

聚类

1. 在无监督学习 (unsupervised learning) 中，训练样本的标记信息是未知的。
2. 无监督学习的目标：通过对无标记训练样本的学习来揭露数据的内在性质以及规律。
3. 一个经典的无监督学习任务：寻找数据的最佳表达 (representation)。常见的有：
 - 低维表达：试图将数据（位于高维空间）中的信息尽可能压缩在一个较低维空间中。
 - 稀疏表达：将数据嵌入到大多数项为零的一个表达中。该策略通常需要进行维度扩张。
 - 独立表达：使数据的各个特征相互独立。
4. 无监督学习应用最广的是聚类 (clustering)。
 - 给定数据集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ ，聚类试图将数据集中的样本划分为 K 个不相交的子集 $\{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$ ，每个子集称为一个簇 cluster。其中： $\mathbb{C}_k \cap_{k \neq l} \mathbb{C}_l = \emptyset$, $\mathbb{D} = \bigcup_{k=1}^K \mathbb{C}_k$ 。
 - 通过这样的划分，每个簇可能对应于一个潜在的概念。这些概念对于聚类算法而言，事先可能是未知的。
 - 聚类过程仅仅能自动形成簇结构，簇所对应的概念语义需要由使用者来提供。
5. 通常用 $\lambda_i \in \{1, 2, \dots, K\}$ 表示样本 \vec{x}_i 的簇标记 cluster label，即 $\vec{x}_i \in \mathbb{C}_{\lambda_i}$ 。于是数据集 \mathbb{D} 的聚类结果可以用包含 N 个元素的簇标记向量 $\vec{\lambda} = (\lambda_1, \lambda_2, \dots, \lambda_N)^T$ 来表示。
6. 聚类的作用：
 - 可以作为一个单独的过程，用于寻找数据内在的分布结构。
 - 也可以作为其他学习任务的前驱过程。如对数据先进行聚类，然后对每个簇单独训练模型。
7. 聚类问题本身是病态的。即：没有某个标准来衡量聚类的效果。
 - 可以简单的度量聚类的性质，如每个聚类的元素到该类中心点的平均距离。
但是实际上不知道这个平均距离对应于真实世界的物理意义。
 - 可能很多不同的聚类都很好地对应了现实世界的某些属性，它们都是合理的。

如：在图片识别中包含的图片有：红色卡车、红色汽车、灰色卡车、灰色汽车。可以聚类成：红色一类、灰色一类；也可以聚类成：卡车一类、汽车一类。

解决该问题的一个做法是：利用深度学习来进行分布式表达，可以对每个车辆赋予两个属性：一个表示颜色、一个表示型号。

一、性能度量

1. 聚类的性能度量也称作聚类的有效性指标 validity index。
2. 直观上看，希望同一簇的样本尽可能彼此相似，不同簇的样本之间尽可能不同。即：簇内相似度 intra-cluster similarity 高，且簇间相似度 inter-cluster similarity 低。
3. 聚类的性能度量分两类：
 - 聚类结果与某个参考模型 reference model 进行比较，称作外部指标 external index。
 - 直接考察聚类结果而不利用任何参考模型，称作内部指标 internal index。

1.1 外部指标

1. 对于数据集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ ，假定通过聚类给出的簇划分为 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$ 。参考模型给出的簇划分为 $\mathcal{C}^* = \{\mathbb{C}_1^*, \mathbb{C}_2^*, \dots, \mathbb{C}_{K'}^*\}$ ，其中 K 和 K' 不一定相等。

令 $\vec{\lambda}, \vec{\lambda}^*$ 分别表示 $\mathcal{C}, \mathcal{C}^*$ 的簇标记向量。定义：

$$\begin{aligned} a &= |SS|, SS = \{(\vec{x}_i, \vec{x}_j) \mid \lambda_i = \lambda_j, \lambda_i^* = \lambda_j^*, i < j\} \\ b &= |SD|, SD = \{(\vec{x}_i, \vec{x}_j) \mid \lambda_i = \lambda_j, \lambda_i^* \neq \lambda_j^*, i < j\} \\ c &= |DS|, DS = \{(\vec{x}_i, \vec{x}_j) \mid \lambda_i \neq \lambda_j, \lambda_i^* = \lambda_j^*, i < j\} \\ d &= |DD|, DD = \{(\vec{x}_i, \vec{x}_j) \mid \lambda_i \neq \lambda_j, \lambda_i^* \neq \lambda_j^*, i < j\} \end{aligned}$$

其中 $|\cdot|$ 表示集合的元素的个数。各集合的意义为：

- SS ：包含了同时隶属于 $\mathcal{C}, \mathcal{C}^*$ 的样本对。
- SD ：包含了隶属于 \mathcal{C} ，但是不隶属于 \mathcal{C}^* 的样本对。
- DS ：包含了不隶属于 \mathcal{C} ，但是隶属于 \mathcal{C}^* 的样本对。
- DD ：包含了既不隶属于 \mathcal{C} ，又不隶属于 \mathcal{C}^* 的样本对。

由于每个样本对 $(\vec{x}_i, \vec{x}_j), i < j$ 仅能出现在一个集合中，因此有 $a + b + c + d = \frac{N(N-1)}{2}$ 。

2. 下述性能度量的结果都在 $[0, 1]$ 之间。这些值越大，说明聚类的性能越好。

1.1.1 Jaccard系数

1. **Jaccard 系数 Jaccard Coefficient:JC** :

$$JC = \frac{a}{a + b + c}$$

它刻画了：所有的同类的样本对（要么在 \mathcal{C} 中属于同类，要么在 \mathcal{C}^* 中属于同类）中，同时隶属于 $\mathcal{C}, \mathcal{C}^*$ 的样本对的比例。

1.1.2 FM指数

1. **FM 指数 Fowlkes and Mallows Index:FMI** :

$$FMI = \sqrt{\frac{a}{a+b} \cdot \frac{a}{a+c}}$$

它刻画的是：

- 在 \mathcal{C} 中同类的样本对中，同时隶属于 \mathcal{C}^* 的样本对的比例为 $p1 = \frac{a}{a+b}$ 。
- 在 \mathcal{C}^* 中同类的样本对中，同时隶属于 \mathcal{C} 的样本对的比例为 $p2 = \frac{a}{a+c}$ 。
- **FMI** 就是 $p1$ 和 $p2$ 的几何平均。

1.1.3 Rand指数

1. **Rand 指数 Rand Index:RI** :

$$RI = \frac{a+d}{N(N-1)/2}$$

它刻画的是：

- 同时隶属于 $\mathcal{C}, \mathcal{C}^*$ 的同类样本对（这种样本对属于同一个簇的概率最大）与既不隶属于 \mathcal{C} 、又不隶属于 \mathcal{C}^* 的非同类样本对（这种样本对不是同一个簇的概率最大）之和，占所有样本对的比例。
- 这个比例其实就是聚类的可靠程度的度量。

1.1.4 ARI指数

1. 使用 RI 有个问题：对于随机聚类，RI 指数不保证接近0（可能还很大）。

ARI 指数就通过利用随机聚类来解决这个问题。

2. 定义一致性矩阵为：

	\mathbb{C}_1^*	\mathbb{C}_2^*	...	$\mathbb{C}_{K'}^*$	sums
\mathbb{C}_1	$n_{1,1}$	$n_{1,2}$...	$n_{1,K'}$	s_1
\mathbb{C}_2	$n_{2,1}$	$n_{2,2}$...	$n_{2,K'}$	s_2
\vdots	\vdots	\vdots	...	\vdots	\vdots
\mathbb{C}_K	$n_{K,1}$	$n_{K,2}$...	$n_{K,K'}$	s_K
sums	t_1	t_2	...	t_K	N

其中：

- s_i 为属于簇 \mathbb{C}_i 的样本的数量， t_i 为属于簇 \mathbb{C}_i^* 的样本的数量。
- $n_{i,j}$ 为同时属于簇 \mathbb{C}_i 和簇 \mathbb{C}_j^* 的样本的数量。

则根据定义有： $a = \sum_i \sum_j C_{n_{i,j}}^2$ ，其中 $C_n^2 = \frac{n(n-1)}{2}$ 表示组合数。数字 2 是因为需要提取两个样本组成样本对。

3. 定义 ARI 指数 Adjusted Rand Index：

$$ARI = \frac{\sum_i \sum_j C_{n_{i,j}}^2 - [\sum_i C_{s_i}^2 \times \sum_j C_{t_j}^2] / C_N^2}{\frac{1}{2} [\sum_i C_{s_i}^2 + \sum_j C_{t_j}^2] - [\sum_i C_{s_i}^2 \times \sum_j C_{t_j}^2] / C_N^2}$$

其中：

- $\sum_i \sum_j C_{n_{i,j}}^2$: 表示同时隶属于 $\mathcal{C}, \mathcal{C}^*$ 的样本对。
 - $\frac{1}{2} [\sum_i C_{s_i}^2 + \sum_j C_{t_j}^2]$: 表示最大的样本对。
- 即：无论如何聚类，同时隶属于 $\mathcal{C}, \mathcal{C}^*$ 的样本对不会超过该数值。
- $[\sum_i C_{s_i}^2 \times \sum_j C_{t_j}^2] / C_N^2$: 表示在随机划分的情况下，同时隶属于 $\mathcal{C}, \mathcal{C}^*$ 的样本对的期望。
 - 随机挑选一对样本，一共有 C_N^2 种情形。
 - 这对样本隶属于 \mathcal{C} 中的同一个簇，一共有 $\sum_i C_{s_i}^2$ 种可能。
 - 这对样本隶属于 \mathcal{C}^* 中的同一个簇，一共有 $\sum_j C_{t_j}^2$ 种可能。
 - 这对样本隶属于 \mathcal{C} 中的同一个簇、且属于 \mathcal{C}^* 中的同一个簇，一共有 $\sum_i C_{s_i}^2 \times \sum_j C_{t_j}^2$ 种可能。
 - 则在随机划分的情况下，同时隶属于 $\mathcal{C}, \mathcal{C}^*$ 的样本对的期望（平均样本对）为：
 $[\sum_i C_{s_i}^2 \times \sum_j C_{t_j}^2] / C_N^2$ 。

