机器学习-聚类算法

2021年4月6日 11:20

参考：
基础聚类 https://www.jianshu.com/p/8890ccdfaf6c
其他聚类 https://www.jianshu.com/p/8890ccdfaf6c

基础聚类

这里将介绍无监督学习算法，也就是聚类算法。在无监督学习中，目标属性是不存在的，也就是所说的不存在“y”值，我们是根据内部存在的数据特征，划分不同的类别，使得类别内的数据比较相似。
我们对数据进行聚类的思想不同可以设计不同的聚类算法，本章主要谈论三种聚类思想以及该聚类思想下的三种聚类算法。
本章主要涉及到的知识点有：

- **“距离”**
- **K-Means算法**
- **几种优化K-Means算法**
- **密度聚类**

算法思想：“物以类聚，人以群分”

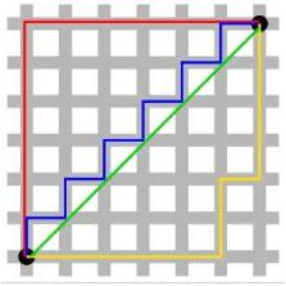
本节首先通过聚类算法的基本思想，引出样本相似度这个概念，并且介绍几种基本的样本相识度方法。

如何将数据划分不同类别

通过计算样本之间的相识度，将相识度大的划分为一个类别。衡量样本之间的相识度的大小的方式有下面几种：

闵可夫斯基距离（Minkowski距离）也就是前面提到的范式距离
当p=1时为曼哈顿距离，公式如下（以二维空间为例）：

$$d = |x_1 - x_2| + |y_1 - y_2|$$



当p=2时，为欧几里得距离，公式如下：

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

当p=为无穷大时候，为切比雪夫距离，公式如下：

$$d = \max(|x_1 - x_2|, |y_1 - y_2|)$$

一般情况下用欧几里得距离比较多，当数据量出现扁平化时候，一般用切夫雪比距离。

夹角余弦相识度

假设两个样本有2个特征，

则这两个样本的夹角余弦相似度公式如下：

$$\cos(\theta) = \frac{a^T * b}{|a||b|}$$

最常见的应用就是计算文本相似度。将两个文本根据他们词，建立两个向量，计算这两个向量的余弦值，就可以知道两个文本在统计学方法中他们的相似度情况。实践证明，这是一个非常有效的方法。

杰卡德相似系数（Jaccard）

适用于样本只有（0,1）的情况，又叫二元相似性，计算公式如下：

$$dist(A,B) = 1 - J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

将杰卡德相似性度量应用到基于物品的协同过滤系统中，并建立起相应的评价分析方法。 与传统相似性度量方法相比，杰卡德方法完善了余弦相似性只考虑用户评分而忽略了其他信息量的弊端，特别适合于应用到稀疏度过高的数据。

类别的定义：簇

前面我们讲到把数据划分为不同类别，机器学习给这个类别定义一个新的名字—簇。

将具有M个样本的数据换分为k个簇，必然k<=M。簇满足以下条件：

1. q 每个簇至少包含一个对象
2. q 每个对象属于且仅属于一个簇
3. q 将上述条件的k个簇成为一个合理的聚类划分

对于给定的类别数目k,首先给定初始划分,通过迭代改变样本和簇的隶属关系,使的每次处理后得到的划分方式比上一次的好(总的数据集之间的距离和变小了）。

下面介绍一种最常用的一种最基本的算法—K-Means算法

K-Means算法

K- means算法,也称为K-平均或者K-均值,是一种使用广泛的最基础的聚类算法,一般作为掌握聚类算法的第一个算法。

K-Means构建步骤

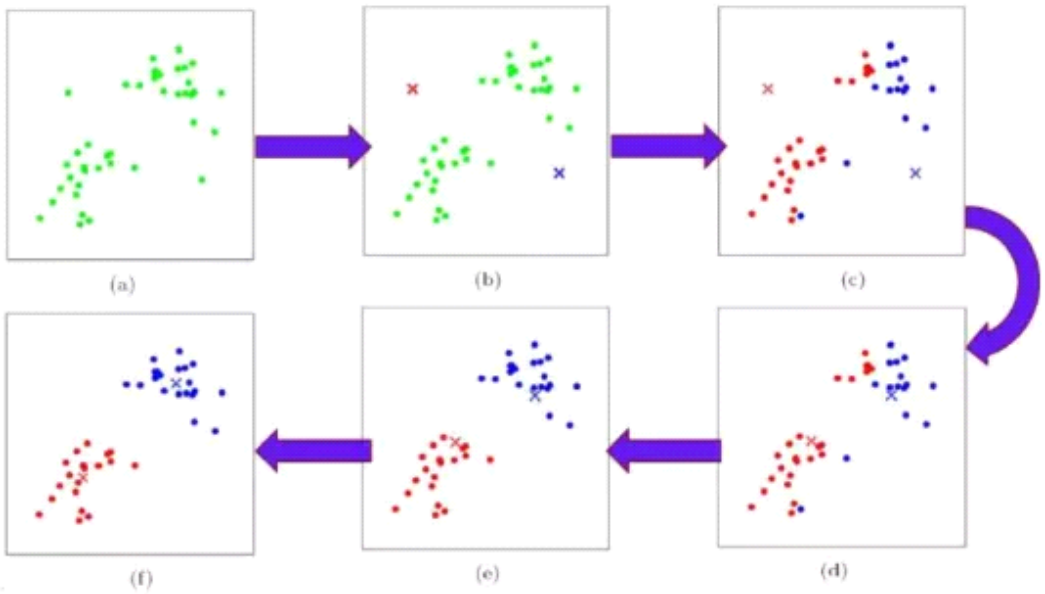
- K-means算法，也称为K-平均或者K-均值，是一种使用广泛的最基础的聚类算法，一般作为掌握聚类算法的第一个算法
- 假设输入样本为T=$X_1,X_2,...,X_m$;则算法步骤为（使用欧几里得距离公式）：
  - 选择初始化的k个类别中心$a_1,a_2,...a_k$;
  - 对于每个样本X_i，将其标记位距离类别中心a_j最近的类别j
  - 更新每个类别的中心点a_j为隶属该类别的所有样本的均值
  - 重复上面两步操作，直到达到某个中止条件
- 中止条件：
  - 迭代次数、最小平方误差MSE、簇中心点变化率

$$label_i = \arg\min_{1<=j<=k}\left\{\sqrt{\sum_{i=1}^{n}(x_i - a_j)^2}\right\}$$

$$a_j = \frac{1}{N(c_j)}\sum_{i \in c_j} x_i$$

K-Means算法过程

根据构建K-Means算法步骤用图表示出来结果如图所示：



我们用语言和公式来还原上述图解的过程：

1. 原始数据集有N个样本，人为给定两个中心点。
2. 分别计算每个样本到两个中心点之间的距离，可选欧几里得距离，计算公式用9.1所提到的公式如下所示：

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

3. 把样本分为了两个簇，计算每个簇中样本点的均值为新的中心点。计算公式如下：

$$a_j = \frac{1}{N(c_j)}\sum_{i \in c_j} x_i$$

4. 重复以上步骤，知道达到前面所说中止条件。

K-Means的损失函数

我们的目的就是使得最终得到的中心点使得，每个样本到中心点和最小，每个样本到中心点距离公式为：

$$J(a_1,a,a_3,?\cdots,a_k)=\frac{1}{2}\sum_{j=1}^{k}\sum_{i=1}^{N_j}(x_i-a_i)^2$$

