聚类算法

OUTLINES

1. 预备知识
2. Clustering 问题定义
3. Clustering 算法概述
4. 算法 (k-means, BIRCH, DBSCAN, Spectral)
5. Clustering算法选型

预备知识 – 评价指标

内部评价

- Compactness (紧密性)(CP)
- Separation (间隔性)(SP)
- Davies-Bouldin Index (戴维森堡丁指数) (分类适确性指标) (DB)(DBI)
- Dunn Validity Index (邓恩指数)(DVI)

外部评价

- Cluster Accuracy (准确性)(CA)
- Rand index (兰德指数)(RI)、Adjusted Rand index (调整兰德指数)(ARI)
- Normalized Mutual Information (标准互信息)(NMI)、Mutual Information (互信息)(MI)

预备知识 – 评价指标

内部评价

- Compactness (紧密性)(CP)
- Separation (间隔性)(SP)
- Davies-Bouldin Index (戴维森堡丁指数) (分类适确性指标) (DB)(DBI)
- Dunn Validity Index (邓恩指数)(DVI)

Compactness(紧密性)(CP)

CP 计算每一个类各点到聚类中心的平均距离

CP 越低意味着类内聚类距离越近

缺点：没有考虑类间效果

$$\overline{CP}_i = \frac{1}{|\Omega_i|} \sum_{x_i \in \Omega_i} \|x_i - w_i\|$$

$$\overline{CP} = \frac{1}{K} \sum_{k=1}^{K} \overline{CP}_k,$$

预备知识 – 评价指标

内部评价

- **Compactness (紧密性)(CP)**
- **Separation (间隔性)(SP)**
- **Davies-Bouldin Index (戴维森堡丁指数) (分类适确性指标) (DB)(DBI)**
- **Dunn Validity Index (邓恩指数)(DVI)**

Separation(间隔性)(SP)

SP 计算各聚类中心两两之间平均距离

SP 越高意味类间聚类距离越远

缺点：没有考虑类内效果

$$\overline{SP} = \frac{2}{k^2 - k} \sum_{i=1}^{k} \sum_{j=i+1}^{k} \|w_i - w_j\|_2$$

有没有综合考虑 CP 和 SP 的Metric？

预备知识 – 评价指标

内部评价

- Compactness (紧密性)(CP)
- Separation (间隔性)(SP)
- Davies-Bouldin Index (戴维森堡丁指数) (分类适确性指标) (DB)(DBI)
- Dunn Validity Index (邓恩指数)(DVI)

Davies-Bouldin Index 戴维森堡丁指数

(分类适确性指标)(DB 、DBI)

DB计算每个类别与其他某个类别的**类内距离**平均距离之和除以两**聚类中心**距离的最大值的平均值

DB越小，意味着类内距离越小，同时类间距离越大

缺点：因使用欧式距离，所以对于环状分布聚类评测很差

$$DB = \frac{1}{k}\sum_{i=1}^{k}\max_{j\neq i}\left(\frac{\overline{C_i}+\overline{C_j}}{\|w_i - w_j\|_2}\right)$$

预备知识 – 评价指标

内部评价

- Compactness (紧密性)(CP)
- Separation (间隔性)(SP)
- Davies-Bouldin Index (戴维森堡丁指数) (分类适确性指标) (DB)(DBI)
- Dunn Validity Index (邓恩指数)(DVI)

Dunn Validity Index (邓恩指数)(DVI)

DVI计算任意两个簇元素的最短距离(类间)除以任意簇中的最大距离(类内)

DVI越大，意味着类间距离越大，同时类内距离越小

缺点：对离散点的聚类测评很高、对环状分布测评效果差

$$DVI = \frac{\min\limits_{0<m\neq n<K}\left\{\min\limits_{\substack{\forall x_i\in\Omega_m \\ \forall x_j\in\Omega_n}}\{\|x_i - x_j\|\}\right\}}{\max\limits_{0<m\leq K}\max\limits_{\forall x_i,x_j\in\Omega_m}\{\|x_i - x_j\|\}}$$

预备知识 – 评价指标

内部评价

- Compactness (紧密性)(CP)
- Separation (间隔性)(SP)
- Davies-Bouldin Index (戴维森堡丁指数) (分类适确性指标) (DB)(DBI)
- Dunn Validity Index (邓恩指数)(DVI)

外部评价

- Cluster Accuracy (准确性)(CA)
- Rand index (兰德指数)(RI)、Adjusted Rand index (调整兰德指数)(ARI)
- Normalized Mutual Information (标准互信息)(NMI)、Mutual Information (互信息)(MI)

预备知识 – 评价指标

外部评价

- **Cluster Accuracy (准确性)(CA)**
- **Rand index (兰德指数)(RI)、Adjusted Rand index (调整兰德指数)(ARI)**
- **Normalized Mutual Information (标准互信息)(NMI)、Mutual Information (互信息)(MI)**

Cluster Accuracy (准确性) (CA)

CA 计算聚类正确的百分比

CA **越大**证明聚类效果**越好**

预备知识 – 评价指标

外部评价

- Cluster Accuracy (准确性)(CA)
- Rand index (兰德指数)(RI) 、Adjusted Rand index (调整兰德指数)(ARI)
- Normalized Mutual Information (标准互信息)(NMI)、Mutual Information (互信息)(MI)

Rand index(兰德指数)(RI)

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**，聚类效果准确性越高，同时每个类内的**纯度越高**

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

其中C表示实际类别信息，K表示聚类结果，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，$C_2^{n_{samples}}$其中表示数据集中可以组成的对数

预备知识 – 评价指标

外部评价

- Cluster Accuracy (准确性)(CA)
- Rand index (兰德指数)(RI)、Adjusted Rand index (调整兰德指数)(ARI)
- Normalized Mutual Information (标准互信息)(NMI)、Mutual Information (互信息)(MI)

Adjusted Rand index(调整兰德指数)(ARI)

为了实现“在聚类结果随机产生的情况下，指标应该接近零”，具有更高的区分度。

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。

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

预备知识 – 评价指标

外部评价

- Cluster Accuracy (准确性)(CA)
- Rand index (兰德指数)(RI)、Adjusted Rand index (调整兰德指数)(ARI)
- Normalized Mutual Information (标准互信息)(NMI)、Mutual Information (互信息)(MI)

Mutual Information(互信息)(MI)

是一个随机变量中包含的关于另一个随机变量的信息量，或者说是一个随机变量由于已知另一个随机变量而减少的不确定性

$$I(X,Y)=\sum_{x,y}p(x,y)\log\frac{p(x,y)}{p(x)p(y)}$$

预备知识 – 评价指标

外部评价

- Cluster Accuracy (准确性)(CA)
- Rand index (兰德指数)(RI)、Adjusted Rand index (调整兰德指数)(ARI)
- Normalized Mutual Information (标准互信息)(NMI)、Mutual Information (互信息)(MI)

Normalized Mutual Information (标准互信息)(NMI)

标准化互聚类信息都是用熵做分母将MI值调整到0与1之间

$$U(X,Y) = 2R = 2\frac{I(X;Y)}{H(X)+H(Y)}$$

$$H(X) = \sum_{i=1}^{n} p(x_i)\, I(x_i) = \sum_{i=1}^{n} p(x_i) \log_b \frac{1}{p(x_i)} = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i),$$

预备知识 – 评价指标

Clustering metrics

See the Clustering performance evaluation section of the user guide for further details.

The `sklearn.metrics.cluster` submodule contains evaluation metrics for cluster analysis results. There are two forms of evaluation:

- supervised, which uses a ground truth class values for each sample.
- unsupervised, which does not and measures the 'quality' of the model itself.

`metrics.adjusted_mutual_info_score` (…)	Adjusted Mutual Information between two clusterings.
`metrics.adjusted_rand_score` (labels_true, …)	Rand index adjusted for chance.
`metrics.calinski_harabaz_score` (X, labels)	Compute the Calinski and Harabaz score.
`metrics.completeness_score` (labels_true, …)	Completeness metric of a cluster labeling given a ground truth.
`metrics.fowlkes_mallows_score` (labels_true, …)	Measure the similarity of two clusterings of a set of points.
`metrics.homogeneity_completeness_v_measure` (…)	Compute the homogeneity and completeness and V-Measure scores at once.
`metrics.homogeneity_score` (labels_true, …)	Homogeneity metric of a cluster labeling given a ground truth.
`metrics.mutual_info_score` (labels_true, …)	Mutual Information between two clusterings.
`metrics.normalized_mutual_info_score` (…)	Normalized Mutual Information between two clusterings.
`metrics.silhouette_score` (X, labels[, …])	Compute the mean Silhouette Coefficient of all samples.
`metrics.silhouette_samples` (X, labels[, metric])	Compute the Silhouette Coefficient for each sample.
`metrics.v_measure_score` (labels_true, labels_pred)	V-measure cluster labeling given a ground truth.

预备知识 – 距离度量

距离度量

- **曼哈顿距离**(Manhattan Distance)：

  来源于城市区块距离，是将多个维度上的距离进行求和后的结果

$$dist(X,Y) = \sum_{i=1}^{n} |x_i - y_i|$$

- **欧几里得距离**（Euclidean Distance）

  最常见的距离度量，衡量的是多维空间中各个点之间的绝对距离

$$dist(X,Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

预备知识 – 距离度量

距离度量

- **明可夫斯基距离**(Minkowski Distance)：

  欧氏距离的推广，是对多个距离度量公式的概括性的表述

$$dist(X,Y) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{1/p}$$

- **切比雪夫距离**（Chebyshev Distance）：

  起源于国际象棋中国王的走法，即当p趋向于无穷大时的明氏距离

$$dist(X,Y) = \lim_{p\to\infty} (\sum_{i=1}^{n} |x_i - y_i|^p)^{1/p}$$

预备知识 – 距离度量

距离度量

- **切比雪夫距离**（Chebyshev Distance）：

  起源于国际象棋中国王的走法，即当p趋向于无穷大时的明氏距离

$$dist(X,Y) = \lim_{p\to\infty}(\sum_{i=1}^{n}|x_i - y_i|^p)^{1/p}$$

  另一种形式： 各坐标数值差绝对值的最大值

  Eg: 空间中的两点 p 和 q ，各个维度记做 p_i 或 q_i

$$D_{Cebyshev}(p,q) := \max(|p_i - q_i|)$$

  在二维空间上，

$$D_{Cebyshev}(p,q) = \max(|x_1 - x_2|, |y_1 - y_2|)$$

预备知识 – 距离度量

相似度量

- **向量空间余弦相似度**(Cosine Similarity)：

  向量空间中两个向量夹角的余弦值作为衡量两个个体间差异的大小。

  余弦相似度更加注重两个向量在方向上的差异，而非距离或长度上

$$sim(X,Y) = cos\theta = \frac{\vec{x} \cdot \vec{y}}{\|x\|\|y\|}$$

预备知识 – 距离度量

相似度量

- **皮尔森相关系数**(Pearson Correlation Coefficient)：

  相关分析中的相关系数r

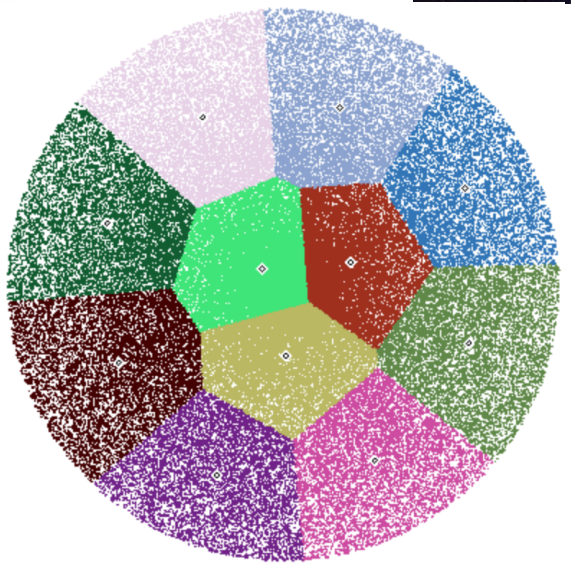
  分别对X和Y基于自身总体标准化后计算空间向量的余弦夹角

$$r(X,Y)=\frac{cov(X,Y)}{\sigma_X\sigma_Y}=\frac{\sum_{i=1}^{n}(X_i-\bar{X})(Y_i-\bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i-\bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i-\bar{Y})^2}}$$

OUTLINES

1. 预备知识
2. Clustering 问题定义
3. Clustering 算法概述
4. 算法 (k-means, BIRCH, DBSCAN, Spectral)
5. Clustering算法选型

Clustering - 问题定义



Clustering - 问题定义

聚类：Clustering

本质：寻找数据的**内容结构**，将数据划分成**有意义**或**有用**的**簇**

目标：**簇内**的对象相互之间是**相似**的，而**簇间**的对象是不同的

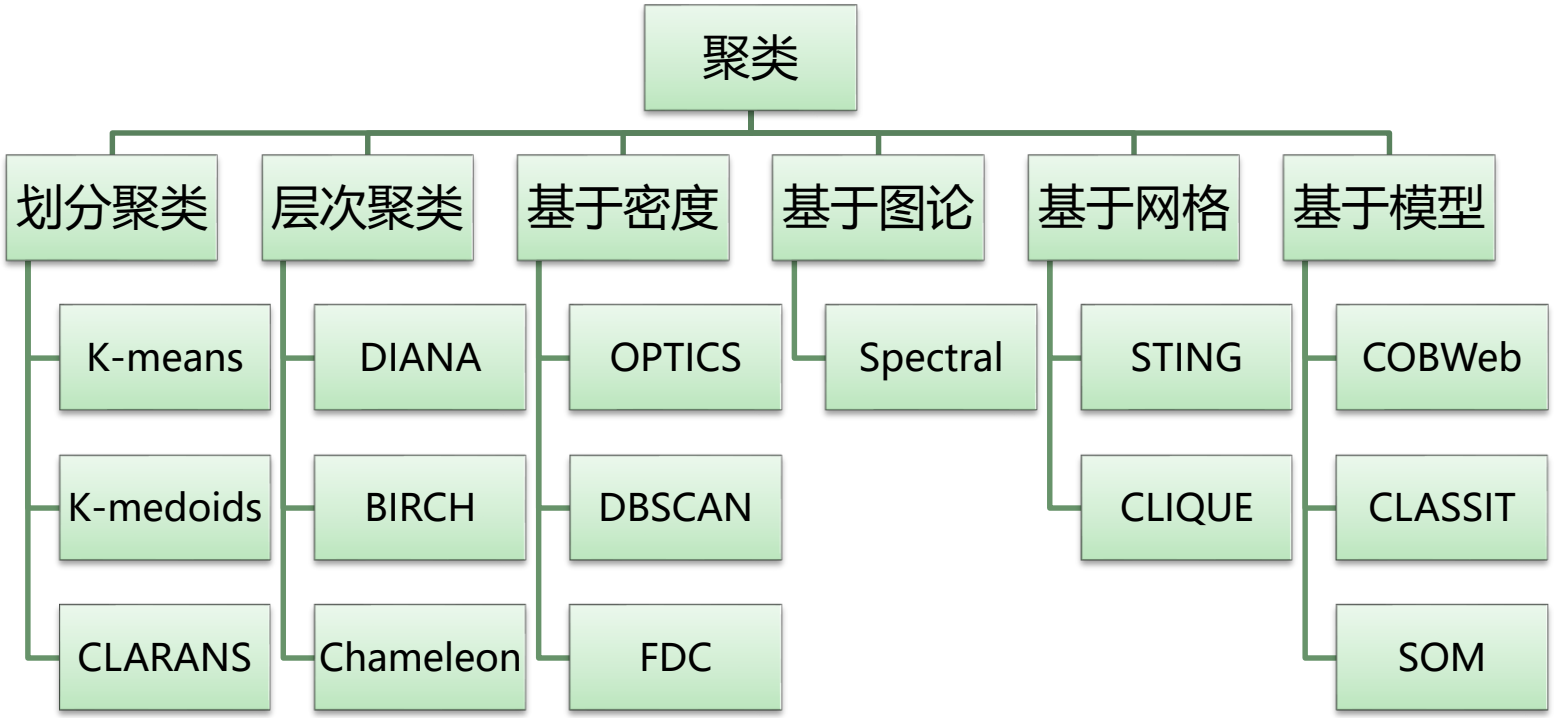
核心：**相似度**计算

通常并不需要使用标签训练数据进行学习，即**unsupervised learning (无监督学习)**。

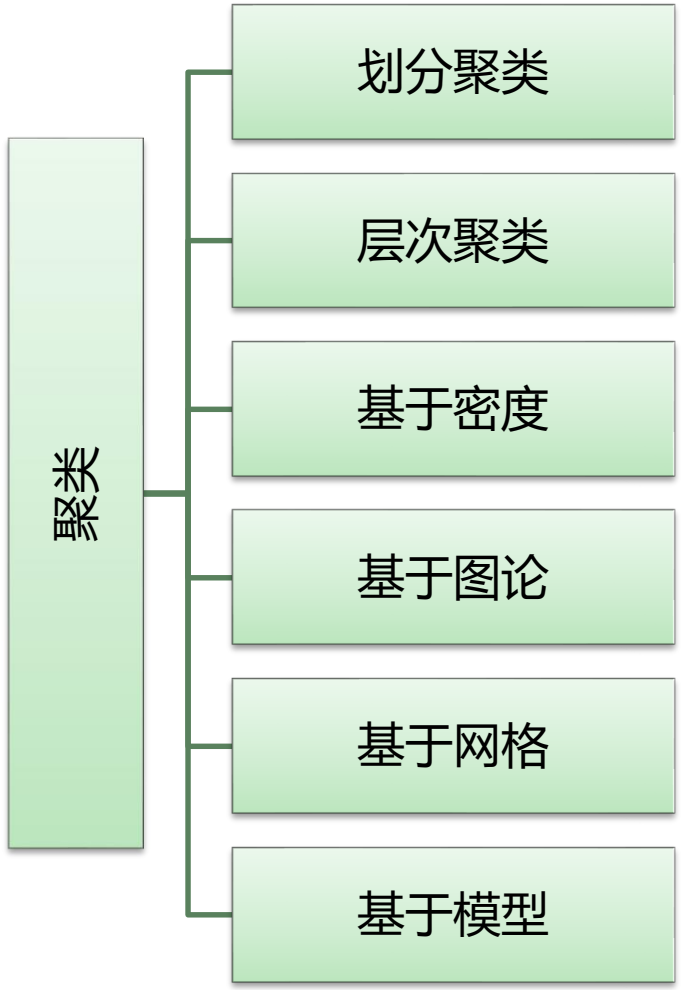
OUTLINES

1. 预备知识
2. Clustering 问题定义
3. Clustering 算法概述
4. 算法 (k-means, BIRCH, DBSCAN, Spectral)
5. Clustering算法选型