1.2 内部指标

1. 对于数据集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ ，假定通过聚类给出的簇划分为 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$ 。

定义：

$$\begin{aligned}\text{avg}(\mathbb{C}_k) &= \frac{2}{|\mathbb{C}_k|(|\mathbb{C}_k|-1)} \sum_{\vec{\mathbf{x}}_i, \vec{\mathbf{x}}_j \in \mathbb{C}_k, i \neq j} \text{distance}(\vec{\mathbf{x}}_i, \vec{\mathbf{x}}_j), \quad k = 1, 2, \dots, K \\ \text{diam}(\mathbb{C}_k) &= \max_{\vec{\mathbf{x}}_i, \vec{\mathbf{x}}_j \in \mathbb{C}_k, i \neq j} \text{distance}(\vec{\mathbf{x}}_i, \vec{\mathbf{x}}_j), \quad k = 1, 2, \dots, K \\ d_{min}(\mathbb{C}_k, \mathbb{C}_l) &= \min_{\vec{\mathbf{x}}_i \in \mathbb{C}_k, \vec{\mathbf{x}}_j \in \mathbb{C}_l} \text{distance}(\vec{\mathbf{x}}_i, \vec{\mathbf{x}}_j), \quad k, l = 1, 2, \dots, K; k \neq l \\ d_{cen}(\mathbb{C}_k, \mathbb{C}_l) &= \text{distance}(\vec{\mu}_k, \vec{\mu}_l), \quad k, l = 1, 2, \dots, K; k \neq l\end{aligned}$$

其中 : $\text{distance}(\vec{\mathbf{x}}_i, \vec{\mathbf{x}}_j)$ 表示两点 $\vec{\mathbf{x}}_i, \vec{\mathbf{x}}_j$ 之间的距离 ; $\vec{\mu}_k$ 表示簇 \mathbb{C}_k 的中心点 , $\vec{\mu}_l$ 表示簇 \mathbb{C}_l 的中心点 , $\text{distance}(\vec{\mu}_k, \vec{\mu}_l)$ 表示簇 $\mathbb{C}_k, \mathbb{C}_l$ 的中心点之间的距离。

上述定义的意义为 :

- $\text{avg}(\mathbb{C}_k)$: 簇 \mathbb{C}_k 中每对样本之间的平均距离。
- $\text{diam}(\mathbb{C}_k)$: 簇 \mathbb{C}_k 中距离最远的两个点的距离。
- $d_{min}(\mathbb{C}_k, \mathbb{C}_l)$: 簇 $\mathbb{C}_k, \mathbb{C}_l$ 之间最近的距离。
- $d_{cen}(\mathbb{C}_k, \mathbb{C}_l)$: 簇 $\mathbb{C}_k, \mathbb{C}_l$ 中心点之间的距离。

1.2.1 DB指数

1. **DB 指数 Davies-Bouldin Index:DBI** :

$$DBI = \frac{1}{K} \sum_{k=1}^K \max_{k \neq l} \left(\frac{\text{avg}(\mathbb{C}_k) + \text{avg}(\mathbb{C}_l)}{d_{cen}(\mathbb{C}_k, \mathbb{C}_l)} \right)$$

其物理意义为 :

- 给定两个簇 , 每个簇样本距离均值之和比上两个簇的中心点之间的距离作为度量。
该度量越小越好。
- 给定一个簇 k , 遍历其它的簇 , 寻找该度量的最大值。
- 对所有的簇 , 取其最大度量的均值。

2. 显然 DBI 越小越好。

- 如果每个簇样本距离均值越小 (即簇内样本距离都很近) , 则 DBI 越小。
- 如果簇间中心点的距离越大 (即簇间样本距离相互都很远) , 则 DBI 越小。

1.2.2 Dunn指数

1. **Dunn 指数 Dunn Index:DI** :

$$DI = \frac{\min_{k \neq l} d_{min}(\mathbb{C}_k, \mathbb{C}_l)}{\max_i \text{diam}(\mathbb{C}_i)}$$

其物理意义为 : 任意两个簇之间最近的距离的最小值 , 除以任意一个簇内距离最远的两个点的距离的最大值。

2. 显然 DI 越大越好。

- 如果任意两个簇之间最近的距离的最小值越大 (即簇间样本距离相互都很远) , 则 DI 越大。
- 如果任意一个簇内距离最远的两个点的距离的最大值越小 (即簇内样本距离都很近) , 则 DI 越大。

1.3 距离度量

1. 距离函数 $\text{distance}(\cdot, \cdot)$ 常用的有以下距离 :

- 闵可夫斯基距离 Minkowski distance :

给定样本 $\vec{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,n})^T$, $\vec{x}_j = (x_{j,1}, x_{j,2}, \dots, x_{j,n})^T$, 则闵可夫斯基距离定义为：

$$\text{distance}(\vec{x}_i, \vec{x}_j) = \left(\sum_{d=1}^n |x_{i,d} - x_{j,d}|^p \right)^{1/p}$$

- 当 $p = 2$ 时, 闵可夫斯基距离就是欧式距离 Euclidean distance :

$$\text{distance}(\vec{x}_i, \vec{x}_j) = \|\vec{x}_i - \vec{x}_j\|_2 = \sqrt{\sum_{d=1}^n |x_{i,d} - x_{j,d}|^2}$$

- 当 $p = 1$ 时, 闵可夫斯基距离就是曼哈顿距离 Manhattan distance :

$$\text{distance}(\vec{x}_i, \vec{x}_j) = \|\vec{x}_i - \vec{x}_j\|_1 = \sum_{d=1}^n |x_{i,d} - x_{j,d}|$$

- VDM 距离 Value Difference Metric :

考虑非数值类属性 (如属性取值为：中国，印度，美国，英国)，令 $m_{d,a}$ 表示 $x_d = a$ 的样本数；
 $m_{d,a,k}$ 表示 $x_d = a$ 且位于簇 C_k 中的样本的数量。则在属性 d 上的两个取值 a, b 之间的 VDM 距离为：

$$VDM_p(a, b) = \left(\sum_{k=1}^K \left| \frac{m_{d,a,k}}{m_{d,a}} - \frac{m_{d,b,k}}{m_{d,b}} \right|^p \right)^{1/p}$$

该距离刻画的是：属性取值在各簇上的频率分布之间的差异。

2. 当样本的属性为数值属性与非数值属性混合时，可以将闵可夫斯基距离与 VDM 距离混合使用。

假设属性 x_1, x_2, \dots, x_{n_c} 为数值属性，属性 $x_{n_c+1}, x_{n_c+2}, \dots, x_n$ 为非数值属性。则：

$$\text{distance}(\vec{x}_i, \vec{x}_j) = \left(\sum_{d=1}^{n_c} |x_{i,d} - x_{j,d}|^p + \sum_{d=n_c+1}^n VDM_p(x_{i,d}, x_{j,d})^p \right)^{1/p}$$

3. 当样本空间中不同属性的重要性不同时，可以采用加权距离。

以加权闵可夫斯基距离为例：

$$\text{distance}(\vec{x}_i, \vec{x}_j) = \left(\sum_{d=1}^n w_d \times |x_{i,d} - x_{j,d}|^p \right)^{1/p}$$

$$w_d \geq 0, d = 1, 2, \dots, n; \quad \sum_{d=1}^n w_d = 1$$

4. 这里的距离函数都是事先定义好的。在有些实际任务中，有必要基于数据样本来确定合适的距离函数。这可以通过距离度量学习 distance metric learning 来实现。

5. 这里的距离度量满足三角不等式： $\text{distance}(\vec{x}_i, \vec{x}_j) \leq \text{distance}(\vec{x}_i, \vec{x}_k) + \text{distance}(\vec{x}_k, \vec{x}_j)$ 。

在某些任务中，根据数据的特点可能需要放松这一性质。如：美人鱼 和 人 距离很近，美人鱼 和 鱼 距离很近，但是 人 和 鱼 的距离很远。这样的距离称作非度量距离 non-metric distance。

二、原型聚类

1. 原型聚类 prototype-based clustering 假设聚类结构能通过一组原型刻画。

常用的原型聚类有：

- **k 均值算法 k-means** 。
- 学习向量量化算法 Learning Vector Quantization:LVQ 。
- 高斯混合聚类 Mixture-of-Gaussian 。

2.1 k-means 算法

2.1.1 k-means

1. 给定样本集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$, 假设一个划分为 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$ 。

定义该划分的平方误差为：

$$err = \sum_{k=1}^K \sum_{\vec{x}_i \in \mathbb{C}_k} \|\vec{x}_i - \vec{\mu}_k\|_2^2$$

其中 $\vec{\mu}_k = \frac{1}{|\mathbb{C}_k|} \sum_{\vec{x}_i \in \mathbb{C}_k} \vec{x}_i$ 是簇 \mathbb{C}_k 的均值向量。

- err 刻画了簇类样本围绕簇均值向量的紧密程度，其值越小，则簇内样本相似度越高。
- **k-means** 算法的优化目标为：最小化 err 。即： $\min_{\mathcal{C}} \sum_{k=1}^K \sum_{\vec{x}_i \in \mathbb{C}_k} \|\vec{x}_i - \vec{\mu}_k\|_2^2$ 。
- 2. **k-means** 的优化目标需要考察 \mathbb{D} 的所有可能的划分，这是一个 **NP** 难的问题。实际上 **k-means** 采用贪心策略，通过迭代优化来近似求解。