为了使损失函数最小，求偏导可以得到中心点的更新公式为：

$$a_j=\frac{1}{N(c_j)}\sum_{i\in c_i}x_i$$

K-Means算法遇到的问题

根据上面我们掌握的K-Means算法原理，发现有两个问题会很大影响K-Means算法。

1. K- means算法在迭代的过程中使用所有点的均值作为新的质点(中心点),如果簇中存在异常点,将导致均值偏差比较严重。

   例如：

   一个簇中有2、4、6、8、100五个数据,那么新的质点为24,显然这个质点离绝大多数点都比较远;在当前情况下,使用中位数6可能比使用均值的想法更好,**使用中位数的聚类方式叫做K-Mediods聚类(K中值聚类)**

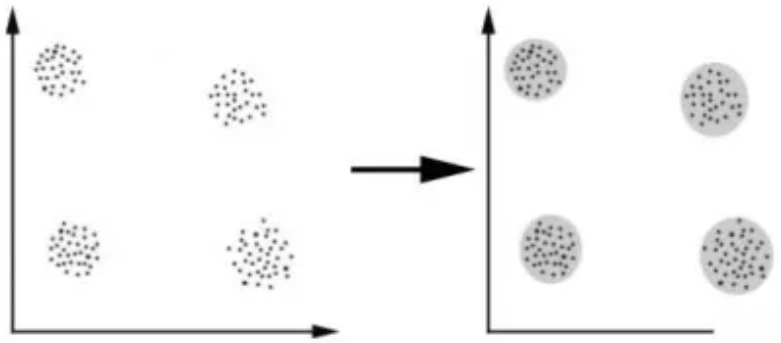
2. **初值敏感**

   K- means算法是初值敏感的,选择不同的初始值可能导致不同的簇划分规则。

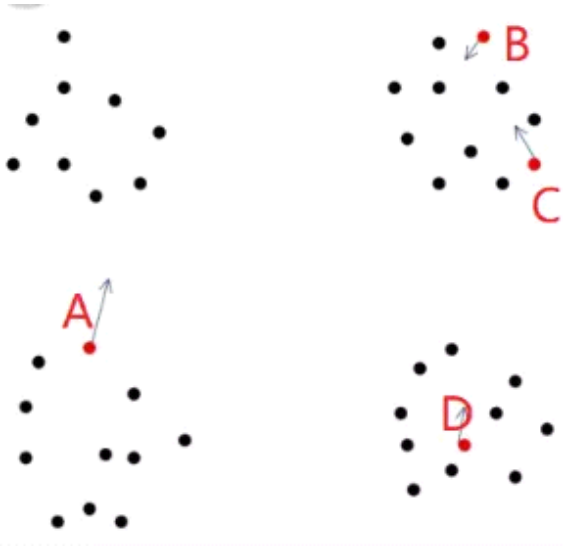
   为了避免这种敏感性导致的最终结果异常性,可以采用初始化多套初始节点构造不同的分类规则,然后选择最优的构造规则。

   又或者改变初始值的选择。这样通过改进的K-Means算法，将在下面进行一一介绍。

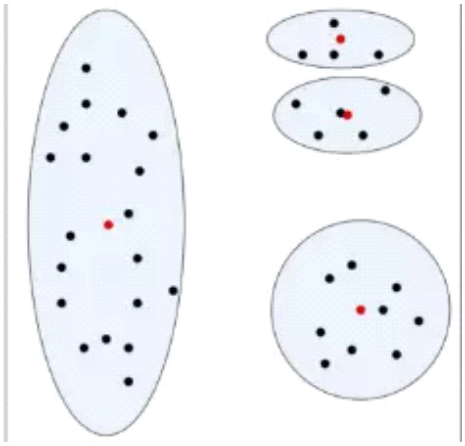
   下面给出一个初始值敏感的直观例子。给定一定的数据点如图9.3所示，我们明显等看到可以划分为四个区域：



   假如我们随机给定的中心点A,B,C,D如图9.3所示：



我们按照 K-Means算法划分的结果如图9.4所示：



K-Means例题（略）

K-Means改进的几种算法

前面简单地介绍了一种聚类算法思想K-Means算法，由于K-Means算法的简单且易于实现，因此K-Means算法得到了很多的应用，但是从K-Means算法的过程中发现，K-Means算法中的聚类中心的个数k需要事先指定，这一点对于一些未知数据存在很大的局限性。其次，在利用K-Means算法进行聚类之前，需要初始化k个聚类中心，在上述的K-Means算法的过程中，使用的是在数据集中随机选择最大值和最小值之间的数作为其初始的聚类中心，但是聚类中心选择不好，对于K-Means算法有很大的影响。介绍几种K-Means改进的算法。

K-Means++算法

K-Means++算法在聚类中心的初始化过程中的基本原则是使得初始的聚类中心之间的相互距离尽可能远，这样可以避免出现上述的问题。从而可以解决K- Means算法对初始簇心比较敏感的问题,K-Means++算法和K- Means算法的区别主要在于初始的K个中心点的选择方面。

K- Means算法使用随机给定的方式,K- Means++算法采用下列步骤给定K个初始质点：

1. q 从数据集中任选一个节点作为第一个聚类中心
2. q 对数据集中的每个点 x ,计算x到所有已有聚类中心点的距离和D(X),基于D(X)采用线性概率

（每个样本被选为下一个中心点的概率）选择出下一个聚类中心点距离较远的一个点成为新增的一个聚类中心点)

3. q 重复步骤2直到找到k个聚类中心点

这种由于依靠中心点和中心点之间的有序性进行中心点的划分，虽然避免了初始值敏感问题，可对于特别离散的数据，效果就不是很好了。

参考文献：Bahman Bahmani,Benjamin Moseley,Andrea Vattani.Scalable K-Means++

K-Meansll算法

k-means++ 最主要的缺点在于其内在的顺序执行特性，得到 k 个聚类中心必须遍历数据集 k 次，并且当前聚类中心的计算依赖于前面得到的所有聚类中心，这使得算法无法并行扩展，极大地限制了算法在大规模数据集上的应用。

这是一种针对K-Means++改进的算法，主要思路是改变每次遍历时候的取样规则,并非按照K-Means++算法每次遍历只获取一个样本,而是每次获取K个样本,重复该取样操作O(klogn)次,然后再将这些抽样出来的样本聚类出K个点,最后使用这K个点作为K- Means算法的初始聚簇中心点。一般情况下，重复几次就可以得到比较好的中心点。

二分K- Means算法

同样是为了解决K- Means算法对初始簇心比较敏感的问题,二分K- Means算法和前面两种寻找其他质心不同，它是一种弱化初始质心的一种算法。

这个算法的思想是：首先将所有点作为一个簇，然后将该簇一分为二。之后选择能最大程度降低聚类代价函数（也就是误差平方和）的簇划分为两个簇（或者选择最大的簇等，选择方法多种）。以此进行下去，直到簇的数目等于用户给定的数目k为止。

算法的步骤如下：

1. q 将所有样本数据作为一个簇放到一个队列中
2. q 从队列中选择一个簇进行K- means算法划分,划分为两个子簇,并将子簇添加到队列中
3. q 循环迭代第二步操作,直到中止条件达到(聚簇数量、最小平方误差、迭代次数等)
4. q 队列中的簇就是最终的分类簇集合