Clustering – 算法概述



Clustering – 算法概述



划分法（Partitioning methods）

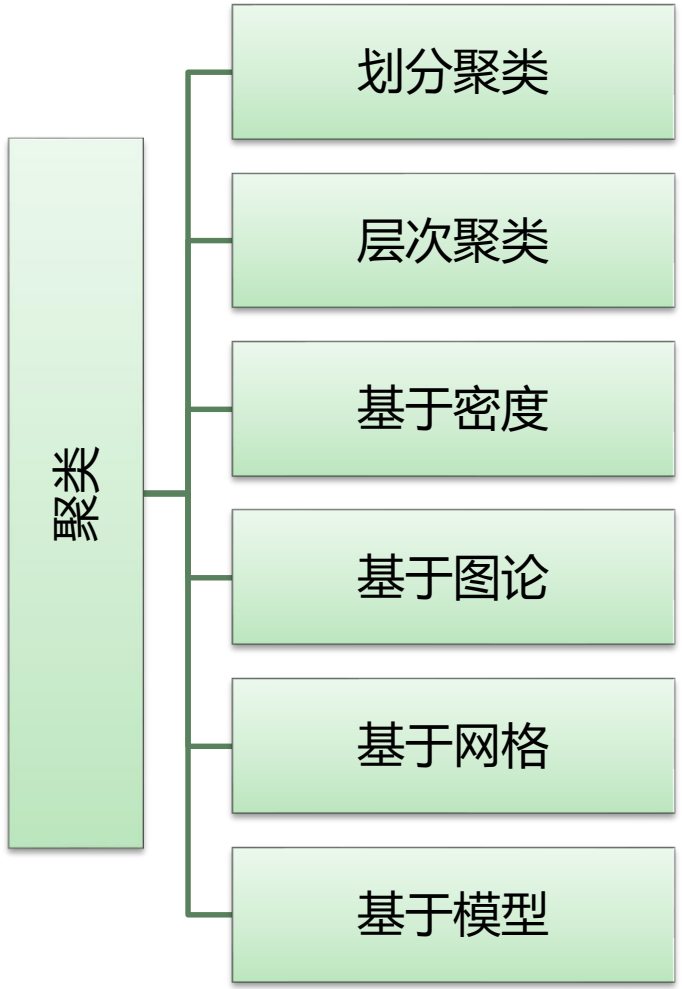
给定一个有N个元组或者纪录的数据集，

分裂法将构造K个分组，每一个分组就代表一个聚类

每一个分组至少包含一个数据纪录

每一个数据纪录属于且仅属于一个分组（模糊聚类除外）

Clustering – 算法概述

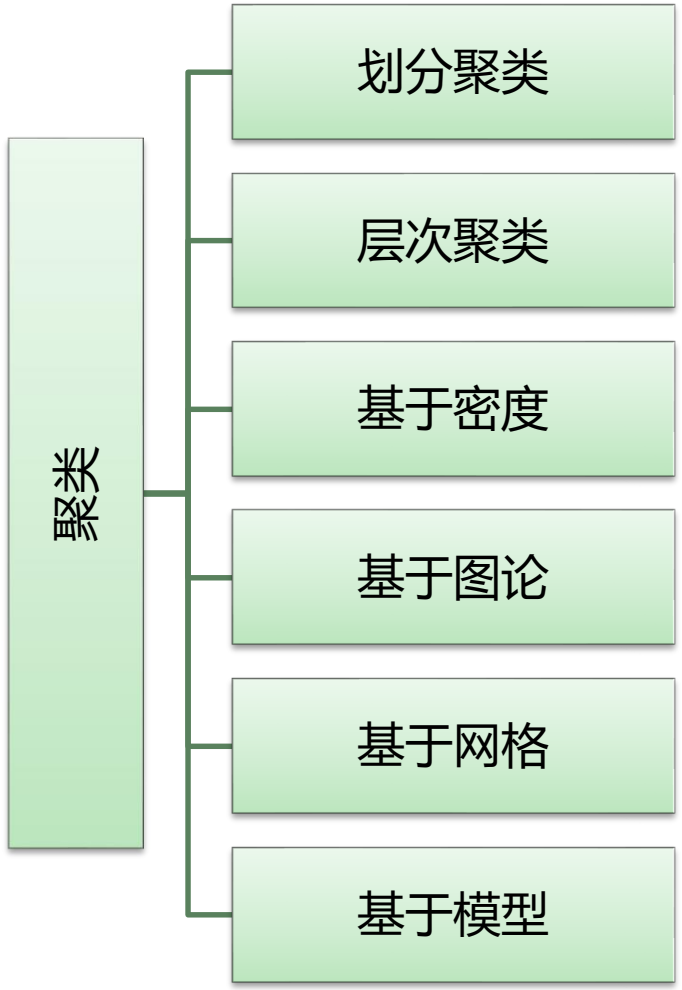


划分法（Partitioning methods）

对于给定的K，
算法首先给出一个初始的分组方法，
之后通过反复迭代的方法改变分组，
使得每一次改进之后的分组方案都较前一次好

标准：
同一分组中的记录越近越好，而不同分组中的纪录越远越好。

Clustering – 算法概述



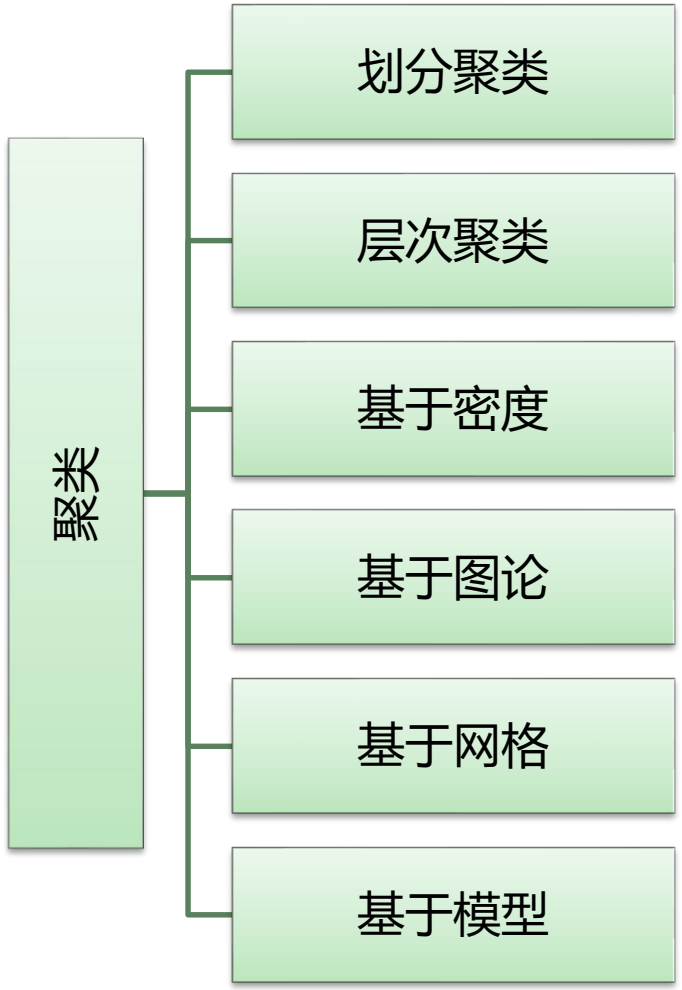
层次法（Hierarchical Clustering）

对给定的数据集进行层次似的分解，

直到某种条件满足为止。

分为“自底向上”和“自顶向下”两种方案。

Clustering – 算法概述



层次法（Hierarchical Clustering）

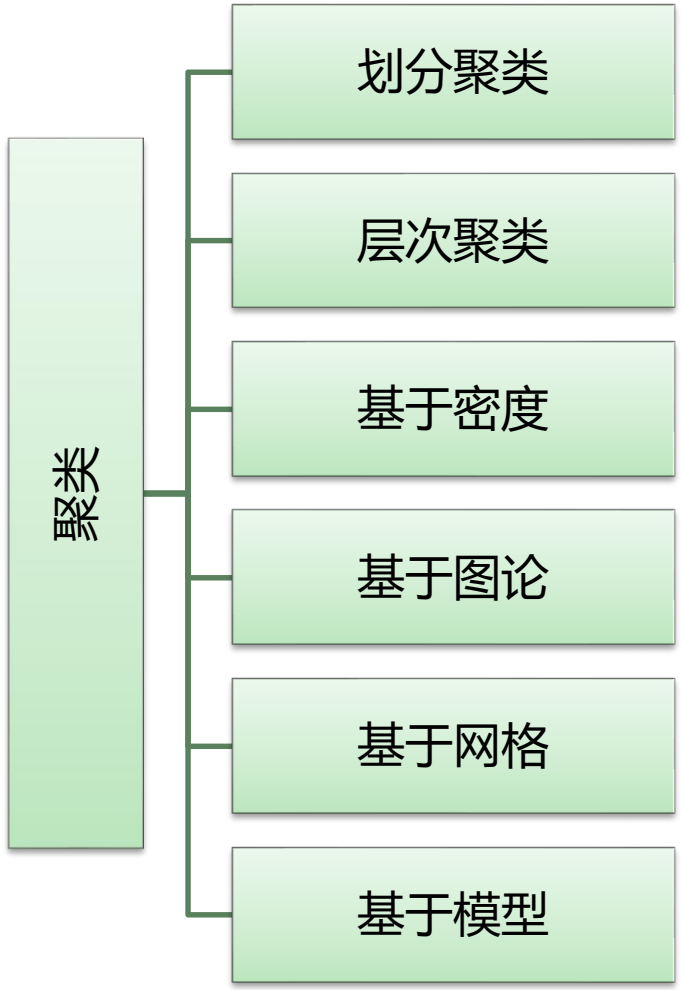
Eg: 在“自底向上”方案中，

初始时每一个数据纪录都组成一个单独的组，

在接下来的迭代中，它把那些相互邻近的组合并成一个组，

直到所有的记录组成一个分组或者某个条件满足为止。

Clustering – 算法概述

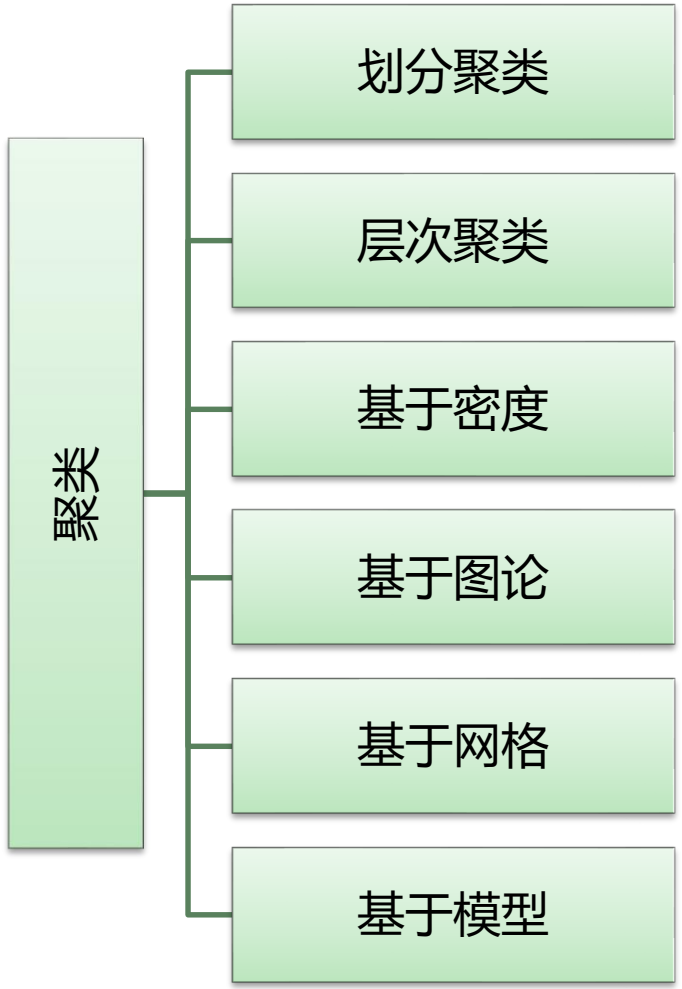


基于密度的聚类（density-based methods）

核心思想：

只要一个区域中的点的密度大过某个阈值，就把它加到与之相近的聚类中去。

Clustering – 算法概述



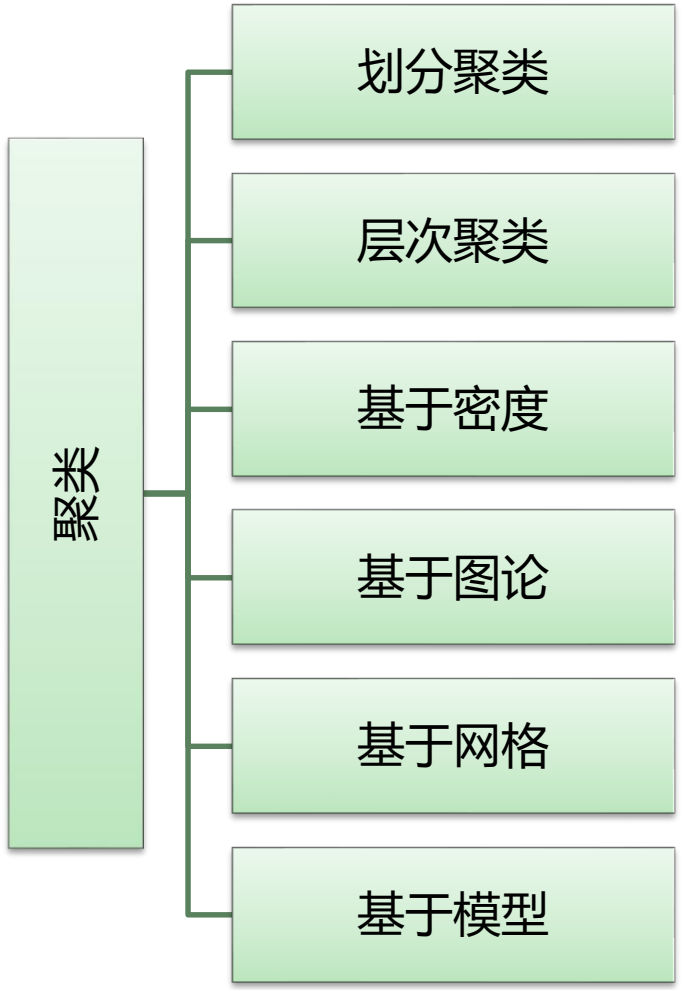
基于密度的聚类（density-based methods）

与其它方法的一个根本区别是：

它不是基于各种各样的距离的，而是基于密度的。

这样能克服基于距离的算法只能发现“类圆形”的聚类的缺点。

Clustering – 算法概述



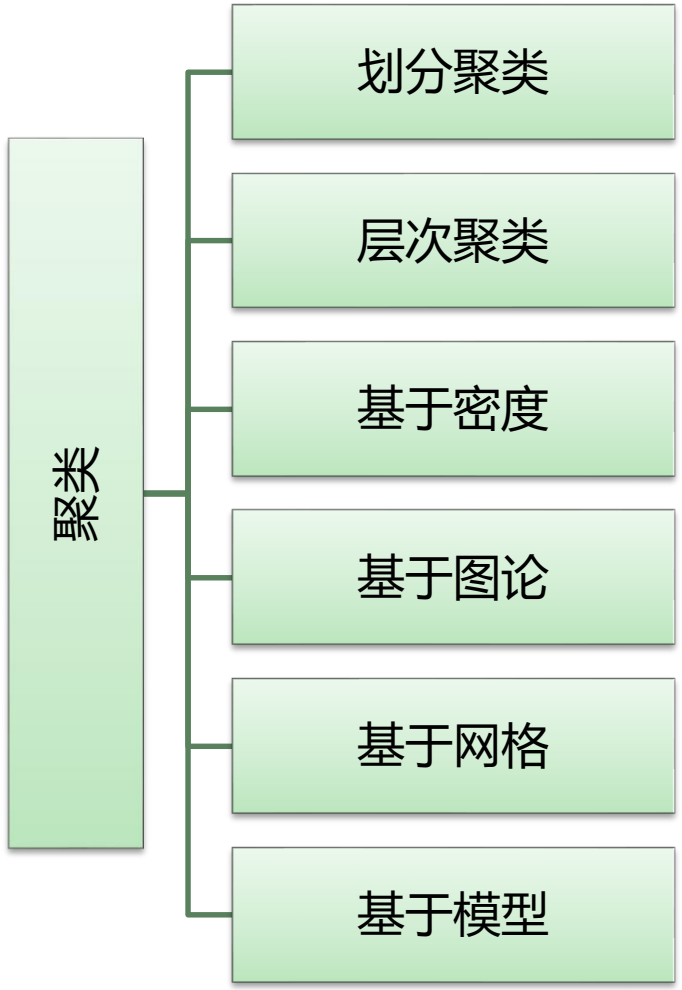
基于图的聚类（Graph-based methods）

建立与问题相适应的图，

图的节点对应被分析数据的最小单元，

图的边（或弧）对应于最小处理单元数据之间的相似性度量。

Clustering – 算法概述

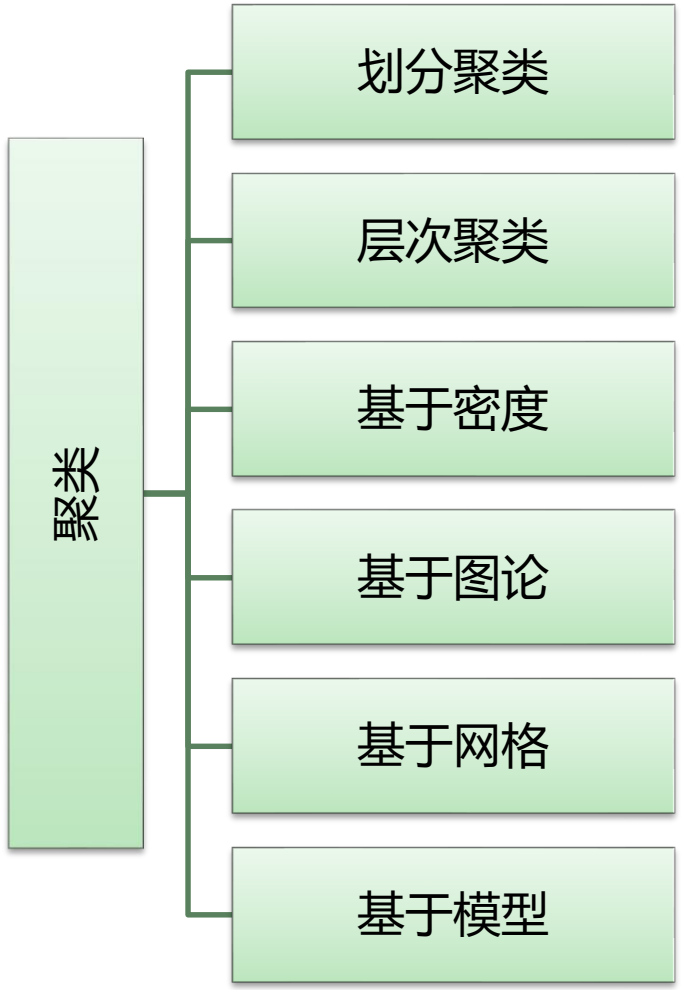


基于图的聚类（Graph-based methods）

每一个最小处理单元数据之间都会有一个度量表达，这就确保了数据的局部特性比较易于处理。

图论聚类法是以样本数据的局域连接特征作为聚类的主要信息源，因而其主要优点是易于处理局部数据的特性。

Clustering – 算法概述



基于网格的方法（grid-based methods)

