于我们的可视化，所以只设定x轴的取值在[-0.1,1]之间
ax1.set_xlim([-0.1, 1])

# 接下来设定纵坐标，通常来说，纵坐标是从0开始，最大值取到X.shape[0]的取值
# 但我们希望，每个簇能够排在一起，不同的簇之间能够有一定的空隙
# 以便我们看到不同的条形图聚合成的块，理解它是对应了哪一个簇
# 因此我们在设定纵坐标的取值范围的时候，在X.shape[0]上，加上一个距离(n_clusters + 1) * 10，留作间隔用
ax1.set_ylim([0, X.shape[0] + (n_clusters + 1) * 10])

# 开始建模，调用聚类好的标签
clusterer = KMeans(n_clusters=n_clusters, random_state=10).fit(X)
cluster_labels = clusterer.labels_

# 调用轮廓系数分数，注意，silhouette_score生成的是所有样本点的轮廓系数均值
# 两个需要输入的参数是，特征矩阵x和聚类完毕后的标签
silhouette_avg = silhouette_score(X, cluster_labels)
#用print来报一下结果，现在的簇数量下，整体的轮廓系数究竟有多少
print("For n_clusters =", n_clusters,
      "The average silhouette_score is :", silhouette_avg)

# 调用silhouette_samples，返回每个样本点的轮廓系数，这就是我们的横坐标
sample_silhouette_values = silhouette_samples(X, cluster_labels)

#设定y轴上的初始取值
y_lower = 10

#接下来，对每一个簇进行循环
for i in range(n_clusters):
    # 从每个样本的轮廓系数结果中抽取出第i个簇的轮廓系数，并对他进行排序
    ith_cluster_silhouette_values = sample_silhouette_values[cluster_labels == i]

    #注意，.sort()这个命令会直接改掉原数据的顺序
    ith_cluster_silhouette_values.sort()

    #查看这一个簇中究竟有多少个样本
```

```
    size_cluster_i = ith_cluster_silhouette_values.shape[0]

    #这一个簇在y轴上的取值，应该是由初始值(y_lower)开始，到初始值+加上这个簇中的样本数量结束(y_upper)
    y_upper = y_lower + size_cluster_i

    #colormap库中的，使用小数来调用颜色的函数
    #在nipy_spectral([输入任意小数来代表一个颜色])
    #在这里我们希望每个簇的颜色是不同的，我们需要的颜色种类刚好是循环的个数的种类
    #在这里，只要能够确保，每次循环生成的小数是不同的，可以使用任意方式来获取小数
    #在这里，我是用i的浮点数除以n_clusters，在不同的i下，自然生成不同的小数
    #以确保所有的簇会有不同的颜色
    color = cm.nipy_spectral(float(i)/n_clusters)

    #开始填充子图1中的内容
    #fill_between是填充曲线与直角之间的空间的函数
    #fill_betweenx的直角是在纵坐标上
    #fill_betweeny的直角是在横坐标上
    #fill_betweenx的参数应该输入(定义曲线的点的横坐标，定义曲线的点的纵坐标，柱状图的颜色)
    ax1.fill_betweenx(np.arange(y_lower, y_upper)
                      ,ith_cluster_silhouette_values
                      ,facecolor=color
                      ,alpha=0.7
                     )

    #为每个簇的轮廓系数写上簇的编号，并且让簇的编号显示坐标轴上每个条形图的中间位置
    #text的参数为(要显示编号的位置的横坐标，要显示编号的位置的纵坐标，要显示的编号内容)
    ax1.text(-0.05
             , y_lower + 0.5 * size_cluster_i
             , str(i))

    # 为下一个簇计算新的y轴上的初始值，是每一次迭代之后，y的上线再加上10
    #以此来保证，不同的簇的图像之间显示有空隙
    y_lower = y_upper + 10

#给图1加上标题，横坐标轴，纵坐标轴的标签
ax1.set_title("The silhouette plot for the various clusters.")
ax1.set_xlabel("The silhouette coefficient values")
ax1.set_ylabel("Cluster label")

#把整个数据集上的轮廓系数的均值以虚线的形式放入我们的图中
ax1.axvline(x=silhouette_avg, color="red", linestyle="--")

#让y轴不显示任何刻度
ax1.set_yticks([])

#让x轴上的刻度显示为我们规定的列表
ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
```

```
#开始对第二个图进行处理，首先获取新颜色，由于这里没有循环，因此我们需要一次性生成多个小数来获取多个颜色
colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)

ax2.scatter(X[:, 0], X[:, 1]
            ,marker='o' #点的形状
            ,s=8 #点的大小
            ,c=colors
           )