从队列中选择划分聚簇的规则一般有两种方式，分别如下：

1. 对所有簇计算误差和SSE(SSE也可以认为是距离函数的一种变种),选择SSE最大的聚簇进行划分操作(优选这种策略)

$$SSE = \sum_{i=1}^{n} a_i (x_i - a_i)^2$$

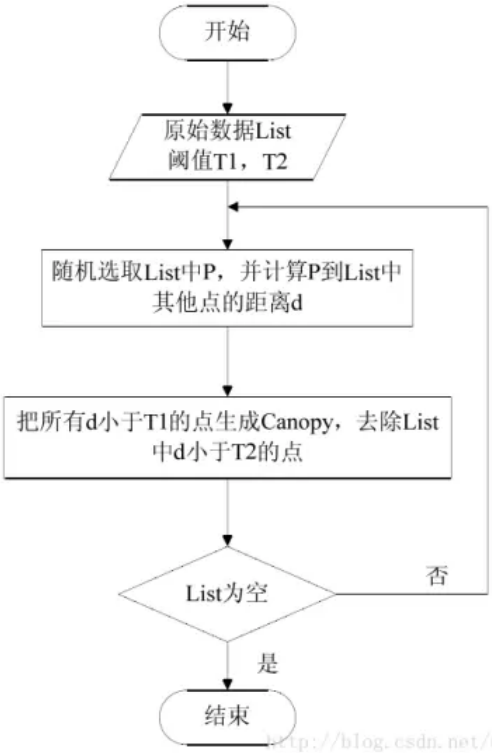
2. 选择样本数据量最多的簇进行划分操作

Canopy算法

Canopy Clustering 这个算法是2000年提出来的，此后与Hadoop配合，已经成为一个比较流行的算法了。确切的说，这个算法获得的并不是最终结果，它是为其他算法服务的，比如k-means算法。它能有效地降低k-means算法中计算点之间距离的复杂度。算法流程如下：

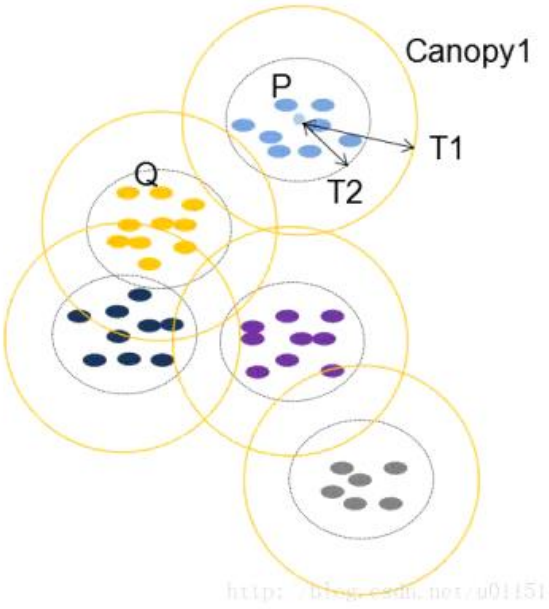
1. 给定样本列表L=x1,,2...,m以及先验值T1和T2(T1>T2)
2. 从列表L中获取一个节点P,计算P到所有聚簇中心点的距离(如果不存在聚簇中心,那么此时点P形成一个新的聚簇),并选择出最小距离D(P,aj)。
3. 如果距离D小于T1,表示该节点属于该聚簇,添加到该聚簇列表中
4. 如果距离D小于T2,表示该节点不仅仅属于该聚簇,还表示和当前聚簇中心点非常近,所以将该聚簇的中心点设置为该簇中所有样本的中心点,并将P从列表L中删除。
5. 如果距离D大于T1,那么节点P形成一个新的聚簇。
6. 直到列表L中的元素数据不再有变化或者元素数量为0的时候,结束循环操作。

该步骤用流程图表示如下图所示:



Canopy算法得到的最终结果的值,聚簇之间是可能存在重叠的,但是不会存在某个对象不属于任何聚簇的情况。

可以看到canopy算法将可以将一堆杂乱的数据大致的划分为几块所以Canopy算法一般会和kmeans算法配合使用来到达使用者的目的在使用Canopy算法时，阈值t1，t2的确定是十分重要的。t1的值过大，会导致更多的数据会被重复迭代，形成过多的Canopy；值过小则导致相反的效果t2的值过大，会导致一个canopy中的数据太多，反之则过少这样的情况都会导致运行的结果不准确。该算法的过程图形说明如图所示:



Mini batch k- Means算法

Mini Batch K-Means使用了一个种叫做Mini Batch（分批处理）的方法对数据点之间的距离进行计算。Mini Batch的好处是计算过程中不必使用所有的数据样本，而是从不同类别的样本中抽取一部分样本来代表各自类型进行计算。由于计算样本量少，所以会相应的减少运行时间，但另一方面抽样也必然会带来准确度的下降。这样使用于存在巨大的数据集合的情况下。

实际上，这种思路不仅应用于K-Means聚类，还广泛应用于梯度下降、深度网络等机器学习和深度学习算法。

该算法的算法流程和k- Means类似，流程如下：

1. 首先抽取部分数据集,使用K- Means算法构建出K个聚簇点的模型。
2. 继续抽取训练数据集中的部分数据集样本数据,并将其添加到模型中,分配给距离最近的聚簇中心点。
3. 更新聚簇的中心点值。
4. 循环迭代第二步和第三步操作,直到中心点稳定或者达到迭代次数,停止计算操作。

应用场景，由于Mini Batch KMeans跟K-Means是极其相似的两种聚类算法，因此应用场景基本一致。

后面我们就以 Mini batch k- Means算法为例子，比较一下它和 k- Means算法的区别。

聚类算法评估

有监督的分类算法的评价指标通常是accuracy, precision, recall, etc；由于聚类算法是无监督的学习算法，评价指标则没有那么简单了。因为聚类算法得到的类别实际上不能说明任何问题，除非这些类别的分布和样本的真实类别分布相似，或者聚类的结果满足某种假设，即同一类别中样本间的相似性高于不同类别间样本的相似性。

下面介绍几种常见的评估方法：

均一性/完整性

一个簇只包含一个类别样本，满足均一性，也可以认为正确率（每个簇中正确分类占该簇总样本的

比），公式如下：

$$p = \frac{1}{k}\sum_{i=1}^{k}\frac{N(C_i = K_i)}{N(C_i)}$$

兰德系数（RI)

兰德系数（Rand index）需要给定实际类别信息C，假设K是聚类结果，a表示在C与K中都是同类别的元素对数，b表示在C与K中都是不同类别的元素对数，则兰德指数为：

$$RI = \frac{a+b}{C_2^{n_{samples}}}$$

分子：属性一致的样本数，即同属于这一类或都不属于这一类。a是真实在同一类、预测也在同一类的样本数；b是真实在不同类、预测也在不同类的样本数；

分母：任意两个样本为一类有多少种组合，是数据集中可以组成的总元素对数；

RI取值范围为[0,1]，值越大意味着聚类结果与真实情况越吻合。

对于随机结果，RI并不能保证分数接近零。为了实现“在聚类结果随机产生的情况下，指标应该接近零”，调整兰德系数（Adjusted rand index）被提出，它具有更高的区分度：

$$ARI = \frac{RI - E[RI]}{\max(RI) - E[RI]}$$

ARI取值范围为[-1,1]，值越大意味着聚类结果与真实情况越吻合。从广义的角度来讲，ARI衡量的是两个数据分布的吻合程度。

优点：

1. 对任意数量的聚类中心和样本数，随机聚类的ARI都非常接近于0；
2. 取值在 [- 1，1] 之间，负数代表结果不好，越接近于1越好；
3. 可用于聚类算法之间的比较。