将数据空间划分成为有限个单元（cell）的网格结构，

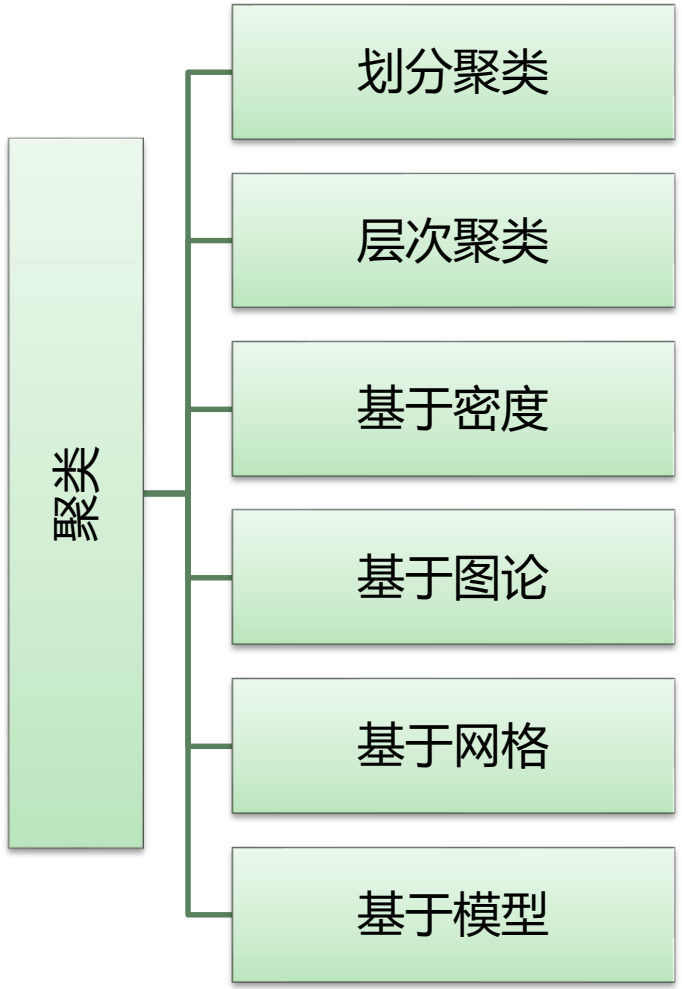
所有的处理都是以单个的单元为对象的。

优点就是处理速度很快，

通常这是与目标数据库中记录的个数无关的，

它只与把数据空间分为多少个单元有关。

Clustering – 算法概述



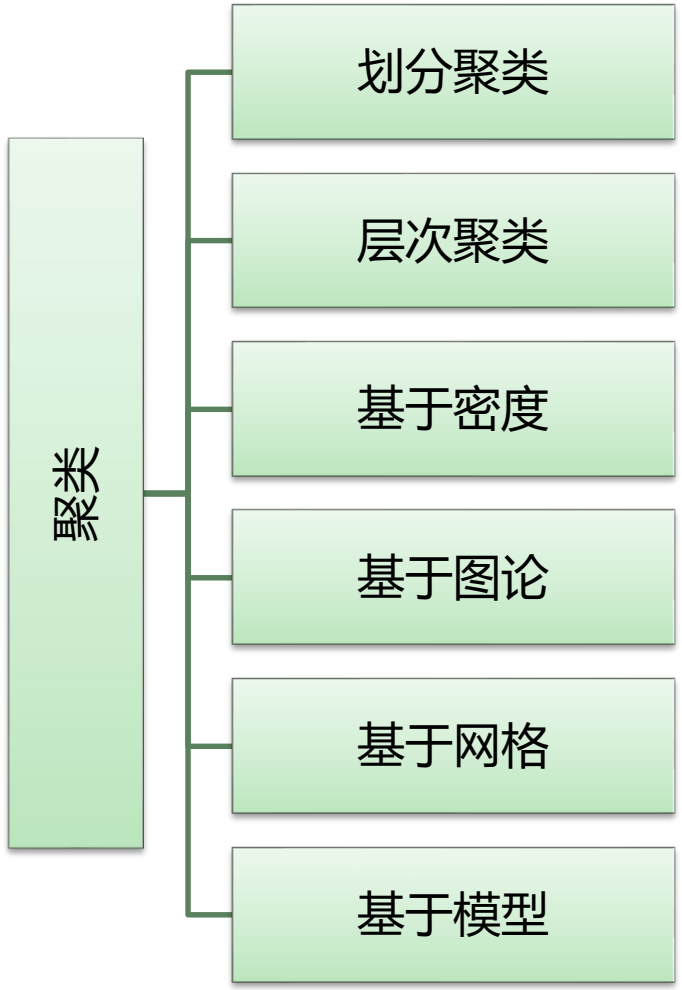
基于模型的方法（model-based methods）

给每一个聚类假定一个模型，

然后去寻找能够很好的满足这个模型的数据集。

这样一个模型可能是数据点在空间中的密度分布函数或者其它。

Clustering – 算法概述



基于模型的方法（model-based methods）

潜在的假定：

目标数据集是由一系列的概率分布所决定的。

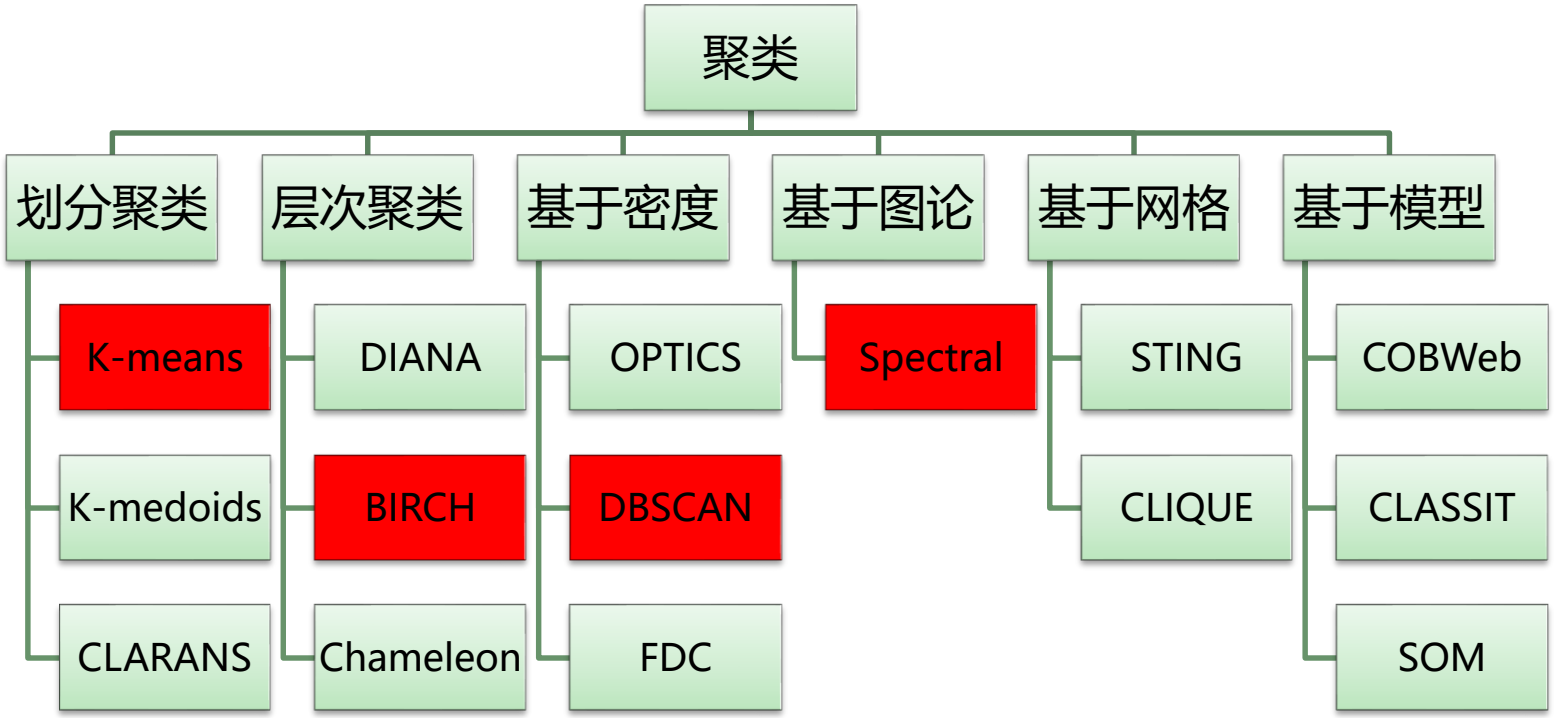
通常有两种尝试方向：

统计的方案和神经网络的方案。

OUTLINES

1. 预备知识
2. Clustering 问题定义
3. Clustering 算法概述
4. 算法 (k-means, BIRCH, DBSCAN, Spectral)
5. Clustering算法选型

Clustering – 算法



Clustering – 算法 - K-means

基本流程：

1. 初始化**常数K**（意味着最终的聚类类别数）
2. **随机**选定初始点为**各个簇的质心**
3. 计算每个样本与质心之间的**距离/相似度**，将样本点归到最相似的类中
4. **更新**每个簇的**质心**(即为簇中心)，
5. **重复**3, 4过程，直到质心不再改变，最终就确定了每个样本所属的类别以及每个簇的质心。

Clustering – 算法 - K-means

K-Means & KNN

两个算法都包含一个过程，即找出和某一个点最近的点。

两者都利用了**最近邻(nearest neighbors)**的思想。

Clustering – 算法 - K-means

K-Means & KNN

K-Means是**无监督学习**的**聚类算法**，没有样本输出；
而KNN是**监督学习**的**分类算法**，有对应的类别输出。

KNN基本不需要训练，对测试集里面的点只需要找到在训练集中最近的k个点，用这最近的k个点的类别来决定测试点的类别。
而K-Means则有明显训练过程，找到k个类别的最佳质心，从而决定样本的簇类别。

Clustering – 算法 - K-means

由于每次都要计算所有的样本与每一个质心之间的相似度，故在大规模的数据集上，K-Means算法的收敛速度比较慢。

从实际问题出发，人工指定比较合理的K值，通过多次随机初始化聚类中心选取比较满意的结果。

Clustering – 算法 - K-means

由于 K-means 算法的分类结果会受到初始点的选取而有所区别，因

此有提出这种算法的改进: K-means++

初始质心选取的基本思路就是:

初始的聚类中心之间的相互距离要尽可能的远。

K-means++ 能显著的改善分类结果的最终误差。

Clustering – 算法 - K-means

K-means++ 算法描述如下：

步骤一：

随机选取一个样本作为第一个聚类中心 c1；

步骤二：

计算每个样本与当前已有类聚中心最短距离（即与最近一个聚类中心的距离），用 D(x)表示；这个值越大，表示被选取作为聚类中心的概率较大；最后，用轮盘法选出下一个聚类中心；

步骤三：

重复步骤二，知道选出 k 个聚类中心。

Clustering – 算法 - K-means

尽管计算初始点时花费了额外的时间，但是在迭代过程中，k-mean 本身能快速收敛，因此算法实际上降低了计算时间。

有人使用真实和合成的数据集测试了他们的方法，速度通常提高了 2 倍，对于某些数据集，误差提高了近 1000 倍。

Clustering – 算法 - K-means

K-Means的主要优点:

1. 原理比较简单，实现也是很容易，收敛速度快。
2. 聚类效果较优。
3. 算法的可解释度比较强。
4. 主要需要调参的参数仅仅是簇数k。

Clustering – 算法 - K-means

K-Means的主要缺点:

1. K值的选取不好把握
2. 对于不是凸的数据集比较难收敛
3. 如果各隐含类别的数据不平衡，比如各隐含类别的数据量严重失衡，或者各隐含类别的方差不同，则聚类效果不佳。
4. 采用迭代方法，得到的结果只是局部最优。
5. 对噪音和异常点比较的敏感。

Clustering – 算法 - K-means

sklearn.cluster.KMeans

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=1, algorithm='auto'*) [source]

n_clusters : int, optional, default: 8

The number of clusters to form as well as the number of centroids to generate.

init : {'k-means++', 'random' or an ndarray}

Method for initialization, defaults to 'k-means++':

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose k observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

Clustering – 算法 - K-means

sklearn.cluster.KMeans

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=1, algorithm='auto'*) [source]

n_init : int, default: 10

Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

max_iter : int, default: 300

Maximum number of iterations of the k-means algorithm for a single run.

tol : float, default: 1e-4

Relative tolerance with regards to inertia to declare convergence

Clustering – 算法 - K-means

sklearn.cluster.KMeans

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=1, algorithm='auto'*) [source]

precompute_distances : {'auto', True, False}

Precompute distances (faster but takes more memory).

'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.

True : always precompute distances

False : never precompute distances

random_state : int, RandomState instance or None, optional, default: None

If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

Clustering – 算法 - K-means

```
from time import time
import numpy as np
import matplotlib.pyplot as plt

from sklearn import metrics
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale

np.random.seed(42)
```

```
# load dataset
digits = load_digits()
data = scale(digits.data) #Center to the mean and component wise scale to unit variance.
```

```
data.shape
```

```
(1797, 64)
```

```
n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target
```

```
labels
```

```
array([0, 1, 2, ..., 8, 9, 8])
```

```
sample_size = 300
print("n_digits: %d, \t n_samples %d, \t n_features %d"
      % (n_digits, n_samples, n_features))
```

```
n_digits: 10,    n_samples 1797,         n_features 64
```

Clustering – 算法 - K-means

```
def bench_k_means(estimator, name, data):
    t0 = time()
    estimator.fit(data)
    print(82 * '_')
    print('init\t\ttime\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette')
    print('%-9s\t%.2fs\t%i\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f'
          % (name, (time() - t0), estimator.inertia_,
             metrics.homogeneity_score(labels, estimator.labels_),
             metrics.completeness_score(labels, estimator.labels_),
             metrics.v_measure_score(labels, estimator.labels_),
             metrics.adjusted_rand_score(labels, estimator.labels_),
             metrics.adjusted_mutual_info_score(labels,  estimator.labels_),
             metrics.silhouette_score(data, estimator.labels_,
                                      metric='euclidean',
                                      sample_size=sample_size)))
```

```
bench_k_means(KMeans(init='k-means++', n_clusters=n_digits, n_init=10),
              name="k-means++", data=data)

bench_k_means(KMeans(init='random', n_clusters=n_digits, n_init=10),
              name="random", data=data)
```

```
__________________________________________________________________________________
init            time    inertia homo    compl   v-meas  ARI     AMI     silhouette
k-means++       0.14s   69753   0.666   0.713   0.688   0.548   0.662   0.146
__________________________________________________________________________________
init            time    inertia homo    compl   v-meas  ARI     AMI     silhouette
random          0.12s   69442   0.604   0.653   0.628   0.468   0.600   0.126
```

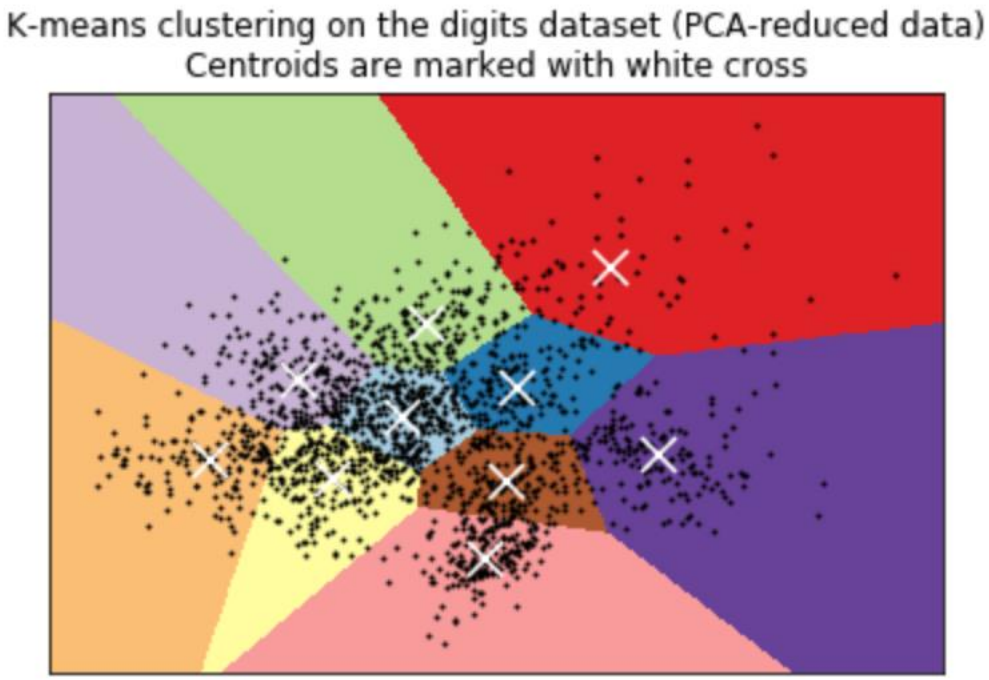
Clustering – 算法 - K-means

```
# Visualize the results on PCA-reduced data

reduced_data = PCA(n_components=2).fit_transform(data)
kmeans = KMeans(init='k-means++', n_clusters=n_digits, n_init=10)
kmeans.fit(reduced_data)

# Step size of the mesh. Decrease to increase the quality of the VQ.
h = .02     # point in the mesh [x_min, x_max]x[y_min, y_max].

# Plot the decision boundary. For that, we will assign a color to each
x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Obtain labels for each point in mesh. Use last trained model.
Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
```