#把生成的质心放到图像中去
centers = clusterer.cluster_centers_
# Draw white circles at cluster centers
ax2.scatter(centers[:, 0], centers[:, 1], marker='x',
            c="red", alpha=1, s=200)

#为图二设置标题，横坐标标题，纵坐标标题
ax2.set_title("The visualization of the clustered data.")
ax2.set_xlabel("Feature space for the 1st feature")
ax2.set_ylabel("Feature space for the 2nd feature")

#为整个图设置标题
plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "
              "with n_clusters = %d" % n_clusters),
             fontsize=14, fontweight='bold')
plt.show()
```

将上述过程包装成一个循环，可以得到：

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

for n_clusters in [2,3,4,5,6,7]:
    n_clusters = n_clusters
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)
    ax1.set_xlim([-0.1, 1])
    ax1.set_ylim([0, X.shape[0] + (n_clusters + 1) * 10])
    clusterer = KMeans(n_clusters=n_clusters, random_state=10).fit(X)
    cluster_labels = clusterer.labels_
    silhouette_avg = silhouette_score(X, cluster_labels)
    print("For n_clusters =", n_clusters,
          "The average silhouette_score is :", silhouette_avg)
    sample_silhouette_values = silhouette_samples(X, cluster_labels)
    y_lower = 10
```

```
    for i in range(n_clusters):
        ith_cluster_silhouette_values =
sample_silhouette_values[cluster_labels == i]
        ith_cluster_silhouette_values.sort()
        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i
        color = cm.nipy_spectral(float(i)/n_clusters)
        ax1.fill_betweenx(np.arange(y_lower, y_upper)
                          ,ith_cluster_silhouette_values
                          ,facecolor=color
                          ,alpha=0.7
                         )
        ax1.text(-0.05
                 , y_lower + 0.5 * size_cluster_i
                 , str(i))
        y_lower = y_upper + 10

    ax1.set_title("The silhouette plot for the various clusters.")
    ax1.set_xlabel("The silhouette coefficient values")
    ax1.set_ylabel("Cluster label")
    ax1.axvline(x=silhouette_avg, color="red", linestyle="--")
    ax1.set_yticks([])
    ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

    colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
    ax2.scatter(X[:, 0], X[:, 1]
                ,marker='o'
                ,s=8
                ,c=colors
               )
    centers = clusterer.cluster_centers_
    # Draw white circles at cluster centers
    ax2.scatter(centers[:, 0], centers[:, 1], marker='x',
                c="red", alpha=1, s=200)

    ax2.set_title("The visualization of the clustered data.")
    ax2.set_xlabel("Feature space for the 1st feature")
    ax2.set_ylabel("Feature space for the 2nd feature")

    plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "
                  "with n_clusters = %d" % n_clusters),
                 fontsize=14, fontweight='bold')
    plt.show()
```

4. 重要参数init & random_state & n_init：初始质心怎么放好?

在K-Means中有一个重要的环节，就是放置初始质心。如果有足够的时间，K-means一定会收敛，但Inertia可能收敛到局部最小值。是否能够收敛到真正的最小值很大程度上取决于质心的初始化。init就是用来帮助我们决定初始化方式的参数。

初始质心放置的位置不同，聚类的结果很可能也会不一样，一个好的质心选择可以让K-Means避免更多的计算，让算法收敛稳定且更快。在之前讲解初始质心的放置时，我们是使用”随机“的方法在样本点中抽取k个样本作为初始质心，这种方法显然不符合”稳定且更快“的需求。为此，我们可以使用random_state参数来控制每次生成的初始质心都在相同位置，甚至可以画学习曲线来确定最优的random_state是哪个整数。

一个random_state对应一个质心随机初始化的随机数种子。如果不指定随机数种子，则sklearn中的K-Means并不会只选择一个随机模式扔出结果，而会在每个随机数种子下运行多次，并使用结果最好的一个随机数种子来作为初始质心。我们可以使用参数n_init来选择，每个随机数种子下运行的次数。这个参数不常用到，默认10次，如果我们希望运行的结果更加精确，那我们可以增加这个参数n_init的值来增加每个随机数种子下运行的次数。

然而这种方法依然是基于随机性的。

为了优化选择初始质心的方法，2007年Arthur, David, and Sergei Vassilvitskii三人发表了论文*“k-means++: The advantages of careful seeding”*，他们开发了”k-means ++“初始化方案，使得初始质心（通常）彼此远离，以此来引导出比随机初始化更可靠的结果。在sklearn中，我们使用参数init ='k-means ++'来选择使用k-means ++作为质心初始化的方案。通常来说，我建议保留默认的"k-means++"的方法。

init：可输入"k-means++"，"random"或者一个n维数组

初始化质心的方法，默认"k-means++"