缺点：

ARI需要真实标签

轮廓系数 Silhouette Coefficient

轮廓系数适用于实际类别信息未知的情况。

- 簇内不相似度：计算样本i倒同簇其它样本的平均距离为a;a越小,表示样本越应该被聚类到该簇,簇C中的所有样本的a的均值被称为簇C的簇不相似度。
- 簇间不相似度： 计算样本i到其它簇C的所有样本的平均距离b;b=min{b;1,bi2,bi};b越大,表示样本越不属于其它簇。
- 轮廓系数：s值越接近1表示样本麋类越合理,越接近-1,表示样本j应该分类到另外的簇中,近似为0,表示样本应该在边界上;所有样本的s的均值被成为聚类结果的轮廓系数。伦敦系数可以写作：

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

Scikit-learn中有兰求德系数方法metrics.silhouette_score。

Python中实现的代码如下：

```
from sklearn import metrics  from sklearn.metrics import pairwise_distances
from sklearn import datasets
dataset = datasets.load_iris()
X = dataset.data
 y = dataset.target
import numpy as np
from sklearn.cluster import KMeans
kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
labels = kmeans_model.labels_
metrics.silhouette_score(X, labels, metric='euclidean')
```

输出：

```
兰德系数为:

0.5730973570083832
```

示例：比较Mini batch k- Means算法和 k- Means算法

要求：

给定较多数据，来比较两种算法的聚类速度，且用刚学到的聚类评估算法对，这两种算法进行评估。

两种的算法的API详情可以参考网址：

<u>http://scikit-learn.org/0.19/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans。</u>

```
#第一个参数表示给定中心点的个数，第二个参数给出初始质心怎么给，第三个参数是多套初始质心，第四个代表的是迭代次数
sklearn.cluster.KMeans(n_clusters=8, init=’k-means++’, n_init=10, max_iter=300) []
(#sklearn.cluster.MiniBatchKMeans "Permalink to this definition")
```

1. 导入模块

```
#导入我们要用的包，包括算法数据创建模块，算法评估模块，算法模块。
```

```
import time  import numpy as np  import matplotlib.pyplot as plt  import matplotlib as mpl
from sklearn.cluster import MiniBatchKMeans, KMeans  from sklearn.metrics.pairwise
import pairwise_distances_argmin  from sklearn.datasets.samples_generator import
make_blobs
```

2. 创建数据

```
#创建呈现3个团状共3000个样本数据，样本有两个特征属性，

#初始化三个中心 centers = [[1, 1], [-1, -1], [1, -1]] clusters = len(centers) #聚类的数目为3 #产生3000组二维的数据，中心是意思三个中心点，标准差是0.7 X, Y = make_blobs(n_samples=3000, centers=centers, cluster_std=0.7, random_state=0)
```

3. 模型构建

```
#构建kmeans算法 k_means = KMeans(init='k-means++', n_clusters=clusters, random_state=28) t0 = time.time() #当前时间 k_means.fit(X) #训练模型 km_batch = time.time() - t0 #使用kmeans训练数据的消耗时间 print ("K-Means算法模型训练消耗时间:%.4fs" % km_batch)

#构建MiniBatchKMeans算法 batch_size = 100 mbk = MiniBatchKMeans(init='k-means++', n_clusters=clusters, batch_size=batch_size, random_state=28) t0 = time.time() mbk.fit(X) mbk_batch = time.time() - t0 print ("Mini Batch K-Means算法模型训练消耗时间:%.4fs" % mbk_batch)

#输出的结果为：

L- Means算法模型训练消耗时间:0.0416s

M- Mini Batch K-Means算法模型训练消耗时间:0.0150s
```

4. 模型的预测

```
#预测结果 km_y_hat = k_means.predict(X) mbkm_y_hat = mbk.predict(X) print(km_y_hat[:10]) print(mbkm_y_hat[:10]) print(k_means.cluster_centers_) print(mbk.cluster_centers_)

#输出的结果：

[0 1 0 2 1 1 1 1 2 1]

[[-1.07159013 -1.00648645]

[ 0.96700708  1.01837274]

[ 1.07705469 -1.06730994]]

[[ 1.02538969 -1.08781328]
```

```
[-1.06046903 -1.01509453]

[ 0.97734743  1.08610316]]
```

5. 求出质心得坐标并进行排序：

```
##获取聚类中心点并聚类中心点进行排序  k_means_cluster_centers = k_means.cluster_centers_#输出kmeans聚类中心点  mbk_means_cluster_centers = mbk.cluster_centers_#输出mbk聚类中心点  print ("K-Means算法聚类中心点:\ncenter=", k_means_cluster_centers)  print ("Mini Batch K-Means算法聚类中心点:\ncenter=", mbk_means_cluster_centers)  order = pairwise_distances_argmin(k_means_cluster_centers, mbk_means_cluster_centers)
```

```
得到的结果如下：
```

```
K-Means算法聚类中心点:

center= [[-1.07159013 -1.00648645]

[ 0.96700708  1.01837274]

[ 1.07705469 -1.06730994]]

### Mini Batch K-Means算法聚类中心点:

center= [[ 1.02538969 -1.08781328]

[-1.06046903 -1.01509453]

[ 0.97734743  1.08610316]]
```

6. 画图，把原始数据和最终预测数据在图上表现出来

```
plt.figure(figsize=(12, 6), facecolor='w')  plt.subplots_adjust(left=0.05, right=0.95, bottom=0.05, top=0.9)  cm = mpl.colors.ListedColormap(['#FFC2CC', '#C2FFCC', '#CCC2FF'])  cm2 = mpl.colors.ListedColormap(['#FF0000', '#00FF00', '#0000FF'])
```

7. 创建各个子图：

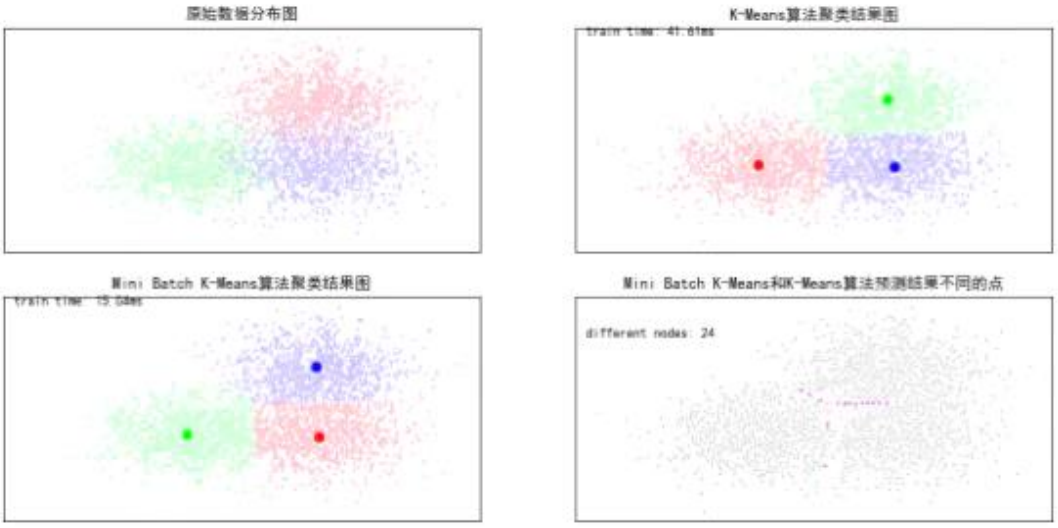
```
#子图1：原始数据  plt.subplot(221)  plt.scatter(X[:, 0], X[:, 1], c=Y, s=6, cmap=cm, edgecolors='none')  plt.title(u'原始数据分布图')  plt.grid(True)  #子图2：K-Means算法聚类结果图  plt.subplot(222)  plt.scatter(X[:,0], X[:,1], c=km_y_hat, s=6, cmap=cm,edgecolors='none')
```

```
plt.scatter(k_means_cluster_centers[:,0], k_means_cluster_centers[:,1],c=range(clusters),s=60,cmap=cm2,edgecolors='none')  plt.title(u'K-Means算法聚类结果图')  plt.text(-3.8, 3,  'train time: %.2fms' % (km_batch*1000))  plt.grid(True)  #子图三Mini Batch K-Means算法聚类结果图  plt.subplot(223)  plt.scatter(X[:,0], X[:,1], c=mbkm_y_hat, s=6, cmap=cm,edgecolors='none')  plt.scatter(mbk_means_cluster_centers[:,0], mbk_means_cluster_centers[:,1],c=range(clusters),s=60,cmap=cm2,edgecolors='none')  plt.title(u'Mini Batch K-Means算法聚类结果图')  plt.text(-3.8, 3,  'train time: %.2fms' % (mbk_batch*1000))  plt.grid(True)
```