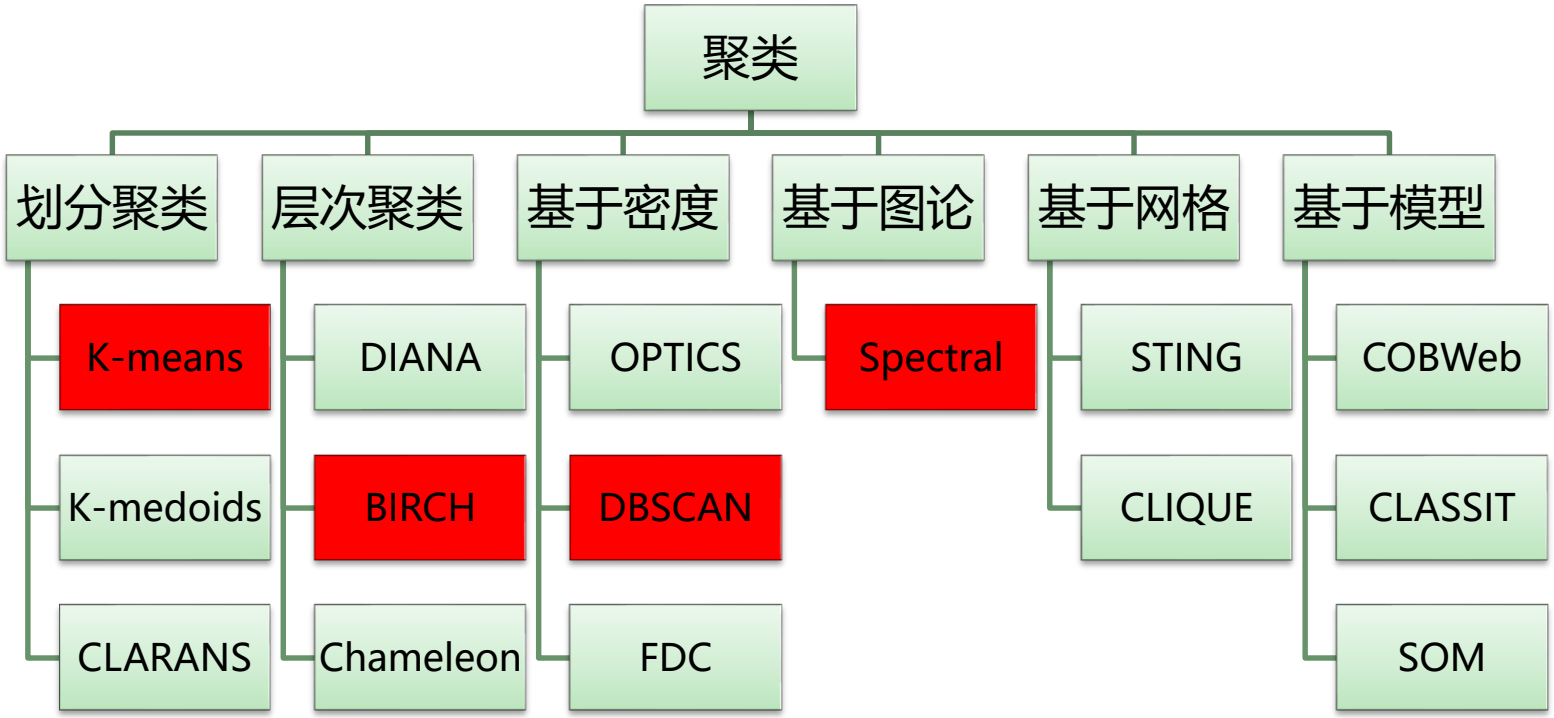
Clustering – 算法 - K-means

```
plt.figure(1)
plt.clf()
plt.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')

plt.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=2)

# Plot the centroids as a white X
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)

plt.title('K-means clustering on the digits dataset (PCA-reduced data)\n'
          'Centroids are marked with white cross')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()
```

Clustering – 算法 - K-means



Clustering – 算法



Clustering – 算法

BIRCH，利用**层次方法**的**平衡迭代规约和聚类**

（Balanced Iterative Reducing and Clustering Using Hierarchies）

BIRCH算法利用了一个树结构来帮助我们快速的聚类，

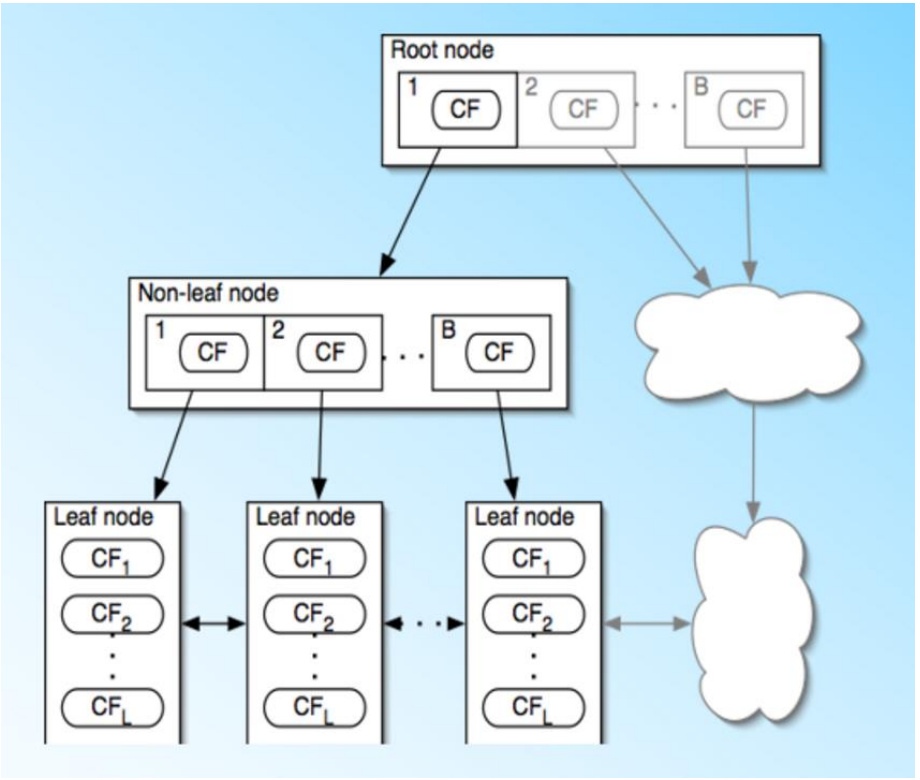
这个树结构类似于平衡B+树，

一般将它称之为**聚类特征树**(Clustering Feature Tree，简称CF Tree)

这颗树的每一个节点是由若干个聚类特征(Clustering Feature，简称CF)组成

Clustering – 算法 - BIRCH

每个节点包括叶子节点都有若干个CF，而内部节点的CF有指向孩子节点的**指针**，所有的叶子节点用一个**双向链表链**接起来。



Clustering – 算法 - BIRCH

每一个CF是一个**三元组**，可以用（N，LS，SS）表示。

其中N代表了这个CF中拥有的样本点的数量

LS代表了这个CF中拥有的样本点各特征维度的和向量

SS代表了这个CF中拥有的样本点各特征维度的平方和

Clustering – 算法 - BIRCH

比如，在CF Tree中的某一个节点的某一个CF中，

有下面5个样本(3,4), (2,6), (4,5), (4,7), (3,8)。

则：

N = 5，

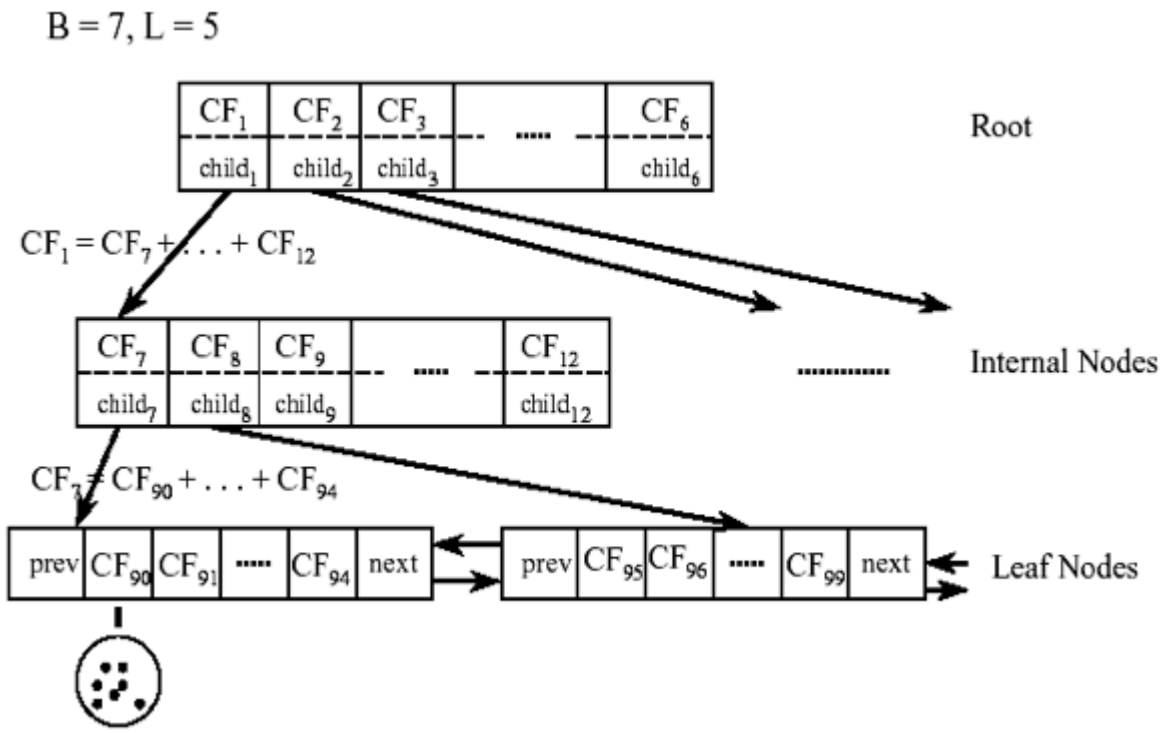
LS = (3+2+4+4+3, 4+6+5+7+8) = (16,30)

SS = (3^2+2^2+4^2+4^2+3^2+4^2+6^2+5^2+7^2+8^2) = (54+190) = 244

Clustering – 算法 - BIRCH

CF 满足线性条件:

$$CF_1 + CF_2 = (N_1 + N_2, LS_1 + LS_2, SS_1 + SS_2)$$



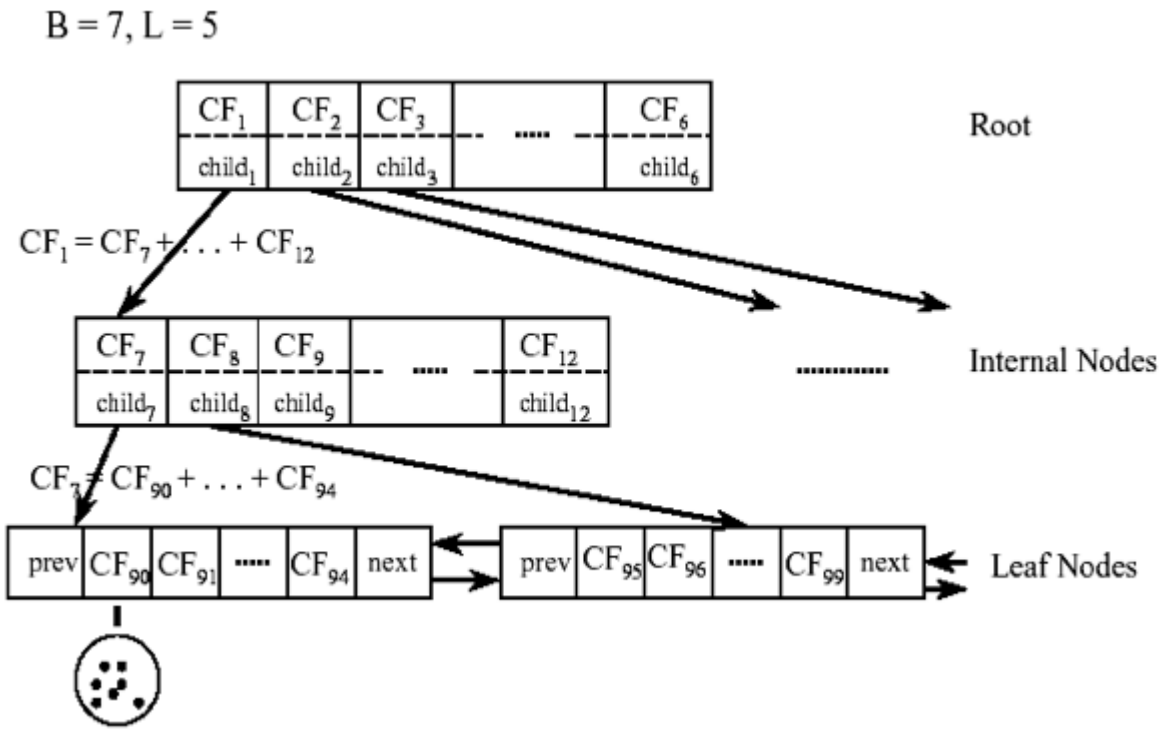
Clustering – 算法 - BIRCH

重要参数：

- 每个**内部节点**的最大CF数B
- 每个**叶子节点**的最大CF数L
- 叶节点每个CF的**最大**样本**半径阈值**T

  一个CF中的所有样本点一定要在半径小于T的一个超球体内

Clustering – 算法 - BIRCH



对于上图中的CF Tree，限定了B=7， L=5， 也就是说内部节点最多有7个CF，而叶子节点最多有5个CF。

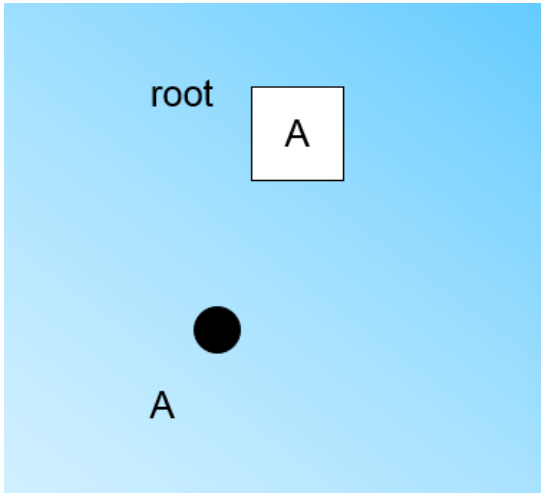
Clustering – 算法 - BIRCH

构建CF Tree

1. 在最开始的时候，CF Tree是空的，没有任何样本，我们从训练集

读入第一个样本点，将它放入一个新的CF三元组A，这个三元组的

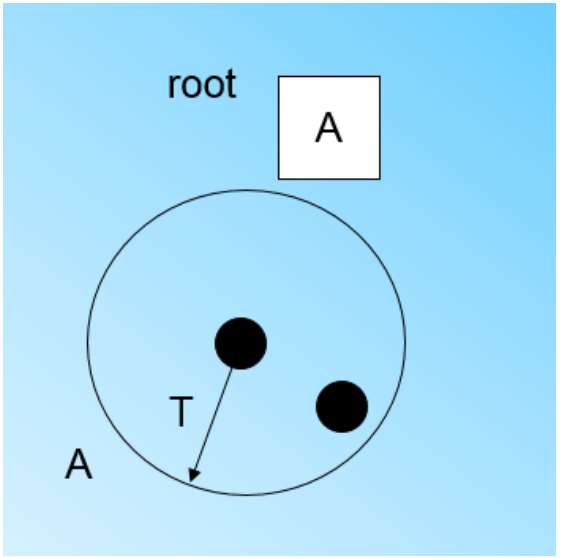
N=1，将这个新的CF放入根节点



Clustering – 算法 - BIRCH

构建CF Tree

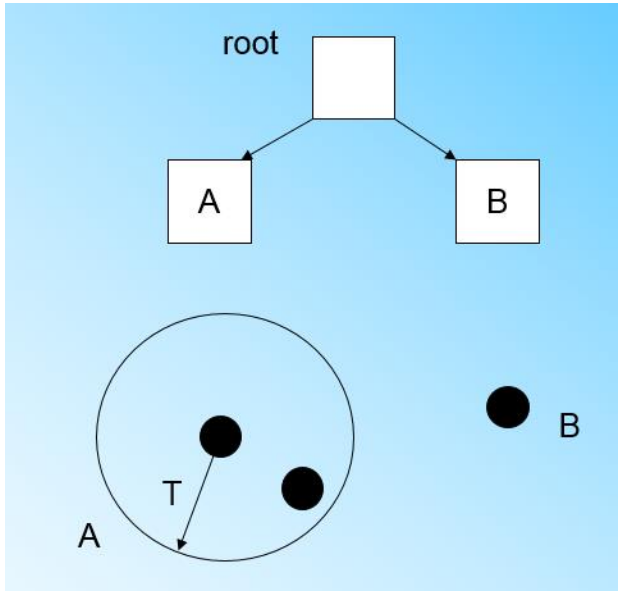
2. 继续读入第二个样本点，我们发现这个样本点和第一个样本点A，在半径为T的超球体范围内，也就是说，他们属于一个CF，我们将第二个点也加入CF A,此时需要更新A的三元组的值。此时A的三元组中N=2