- 首先假设一组均值向量。
- 然后根据假设的均值向量给出了 \mathbb{D} 的一个划分。
- 再根据这个划分来计算真实的均值向量：
 - 如果真实的均值向量等于假设的均值向量，则说明假设正确。根据假设均值向量给出的 \mathbb{D} 的一个划分确实是原问题的解。
 - 如果真实的均值向量不等于假设的均值向量，则可以将真实的均值向量作为新的假设均值向量，继续迭代求解。
- 3. 这里的一个关键就是：给定一组假设的均值向量，如何计算出 \mathbb{D} 的一个簇划分？

k 均值算法的策略是：样本离哪个簇的均值向量最近，则该样本就划归到那个簇。

4. **k-means** 算法：

- 输入：
 - 样本集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ 。
 - 聚类簇数 K 。
- 输出：簇划分 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$ 。
- 算法步骤：
 - 从 \mathbb{D} 中随机选择 K 个样本作为初始均值向量 $\{\vec{\mu}_1, \vec{\mu}_2, \dots, \vec{\mu}_K\}$ 。
 - 重复迭代直到算法收敛，迭代过程：
 - 初始化阶段：取 $\mathbb{C}_k = \phi, k = 1, 2, \dots, K$
 - 划分阶段：令 $i = 1, 2, \dots, N$ ：
 - 计算 \vec{x}_i 的簇标记： $\lambda_i = \arg \min_k \|\vec{x}_i - \vec{\mu}_k\|_2, k \in \{1, 2, \dots, K\}$ 。
 - 即：将 \vec{x}_i 离哪个簇的均值向量最近，则该样本就标记为那个簇。

- 然后将样本 \vec{x}_i 划入相应的簇： $\mathbb{C}_{\lambda_i} = \mathbb{C}_{\lambda_i} \cup \{\vec{x}_i\}$ 。
- 重计算阶段：计算 $\hat{\mu}_k$ ： $\hat{\mu}_k = \frac{1}{|\mathbb{C}_k|} \sum_{\vec{x}_i \in \mathbb{C}_k} \vec{x}_i$ 。
- 终止条件判断：
 - 如果对所有的 $k \in \{1, 2, \dots, K\}$ ，都有 $\hat{\mu}_k = \vec{\mu}_k$ ，则算法收敛，终止迭代。
 - 否则重赋值 $\vec{\mu}_k = \hat{\mu}_k$ 。

5. k-means 优点：

- 计算复杂度低，为 $O(N \times K \times q)$ ，其中 q 为迭代次数。
- 通常 K 和 q 要远远小于 N ，此时复杂度相当于 $O(N)$ 。
- 思想简单，容易实现。

6. k-means 缺点：

- 需要首先确定聚类的数量 K 。
- 分类结果严重依赖于分类中心的初始化。
通常进行多次 k-means，然后选择最优的那次作为最终聚类结果。
- 结果不一定是全局最优的，只能保证局部最优。
- 对噪声敏感。因为簇的中心是取平均，因此聚类簇很远地方的噪音会导致簇的中心点偏移。
- 无法解决不规则形状的聚类。
- 无法处理离散特征，如：国籍、性别等。

7. k-means 性质：

- k-means 实际上假设数据是呈现球形分布，实际任务中很少有这种情况。
与之相比，GMM 使用更加一般的数据表示，即高斯分布。
- k-means 假设各个簇的先验概率相同，但是各个簇的数据量可能不均匀。
- k-means 使用欧式距离来衡量样本与各个簇的相似度。这种距离实际上假设数据的各个维度对于相似度的作用是相同的。
- k-means 中，各个样本点只属于与其相似度最高的那个簇，这实际上是硬分簇。
- k-means 算法的迭代过程实际上等价于 EM 算法。具体参考 EM 算法章节。

2.1.2 k-means++

1. k-means++ 属于 k-means 的变种，它主要解决 k-means 严重依赖于分类中心初始化的问题。
2. k-means++ 选择初始均值向量时，尽量安排这些初始均值向量之间的距离尽可能的远。
3. k-means++ 算法：

- 输入：
 - 样本集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ 。
 - 聚类簇数 K 。
- 输出：簇划分 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$ 。
- 算法步骤：
 - 从 \mathbb{D} 中随机选择1个样本作为初始均值向量组 $\{\vec{\mu}_1, \dots\}$ 。
 - 迭代，直到初始均值向量组有 K 个向量。

假设初始均值向量组为 $\{\vec{\mu}_1, \dots, \vec{\mu}_m\}$ 。迭代过程如下：

- 对每个样本 \vec{x}_i ，分别计算其距 $\vec{\mu}_1, \dots, \vec{\mu}_m$ 的距离。这些距离的最小值记做 $d_i = \min_{\vec{\mu}_j} \|\vec{x}_i - \vec{\mu}_j\|$ 。
- 对样本 \vec{x}_i ，其设置为初始均值向量的概率正比于 d_i 。即：离所有的初始均值向量越远，则越可能被选中为下一个初始均值向量。
- 以概率分布 $P = \{d_1, d_2, \dots, d_N\}$ （未归一化的）随机挑选一个样本作为下一个初始均值向量 $\vec{\mu}_{m+1}$ 。
- 一旦挑选出初始均值向量组 $\{\vec{\mu}_1, \dots, \vec{\mu}_K\}$ ，剩下的迭代步骤与 **k-means** 相同。

2.1.3 k-modes

1. **k-modes** 属于 **k-means** 的变种，它主要解决 **k-means** 无法处理离散特征的问题。

2. **k-modes** 与 **k-means** 有两个不同点（假设所有特征都是离散特征）：

- 距离函数不同。在 **k-modes** 算法中，距离函数为：

$$\text{distance}(\vec{x}_i, \vec{x}_j) = \sum_{d=1}^n I(x_{i,d} \neq x_{j,d})$$

其中 $I(\cdot)$ 为示性函数。

上式的意义为：样本之间的距离等于它们之间不同属性值的个数。

- 簇中心的更新规则不同。在 **k-modes** 算法中，簇中心每个属性的取值为：簇内该属性出现频率最大的那个值。

$$\hat{\mu}_{k,d} = \arg \max_v \sum_{\vec{x}_i \in C_k} I(x_{i,d} = v)$$

其中 v 的取值空间为所有样本在第 d 个属性上的取值。

2.1.4 k-medoids

1. **k-medoids** 属于 **k-means** 的变种，它主要解决 **k-means** 对噪声敏感的问题。

2. **k-medoids** 算法：

- 输入：

- 样本集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ 。
- 聚类簇数 K 。
- 输出：簇划分 $\mathcal{C} = \{C_1, C_2, \dots, C_K\}$ 。

- 算法步骤：

- 从 \mathbb{D} 中随机选择 K 个样本作为初始均值向量 $\{\vec{\mu}_1, \vec{\mu}_2, \dots, \vec{\mu}_K\}$ 。
- 重复迭代直到算法收敛，迭代过程：

- 初始化阶段：取 $C_k = \phi, k = 1, 2, \dots, K$ 。

遍历每个样本 $\vec{x}_i, i = 1, 2, \dots, N$ ，计算它的簇标记：

$\lambda_i = \arg \min_k \|\vec{x}_i - \vec{\mu}_k\|_2, k \in \{1, 2, \dots, K\}$ 。即：将 \vec{x}_i 离哪个簇的均值向量最近，则该样本就标记为那个簇。

然后将样本 \vec{x}_i 划入相应的簇： $C_{\lambda_i} = C_{\lambda_i} \cup \{\vec{x}_i\}$ 。

- 重计算阶段：

遍历每个簇 $\mathbb{C}_k, k = 1, 2, \dots, K$:

- 计算簇心 $\vec{\mu}_k$ 距离簇内其它点的距离 $d_{\mu}^{(k)} = \sum_{\vec{x}_j^{(k)} \in \mathbb{C}_k} \|\vec{\mu}_k - \vec{x}_j^{(k)}\|$ 。
- 计算簇 \mathbb{C}_k 内每个点 $\vec{x}_i^{(k)}$ 距离簇内其它点的距离 $d_i^{(k)} = \sum_{\vec{x}_j^{(k)} \in \mathbb{C}_k} \|\vec{x}_i^{(k)} - \vec{x}_j^{(k)}\|$ 。
- 如果 $d_i^{(k)} < d_{\mu}^{(k)}$, 则重新设置簇中心 : $\vec{\mu}_k = \vec{x}_i^{(k)}$, $d_{\mu}^{(k)} = d_i^{(k)}$ 。

- 终止条件判断 : 遍历一轮簇 $\mathbb{C}_1, \dots, \mathbb{C}_K$ 之后 , 簇心保持不变。

3. **k-medoids** 算法在计算新的簇心时 , 不再通过簇内样本的均值来实现 , 而是挑选簇内距离其它所有点都最近的样本来实现。这就减少了孤立噪声带来的影响。
4. **k-medoids** 算法复杂度较高 , 为 $O(N^2)$ 。其中计算代价最高的是计算每个簇内每对样本之间的距离。
通常会在算法开始时计算一次 , 然后将结果缓存起来 , 以便后续重复使用。

2.1.5 mini-batch k-means

1. **mini-batch k-means** 属于 **k-means** 的变种 , 它主要用于减少 **k-means** 的计算时间。
2. **mini-batch k-means** 算法每次训练时随机抽取小批量的数据 , 然后用这个小批量数据训练。这种做法减少了 **k-means** 的收敛时间 , 其效果略差于标准算法。
3. **mini-batch k-means** 算法 :

◦ 输入 :

- 样本集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ 。
- 聚类簇数 K 。

◦ 输出 : 簇划分 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$ 。

◦ 算法步骤 :