输入"k-means++"：一种为K均值聚类选择初始聚类中心的聪明的办法，以加速收敛

如果输入了n维数组，数组的形状应该是(n_clusters，n_features)并给出初始质心

random_state：控制每次质心随机初始化的随机数种子

n_init：整数，默认10

使用不同的质心随机初始化的种子来运行k-means算法的次数。最终结果会是基于Inertia来计算的n_init次连续运行后的最佳输出。

```
X
y

plus = KMeans(n_clusters = 10).fit(X)
plus.n_iter_

random = KMeans(n_clusters = 10,init="random",random_state=420).fit(X)
random.n_iter_
```

5. 重要参数max_iter & tol：让迭代停下来

在之前描述K-Means的基本流程时我们提到过，当质心不再移动，K-Means算法就会停下来。但在完全收敛之前，我们也可以使用max_iter，最大迭代次数，或者tol，两次迭代间Inertia下降的量，这两个参数来让迭代提前停下来。有时候，当我们的n_clusters选择不符合数据的自然分布，或者我们为了业务需求，必须要填入与数据的自然分布不合的n_clusters，提前让迭代停下来反而能够提升模型的表现。

参数	说明
max_iter	整数，默认300 单次运行K-Means允许的最大迭代次数
tol	浮点数，默认1e-4 两次迭代间Inertia下降的量，如果两次迭代之间Inertia下降的值小于tol所设定的值，迭代就会停下

```
random = KMeans(n_clusters =
                10,init="random",max_iter=10,random_state=420).fit(X)
y_pred_max10 = random.labels_
silhouette_score(X,y_pred_max10)

random = KMeans(n_clusters =
                10,init="random",max_iter=20,random_state=420).fit(X)
y_pred_max20 = random.labels_
silhouette_score(X,y_pred_max20)
```

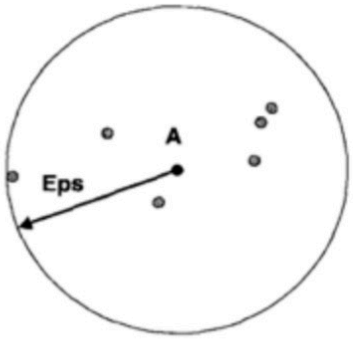
六、基于密度的聚类算法：DBSCAN

在之前的聚类分析学习中，我们主要了解了K-Means算法，包括其基本原理、背后蕴藏的机器学习思维，以及手动Python实现。K-Means是最常见的聚类方法，能够实现快速聚类，但局限于其算法构架，K-Means对非球形边界的数据很难达到一个较好的分类效果，因此我们需要借助另外的数学理论工具对其进行进一步的完善。这里介绍一种利用密度进行聚类的方法：DBSCAN。一方面作为聚类算法的补充内容，一方面需要掌握面对一个新算法如何利用scikit-learn快速上手应用实践的方法，当然最重要的一点，是要树立关于密度的基本认知，为后续学习核函数做铺垫。

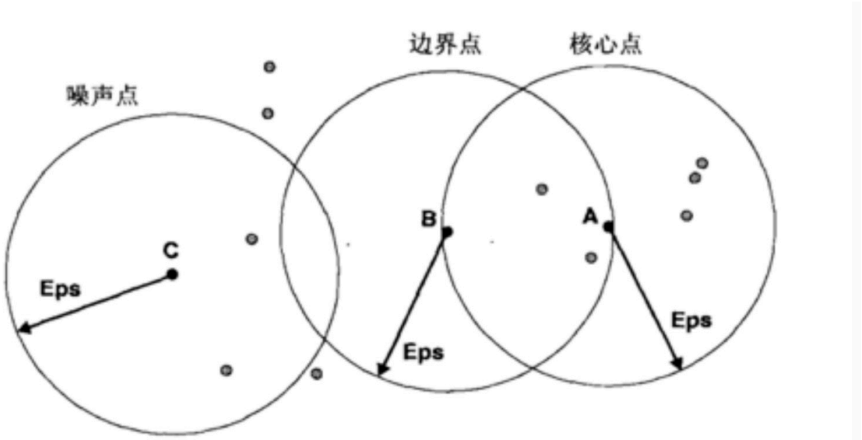
1. DBSCAN基本原理

DBSCAN是一种基于密度的聚类方法，旨在寻找被低密度区域分离的高密度区域。DBSCAN是一种简单、有效的基于密度的聚类算法，且包含了基于密度的许多方法中的重要概念。尽管定义密度的方法没有定义相似度的方法多，但仍存在几种不同的方法，本部分中，我们讨论DBSCAN使用的基于中心的定义是否相似的方法。