Clustering – 算法 - BIRCH

构建CF Tree

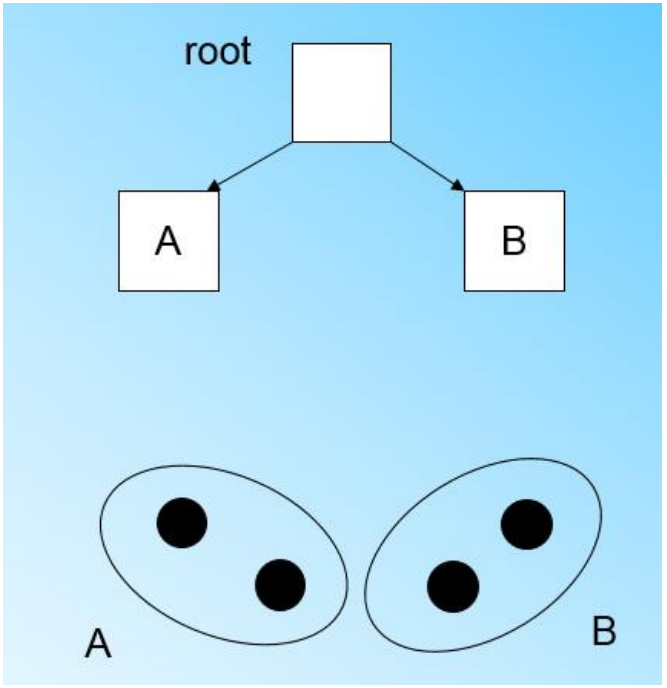
3. 读入第三个节点，结果我们发现这个节点不能融入刚才前面的节点形成的超球体内，也就是说，我们需要一个新的CF三元组B，来容纳这个新的值。此时根节点有两个CF三元组A和B



Clustering – 算法 - BIRCH

构建CF Tree

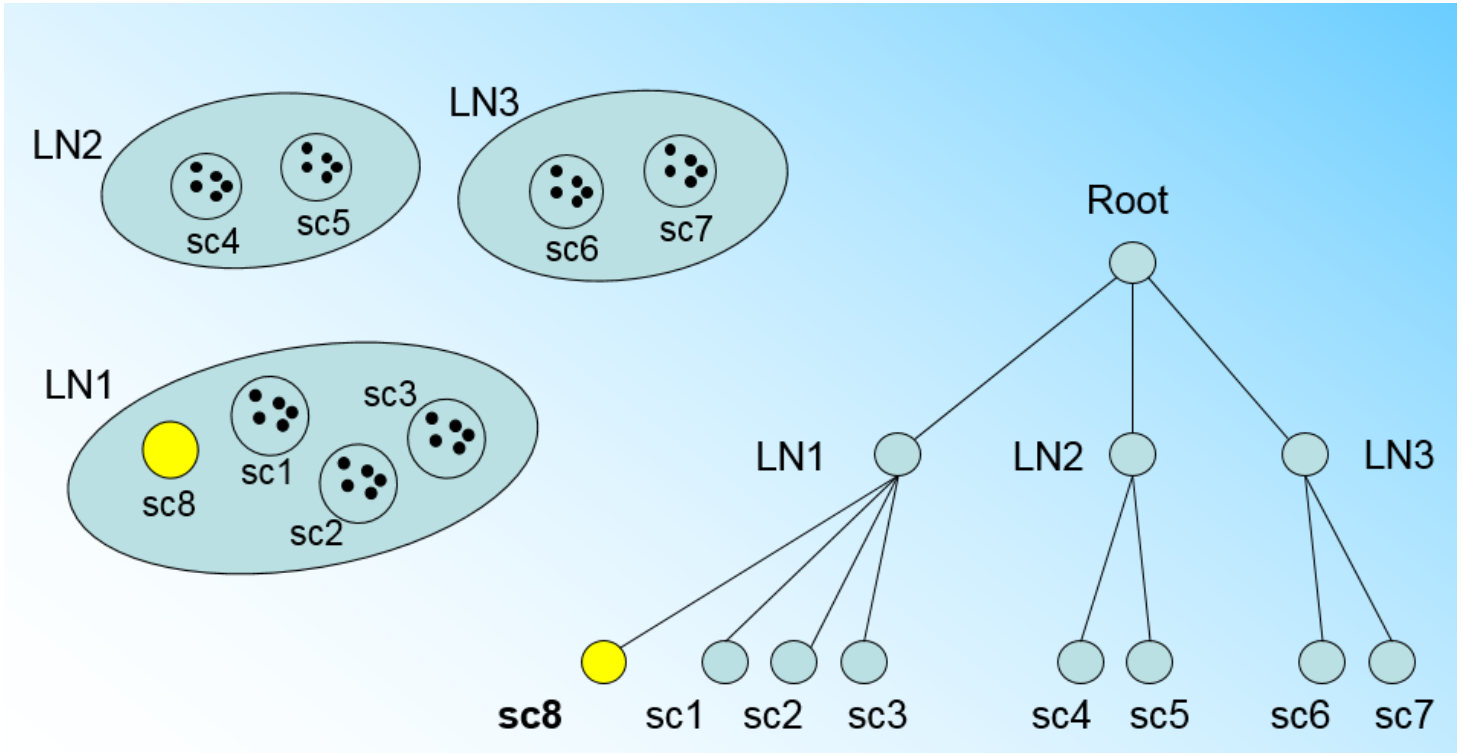
4. 读入第四个样本点的时候，我们发现和B在半径小于T的超球体



Clustering – 算法 - BIRCH

构建CF Tree

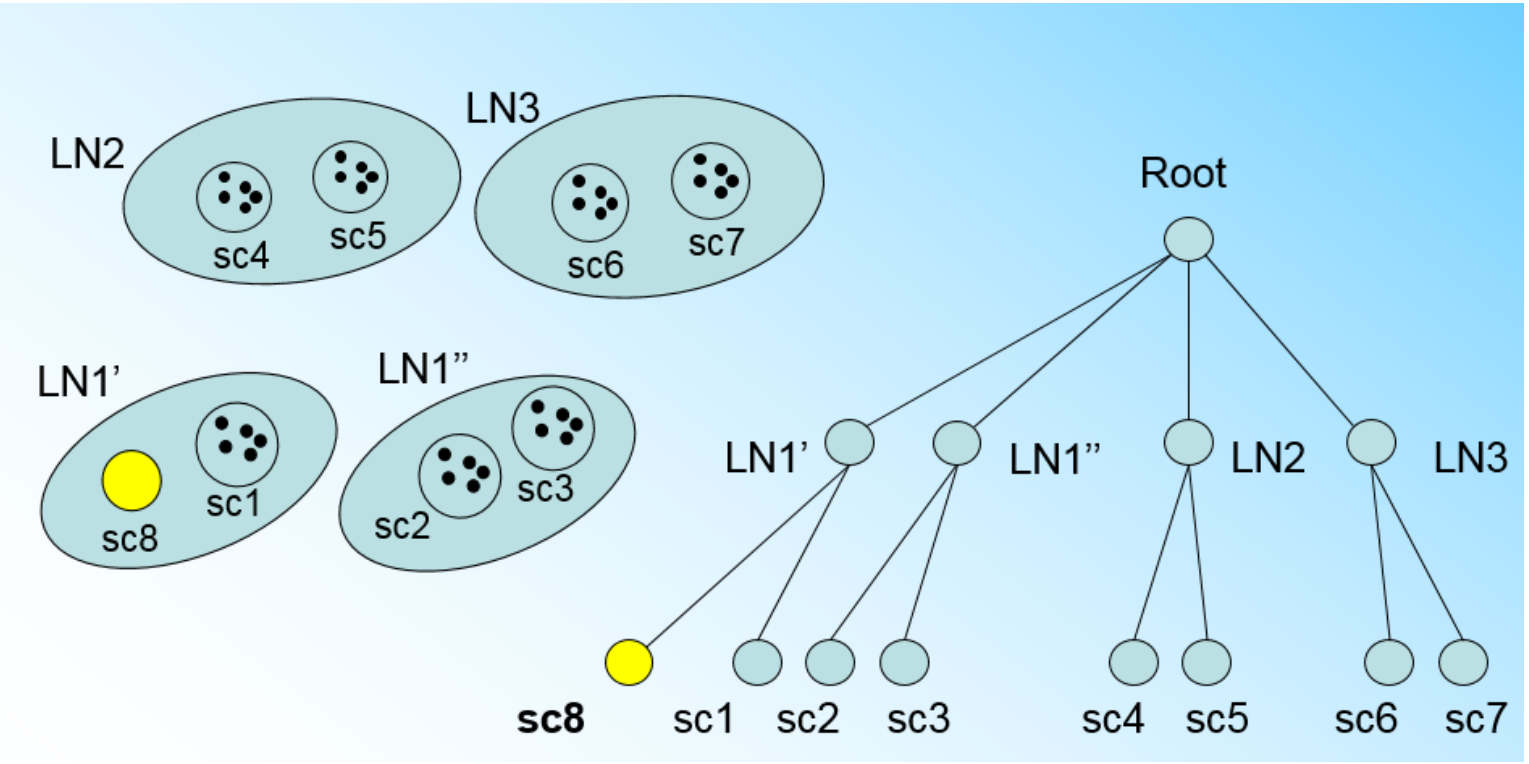
当构造CF超过叶子节点的最大CF数L时，需要进行结点分裂



Clustering – 算法 - BIRCH

构建CF Tree

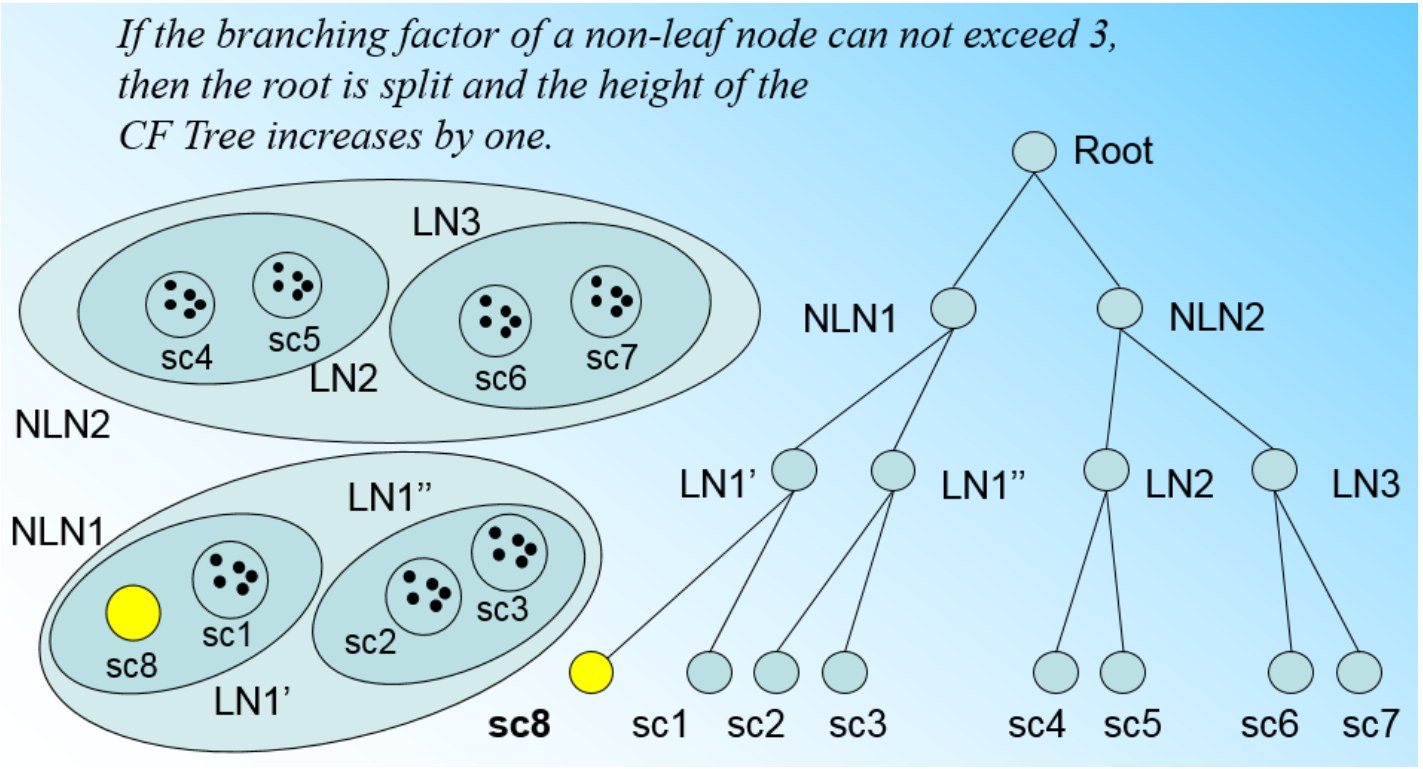
当构造CF超过内部节点的最大CF数B时，需要进行结点分裂



Clustering – 算法 - BIRCH

构建CF Tree

当构造CF超过根节点的最大CF数时，需要进行结点分裂



Clustering – 算法 - BIRCH

CF Tree的插入：

1. 从根节点向下寻找和新样本**距离最近的叶子节点**和叶子节点里**最近的CF节点**

2. 如果新样本加入后，这个CF节点对应的超球体半径仍然**满足小于阈值T**，则更新路径上所有的CF三元组，插入结束。否则转入3.

Clustering – 算法 - BIRCH

CF Tree的插入：

3. 如果当前叶子节点的CF节点个数小于阈值L，则创建一个新的CF节点，放入新样本，将新的CF节点放入这个叶子节点，更新路径上所有的CF三元组，插入结束。否则转入4。

Clustering – 算法 - BIRCH

CF Tree的插入：

4. 将当前叶子节点划分为两个新叶子节点，选择旧叶子节点中所有CF元组里超球体距离最远的两个CF元组，分别作为两个新叶子节点的第一个CF节点。将其他元组和新样本元组按照距离远近原则放入对应的叶子节点。依次向上检查父节点是否也要分裂，如果需要按和叶子节点分裂方式相同。

Clustering – 算法 - BIRCH

BIRCH算法流程

1. 将**所有的样本依次读入**，在内存中建立一颗CF Tree

2. （可选）将第一步建立的CF Tree进行筛选，去除一些异常CF节点，这些节点一般里面的样本点很少。对于一些超球体距离非常近的元组进行合并

Clustering – 算法 - BIRCH

BIRCH算法流程

3. （可选）利用其它的一些聚类算法比如K-Means对所有的CF元组进行聚类，得到一颗比较好的CF Tree.（主要目的是消除由于样本读入顺序导致的不合理的树结构，以及一些由于节点CF个数限制导致的树结构分裂）

Clustering – 算法 - BIRCH

BIRCH算法流程

4. （可选）利用第三步生成的CF Tree的所有CF节点的质心，作为初始质心点，对所有的样本点按距离远近进行聚类，从而进一步减少了由于CF Tree的一些限制导致的聚类不合理的情况

Clustering – 算法 - BIRCH

BIRCH算法可以不用输入类别数K值，这点和K-Means不同。

如果不输入K值，则最后的**CF元组的组数**即为最终的K，

否则会按照输入的K值对CF元组按距离大小进行合并。

对于调参，BIRCH要比K-Means复杂，

因为它需要对CF Tree的几个关键的参数进行调参，

这几个参数对CF Tree的最终形式影响很大

Clustering – 算法 - BIRCH

BIRCH算法适用于**样本量较大**的情况，也适用于**类别数比较大**的情况。

BIRCH除了聚类还可以：

异常点检测和数据初步按类别规约的预处理。

但是如果数据特征的维度非常大，则BIRCH不太适合，

比如大于20，此时Mini Batch K-Means的表现较好。

Clustering – 算法 - BIRCH

BIRCH算法的主要优点有：

1) 节约内存，所有的样本都在磁盘上，CF Tree仅仅存了CF节点和对应的指针。

2) 聚类速度快，只需要一遍扫描训练集就可以建立CF Tree，CF Tree的增删改都很快。

3) 可以识别噪音点，还可以对数据集进行初步分类的预处理

Clustering – 算法 - BIRCH

BIRCH算法的主要缺点有：

1) 由于CF Tree对每个节点的CF个数有限制，导致聚类的结果可能和真实的类别分布不同.

2) 对高维特征的数据聚类效果不好。此时可以选择Mini Batch K-Means

3) 如果数据集的分布簇不是类似于超球体，或者说不是凸的，则聚类效果不好。

Clustering – 算法 - BIRCH

sklearn.cluster.Birch

class sklearn.cluster. **Birch** (*threshold=0.5, branching_factor=50, n_clusters=3, compute_labels=True, copy=True*)
[source]

threshold : float, default 0.5

The radius of the subcluster obtained by merging a new sample and the closest subcluster should be lesser than the threshold. Otherwise a new subcluster is started. Setting this value to be very low promotes splitting and vice-versa.

branching_factor : int, default 50

Maximum number of CF subclusters in each node. If a new samples enters such that the number of subclusters exceed the branching_factor then that node is split into two nodes with the subclusters redistributed in each. The parent subcluster of that node is removed and two new subclusters are added as parents of the 2 split nodes.

Clustering – 算法 - BIRCH

sklearn.cluster.Birch

class sklearn.cluster. **Birch** (*threshold=0.5, branching_factor=50, n_clusters=3, compute_labels=True, copy=True*)
[source]

n_clusters : int, instance of sklearn.cluster model, default 3

Number of clusters after the final clustering step, which treats the subclusters from the leaves as new samples.

- None : the final clustering step is not performed and the subclusters are returned as they are.
- sklearn.cluster Estimator : If a model is provided, the model is fit treating the subclusters as new samples and the initial data is mapped to the label of the closest subcluster.
- int : the model fit is `AgglomerativeClustering` with n_clusters set to be equal to the int.

compute_labels : bool, default True

Whether or not to compute labels for each fit.

copy : bool, default True

Whether or not to make a copy of the given data. If set to False, the initial data will be overwritten.