- 从 \mathbb{D} 中随机选择 K 个样本作为初始均值向量 $\{\vec{\mu}_1, \vec{\mu}_2, \dots, \vec{\mu}_K\}$ 。
- 重复迭代直到算法收敛 , 迭代过程 :
 - 初始化阶段 : 取 $\mathbb{C}_k = \phi, k = 1, 2, \dots, K$
 - 划分阶段 : 随机挑选一个 **Batch** 的样本集合 $\mathbb{B} = \vec{x}_{b_1}, \dots, \vec{x}_{b_M}$, 其中 M 为批大小。
 - 计算 $\vec{x}_i, i = b_1, \dots, b_M$ 的簇标记 : $\lambda_i = \arg \min_k \|\vec{x}_i - \vec{\mu}_k\|_2, k \in \{1, 2, \dots, K\}$
 - 即 : 将 \vec{x}_i 离哪个簇的均值向量最近 , 则该样本就标记为那个簇。
 - 然后将样本 $\vec{x}_i, i = b_1, \dots, b_M$ 划入相应的簇 : $\mathbb{C}_{\lambda_i} = \mathbb{C}_{\lambda_i} \cup \{\vec{x}_i\}$ 。
 - 重计算阶段 : 计算 $\hat{\vec{\mu}}_k : \hat{\vec{\mu}}_k = \frac{1}{|\mathbb{C}_k|} \sum_{\vec{x}_i \in \mathbb{C}_k} \vec{x}_i$ 。
 - 终止条件判断 :
 - 如果对所有的 $k \in \{1, 2, \dots, K\}$, 都有 $\hat{\vec{\mu}}_k = \vec{\mu}_k$, 则算法收敛 , 终止迭代。
 - 否则重赋值 $\vec{\mu}_k = \hat{\vec{\mu}}_k$ 。

2.2 学习向量量化

1. 与一般聚类算法不同 , 学习向量量化 **Learning Vector Quantization:LVQ** 假设数据样本带有类别标记 , 学习过程需要利用样本的这些监督信息来辅助聚类。

2. 给定样本集 $\mathbb{D} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots, (\vec{x}_N, y_N)\}$, $\vec{x} \in \mathcal{X}, y \in \mathcal{Y}$, LVQ 的目标是从特征空间中挑选一组样本作为原型向量 $\{\vec{p}_1, \vec{p}_2, \dots, \vec{p}_Q\}$ 。

- 每个原型向量代表一个聚类簇，簇标记 $y_{p_q} \in \mathcal{Y}, q = 1, 2, \dots, Q$ 。即：簇标记从类别标记中选取。
 - 原型向量从特征空间中取得，它们不一定就是 \mathbb{D} 中的某个样本。
3. LVQ 的想法是：通过从样本中挑选一组样本作为原型向量 $\{\vec{p}_1, \vec{p}_2, \dots, \vec{p}_Q\}$ ，可以实现对样本空间 \mathcal{X} 的簇划分。

- 对任意样本 \vec{x} ，它被划入与距离最近的原型向量所代表的簇中。
- 对于每个原型向量 \vec{p}_q ，它定义了一个与之相关的一个区域 \mathbf{R}_q ，该区域中每个样本与 \vec{p}_q 的距离都不大于它与其他原型向量 $\vec{p}_{q'}$ 的距离。

$$\mathbf{R}_q = \{\vec{x} \in \mathcal{X} \mid \|\vec{x} - \vec{p}_q\|_2 \leq \min_{q' \neq q} \|\vec{x} - \vec{p}_{q'}\|_2\}$$

- 区域 $\{\mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_Q\}$ 对样本空间 \mathcal{X} 形成了一个簇划分，该划分通常称作 Voronoi 割分。

4. 问题是如何从样本中挑选一组样本作为原型向量？LVQ 的思想是：

- 首先挑选一组样本作为假设的原型向量。
- 然后对于训练集中的每一个样本 \vec{x}_i ，找出假设的原型向量中，距离该样本最近的原型向量 \vec{p}_{q_i} ：
 - 如果 \vec{x}_i 的标记与 \vec{p}_{q_i} 的标记相同，则更新 \vec{p}_{q_i} ，将该原型向量更靠近 \vec{x}_i 。
 - 如果 \vec{x}_i 的标记与 \vec{p}_{q_i} 的标记不相同，则更新 \vec{p}_{q_i} ，将该原型向量更远离 \vec{x}_i 。
- 不停进行这种更新，直到迭代停止条件（如以到达最大迭代次数，或者原型向量的更新幅度很小）。

5. LVQ 算法：

- 输入：
 - 样本集 $\mathbb{D} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots, (\vec{x}_N, y_N)\}$
 - 原型向量个数 Q
 - 各原型向量预设的类别标记 $\{y_{p_1}, y_{p_2}, \dots, y_{p_Q}\}$
 - 学习率 $\eta \in (0, 1)$
- 输出：原型向量 $\{\vec{p}_1, \vec{p}_2, \dots, \vec{p}_Q\}$
- 算法步骤：
 - 依次随机从类别 $\{y_{p_1}, y_{p_2}, \dots, y_{p_Q}\}$ 中挑选一个样本，初始化一组原型向量 $\{\vec{p}_1, \vec{p}_2, \dots, \vec{p}_Q\}$ 。
 - 重复迭代，直到算法收敛。迭代过程如下：
 - 从样本集 \mathbb{D} 中随机选取样本 (\vec{x}_i, y_i) ，挑选出距离 (\vec{x}_i, y_i) 最近的原型向量 \vec{p}_{q_i} ：

$$q_i = \arg \min_q \|\vec{x}_i - \vec{p}_q\|$$
 - 如果 \vec{p}_{q_i} 的类别等于 y_i ，则： $\vec{p}_{q_i} \leftarrow \vec{p}_{q_i} + \eta(\vec{x}_i - \vec{p}_{q_i})$ 。
 - 如果 \vec{p}_{q_i} 的类别不等于 y_i ，则： $\vec{p}_{q_i} \leftarrow \vec{p}_{q_i} - \eta(\vec{x}_i - \vec{p}_{q_i})$ 。

6. 在原型向量的更新过程中：

- 如果 \vec{p}_{q_i} 的类别等于 y_i ，则更新后， \vec{p}_{q_i} 与 \vec{x}_i 距离为：

$$\|\vec{p}_{q_i} - \vec{x}_i\|_2 = \|\vec{p}_{q_i} + \eta(\vec{x}_i - \vec{p}_{q_i}) - \vec{x}_i\|_2 = (1 - \eta)\|\vec{p}_{q_i} - \vec{x}_i\|_2$$

则更新后的原型向量 \vec{p}_{q_i} 距离 \vec{x}_i 更近。

- 如果 \vec{p}_{q_i} 的类别不等于 y_i ，则更新后， \vec{p}_{q_i} 与 \vec{x}_i 距离为：

$$\|\vec{p}_{q_i} - \vec{x}_i\|_2 = \|\vec{p}_{q_i} - \eta(\vec{x}_i - \vec{p}_{q_i}) - \vec{x}_i\|_2 = (1 + \eta)\|\vec{p}_{q_i} - \vec{x}_i\|_2$$

则更新后的原型向量 \vec{p}_{q_i} 距离 \vec{x}_i 更远。

7. 这里有一个隐含假设：即计算得到的样本 $\vec{p}_{q_i} \pm \eta(\vec{x}_i - \vec{p}_{q_i})$ （该样本可能不在样本集中）的标记就是更新之前 \vec{p}_{q_i} 的标记。

即：更新操作只改变原型向量的样本值，但是不改变该原型向量的标记。

2.3 高斯混合聚类

1. 高斯混合聚类采用概率模型来表达聚类原型。

2. 对于 n 维样本空间 \mathcal{X} 中的随机向量 \vec{x} ，若 \vec{x} 服从高斯分布，则其概率密度函数为：

$$p(\vec{x} | \vec{\mu}, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2} (\vec{x} - \vec{\mu})^T \Sigma^{-1} (\vec{x} - \vec{\mu})\right)$$

其中 $\vec{\mu} = (\mu_1, \mu_2, \dots, \mu_n)^T$ 为 n 维均值向量， Σ 是 $n \times n$ 的协方差矩阵。 \vec{x} 的概率密度函数由参数 $\vec{\mu}, \Sigma$ 决定。

3. 定义高斯混合分布： $p_{\mathcal{M}} = \sum_{k=1}^K \alpha_k p(\vec{x} | \vec{\mu}_k, \Sigma_k)$ 。该分布由 K 个混合成分组成，每个混合成分对应一个高斯分布。其中：

- $\vec{\mu}_k, \Sigma_k$ 是第 k 个高斯混合成分的参数。
- $\alpha_k > 0$ 是相应的混合系数，满足 $\sum_{k=1}^K \alpha_k = 1$ 。

4. 假设训练集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ 的生成过程是由高斯混合分布给出。

令随机变量 $Z \in \{1, 2, \dots, K\}$ 表示生成样本 \vec{x} 的高斯混合成分序号， Z 的先验概率 $P(Z = k) = \alpha_k$ 。

生成样本的过程分为两步：

- 首先根据概率分布 $\alpha_1, \alpha_2, \dots, \alpha_K$ 生成随机变量 Z 。
- 再根据 Z 的结果，比如 $Z = k$ ，根据概率 $p(\vec{x} | \vec{\mu}_k, \Sigma_k)$ 生成样本。

5. 根据贝叶斯定理，若已知输出为 \vec{x}_i ，则 Z 的后验分布为：

$$p_{\mathcal{M}}(Z = k | \vec{x}_i) = \frac{P(Z = k)p_{\mathcal{M}}(\vec{x}_i | Z = k)}{p_{\mathcal{M}}(\vec{x}_i)} = \frac{\alpha_k p(\vec{x}_i | \vec{\mu}_k, \Sigma_k)}{\sum_{l=1}^K \alpha_l p(\vec{x}_i | \vec{\mu}_l, \Sigma_l)}$$

其物理意义为：所有导致输出为 \vec{x}_i 的情况下， $Z = k$ 发生的概率。

6. 当高斯混合分布已知时，高斯混合聚类将样本集 \mathbb{D} 划分成 K 个簇 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$ 。