8. 创建两则分不同的点子图：

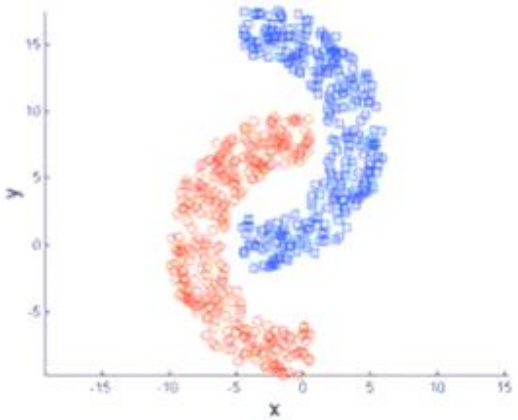
```
different = list(map(lambda x: (x!=0) & (x!=1) & (x!=2), mbkm_y_hat))  for k in range(clusters):      different += ((km_y_hat == k) != (mbkm_y_hat == order[k]))  identic = np.logical_not(different)  different_nodes = len(list(filter(lambda x:x, different)))  plt.subplot(224)  # 两者预测相同的  plt.plot(X[identic, 0], X[identic, 1], 'w', markerfacecolor='#bbbbbb', marker='.')  # 两者预测不相同的  plt.plot(X[different, 0], X[different, 1], 'w', markerfacecolor='m', marker='.')  plt.title(u'Mini Batch K-Means和K-Means算法预测结果不同的点')  plt.xticks(())  plt.yticks(())  plt.text(-3.8, 2,  'different nodes: %d' % (different_nodes))  plt.show()
```

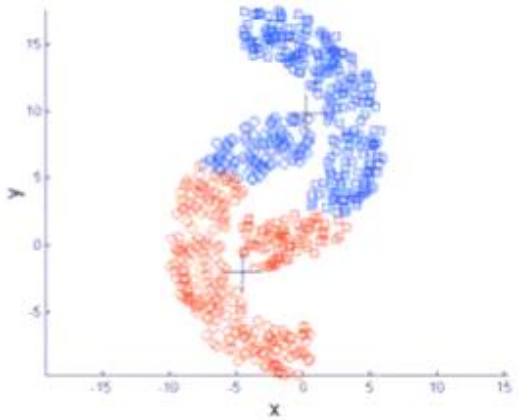
9. 输出结果如下



分析：从上图，我们看出Mini batch k- Means算法比 k- Means算法速度快了不止一倍，而效果还是不错的。

思考：如果出现如图9.7所示出现的数据类型用类 k- Means算法就不能正确地对他们进行聚类了，因为他们属于非凸类数据。这时候就要转变聚类思想了，采用别的聚类方法了。





本章小结

本章主要介绍了聚类中的一种最常见的算法—K-Means算法以及其优化算法，聚类是一种无监督学习的方法。我们通过簇来把数据划分为不同的种类，介绍了该算法的构建流程，根据构建的流程中的初始值敏感问题，我们对最基本的K-Means算法进行了改进，其中Mini batch k- Means算法可以适用于数据量比较多的情况，且效果也不错。

本章的最后我们根据样本不同的特征，有时候不适合采用K-Means算法，我们将在下一章介绍几种其他思想产生的算法。

其他聚类

紧接上章，本章主要是介绍和K-Means算法思想不同而的其他聚类思想形成的聚类算法。

k-means算法却是一种方便好用的聚类算法，但是始终有K值选择和初始聚类中心点选择的问题，而这些问题也会影响聚类的效果。为了避免这些问题，我们可以选择另外一种比较实用的聚类算法-层次聚类算法。顾名思义，层次聚类就是一层一层的进行聚类，可以由上向下把大的类别（cluster）分割，叫作分裂法；也可以由下向上对小的类别进行聚合，叫作凝聚法；但是一般用的比较多的是由下向上的凝聚方法。

本章主要涉及到的知识点有：

- **层次聚类**
- **BIRCH算法**

层次聚类

层次聚类方法对给定的数据集进行层次的分解，直到满足某种条件为止，传统的层次聚类算法主要分为两大类算法:分裂的层次聚类和凝聚的层次聚类。

分裂的层次聚类

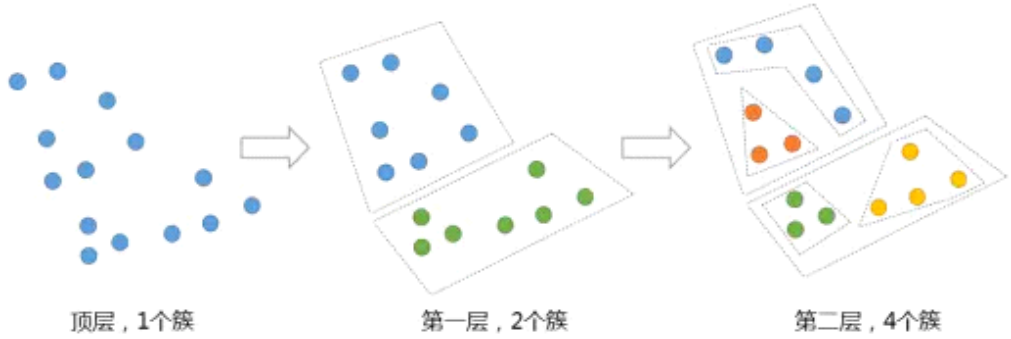
DIANA算法(Divisive analysis)采用自顶向下的策略。首先将所有对象置于一个簇中，然后按照某种