Clustering – 算法 - BIRCH

```
# clustering - BIRCH
```

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

```
# Generate centers for the blobs so that it forms a 10 X 10 grid.
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.newaxis, convenient alias for None
                       np.ravel(yy)[:, np.newaxis]))
```

```
# Generate blobs to do a comparison between MiniBatchKMeans and Birch.
X, y = make_blobs(n_samples=100000, centers=n_centres, random_state=0)
```

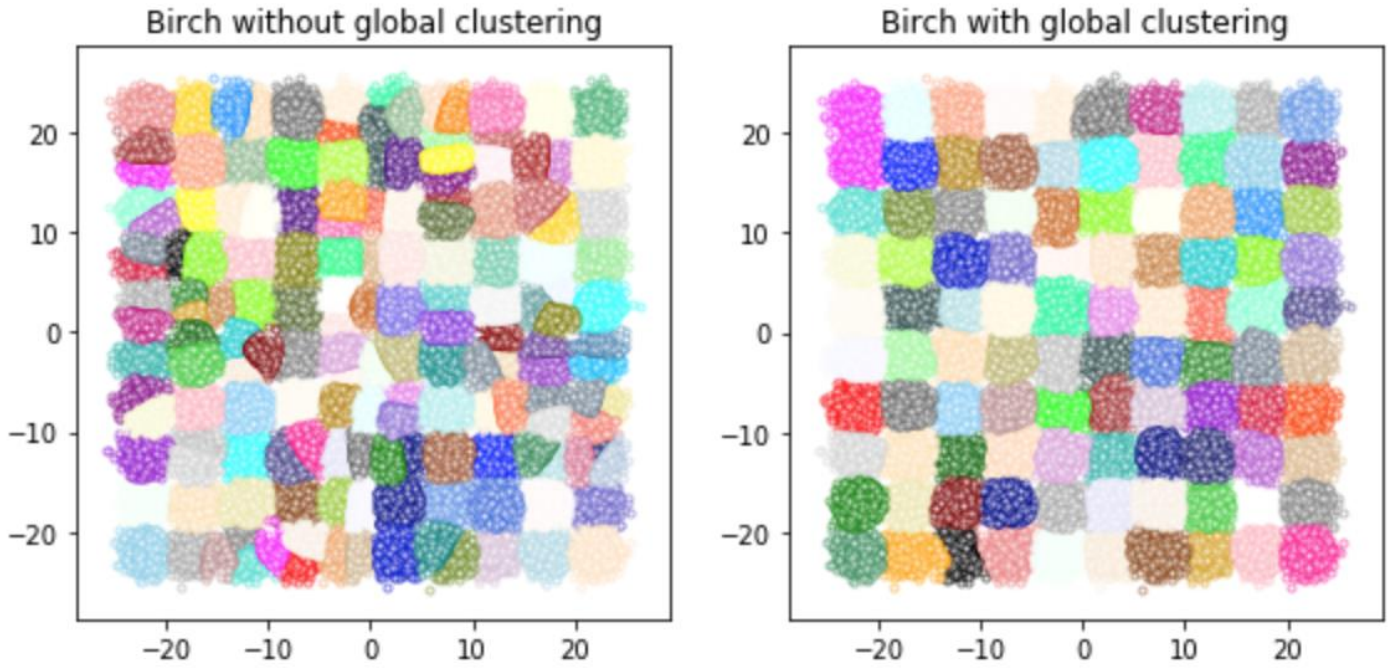
```
# model
brich_0 = Birch(threshold=1.7, n_clusters=None)
brich_1 = Birch(threshold=1.7, n_clusters=100)

models = [brich_0, brich_1]
models_info = ['without global clustering', 'with global clustering']
```

Clustering – 算法 - BIRCH

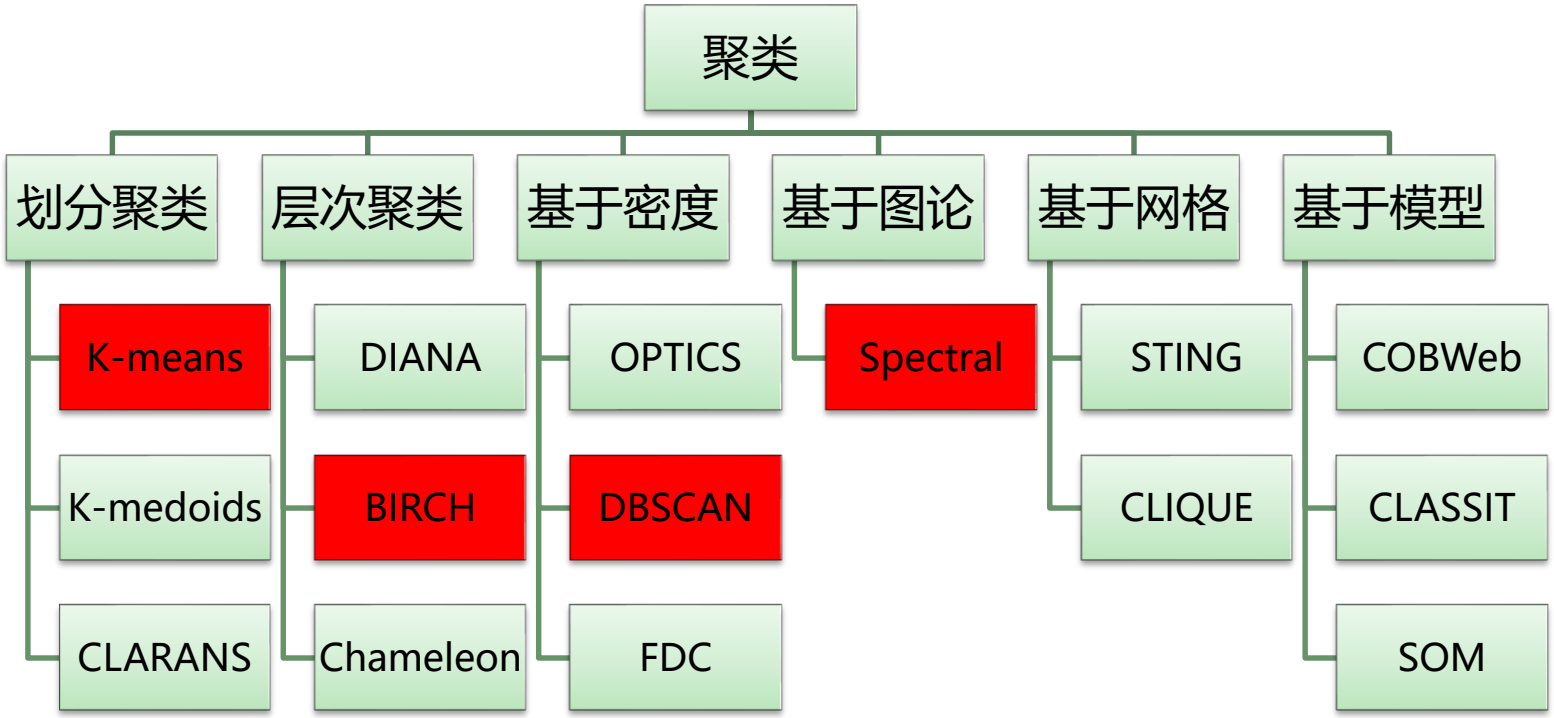
```
# trainning
from itertools import cycle
import matplotlib.colors as colors

# Use all colors that matplotlib provides by default.
colors_ = cycle(colors.cnames.keys())

fig = plt.figure(figsize=(12, 4))
fig.subplots_adjust(left=0.04, right=0.98, bottom=0.1, top=0.9)

for idx, (model, info) in enumerate(zip(models, models_info)):
    t_s = time()
    model.fit(X)
    t_d = time() - t_s

    labels = model.labels_
    centroids = model.subcluster_centers_
    n_clusters = np.unique(labels).size

    print("Birch %s: n_clusters %d, take %0.2f seconds" % (info, n_clusters, t_d))

    ax = fig.add_subplot(1, 3, idx + 1)
    for this_centroid, k, col in zip(centroids, range(n_clusters), colors_):
        mask = labels == k
        ax.scatter(X[mask, 0], X[mask, 1], c='w', edgecolor=col, marker='.', alpha=0.5)
    ax.set_title('Birch %s' % info)
```

Clustering – 算法 - BIRCH

```
Birch without global clustering: n_clusters 158, take 2.15 seconds
Birch with global clustering: n_clusters 100, take 2.14 seconds
```



Clustering – 算法



Clustering – 算法 - DBSCAN

DBSCAN，具有噪声的**基于密度**的聚类方法

Density-Based Spatial Clustering of Applications with Noise

一种很典型的密度聚类算法，假定类别可以通过样本分布的**紧密程度**决定

和K-Means，BIRCH这些一般只适用于凸样本集的聚类相比，

DBSCAN既可以适用于凸样本集，也可以适用于非凸样本集

Clustering – 算法 - DBSCAN

DBSCAN是基于一组邻域来描述样本集的紧密程度的，

参数$(\epsilon, MinPts)$用来描述邻域的样本分布紧密程度。

其中，

ϵ 描述了某一样本的邻域距离阈值，

$MinPts$描述了某一样本的距离为ϵ的邻域中样本个数的阈值。

Clustering – 算法 - DBSCAN

假设我的样本集是 D=(x_1,x_2,...,x_m) DBSCAN具体的密度描述定义如下：

1. ϵ-邻域：

对于$x_j \in D$，其ϵ-邻域包含样本集D中与x_j的距离不大于ϵ的子样本集，即

$$N_\epsilon(x_j) = \{x_i \in D | distance(x_i, xj) \leq \epsilon\}$$

这个子样本集的个数记为$|N\epsilon(xj)|$

2. 核心对象：

对于任一样本$x_j \in D$，如果其ϵ-邻域对应的$N\epsilon(x_j)$**至少包含$MinPts$个样本**，

即如果$|N\epsilon(x_j)| \geq MinPts$，则$x_j$是核心对象。

Clustering – 算法 - DBSCAN

假设我的样本集是 $D=(x_1,x_2,...,x_m)$ DBSCAN具体的密度描述定义如下：

3. 密度直达：

如果x_i位于x_j的ϵ-邻域中，且x_j是核心对象，则称x_i由x_j密度直达。

注意反之不一定成立，即此时不能说x_j由x_i密度直达, 除非且x_i也是核心对象。

Clustering – 算法 - DBSCAN

假设我的样本集是 D=(x_1,x_2,...,x_m) DBSCAN具体的密度描述定义如下：

4. 密度可达：

对于x_i和x_j, 如果存在样本序列$p_1, p_2, \dots, p_T$, 满足$p_1 = x_i, p_T = x_j$, **且**

$\boldsymbol{p_{t+1}}$**由**$\boldsymbol{p_t}$**密度直达**，则称x_j由x_i密度可达。也就是说，密度可达满足

传递性。

此时序列中的传递样本$p_1, p_2, \dots, p_{T-1}$均为核心对象

因为只有核心对象才能使其他样本密度直达

注意密度可达也不满足对称性，这个可以由密度直达的不对称性得

出

Clustering – 算法 - DBSCAN

假设我的样本集是 D=(x_1,x_2,...,x_m) DBSCAN具体的密度描述定义如下：

5. 密度相连：

对于x_i和x_j，如果存在核心对象样本x_k，使x_i和x_j均由x_k密度可达，则称x_i和x_j密度相连。

注意密度相连关系是满足**对称性**的。

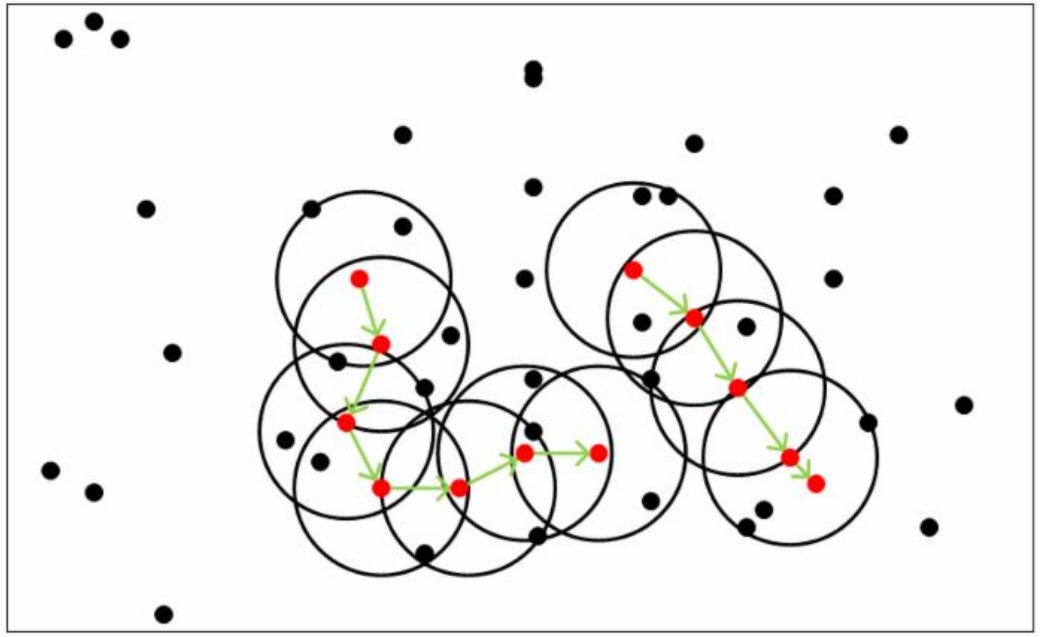
由密度可达关系导出的最大密度相连的样本集合，即为我们最终聚类的一个类别，或者说一个簇。

Clustering – 算法 - DBSCAN

Eg: MinPts=5，红色的点都是核心对象，因为其ϵ-邻域至少有5个样本

黑色的样本是非核心对象，绿色箭头连起来的核心对象组成了密度可达的

样本序列



Clustering – 算法 - DBSCAN

Prob_1: 异常样本点

一些异常样本点或者说少量游离于簇外的样本点，

这些点不在任何一个核心对象在周围，

在DBSCAN中，我们一般将这些样本点标记为噪音点。

Clustering – 算法 - DBSCAN

Prob_2: 距离的度量问题

即如何计算某样本和核心对象样本的距离。

在DBSCAN中，一般采用**最近邻思想**，采用某一种距离度量来衡量样本距离，比如欧式距离。

这和KNN分类算法的最近邻思想完全相同。

对应少量的样本，寻找最近邻可以直接去计算所有样本的距离，

如果样本量较大，则一般采用KD树或者球树来快速的搜索最近邻。

Clustering – 算法 - DBSCAN

Prob_3: 争议样本点

某些样本可能到两个核心对象的距离都小于ϵ，

但是这两个核心对象由于不是密度直达，又不属于同一个聚类簇，

那么如果界定这个样本的类别呢？

一般来说，此时DBSCAN采用**先来后到**，先进行聚类的类别簇会标记这个样本为它的类别。也就是说BDSCAN的算法不是完全稳定的算法。

Clustering – 算法 - DBSCAN

DBSCAN的主要优点：

1. 可以对任意形状的稠密数据集进行聚类，相对的，K-Means之类的聚类算法一般只适用于凸数据集。
2. 可以在聚类的同时发现异常点，对数据集中的异常点不敏感。
3. 聚类结果没有偏倚，相对的，K-Means之类的聚类算法初始值对聚类结果有很大影响。

Clustering – 算法 - DBSCAN

DBSCAN的主要缺点：

1. 如果样本集的密度不均匀、聚类间距差相差很大时，聚类质量较差，这时用DBSCAN聚类一般不适合。
2. 如果样本集较大时，聚类收敛时间较长，此时可以对搜索最近邻时建立的KD树或者球树进行规模限制来改进。
3. 调参相对于传统的K-Means之类的聚类算法稍复杂，主要需要对**距离阈值ϵ**，**邻域样本数阈值MinPts**联合调参，不同的参数组合对最后的聚类效果有较大影响。

Clustering – 算法 - DBSCAN

sklearn.cluster.DBSCAN

class sklearn.cluster. **DBSCAN** (*eps=0.5, min_samples=5, metric='euclidean', metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=1*) [source]

eps : float, optional

The maximum distance between two samples for them to be considered as in the same neighborhood.

min_samples : int, optional

The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.

metric : string, or callable

The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by metrics.pairwise.calculate_distance for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a sparse matrix, in which case only "nonzero" elements may be considered neighbors for DBSCAN.

Clustering – 算法 - DBSCAN

sklearn.cluster.DBSCAN

class sklearn.cluster. **DBSCAN** (*eps=0.5, min_samples=5, metric='euclidean', metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=1*) [source]

metric_params : dict, optional

Additional keyword arguments for the metric function.

New in version 0.19.

algorithm : {'auto', 'ball_tree', 'kd_tree', 'brute'}, optional

The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. See NearestNeighbors module documentation for details.

leaf_size : int, optional (default = 30)

Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

Clustering – 算法 - DBSCAN

sklearn.cluster.DBSCAN

class sklearn.cluster. **DBSCAN** (*eps=0.5, min_samples=5, metric='euclidean', metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=1*) [source]

p : float, optional

The power of the Minkowski metric to be used to calculate distance between points.

n_jobs : int, optional (default = 1)

The number of parallel jobs to run. If `-1`, then the number of jobs is set to the number of CPU cores.