对于每个样本 \vec{x}_i ，给出它的簇标记 λ_i 为：

$$\lambda_i = \arg \max_k p_{\mathcal{M}}(Z = k | \vec{x}_i)$$

即：如果 \vec{x}_i 最有可能是 $Z = k$ 产生的，则将该样本划归到簇 \mathbb{C}_k 。

这就是通过最大后验概率确定样本所属的聚类。

7. 现在的问题是，如何学习高斯混合分布的参数。由于涉及到隐变量 Z ，可以采用 EM 算法求解。

具体求解参考 EM 算法的章节部分。

三、密度聚类

1. 密度聚类 density-based clustering 假设聚类结构能够通过样本分布的紧密程度确定。

2. 密度聚类算法从样本的密度的角度来考察样本之间的可连接性，并基于可连接样本的不断扩张聚类簇，从而获得最终的聚类结果。

3.1 DBSCAN 算法

1. DBSCAN 是一种著名的密度聚类算法，它基于一组邻域参数 $(\epsilon, MinPts)$ 来刻画样本分布的紧密程度。
2. 给定数据集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ ，定义：
 - ϵ -邻域： $N_\epsilon(\vec{x}_i) = \{\vec{x}_j \in \mathbb{D} \mid distance(\vec{x}_i, \vec{x}_j) \leq \epsilon\}$ 。
即： $N_\epsilon(\vec{x}_i)$ 包含了样本集 \mathbb{D} 中与 \vec{x}_i 距离不大于 ϵ 的所有的样本。
 - 核心对象 core object：若 $|N_\epsilon(\vec{x}_i)| \geq MinPts$ ，则称 \vec{x}_i 是一个核心对象。
即：若 \vec{x}_i 的 ϵ -邻域中至少包含 $MinPts$ 个样本，则 \vec{x}_i 是一个核心对象。
 - 密度直达 density-reachable：若 \vec{x}_i 是一个核心对象，且 $\vec{x}_j \in N_\epsilon(\vec{x}_i)$ ，则称 \vec{x}_j 由 \vec{x}_i 密度直达，记作 $\vec{x}_i \rightarrowtail \vec{x}_j$ 。
 - 密度可达 density-reachable：对于 \vec{x}_i 和 \vec{x}_j ，若存在样本序列 $(\vec{p}_0, \vec{p}_1, \vec{p}_2, \dots, \vec{p}_m, \vec{p}_{m+1})$ ，其中 $\vec{p}_0 = \vec{x}_i, \vec{p}_{m+1} = \vec{x}_j, \vec{p}_s \in \mathbb{D}$ ，如果 \vec{p}_{s+1} 由 \vec{p}_s 密度直达，则称 \vec{x}_j 由 \vec{x}_i 密度可达，记作 $\vec{x}_i \rightsquigarrow \vec{x}_j$ 。
 - 密度相连 density-connected：对于 \vec{x}_i 和 \vec{x}_j ，若存在 \vec{x}_k ，使得 \vec{x}_i 与 \vec{x}_j 均由 \vec{x}_k 密度可达，则称 \vec{x}_i 与 \vec{x}_j 密度相连，记作 $\vec{x}_i \sim \vec{x}_j$ 。
3. DBSCAN 算法的簇定义：给定邻域参数 $(\epsilon, MinPts)$ ，一个簇 $\mathbb{C} \subseteq \mathbb{D}$ 是满足下列性质的非空样本子集：
 - 连接性 connectivity：若 $\vec{x}_i \in \mathbb{C}, \vec{x}_j \in \mathbb{C}$ ，则 $\vec{x}_i \sim \vec{x}_j$ 。
 - 最大性 maximality：若 $\vec{x}_i \in \mathbb{C}$ ，且 $\vec{x}_i \rightsquigarrow \vec{x}_j$ ，则 $\vec{x}_j \in \mathbb{C}$ 。
 即一个簇是由密度可达关系导出的最大的密度相连样本集合。
4. DBSCAN 算法的思想：若 \vec{x} 为核心对象，则 \vec{x} 密度可达的所有样本组成的集合记作 $\mathbb{X} = \{\vec{x}' \in \mathbb{D} \mid \vec{x} \rightsquigarrow \vec{x}'\}$ 。
可以证明： \mathbb{X} 就是满足连接性与最大性的簇。
于是 DBSCAN 算法首先任选数据集中的一个核心对象作为种子 seed，再由此出发确定相应的聚类簇。
5. DBSCAN 算法：
 - 输入：
 - 数据集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$
 - 邻域参数 $(\epsilon, MinPts)$
 - 输出：簇划分 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$
 - 算法步骤：
 - 初始化核心对象集合为空集： $\Omega = \emptyset$
 - 寻找核心对象：
 - 遍历所有的样本点 $\vec{x}_i, i = 1, 2, \dots, N$ ，计算 $N_\epsilon(\vec{x}_i)$
 - 如果 $|N_\epsilon(\vec{x}_i)| \geq MinPts$ ，则 $\Omega = \Omega \cup \{\vec{x}_i\}$
 - 迭代：以任一未访问过的核心对象为出发点，找出有其密度可达的样本生成的聚类簇，直到所有核心对象都被访问为止。
6. 注意：
 - 若在核心对象 \vec{o}_1 的寻找密度可达的样本的过程中，发现核心对象 \vec{o}_2 是由 \vec{o}_1 密度可达的，且 \vec{o}_2 尚未被访问，则将 \vec{o}_2 加入 \vec{o}_1 所属的簇，并且标记 \vec{o}_2 为已访问。
 - 对于 \mathbb{D} 中的样本点，它只可能属于某一个聚类簇。因此在核心对象 \vec{o}_i 的寻找密度可达的样本的过程中，它只能在标记为未访问的样本中寻找（标记为已访问的样本已经属于某个聚类簇了）。
7. DBSCAN 算法的优点：

- 簇的数量由算法自动确定，无需人工指定。
- 基于密度定义，能够对抗噪音。
- 可以处理任意形状和大小的簇。

8. DBSCAN 算法的缺点：

- 若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。因为此时参数 ϵ 和 $MinPts$ 的选择比较困难。
- 无法应用于密度不断变化的数据集中。

3.2 Mean-Shift 算法

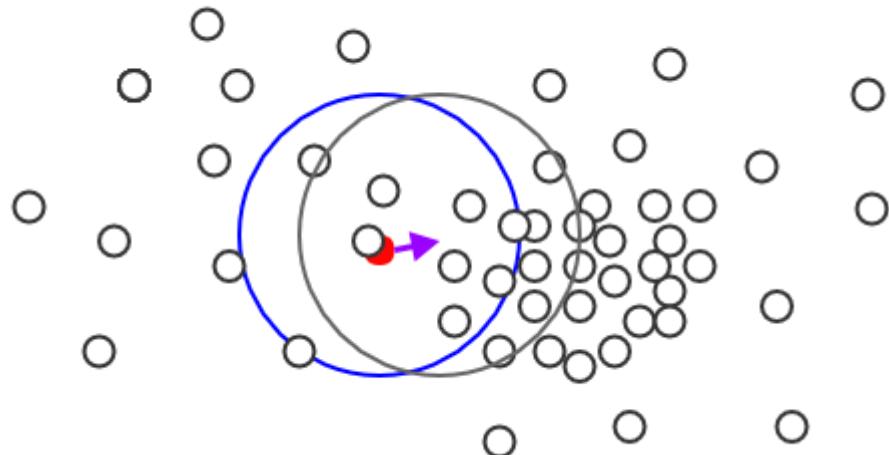
1. Mean-Shift 是基于核密度估计的爬山算法，可以用于聚类、图像分割、跟踪等领域。
2. 给定 n 维空间的 N 个样本组成的数据集 $\mathbb{D} = \{\vec{x}_1, \dots, \vec{x}_N\}$ ，给定一个中心为 \vec{x} 、半径为 h 的球形区域 \mathbb{S} （称作兴趣域），定义其 mean shift 向量为： $\vec{M}(\vec{x}) = \frac{1}{|\mathbb{S}|} \sum_{\vec{x}_i \in \mathbb{S}} (\vec{x}_i - \vec{x})$ 。

3. Mean-Shift 算法的基本思路是：

- 首先任选一个点作为聚类的中心来构造兴趣域。
- 然后计算当前的 mean shift 向量，兴趣域的中心移动为： $\vec{x} \leftarrow \vec{x} + \vec{M}(\vec{x})$ 。
移动过程中，兴趣域范围内的所有样本都标记为同一个簇。
- 如果 mean shift 向量为 0，则停止移动，说明兴趣域已到达数据点最密集的区域。

因此 Mean-Shift 会向着密度最大的区域移动。

下图中：蓝色为当前的兴趣域，红色为当前的中心点 \vec{x} ；紫色向量为 mean shift 向量 $\vec{M}(\vec{x})$ ，灰色为移动之后的兴趣域。



4. 在计算 mean shift 向量的过程中假设每个样本的作用都是相等的。实际上随着样本与中心点的距离不同，样本对于 mean shift 向量的贡献不同。

定义高斯核函数为： $g(x) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{x^2}{2})$ ，则重新 mean shift 向量定义为：

$$\vec{M}(\vec{x}) = \vec{m}(\vec{x}) - \vec{x}, \quad \vec{m}(\vec{x}) = \frac{\sum_{\vec{x}_i \in \mathbb{S}} \vec{x}_i g\left(\left\|\frac{\vec{x}_i - \vec{x}}{h}\right\|^2\right)}{\sum_{\vec{x}_i \in \mathbb{S}} g\left(\left\|\frac{\vec{x}_i - \vec{x}}{h}\right\|^2\right)}$$

其中 h 称做带宽。 $\left\| \frac{\vec{x}_i - \vec{x}}{h} \right\|^2$ 刻画了样本 \vec{x}_i 距离中心点 \vec{x} 相对于半径 h 的相对距离。

5. Mean_Shift 算法：

◦ 输入：

- 数据集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$
- 带宽参数 h
- 迭代阈值 ϵ_1 ，簇阈值 ϵ_2

◦ 输出：簇划分 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots\}$

◦ 算法步骤：

迭代，直到所有的样本都被访问过。迭代过程为（设已有的簇为 $1, 2, \dots, L-1$ ）：

- 在未访问过的样本中随机选择一个点作为中心点 \vec{x} ，找出距它半径为 h 的兴趣域，记做 \mathbb{S} 。
- 将 \mathbb{S} 中的样本的簇标记设置为 L （一个新的簇）。
- 计算当前的 mean shift 向量，兴趣域中心的移动为：

$$\vec{x} \leftarrow \vec{x} + \vec{M}(\vec{x}) = \vec{m}(\vec{x}) = \frac{\sum_{\vec{x}_i \in \mathbb{S}} \vec{x}_i g\left(\left\| \frac{\vec{x}_i - \vec{x}}{h} \right\|^2\right)}{\sum_{\vec{x}_i \in \mathbb{S}} g\left(\left\| \frac{\vec{x}_i - \vec{x}}{h} \right\|^2\right)}$$