既定的规则逐渐细分为越来越小的簇(比如最大的欧式距离)，直到达到某个终结条件(簇数目或者簇距离达到阈值)。

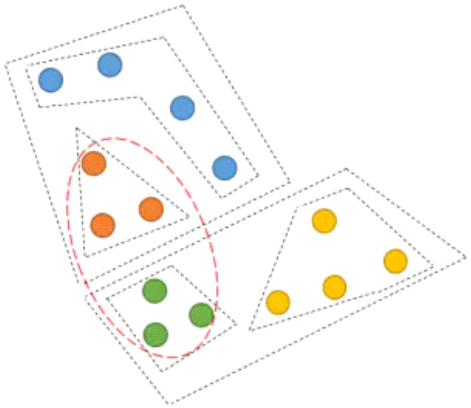
分裂法指的是初始时将所有的样本归为一个类簇，然后依据某种准则进行逐渐的分裂，直到达到某种条件或者达到设定的分类数目。算法构建步骤：

1. 将样本集中的所有的样本归为一个类簇；
2. 在同一个类簇（计为c）中计算两两样本之间的距离，找出距离最远的两个样本a，b；
3. 将样本a，b分配到不同的类簇c1和c2中；
4. 计算原类簇（c）中剩余的其他样本点和a，b的距离，若是dis(a)<dis(b)，则将样本点归到c1中，否则归到c2中；
5. 重复以上步骤，直到达到聚类的数目或者达到设定的条件。

这里我们可以看出，上一章所说的 K-Means算法及其优化算法就是运用这种自顶向下的策略。例如：有一些点，按照上面所说的算法构建步骤的过程可以如下图表示出来：



常见的自顶向下的算法有K-means层次聚类算法。但值得注意的是：对于以上的例子，红色椭圆框中的对象聚类成一个簇可能是更优的聚类结果，但是由于橙色对象和绿色对象在第一次K-means就被划分到不同的簇，之后也不再可能被聚类到同一个簇。



两个簇之间最近的两个点的距离作为簇之间的距离，该方式的缺陷是受噪点影响大，容易产生长条状的簇。

凝聚的层次聚类

AGNES算法(Agglomerative Nesting)采用自底向上的策略。最初将每个对象作为一个簇，然后这些簇根据某些准则被一步一步合并，两个簇间的距离可以由这两个不同簇中距离最近的数据点的相似度来确定;聚类的合并过程反复进行直到所有的对象满足簇数目。

凝聚法指的是初始时将每个样本点当做一个类簇，所以原始类簇的大小等于样本点的个数，然后依据

某种准则合并这些初始的类簇，直到达到某种条件或者达到设定的分类数目。算法步骤：

1. 将样本集中的所有的样本点都当做一个独立的类簇；
2. 计算两两类簇之间的距离（后边会做介绍），找到距离最小的两个类簇c1和c2；
3. 合并类簇c1和c2为一个类簇；
4. 重复以上步骤，直到达到聚类的数目或者达到设定的条件。

簇间距离

在以上算法的构建过程中，我们提到了两个簇之间的距离的计算，例如对于两个簇C_1和C_2，有三种方法计算塔门之间的距离：

单连锁（Single link）

两个簇之间的最小距离:

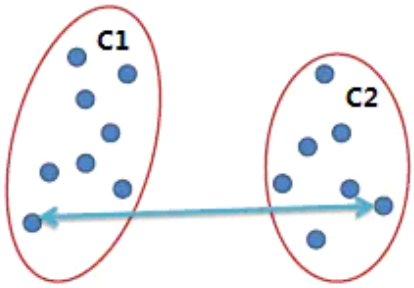
$$D(C_1, C_2) = \mathop{\mathbf{MIN}}_{x_1 \in C_1, x_2 \in C_2} D(x_1, x_2)$$

两个簇之间最近的两个点的距离作为簇之间的距离，该方式的缺陷是受噪点影响大，容易产生长条状的簇。

全连锁（Complete link）

两个簇之间的最大距离:

$$D(C_1, C_2) = \max_{x_1 \in C_1, x_2 \in C_2} D(x_1, x_2)$$

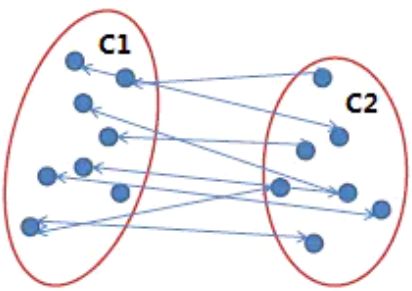


两个簇之间最远的两个点的距离作为簇之间的距离，采用该距离计算方式得到的聚类比较紧凑。

平均连锁（Average link）

两个簇之间的平均距离:

$$D(C_1, C_2) = \frac{1}{|C_1|}\frac{1}{C_2}\sum_{x_1 \in C_1}\sum_{x_2 \in C_2} D(x_1, x_2)$$



两个簇之间两两点之间距离的平均值，该方式可以有效地排除噪点的影响。

层次聚类小结

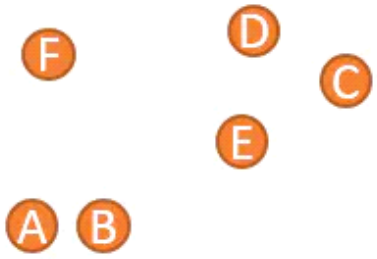
层次聚类的优缺点：

- 简单，理解容易
- 合并点/分裂点选择不太容易
- 合并/分类的操作不能进行撤销
- 大数据集不太适合
- 执行效率较低Ot*n2)，t为迭代次数，n为样本点数

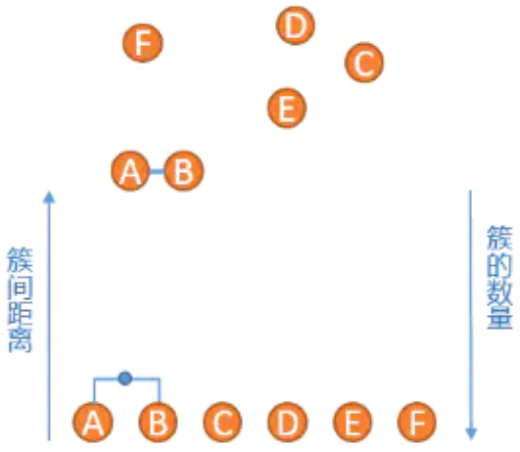
层次聚类算法示例

Agglomerative算法

对于如下数据：



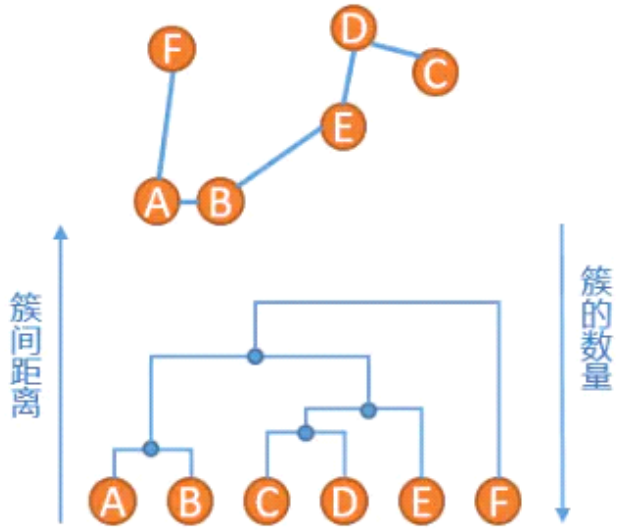
1. 将A到F六个点，分别生成6个簇；
   找到当前簇中距离最短的两个点，这里我们使用单连锁的方式来计算距离，发现A点和B点距离最短，将A和B组成一个新的簇，此时簇列表中包含五个簇，分别是{A，B}，{C}，{D}，{E}，{F}，如下图所示；