Clustering – 算法 - DBSCAN

```
# clustering - DBSCAN
```

```
from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
```

```
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=750, centers=centers, cluster_std=0.4, random_state=0)

X = StandardScaler().fit_transform(X)
```

```
# model
dbs = DBSCAN(eps=0.3, min_samples=10)
dbs.fit(X)
```

```
DBSCAN(algorithm='auto', eps=0.3, leaf_size=30, metric='euclidean',
    metric_params=None, min_samples=10, n_jobs=None, p=None)
```

Clustering – 算法 - DBSCAN

```
# core_samples selection, excluding noise sample
core_samples_mask = np.zeros_like(dbs.labels_, dtype=bool)
core_samples_mask[dbs.core_sample_indices_] = True
labels = dbs.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)
```

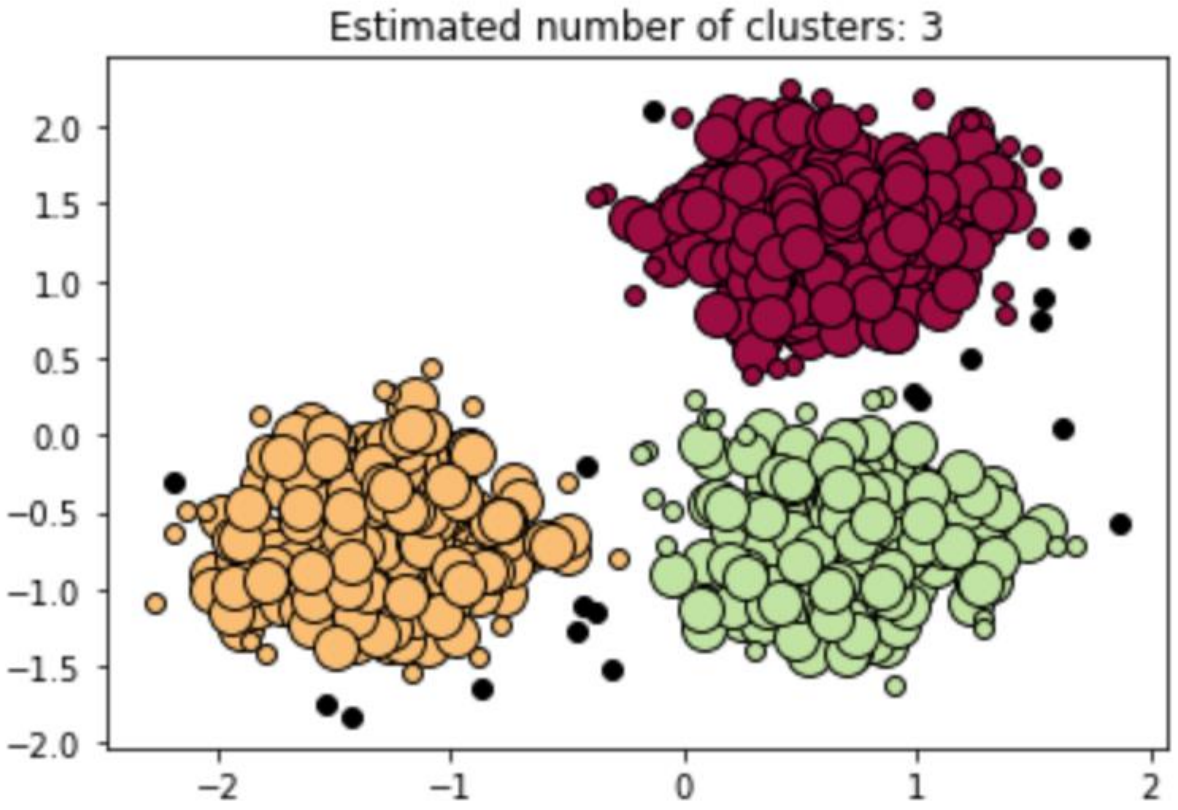
```
print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true, labels))
print("Completeness: %0.3f" % metrics.completeness_score(labels_true, labels))
print("Adjusted Rand Index: %0.3f"% metrics.adjusted_rand_score(labels_true, labels))
print("Adjusted Mutual Information: %0.3f"% metrics.adjusted_mutual_info_score(labels_true, labels))
```

```
Estimated number of clusters: 3
Estimated number of noise points: 18
Homogeneity: 0.953
Completeness: 0.883
Adjusted Rand Index: 0.952
Adjusted Mutual Information: 0.883
```

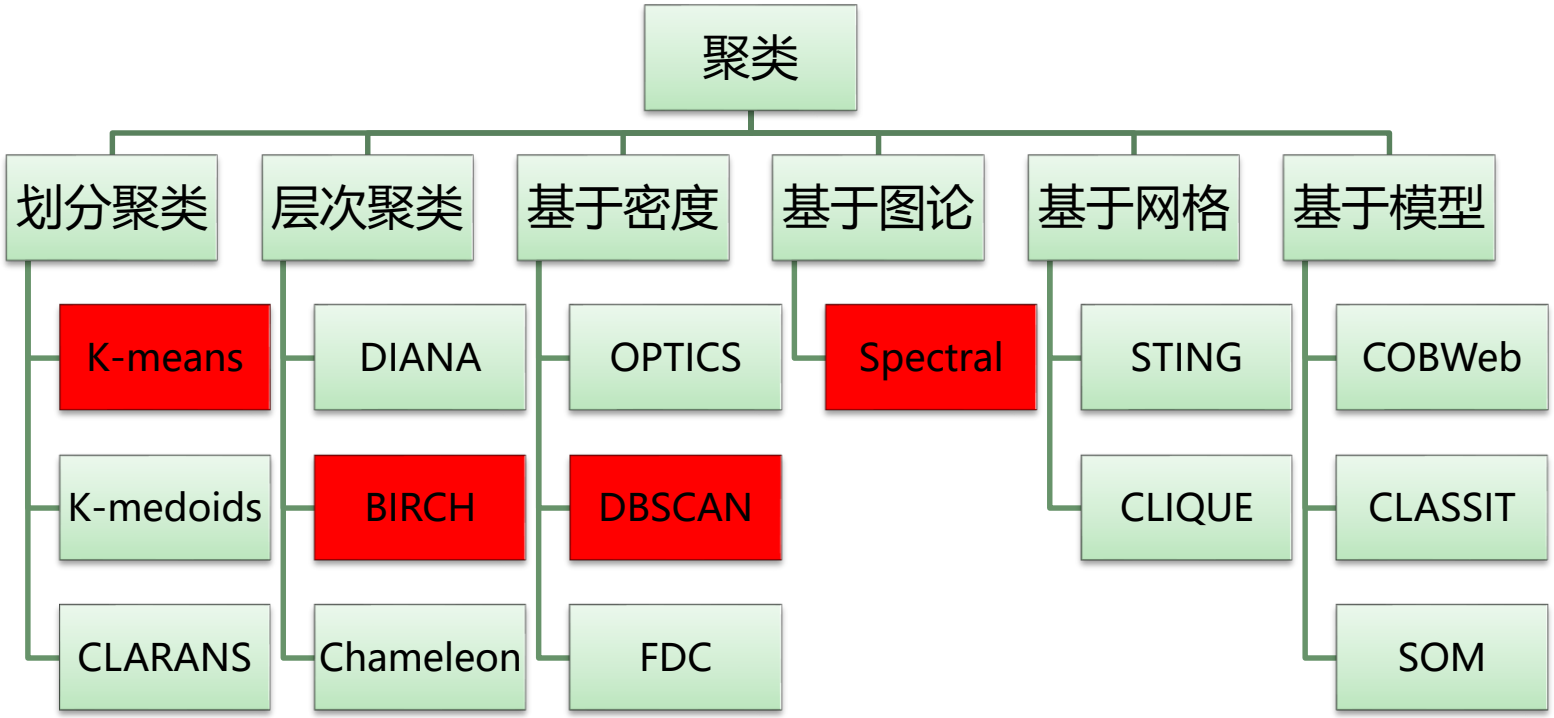
Clustering – 算法 - DBSCAN

```
# Plot result
import matplotlib.pyplot as plt

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]

for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k) # select specific cluster

    xy = X[class_member_mask & core_samples_mask] # make sure of a cluster and core sample
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col), markeredgecolor='k', markersize=14)

    xy = X[class_member_mask & ~core_samples_mask] # make sure of a cluster and noise sample
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col), markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
```

Clustering – 算法 - DBSCAN



Clustering – 算法



Clustering – 算法 - SPECTRAL

谱聚类（spectral clustering)

是从图论中演化出来的算法，广泛使用的聚类算法

比起传统的K-Means算法，谱聚类对数据分布的适应性更强，聚类效果也很优秀，同时聚类的计算量也小很多，更加难能可贵的是实现起来也不复杂。

Clustering – 算法 - SPECTRAL

核心思想：

把所有的数据看做空间中的点，这些点之间可以用边连接起来

距离较远的两点之间的边权重值较低，而距离较近的两点之间的边权重值较高

通过对所有数据点组成的图进行切图，让切图后不同的子图间边权重和尽可能的低，而子图内的边权重和尽可能的高，从而达到聚类的目的

Clustering – 算法 - SPECTRAL

无向权重图

一个图G，我们一般用**点的集合V**和**边的集合E**来描述，即为$G(V,E)$。

其中V即为我们数据集里面所有的点$(v_1, v_2, \dots v_n)$。

对于V中的任意两个点，可以有边连接，也可以没有边连接。

我们定义权重w_{ij}为点v_i和点v_j之间的权重。

由于我们是无向图，所以$w_{ij} = w_{ji}$。

有边连接的两个点v_i和v_j，$w_{ij} > 0$；没有边连接的两个点v_i和v_j，$w_{ij} = 0$

Clustering – 算法 - SPECTRAL

度d_i定义为和它相连的所有边的权重之和

$$d_i = \sum_{j=1}^{n} w_{ij}$$

度矩阵D，对角矩阵，对应第i行的第i个点的度数

$$D_{n\times n} = \begin{bmatrix} d_1 & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \cdots & \cdots & d_n \end{bmatrix}$$

图的邻接矩阵W，第i行的第j个值对应我们的权重w_{ij}

$$W_{n\times n} = \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nn} \end{bmatrix}$$

w_{ij} 怎么计算？邻接矩阵W哪里来？

Clustering – 算法 - SPECTRAL

构建邻接矩阵W： **ϵ-邻近法**，**K邻近法**和**全连接法**

- **ϵ-邻近法**

设置距离阈值ϵ，然后用欧式距离s_{ij}度量任意两点x_i和x_j的距离。

即相似矩阵的$s_{ij} = ||x_i - x_j||^2$,

根据s_{ij}和ϵ的大小关系，来定义邻接矩阵W

$$W_{ij} = \begin{cases} 0 & s_{ij} > \epsilon \\ \epsilon & s_{ij} \leq \epsilon \end{cases}$$

距离远近度量很不精确，因此在实际应用中，我们很少使用ϵ -邻近法。

Clustering – 算法 - SPECTRAL

构建邻接矩阵W：**ϵ-邻近法**，**K邻近法**和**全连接法**

- **K邻近法**

利用KNN算法遍历所有的样本点，取每个样本最近的k个点作为近邻，

只有和样本距离最近的k个点之间的$w_{ij} > 0$。

但是这种方法会造成重构之后的邻接矩阵W非对称

Clustering – 算法 - SPECTRAL

构建邻接矩阵W： **ϵ-邻近法**，**K邻近法**和**全连接法**

- **全连接法**

所有的点之间的权重值都大于0

可以选择不同的核函数来定义边权重，

常用的有多项式核函数，高斯核函数和Sigmoid核函数。

使用 基于高斯核函数的全连接法 来建立邻接矩阵是最普遍的

$$W_{ij} = S_{ij} = \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})$$

Clustering – 算法 - SPECTRAL

构建拉普拉斯矩阵L

$$L = D - W$$

1. 拉普拉斯矩阵是对称矩阵，这可以由D和W都是对称矩阵而得。

2. 由于拉普拉斯矩阵是对称矩阵，则它的所有的**特征值都是实数**。

3. 利用拉普拉斯矩阵的定义可知，对于任意的向量f，有

$$f^T L f = \frac{1}{2}\sum_{i,j=1}^{n} w_{ij}(f_i - f_j)^2$$

4. 拉普拉斯矩阵是半正定的，且对应的n个实数特征值都大于等于0，即$0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$，且最小的特征值为0，这个由性质3很容易得出。

Clustering – 算法 - SPECTRAL

无向图切图

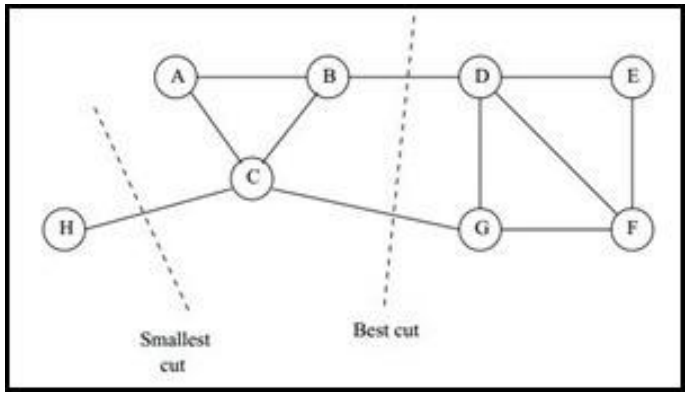
对于无向图G的切图，目标是将图$G(V,E)$切成相互没有连接的k个子图，

每个子图点的集合为：$A_1, A_2, ..Ak$，

它们满足$A_i \cap Aj = \emptyset$，且$A_1 \cup A_2 \cup \cdots \cup A_k = V$

$$W(A,B) = \sum_{i \in A, j \in B} w_{ij}$$

$$cut(A_1, A_1, \ldots A_k) = \frac{1}{2}\sum_{i=1}^{k} W(A_i, \bar{A}_i)$$



Which cutting is Best ? Smallest cut ?

Clustering – 算法 - SPECTRAL

切图聚类 （RatioCut，Ncut)

RatioCut

不光考虑最小化$cut(A_1, A_2, \dots, A_k)$，

它还同时考虑最大化每个子图点的个数

$$RatioCut(A_1, A_2, \dots, A_k) = \frac{1}{2}\sum_{i=1}^{k}\frac{W(A_i, \bar{A}_i)}{|A_i|}$$

目标：最小化$RatioCut(A_1, A_2, \dots, A_k)$

Clustering – 算法 - SPECTRAL

$$RatioCut(A_1, A_2, \dots, A_k) = \frac{1}{2}\sum_{i=1}^{k}\frac{W(A_i, \bar{A}_i)}{|A_i|}$$

RatioCut 新表达：引入指示向量 $h_{ji} = \begin{cases} 0 & v_i \notin A_j \\ \frac{1}{\sqrt{|A_i|}} & v_i \in A_j \end{cases}$

$$h_i^T L h_i = \frac{1}{2}\sum_{m=1}\sum_{n=1} w_{mn}(h_{im} - h_{in})^2$$

$$= \frac{1}{2}\left(\sum_{m \in A_i, n \notin A_i} w_{mn}\left(\frac{1}{\sqrt{|A_i|}} - 0\right)^2 + \sum_{m \notin A_i, n \in A_i} w_{mn}\left(0 - \frac{1}{\sqrt{|A_i|}}\right)^2\right)$$

$$= \frac{1}{2}\left(\sum_{m \in A_i, n \notin A_i} w_{mn}\frac{1}{|A_i|} + \sum_{m \notin A_i, n \in A_i} w_{mn}\frac{1}{|A_i|}\right)$$

$$= \frac{1}{2}\left(cut(A_i, \bar{A}_i)\frac{1}{|A_i|} + cut(\bar{A}_i, A_i)\frac{1}{|A_i|}\right) = \frac{cut(A_i, \bar{A}_i)}{|A_i|} = RatioCut(A_i, \bar{A}_i)$$

Clustering – 算法 - SPECTRAL

$$RatioCut(A_1, A_2, \dots, A_k) = \frac{1}{2}\sum_{i=1}^{k}\frac{W(A_i, \bar{A}_i)}{|A_i|}$$

RatioCut 新表达：引入指示向量 $h_{ji} = \begin{cases} 0 & v_i \notin A_j \\ \dfrac{1}{\sqrt{|A_i|}} & v_i \in A_j \end{cases}$

每个子图 i 的RatioCut可以表示为：$h_i^T L h_i = RatioCut(A_i, \bar{A}_i)$

整个数据集上，k个子图的表达?

$$RatioCut(A_1, A_2, \dots Ak) = \sum_{i=1}^{k} h_i^T L h_i = \sum_{i=1}^{k} H^T L H = tr(H^T L H)$$

目标：$\arg\min tr(H^T L H)\ s.t.\ H^T H = I$

Clustering – 算法 - SPECTRAL

切图聚类 （RatioCut，Ncut)

Ncut

和RatioCut切图很类似，但是把Ratiocut的分母$|A_i|$换成$vol(A_i)$。

由于子图样本的个数多并不一定权重就大，

我们切图时基于权重也更合我们的目标，

因此一般来说Ncut切图优于RatioCut切图。

$$NCut(A_1, A_2, \dots, A_k) = \frac{1}{2}\sum_{i=1}^{k}\frac{W(A_i, \bar{A}_i)}{vol(A_i)}$$

$$vol(A) = \sum_{i \in A} d_i$$

Clustering – 算法 - SPECTRAL

谱聚类算法的主要优点：

1. 谱聚类只需要数据之间的相似度矩阵，因此对于处理稀疏数据的聚类很有效。这点传统聚类算法比如K-Means很难做到

2. 由于使用了降维，因此在处理高维数据聚类时的复杂度比传统聚类算法好。

Clustering – 算法 - SPECTRAL

谱聚类算法的主要缺点：

1. 如果最终聚类的维度非常高，则由于降维的幅度不够，谱聚类的运行速度和最后的聚类效果均不好。

2. 聚类效果依赖于相似矩阵，不同的相似矩阵得到的最终聚类效果可能很不同

Clustering – 算法 - SPECTRAL

sklearn.cluster.SpectralClustering

class sklearn.cluster. **SpectralClustering** (*n_clusters=8, eigen_solver=None, random_state=None, n_init=10, gamma=1.0, affinity='rbf', n_neighbors=10, eigen_tol=0.0, assign_labels='kmeans', degree=3, coef0=1, kernel_params=None, n_jobs=1*) [source]

n_clusters : integer, optional

The dimension of the projection subspace.

eigen_solver : {None, 'arpack', 'lobpcg', or 'amg'}

The eigenvalue decomposition strategy to use. AMG requires pyamg to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities

random_state : int, RandomState instance or None, optional, default: None

A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when eigen_solver == 'amg' and by the K-Means initialization. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

Clustering – 算法 - SPECTRAL

sklearn.cluster.SpectralClustering

class sklearn.cluster. **SpectralClustering** (*n_clusters=8, eigen_solver=None, random_state=None, n_init=10, gamma=1.0, affinity='rbf', n_neighbors=10, eigen_tol=0.0, assign_labels='kmeans', degree=3, coef0=1, kernel_params=None, n_jobs=1*) [source]

n_init : int, optional, default: 10

Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

gamma : float, default=1.0

Kernel coefficient for rbf, poly, sigmoid, laplacian and chi2 kernels. Ignored for `affinity='nearest_neighbors'`.

affinity : string, array-like or callable, default 'rbf'

If a string, this may be one of 'nearest_neighbors', 'precomputed', 'rbf' or one of the kernels supported by sklearn.metrics.pairwise_kernels.

Only kernels that produce similarity scores (non-negative values that increase with similarity) should be used. This property is not checked by the clustering algorithm.

Clustering – 算法 - SPECTRAL

sklearn.cluster.SpectralClustering

class sklearn.cluster. **SpectralClustering** (*n_clusters=8, eigen_solver=None, random_state=None, n_init=10, gamma=1.0, affinity='rbf', n_neighbors=10, eigen_tol=0.0, assign_labels='kmeans', degree=3, coef0=1, kernel_params=None, n_jobs=1*) [source]

n_neighbors : integer

Number of neighbors to use when constructing the affinity matrix using the nearest neighbors method. Ignored for `affinity='rbf'`.

eigen_tol : float, optional, default: 0.0

Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.

assign_labels : {'kmeans', 'discretize'}, default: 'kmeans'

The strategy to use to assign labels in the embedding space. There are two ways to assign labels after the laplacian embedding. k-means can be applied and is a popular choice. But it can also be sensitive to initialization. Discretization is another approach which is less sensitive to random initialization.

Clustering – 算法 - SPECTRAL

sklearn.cluster.SpectralClustering

class sklearn.cluster. **SpectralClustering** (*n_clusters=8, eigen_solver=None, random_state=None, n_init=10, gamma=1.0, affinity='rbf', n_neighbors=10, eigen_tol=0.0, assign_labels='kmeans', degree=3, coef0=1, kernel_params=None, n_jobs=1*) [source]

degree : float, default=3

Degree of the polynomial kernel. Ignored by other kernels.

coef0 : float, default=1

Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.

kernel_params : dictionary of string to any, optional

Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

n_jobs : int, optional (default = 1)

The number of parallel jobs to run. If `-1`, then the number of jobs is set to the number of CPU cores.