在移动过程中，兴趣域内的所有点标记为访问过，并且将它们的簇标记设置为 L 。

- 如果 $\|\vec{M}(\vec{x})\| \leq \epsilon_1$ ，则本次结束本次迭代。
- 设已有簇中，簇 l 的中心点 $\vec{x}^{(l)}$ 与 \vec{x} 距离最近，如果 $\|\vec{x}^{(l)} - \vec{x}\| \leq \epsilon_2$ ，则将当前簇和簇 l 合并。
合并时，当前簇中的样本的簇标记重新修改为 l 。

当所有的样本都被访问过时，重新分配样本的簇标记（因为可能有的样本被多个簇标记过）：若样本被多个簇标记过，则选择最大的那个簇作为该样本的簇标记。即：尽可能保留大的簇。

6. 可以证明：Mean_Shift 算法每次移动都是向着概率密度函数增加的方向移动。

在 n 维欧式空间中，对空间中的点 \vec{x} 的概率密度估计为：

$$\hat{f}(\vec{x}) = \frac{1}{N} \sum_{i=1}^N K_H(\vec{x} - \vec{x}_i), \quad K_H(\vec{x}) = |\mathbf{H}|^{-\frac{1}{2}} K(\mathbf{H}^{-\frac{1}{2}} \vec{x})$$

其中：

- $K(\vec{x})$ 表示空间中的核函数， \mathbf{H} 为带宽矩阵。
- 通常 $K(\cdot)$ 采用放射状对称核函数 $K(\vec{x}) = c_k \times k(\|\vec{x}\|^2)$ ， $k(\cdot)$ 为 $K(\cdot)$ 的轮廓函数， c_k （一个正数）为标准化常数从而保证 $K(\vec{x})$ 的积分为 1。
- 通常带宽矩阵采用 $\mathbf{H} = h^2 \mathbf{I}$ ， h 为带宽参数。

因此有： $\hat{f}(\vec{x}) = \frac{c_k}{Nh^n} \sum_{k=1}^N k\left(\left\| \frac{\vec{x} - \vec{x}_i}{h} \right\|^2\right)$ 。则有梯度：

$$\nabla_{\vec{x}} \hat{f}(\vec{x}) = \frac{2c_k}{Nh^{n+2}} \sum_{k=1}^N (\vec{x} - \vec{x}_i) k'\left(\left\| \frac{\vec{x} - \vec{x}_i}{h} \right\|^2\right)$$

记 $k(\cdot)$ 的导数为 $g(\cdot) = k'(\cdot)$ 。取 $g(\cdot)$ 为新的轮廓函数， c_g （一个正数）为新的标准化常数， $G(\vec{x}) = c_g \times g(\|\vec{x}\|^2)$ 。

则有：

$$\begin{aligned}\nabla_{\vec{x}} \hat{f}(\vec{x}) &= \frac{2c_k}{Nh^{n+2}} \sum_{i=1}^N (\vec{x} - \vec{x}_i) g\left(\left\|\frac{\vec{x} - \vec{x}_i}{h}\right\|^2\right) \\ &= \frac{2c_k}{h^2 c_g} \left[\frac{c_g}{Nh^n} \sum_{i=1}^N g\left(\left\|\frac{\vec{x} - \vec{x}_i}{h}\right\|^2\right) \right] \left[\frac{\sum_{i=1}^N \vec{x}_i g\left(\left\|\frac{\vec{x} - \vec{x}_i}{h}\right\|^2\right)}{\sum_{i=1}^N g\left(\left\|\frac{\vec{x} - \vec{x}_i}{h}\right\|^2\right)} - \vec{x} \right]\end{aligned}$$

定义 $\hat{f}_g(\vec{x}) = \frac{c_g}{Nh^n} \sum_{i=1}^N g\left(\left\|\frac{\vec{x} - \vec{x}_i}{h}\right\|^2\right)$ ，则它表示基于核函数 $G(\cdot)$ 的概率密度估计，始终为非负数。

根据前面定义： $\vec{M}(\vec{x}) = \frac{\sum_{i=1}^N \vec{x}_i g\left(\left\|\frac{\vec{x} - \vec{x}_i}{h}\right\|^2\right)}{\sum_{i=1}^N g\left(\left\|\frac{\vec{x} - \vec{x}_i}{h}\right\|^2\right)} - \vec{x}$ ，则有： $\nabla_{\vec{x}} \hat{f}(\vec{x}) = \frac{2c_k}{h^2 c_g} \times \hat{f}_g(\vec{x}) \times \vec{M}(\vec{x})$ 。

因此 $\vec{M}(\vec{x})$ 正比于 $\nabla_{\vec{x}} \hat{f}(\vec{x})$ ，因此 mean shift 向量的方向始终指向概率密度增加最大的方向。

上式计算 $\sum_{i=1}^N$ 时需要考虑所有的样本，计算复杂度太大。作为一个替代，可以考虑使用 \vec{x} 距离 h 内的样本，即兴趣域内的样本。即可得到： $\vec{M}(\vec{x}) = \frac{\sum_{\vec{x}_i \in \mathbb{S}} \vec{x}_i g\left(\left\|\frac{\vec{x}_i - \vec{x}}{h}\right\|^2\right)}{\sum_{\vec{x}_i \in \mathbb{S}} g\left(\left\|\frac{\vec{x}_i - \vec{x}}{h}\right\|^2\right)} - \vec{x}$ 。

7. Mean-Shift 算法优点：

- 簇的数量由算法自动确定，无需人工指定。
- 基于密度定义，能够对抗噪音。
- 可以处理任意形状和大小的簇。
- 没有局部极小值点，因此当给定带宽参数 h 时，其聚类结果就是唯一的。

8. Mean_Shift 算法缺点：

- 无法控制簇的数量。
- 无法区分有意义的簇和无意义的簇。如：在 Mean_Shift 算法中，异常点也会形成它们自己的簇。

四、层次聚类

1. 层次聚类 hierarchical clustering 试图在不同层次上对数据集进行划分，从而形成树形的聚类结构。

4.1 AGNES 算法

1. AGglomerative NESting : AGNES 是一种常用的采用自底向上聚合策略的层次聚类算法。

2. AGNES 首先将数据集中的每个样本看作一个初始的聚类簇，然后在算法运行的每一步中，找出距离最近的两个聚类簇进行合并。

合并过程不断重复，直到达到预设的聚类簇的个数。

3. 这里的关键在于：如何计算聚类簇之间的距离？

由于每个簇就是一个集合，因此只需要采用关于集合的某个距离即可。给定聚类簇 C_i, C_j ，有三种距离：

- 最小距离： $d_{min}(C_i, C_j) = \min_{\vec{x}_i \in C_i, \vec{x}_j \in C_j} distance(\vec{x}_i, \vec{x}_j)$ 。
最小距离由两个簇的最近样本决定。
- 最大距离： $d_{max}(C_i, C_j) = \max_{\vec{x}_i \in C_i, \vec{x}_j \in C_j} distance(\vec{x}_i, \vec{x}_j)$ 。
最大距离由两个簇的最远样本决定。
- 平均距离： $d_{avg}(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{\vec{x}_i \in C_i} \sum_{\vec{x}_j \in C_j} distance(\vec{x}_i, \vec{x}_j)$ 。

平均距离由两个簇的所有样本决定。

4. AGNES 算法可以采取上述任意一种距离：

- 当 AGNES 算法的聚类簇距离采用 d_{min} 时，称作单链接 single-linkage 算法。
- 当 AGNES 算法的聚类簇距离采用 d_{max} 时，称作全链接 complete-linkage 算法。
- 当 AGNES 算法的聚类簇距离采用 d_{avg} 时，称作均链接 average-linkage 算法。

5. AGNES 算法：

- 输入：

- 数据集 $\mathbb{D} = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$
- 聚类簇距离度量函数 $d(\cdot, \cdot)$
- 聚类簇数量 K

- 输出：簇划分 $\mathcal{C} = \{\mathbb{C}_1, \mathbb{C}_2, \dots, \mathbb{C}_K\}$

- 算法步骤：

- 初始化：将每个样本都作为一个簇

$$\mathbb{C}_i = \{\vec{x}_i\}, i = 1, 2, \dots, N$$

- 迭代，终止条件为聚类簇的数量为 K 。迭代过程为：

计算聚类簇之间的距离，找出距离最近的两个簇，将这两个簇合并。

每进行一次迭代，聚类簇的数量就减少一些。

6. AGNES 算法的优点：

- 距离容易定义，使用限制较少。
- 可以发现聚类的层次关系。

7. AGNES 算法的缺点：

- 计算复杂度较高。
- 算法容易聚成链状。

4.2 BIRCH 算法

1. BIRCH: Balanced Iterative Reducing and Clustering Using Hierarchies 算法通过聚类特征树 CF Tree: Clustering Feature True 来执行层次聚类，适合于样本量较大、聚类类别数较大的场景。

4.2.1 聚类特征

1. 聚类特征 CF：每个 CF 都是刻画一个簇的特征的三元组： $CF = (\text{num}, \vec{\Sigma}_l, \Sigma_s)$ 。其中：

- num：表示簇内样本数量的数量。
- $\vec{\Sigma}_l$ ：表示簇内样本的线性求和： $\vec{\Sigma}_l = \sum_{\vec{x}_i \in \mathbb{S}} \vec{x}_i$ ，其中 \mathbb{S} 为该 CF 对应的簇。
- Σ_s ：表示簇内样本的长度的平方和。 $\Sigma_s = \sum_{\vec{x}_i \in \mathbb{S}} \|\vec{x}_i\|^2 = \sum_{\vec{x}_i \in \mathbb{S}} \vec{x}_i^T \vec{x}_i$ ，其中 \mathbb{S} 为该 CF 对应的簇。