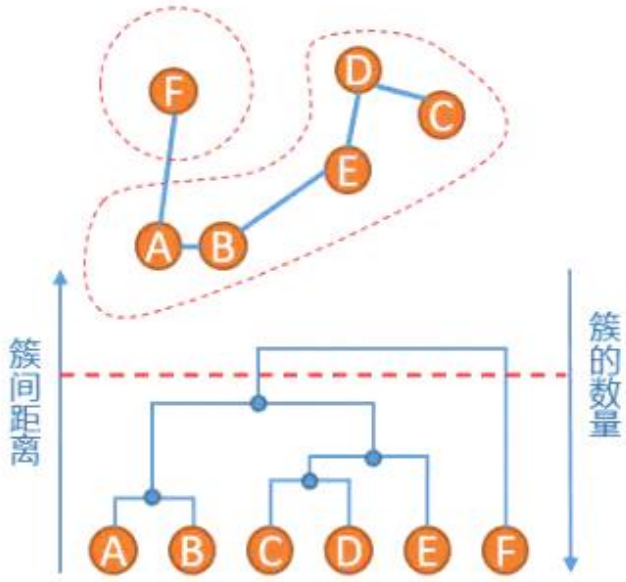
2. 找到当前簇中距离最短的两个点，这里我们使用单连锁的方式来计算距离，发现A点和B点距离最短，将A和B组成一个新的簇，此时簇列表中包含五个簇，分别是{A，B}，{C}，{D}，{E}，{F}，如下图所示；

3. 重复步骤二、发现{C}和{D}的距离最短，连接之，然后是簇{C，D}和簇{E}距离最短，依次类推，直到

最后只剩下一个簇，得到如下所示的示意图：



4. 此时原始数据的聚类关系是按照层次来组织的，选取一个簇间距离的阈值，可以得到一个聚类结果，比如在如下红色虚线的阈值下，数据被划分为两个簇：簇{A，B，C，D，E}和簇{F}



Agglomerative聚类算法的优点是能够根据需要在不同的尺度上展示对应的聚类结果，缺点同Hierarchical K-means算法一样，一旦两个距离相近的点被划分到不同的簇，之后也不再可能被聚类到同一个簇，即无法撤销先前步骤的工作。另外，Agglomerative性能较低，并且因为聚类层次信息需要存储在内存中，内存消耗大，不适用于大量级的数据聚类，下面介绍一种针对大数据量级的聚类算法BIRCH。

BIRCH算法

B|RCH算法(平衡迭代削减聚类法):聚类特征使用3元组进行一个簇的相关信息，通过构建满足分枝因子和簇直径限制的聚类特征树来求聚类，聚类特征树其实是个具有两个参数分枝因子和类直径的高度平衡树;分枝因子规定了树的每个节点的子女的最多个数，而类直径体现了对这一类点的距离范围;非

叶子节点为它子女的最大特征值;聚类特征树的构建可以是动态过程的，可以随时根据数据对模型进行更新操作。

BIRCH算法的全称是Balanced Iterative Reducing and Clustering using Hierarchies，它使用聚类特征来表示一个簇，使用聚类特征树（CF-树）来表示聚类的层次结构，算法思路也是“自底向上”的。该算法的步骤如下：

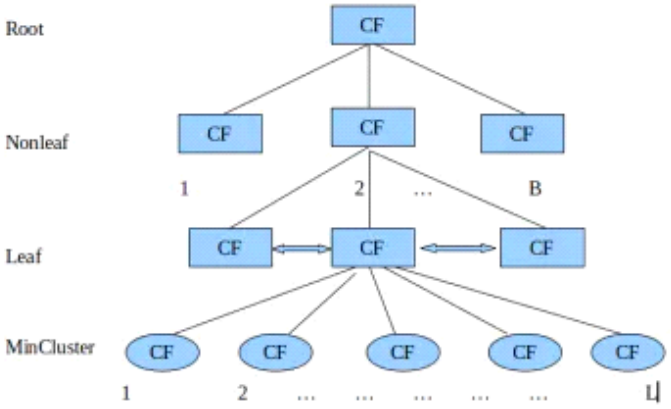
1. 构建BIRCH算法的核心—CF-树（Clustering Feature Tree），而CF-树种每个节点代表一个簇的抽象特征，包含三个数据：簇中数据个数，n个点的线性和，n个数据点的平方和；
2. 从根节点开始，自上而下选择最近的孩子节点；

到达叶子节点后，检查距离其最近的CF能否吸收此数据点：

a. 是，更新CF值
b. 否，创建一个新的CF节点，检查该节点能否加入到当前叶子节点
   i. 能，添加到当前叶子节点；
   ii. 否，分裂最远的一对CF节点，按最近距离重新分配其它节点；

3. 更新每个非叶子节点的CF信息，如果分裂节点，在父节点中插入新的CF节点，直到root；

算法示意图如下图：



示例

基于scikit包中的创建的模拟数据的API进行数据的创建。使用BIRCH算法对数据进行数据进行划分类，比较不同模型数量对算法的图像的影响。

1. 导入模块

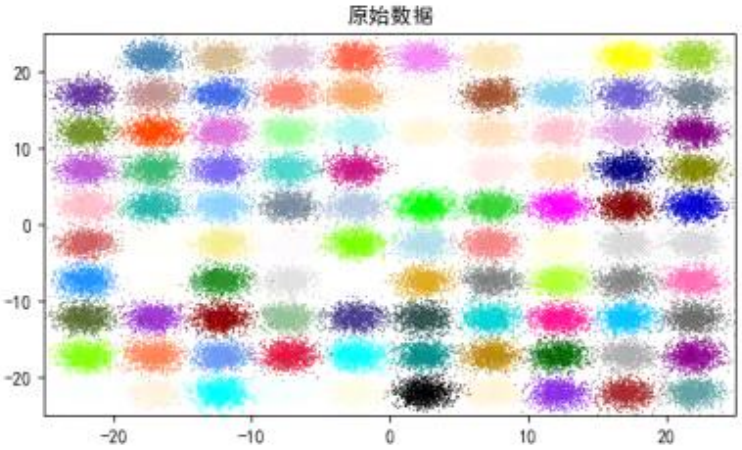
```
#导入我们要用的包，包括算法数据创建模块，算法评估模块，算法模块。
from itertools import cycle
from time import time
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.cluster import Birch
from sklearn.datasets.samples_generator import make_blobs
```

2. 创建数据

```
#创建呈现3个团状共3000个样本数据，样本有两个特征属性，
## 设置属性防止中文乱码  mpl.rcParams['font.sans-serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False
## 产生模拟数据
xx = np.linspace(-22， 22， 10)
yy = np.linspace(-22， 22， 10)
xx， yy = np.meshgrid(xx， yy)
n_centres = np.hstack((np.ravel(xx)[:， np.newaxis]，
np.ravel(yy)[:， np.newaxis]))
 #产生10万条特征属性是2，类别是100，符合高斯分布的数据集
 X， y = make_blobs(n_samples=100000，n_features=2， centers=n_centres，
random_state=28)
```

原始的数据集如下图所示：



3. 模型构建

```
#创建不同的参数（簇直径）Birch层次聚类
birch_models = [
    Birch(threshold=1.7, n_clusters=100),
    Birch(threshold=1.7, n_clusters=None)
]
#threshold：簇直径的阈值，     branching_factor：大叶子个数
#我们也可以加参数来试一下效果，比如加入分支因子branching_factor，给定不同的参数值，看聚类的结果
final_step = [ u'直径=1.7;n_clusters=100',
               u'直径=1.7;n_clusters=None'
```

```
        ]

plt.figure(figsize=(12, 8), facecolor='w')
plt.subplots_adjust(left=0.02, right=0.98, bottom=0.1, top=0.9)
colors_ = cycle(colors.cnames.keys())
cm = mpl.colors.ListedColormap(colors.cnames.keys())
```