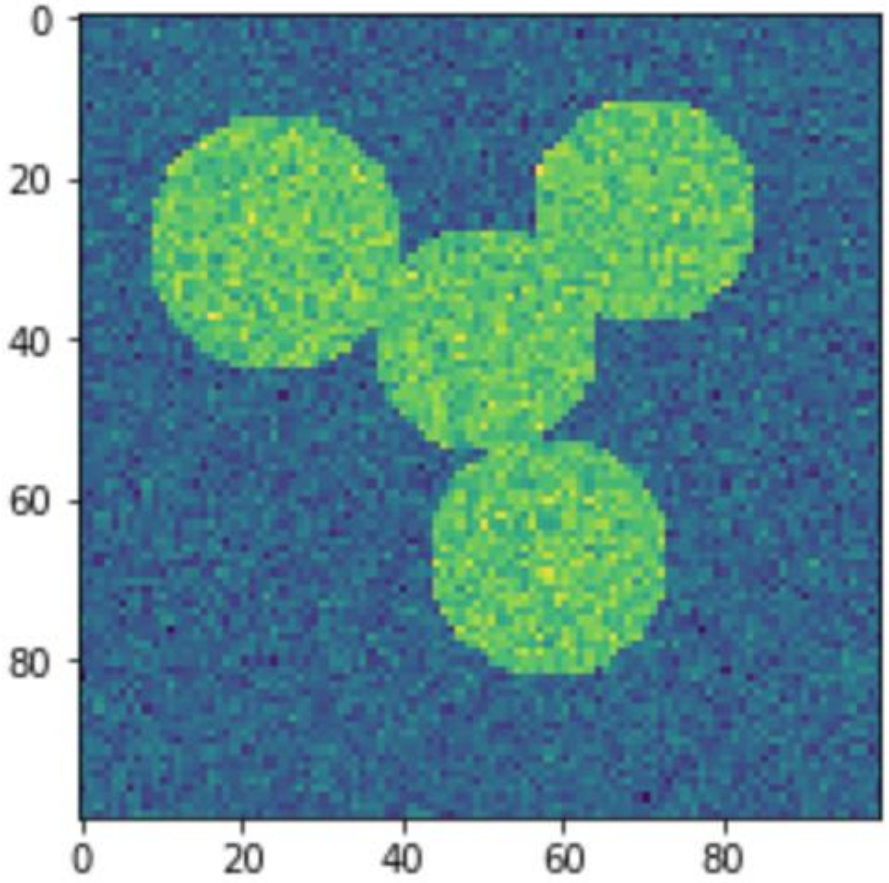
Clustering – 算法 - SPECTRAL

```
# clustering - SPECTRAL
```

```
from sklearn.feature_extraction import image
from sklearn.cluster import spectral_clustering
```

```
l = 100
x, y = np.indices((l, l))

center1 = (28, 24)
center2 = (40, 50)
center3 = (67, 58)
center4 = (24, 70)

radius1, radius2, radius3, radius4 = 16, 14, 15, 14

circle1 = (x - center1[0]) ** 2 + (y - center1[1]) ** 2 < radius1 ** 2
circle2 = (x - center2[0]) ** 2 + (y - center2[1]) ** 2 < radius2 ** 2
circle3 = (x - center3[0]) ** 2 + (y - center3[1]) ** 2 < radius3 ** 2
circle4 = (x - center4[0]) ** 2 + (y - center4[1]) ** 2 < radius4 ** 2

# generating image
img = circle1 + circle2 + circle3 + circle4

# image masker
mask = img.astype(bool)

# add noise
img = img.astype(float)
img += 1 + 0.2 * np.random.randn(*img.shape)

plt.imshow(img)
```

Clustering – 算法 - SPECTRAL

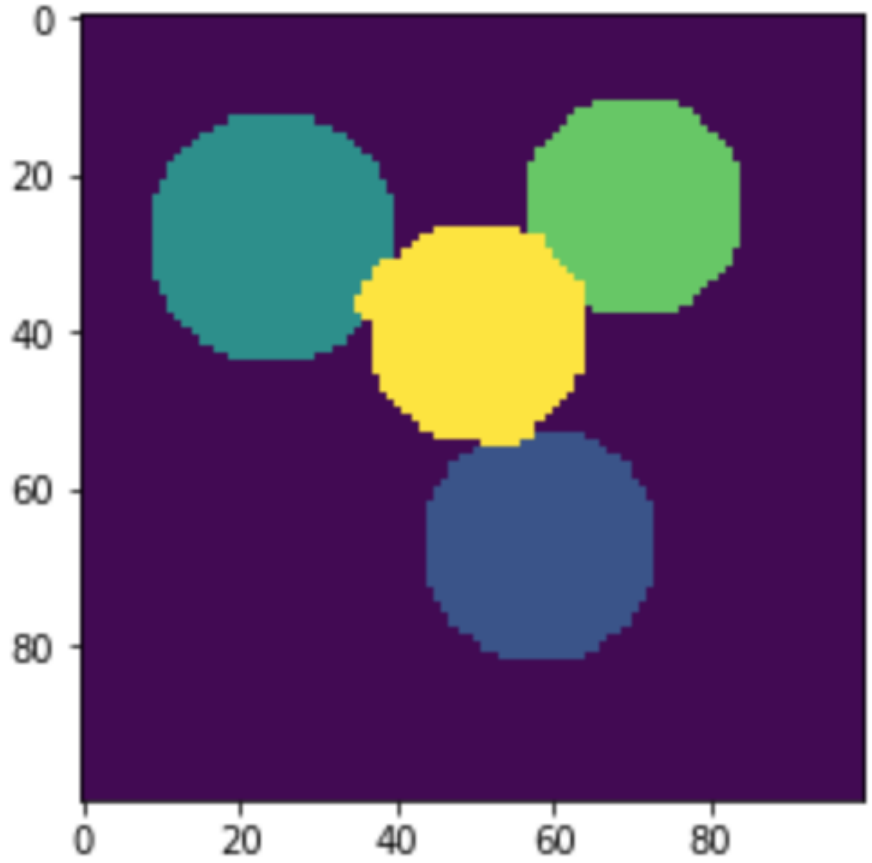


Clustering – 算法 - SPECTRAL

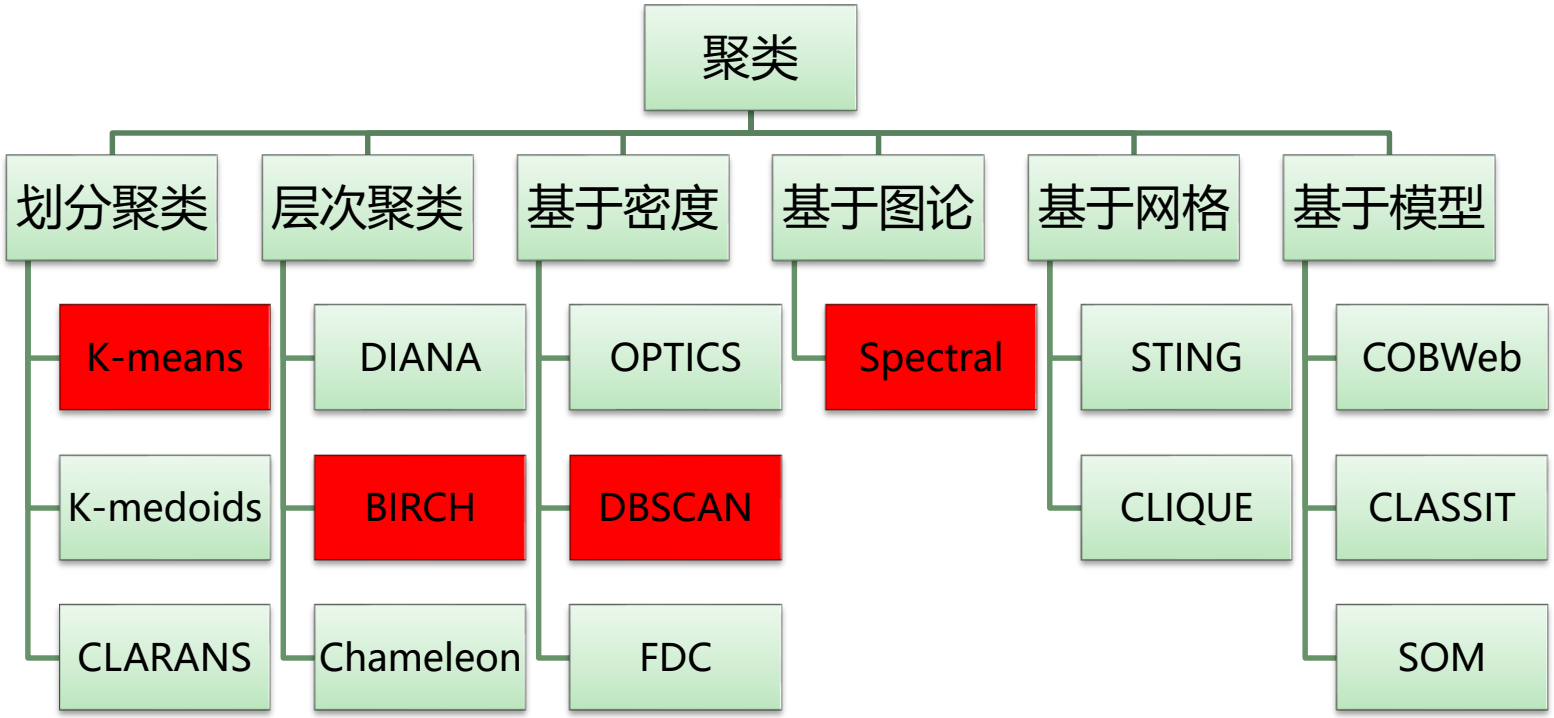
```
# data prprocessing for clustering
graph = image.img_to_graph(img, mask=mask)
graph.data = np.exp(-graph.data / graph.data.std())
```

```
# model
labels = spectral_clustering(graph, n_clusters=4, eigen_solver=
label_im = np.full(mask.shape, -1.)
label_im[mask] = labels
```

```
plt.imshow(label_im)
```



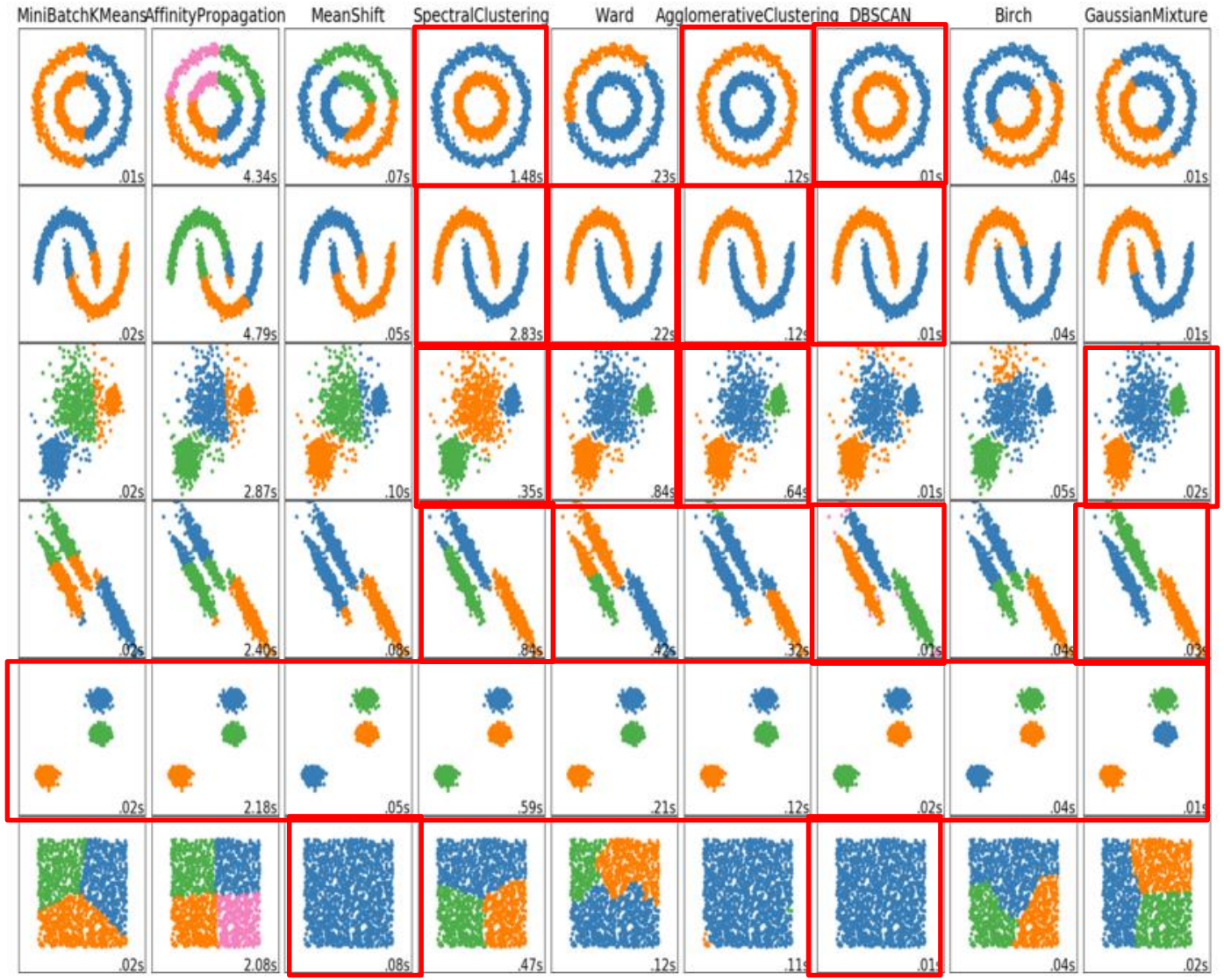
Clustering – 算法



OUTLINES

1. 预备知识
2. Clustering 问题定义
3. Clustering 算法概述
4. 算法 (k-means, BIRCH, DBSCAN, Spectral)
5. Clusering 算法选型

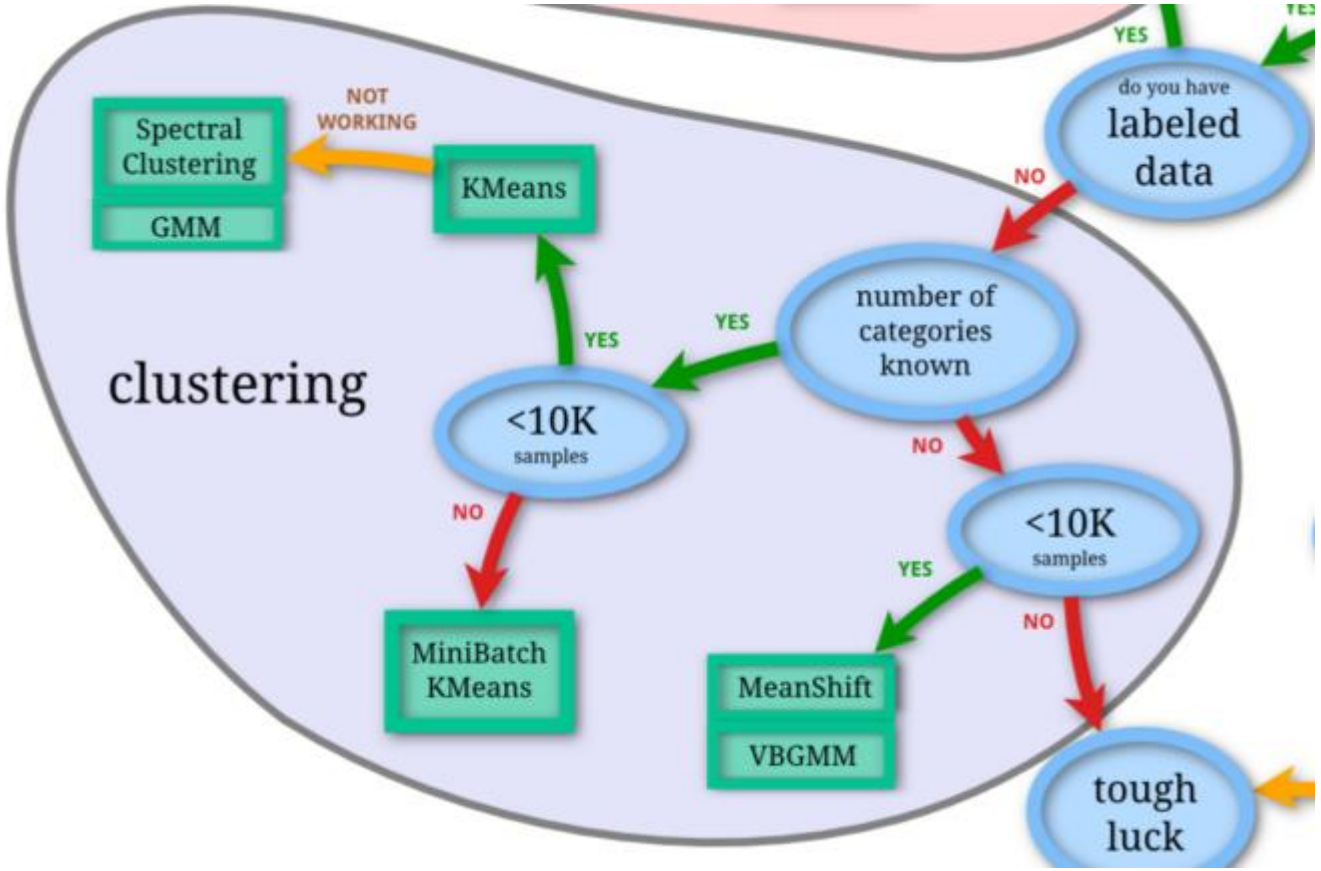
Clustering – 算法选型



Clustering – 算法选型

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n_samples`, medium `n_clusters` with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters, linkage type, distance	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large `n_clusters` and `n_samples`	Large dataset, outlier removal, data reduction.	Euclidean distance between points

Clustering – 算法选型



http://scikit-learn.org/stable/tutorial/machine_learning_map/

总 结

1. 预备知识 – Clustering Metrics and 距离计算
2. Clustering 问题定义
3. Clustering 算法概述
4. 算法 (k-means, BIRCH, DBSCAN, Spectral)
5. Clusering 算法选型