2. 根据 CF 的定义可知：如果 CF1 和 CF2 分别表示两个不相交的簇的特征，如果将这两个簇合并成一个大簇，则大簇的特征为： $CF_{merge} = CF_1 + CF_2$ 。

即：CF 满足可加性。

3. 给定聚类特征 CF，则可以统计出簇的一些统计量：

- 簇心： $\bar{\vec{x}} = \frac{\vec{\Sigma}_l}{\text{num}}$ 。

- 簇内数据点到簇心的平均距离（也称作簇的半径）： $\rho = \sqrt{\frac{\text{num} \times \Sigma_s - \|\vec{\Sigma}_l\|^2}{\text{num}}}$ 。
 - 簇内两两数据点之间的平均距离（也称作簇的直径）： $\delta = \sqrt{\frac{2 \times \text{num} \times \Sigma_s - 2\|\vec{\Sigma}_l\|^2}{\text{num} \times \text{num}-1}}$ 。
4. 给定两个不相交的簇，其特征分别为 $CF_1 = (\text{num}_1, \vec{\Sigma}_{l,1}, \Sigma_{s,1})$ 和 $CF_2 = (\text{num}_2, \vec{\Sigma}_{l,2}, \Sigma_{s,2})$ 。

假设合并之后的簇为 $CF_3 = (\text{num}_3, \vec{\Sigma}_{l,3}, \Sigma_{s,3})$ ，其中 $\text{num}_3 = \text{num}_1 + \text{num}_2$ ， $\vec{\Sigma}_{l,3} = \vec{\Sigma}_{l,1} + \vec{\Sigma}_{l,2}$ ， $\Sigma_{s,3} = \Sigma_{s,1} + \Sigma_{s,2}$ 。

可以通过下列的距离来度量 CF_1 和 CF_2 的相异性：

- 簇心欧氏距离 `centroid Euclidian distance`： $d_0 = \sqrt{\|\vec{x}_1 - \vec{x}_2\|_2^2}$ ，其中 \vec{x}_1, \vec{x}_2 分别为各自的簇心。
- 簇心曼哈顿距离 `centroid Manhattan distance`： $d_1 = \|\vec{x}_1 - \vec{x}_2\|_1$ 。
- 簇连通平均距离 `average inter-cluster distance`：

$$d_2 = \sqrt{\frac{\sum_{\vec{x}_i \in S_1} \sum_{\vec{x}_j \in S_2} \|\vec{x}_i - \vec{x}_j\|_2^2}{\text{num}_1 \times \text{num}_2}} = \sqrt{\frac{\Sigma_{s,1}}{\text{num}_1} + \frac{\Sigma_{s,2}}{\text{num}_2} - 2 \frac{\vec{\Sigma}_{l,1}^T \vec{\Sigma}_{l,2}}{\text{num}_1 \times \text{num}_2}}$$

- 全连通平均距离 `average intra-cluster distance`：

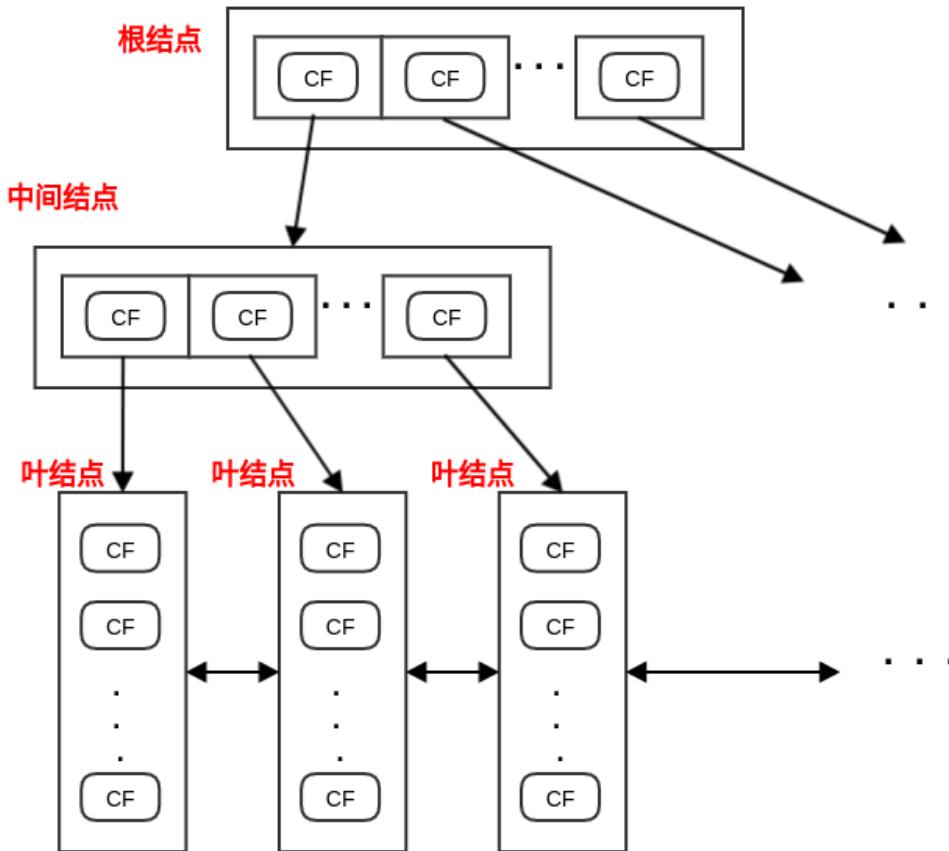
$$\begin{aligned} d_3 &= \sqrt{\frac{\sum_{\vec{x}_i \in S_3} \sum_{\vec{x}_j \in S_3} \|\vec{x}_i - \vec{x}_j\|_2^2}{(\text{num}_1 + \text{num}_2) \times (\text{num}_1 + \text{num}_2 - 1)}} \\ &= \sqrt{\frac{2(\text{num}_1 + \text{num}_2)(\Sigma_{s,1} + \Sigma_{s,2}) - 2\|\vec{\Sigma}_{l,1} - \vec{\Sigma}_{l,2}\|_2^2}{(\text{num}_1 + \text{num}_2) \times (\text{num}_1 + \text{num}_2 - 1)}} \end{aligned}$$

- 方差恶化距离 `variance increass distance`： $d_4 = \rho_3 - \rho_1 - \rho_2$ 。

4.2.2 CF 树

1. `CF` 树的结构类似于平衡 `B+` 树。树由三种结点构成：根结点、中间结点、叶结点。

- 根结点、中间结点：由若干个聚类特征 `CF`，以及这些 `CF` 指向子结点的指针组成。
- 叶结点：由若干个聚类特征 `CF` 组成。
 - 叶结点没有子结点，因此 `CF` 没有指向子结点的指针。
 - 所有的叶结点通过双向链表连接起来。
 - 在 `BIRCH` 算法结束时，叶结点的每个 `CF` 对应的样本集就对应了一个簇。



2. **CF** 树有三个关键参数：

- 枝平衡因子 β ：非叶结点中，最多不能包含超过 β 个 **CF**。
- 叶平衡因子 λ ：叶结点中，最多不能包含超过 λ 个 **CF**。
- 空间阈值 τ ：叶结点中，每个 **CF** 对应的子簇的大小（通过簇半径 ρ 来描述）不能超过 τ 。

3. 由于 **CF** 的可加性，所以 **CF** 树中，每个父结点的 **CF** 等于它所有子结点的所有 **CF** 之和。

4. **CF** 树的生成算法：

- 输入：
 - 样本集 $\mathbb{D} = \{\vec{x}_1, \dots, \vec{x}_N\}$
 - 枝平衡因子 β
 - 叶平衡因子 λ
 - 空间阈值 τ
- 输出：**CF** 树
- 算法步骤：
 - 初始化：**CF** 树的根结点为空。
 - 随机从样本集 \mathbb{D} 中选出一个样本，放入一个新的 **CF** 中，并将该 **CF** 放入到根结点中。
 - 遍历 \mathbb{D} 中的样本，并向 **CF** 树中插入。迭代停止条件为：样本集 \mathbb{D} 中所有样本都插入到 **CF** 树中。
- 迭代过程如下：
 - 随机从样本集 \mathbb{D} 中选出一个样本 \vec{x}_i ，从根结点向下寻找与 \vec{x}_i 距离最近的叶结点 $leaf_j$ ，和 $leaf_j$ 里与 \vec{x}_i 最近的 CF_k 。

- 如果 \vec{x}_i 加入到 CF_k 对应的簇中之后，该簇的簇半径 $\rho \leq \tau$ ，则将 \vec{x}_i 加入到 CF_k 对应的簇中，并更新路径上的所有 CF 。本次插入结束。
- 否则，创建一个新的 CF ，将 \vec{x}_i 放入该 CF 中，并将该 CF 添加到叶结点 $leaf_j$ 中。
如果 $leaf_j$ 的 CF 数量小于 λ ，则更新路径上的所有 CF 。本次插入结束。
- 否则，将叶结点 $leaf_j$ 分裂为两个新的叶结点 $leaf_{j,1}, leaf_{j,2}$ 。分裂方式：
 - 选择叶结点 $leaf_j$ 中距离最远的两个 CF ，分别作为 $leaf_{j,1}, leaf_{j,2}$ 中的首个 CF 。
 - 将叶结点 $leaf_j$ 中剩下的 CF 按照距离这两个 CF 的远近，分别放置到 $leaf_{j,1}, leaf_{j,2}$ 中。
- 依次向上检查父结点是否也需要分裂。如果需要，则按照和叶子结点分裂方式相同。