4. **模型的构建**

```
for ind, (birch_model, info) in enumerate(zip(birch_models, final_step)):
    t = time()
    birch_model.fit(X)
    time_ = time() - t
    # 获取模型结果（label和中心点）
    labels = birch_model.labels_
    centroids = birch_model.subcluster_centers_
    n_clusters = len(np.unique(centroids))
    print("Birch算法，参数信息为：%s；模型构建消耗时间为:%.3f秒；聚类中心数目:%d" % (info,
time_, len(np.unique(labels))))
```

5. **画图预测模型**

```
## 画图
    subinx = 221 + ind
    plt.subplot(subinx)
    for this_centroid, k, col in zip(centroids, range(n_clusters), colors_):
        mask = labels == k
        plt.plot(X[mask, 0], X[mask, 1], 'w', markerfacecolor=col, marker='.')
        if birch_model.n_clusters is None:
            plt.plot(this_centroid[0], this_centroid[1], '*', markerfacecolor=col,
markeredgecolor='k', markersize=2)
    plt.ylim([-25, 25])
    plt.xlim([-25, 25])
    plt.title(u'Birch算法%s，耗时%.3fs' % (info, time_))
    plt.grid(False)
```

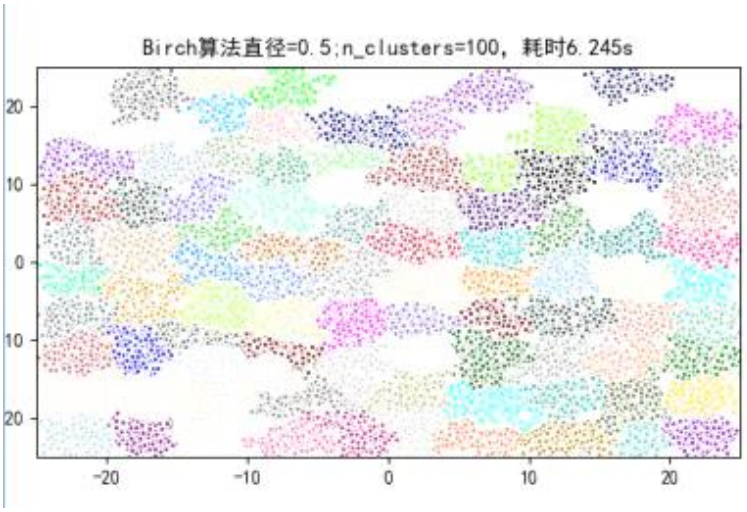
画出原始数据的图

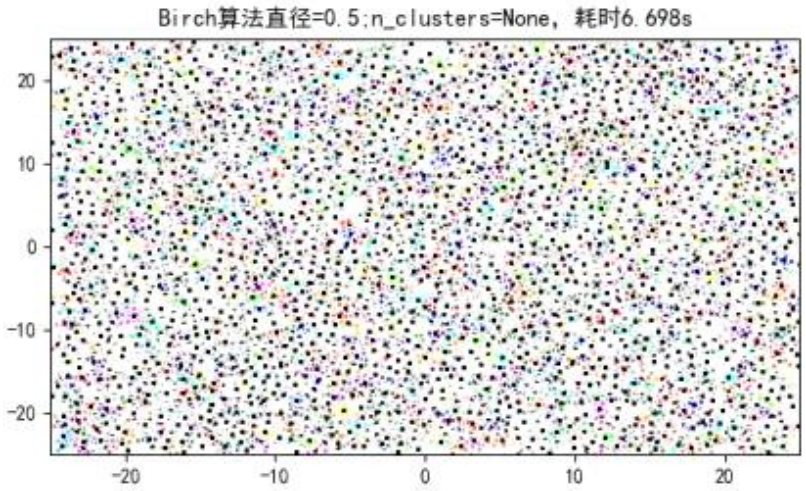
```
# 原始数据集显示
plt.subplot(224)
plt.scatter(X[:, 0], X[:, 1], c=y, s=1, cmap=cm, edgecolors='none')
```

```
plt.ylim([-25, 25])
plt.xlim([-25, 25])
plt.title(u'原始数据')
plt.grid(False)
plt.show()
```

- 当簇的直径为0.5时候，处的数量分别为100和None时候输出的结果：
  Birch算法，参数信息为：直径=0.5;n_clusters=100；模型构建消耗时间为:6.762秒；聚类中心数目:100
  Birch算法，参数信息为：直径=0.5;n_clusters=None；模型构建消耗时间为:6.698秒；聚类中心数目:3205

  输出的图画：

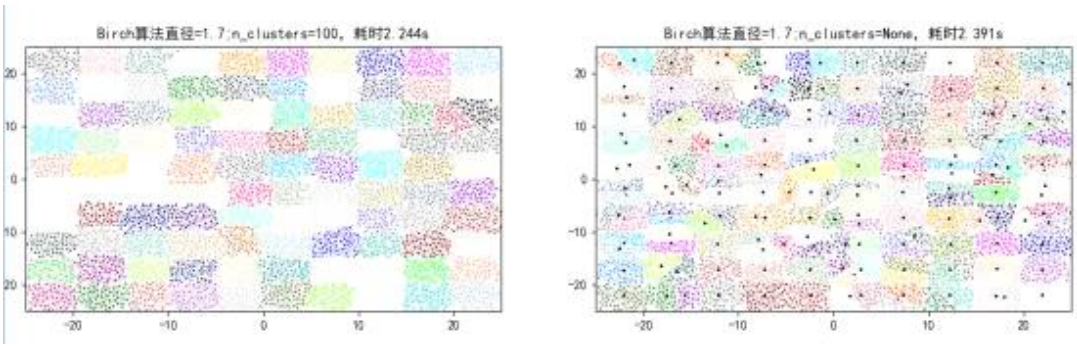




- 当簇的直径为0.5时候，处的数量分别为100和None时候输出的结果：
  Birch算法，参数信息为：直径=1.7;n_clusters=100；模型构建消耗时间为:2.244秒；聚类中心数目:100

Birch算法，参数信息为：直径=1.7;n_clusters=None；模型构建消耗时间为:2.391秒；聚类中心数目:171

输出的结果如下图所示：



BIRCH算法相比Agglomerative凝聚算法具有如下特点：

1. 解决了Agglomerative算法不能撤销先前步骤的工作的缺陷；
2. CF-树只存储原始数据的特征信息，并不需要存储原始数据信息，内存开销上更优；
3. BIRCH算法只需要遍历一遍原始数据，而Agglomerative算法在每次迭代都需要遍历一遍数据，所以BIRCH在性能也优于Agglomerative；
4. 支持对流数据的聚类，BIRCH一开始并不需要所有的数据；

小结

本章主要介绍了聚类中的其他聚类算法的思想—层次聚类，着重介绍了算法—Agglomerative算法，BIRCH算法。以上所有的算法的实现都是依赖于机器学习库—scikit-learn库，当然还有其他聚类比如，谱聚类，Apriori关联分析等都有很好的聚类分析能力。只要掌握其思想，才能对各种聚算法融会贯通。