在基于中心的方法中，数据集中特定点的密度通过对该店*Eps*半径之内的点计数（包括点本身）来估计，如下图所示。点A的*Eps*半径内点的个数为7，包括A本身。该方法实现简单，但是点的密度取决于指定的半径。例如，如果半径足够大，则所有点的密度都等于数据集中的点数*m*。同理，如果半径太小，则所有点的密度都是1。对于低维数据，一种确定合适半径的方法在讨论DBSCAN算法时给出。



密度的基于中心的方法使得我们可以将点分类为（1）稠密区域内部的点（核心点），（2）稠密区域边缘上的点（边界点），（3）稀疏区域中的点（噪声或背景点）。下图使用二维点集图示了核心点、边界点和噪声点的概念。下文给出更详尽的描述。



- 核心点（core point）：这些点在基于密度的簇内部。点的领域由距离函数和用户指定的距离参数*Eps*决定。核心点的定义是，如果该点的给定领域内的点的个数超过给定的阈值*MinPts*，其中*MinPts*也是一个用户指定的参数。在上图中，如果*MinPts*≤7，则对于给定的半径（*Eps*），点A是核心点。
- 边界点（border point）：边界点不是核心点，但它落在某个核心点的领域内。在上图中，点B是边界点。边界点可能落在多个核心点的领域内。
- 噪声点（noise point）：噪声点是既非核心点也非核心点也非边界点的任何点。在上图中，点C是噪声点。

2. DBSCAN算法执行过程

给定核心点、边界点和噪声点的定义，DBSCAN算法可以非正式地描述如下。任意两个足够靠近（相互之间的距离在*Eps*之内）的核心点将放在同一个簇中。同样，任何与核心点足够靠近的边界点也放到与核心点相同的簇中。（如果一个边界点靠近不同簇的核心点，则可能需要解决平局问题。）噪声点被丢弃。算法的细节在伪代码中给出。

聚类分析：DBSCAN算法

---

1. 将所有点标记为核心点、边界点或噪声点
2. 删除噪声点
3. 为距离在*Eps*之内的所有核心点之间赋予一条边
4. 每个彼此联通的核心点组成一个簇
5. 将每个边界点指派到一个与之关联的核心点的簇当中

---

3. Scikit-Learn实践

接下来，利用Scikit-Learn快速实践DBSCAN算法。

- 生成数据集

  为了更好的对比K-Means快速聚类和DBSCAN聚类，此处利用sklearn生成一个非线性边界的数据集。这里需要注意，sklearn提供了非常丰富的生成数据的方法，用以在无外部数据集的情况下测试模型性能，该功能主要由`datasets`模块提供。此处利用该模块的make_moons方法生成非线性边界的数据集。

  ```
  from sklearn.datasets import make_moons
  X,y = make_moons(200, noise = 0.05, random_state=0)
  ```

  此时生成了一个二维数组数据集X，数据集形如弯月，且内部每个元素根据初始簇设置有对应的标签，该初始簇归属情况由原始质心划分方法决定。数据集基本情况如下所示：

  ```
  plt.scatter(X[:,0],X[:,1], c=y)
  ```

- K_Means聚类实验

  该数据集用K_Means聚类很难捕捉到原始数据集的非线性边界下的自然结构。sklearn的聚类算法保存在cluster模块中，我们首先尝试使用K-Means对其进行聚类处理。

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
```

这里需要注意，虽然K-Means是无监督学习，但在sklearn中该算法也作为评估器而存在，因此，使用方法也要遵照评估器的一般使用规则。在模型训练完成后，可使用模型的`labels_`属性查看各训练数据所属簇的划分情况，也可调用predict方法在对原始数据进行预测，从而生成各个点的所属簇标签。

```
labels = kmeans.labels_
kmeans.predict(X)
```

同时模型`cluster_centers_`属性保存了聚类结果的中心点：

```
centers = kmeans.cluster_centers_
```

然后利用可视化图形查看最终聚类结果：

```
plt.scatter(X[:,0],X[:,1], c = y)
plt.plot(centers[:,0], centers[:,1],'ro')
```

很明显K-Means并没有准确的捕捉数据集背后客观分类规律，接下来尝试用DBSCAN进行聚类。

- DBSCAN聚类实践

  接下来，尝试使用DBSCAN对上述数据集进行聚类，在调用sklearn中DBSCAN算法时，需要设置eps半径长度以及核心点的判别条件。

  ```
  from sklearn.cluster import DBSCAN
  db = DBSCAN(eps=0.3, min_samples=10)
  db.fit(X)
  ```

  模型训练完成后，数据集各点所属类同样保存在db.labels_属性中：

  ```
  db.labels_
  ```

  然后利用图形可视化展示聚类结果：

  ```
  plt.scatter(X[:,0],X[:,1], c = db.labels_)
  ```

  能够看出，DBSCAN成功捕捉数据集分类结构。