4.2.3 BIRCH 算法

1. BIRCH 算法的主要步骤是建立 CF 树，除此之外还涉及到 CF 树的瘦身、离群点的处理。
2. BIRCH 需要对 CF 树瘦身，有两个原因：
 - 将数据点插入到 CF 树的过程中，用于存储 CF 树结点及其相关信息的内存有限，导致部分数据点生长形成的 CF 树占满了内存。因此需要对 CF 树瘦身，从而使得剩下的数据点也能插入到 CF 树中。
 - CF 树生长完毕后，如果叶结点中的 CF 对应的簇太小，则会影响后续聚类的速度和质量。
3. BIRCH 瘦身是在将 τ 增加的过程。算法会在内存中同时存放旧树 T 和新树 T' ，初始时刻 T' 为空。
 - 算法同时处理 T 和 T' ，将 T 中的 CF 迁移到 T' 中。
 - 在完成所有的 CF 迁移之后， T 为空， T' 就是瘦身后的 CF 树。
4. BIRCH 离群点的处理：
 - 在对 CF 瘦身之后，搜索所有叶结点中的所有子簇，寻找那些稀疏子簇，并将稀疏子簇放入待定区。
稀疏子簇：簇内数据点的数量远远少于所有子簇的平均数据点的那些子簇。
 - || 将稀疏子簇放入待定区时，需要同步更新 CF 树上相关路径及结点。
 - 当 D 中所有数据点都被插入之后，扫描待定区中的所有数据点（这些数据点就是候选的离群点），并尝试将其插入到 CF 树中。
如果数据点无法插入到 CF 树中，则可以确定为真正的离群点。
5. BIRCH 算法：
 - 输入：
 - 样本集 $D = \{\vec{x}_1, \dots, \vec{x}_N\}$
 - 枝平衡因子 β
 - 叶平衡因子 λ
 - 空间阈值 τ
 - 输出： CF 树
 - 算法步骤：
 - 建立 CF 树。
 - （可选）对 CF 树瘦身、去除离群点，以及合并距离非常近的 CF 。
 - （可选）利用其它的一些聚类算法（如： $k\text{-means}$ ）对 CF 树的所有叶结点中的 CF 进行聚类，得到新的 CF 树。

这是为了消除由于样本读入顺序不同导致产生不合理的树结构。

这一步是对 CF 结构进行聚类。由于每个 CF 对应一组样本，因此对 CF 聚类就是对 \mathbb{D} 进行聚类。

- (可选) 将上一步得到的、新的 CF 树的叶结点中每个簇的中心点作为簇心，对所有样本按照它距这些中心点的距离远近进行聚类。

这是对上一步的结果进行精修。

6. $BIRCH$ 算法优点：

- 节省内存。所有样本都存放在磁盘上，内存中仅仅存放 CF 结构。
- 计算速度快。只需要扫描一遍就可以建立 CF 树。
- 可以识别噪声点。

7. $BIRCH$ 算法缺点：

- 结果依赖于数据点的插入顺序。原本属于同一个簇的两个点可能由于插入顺序相差很远，从而导致分配到不同的簇中。
甚至同一个点在不同时刻插入，也会被分配到不同的簇中。
- 对非球状的簇聚类效果不好。这是因为簇直径 δ 和簇间距离的计算方法导致。
- 每个结点只能包含规定数目的子结点，最后聚类的簇可能和真实的簇差距很大。

8. $BIRCH$ 可以不用指定聚类的类别数 K 。

- 如果不指定 K ，则最终叶结点中 CF 的数量就是 K 。
- 如果指定 K ，则需要将叶结点按照距离远近进行合并，直到叶结点中 CF 数量等于 K 。

五、谱聚类

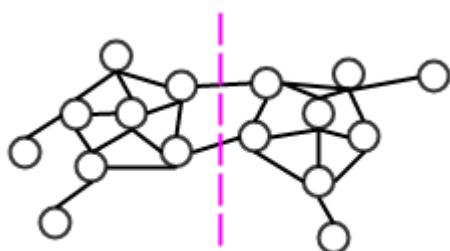
1. 谱聚类 `spectral clustering` 是一种基于图论的聚类方法。

2. 谱聚类的主要思想是：基于数据集 $\mathbb{D} = \{\vec{x}_1, \dots, \vec{x}_N\}$ 来构建图 $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ ，其中：

- 顶点 \mathbb{V} ：由数据集中的数据点组成： $\mathbb{V} = \{1, 2, \dots, N\}$ 。
- 边 \mathbb{E} ：任意一对顶点之间存在边。

距离越近的一对顶点，边的权重越高；距离越远的一对顶点，边的权重越低。

通过对图 \mathcal{G} 进行切割，使得切割之后：不同子图之间的边的权重尽可能的低、各子图内的边的权重尽可能的高。这样就完成了聚类。



5.1 邻接矩阵

1. 在图 $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ 中，定义权重 $w_{i,j}$ 为顶点 i 和 j 之间的权重，其中 $i, j \in \mathbb{V}$ 。

定义 $\mathbf{W} = (w_{i,j})_{N \times N}$ 为邻接矩阵：

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,N} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N,1} & w_{N,2} & \cdots & w_{N,N} \end{bmatrix}$$

由于 \mathcal{G} 为无向图，因此 $w_{i,j} = w_{j,i}$ 。即： $\mathbf{W} = \mathbf{W}^T$ 。

- 对图中顶点 i ，定义它的度 d_i 为：所有与顶点 i 相连的边的权重之和： $d_i = \sum_{j=1}^N w_{i,j}$ 。

定义度矩阵 \mathbf{D} 为一个对角矩阵，其中对角线分别为各顶点的度：

$$\mathbf{D} = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_N \end{bmatrix}$$

- 对于顶点集合 \mathbb{V} 的一个子集 $\mathbb{A} \subset \mathbb{V}$ ，定义 $|\mathbb{A}|$ 为子集 \mathbb{A} 中点的个数；定义 $vol(\mathbb{A}) = \sum_{i \in \mathbb{A}} d_i$ ，为子集 \mathbb{A} 中所有点的度之和。

2. 事实上在谱聚类中，通常只给定数据集 $\mathbb{D} = \{\vec{x}_1, \dots, \vec{x}_N\}$ ，因此需要计算出邻接矩阵 \mathbf{W} 。

- 基本思想是：距离较近的一对点（即相似度较高），边的权重较高；距离较远的一对点（即相似度较低），边的权重较低。
- 基本方法是：首先构建相似度矩阵 $\mathbf{S} = (s_{i,j})_{N \times N}$ ，然后使用 ϵ -近邻法、 K 近邻法、或者全连接法。

$$\mathbf{S} = \begin{bmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,N} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ s_{N,1} & s_{N,2} & \cdots & s_{N,N} \end{bmatrix}$$

- 通常相似度采用高斯核： $s_{i,j} = \exp\left(-\frac{\|\vec{x}_i - \vec{x}_j\|_2^2}{2\sigma^2}\right)$ 。此时有 $s_{i,j} = s_{j,i}$ 。即： $\mathbf{S} = \mathbf{S}^T$ 。
- 也可以选择不同的核函数，如：多项式核函数、高斯核函数、sigmoid 核函数。

3. ϵ -近邻法：设置一个距离阈值 ϵ ，定义邻接矩阵 \mathbf{W} 为：

$$w_{i,j} = \begin{cases} 0, & s_{i,j} > \epsilon \\ \epsilon, & s_{i,j} \leq \epsilon \end{cases}$$

即：一对相似度小于 ϵ 的点，边的权重为 ϵ ；否则边的权重为 0。

ϵ -近邻法得到的权重要么是 0，要么是 ϵ ，权重量很不精确，因此实际应用较少。

4. K 近邻法：利用 KNN 算法选择每个样本最近的 K 个点作为近邻，其它点与当前点之间的边的权重为 0。

这种做法会导致邻接矩阵 \mathbf{W} 非对称，因为当 \vec{x}_j 是 \vec{x}_i 的 K 近邻时， \vec{x}_i 不一定是 \vec{x}_j 的 K 近邻。

为了解决对称性问题，有两种做法：

- 只要一个点在另一个点的 K 近邻中，则认为是近邻。即：取并集。

$$w_{i,j} = w_{j,i} = \begin{cases} 0, & \vec{x}_i \notin KNN(\vec{x}_j) \text{ and } \vec{x}_j \notin KNN(\vec{x}_i) \\ s_{i,j}, & \vec{x}_i \in KNN(\vec{x}_j) \text{ or } \vec{x}_j \in KNN(\vec{x}_i) \end{cases}$$

- 只有两个点互为对方的 K 近邻中，则认为是近邻。即：取交集。

$$w_{i,j} = w_{j,i} = \begin{cases} 0, & \vec{x}_i \notin KNN(\vec{x}_j) \text{ or } \vec{x}_j \notin KNN(\vec{x}_i) \\ s_{i,j}, & \vec{x}_i \in KNN(\vec{x}_j) \text{ and } \vec{x}_j \in KNN(\vec{x}_i) \end{cases}$$

5. 全连接法：所有点之间的权重都大于 0 : $w_{i,j} = s_{i,j}$ 。

5.2 拉普拉斯矩阵

1. 定义拉普拉斯矩阵 $\mathbf{L} = \mathbf{D} - \mathbf{W}$ ，其中 \mathbf{D} 为度矩阵、 \mathbf{W} 为邻接矩阵。

2. 拉普拉斯矩阵 \mathbf{L} 的性质：

- \mathbf{L} 是对称矩阵，即 $\mathbf{L} = \mathbf{L}^T$ 。这是因为 \mathbf{D}, \mathbf{W} 都是对称矩阵。
- 因为 \mathbf{L} 是实对称矩阵，因此它的特征值都是实数。
- 对任意向量 $\vec{f} = (f_1, f_2, \dots, f_N)^T$ ，有： $\vec{f}^T \mathbf{L} \vec{f} = \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N w_{i,j} (f_i - f_j)^2$ 。
- \mathbf{L} 是半正定的，且对应的 N 个特征值都大于等于 0，且最小的特征值为 0。
设其 N 个实特征值从小到大为 $\lambda_1, \dots, \lambda_N$ ，即： $0 = \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_N$ 。

5.3 谱聚类算法

1. 给定无向图 $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ ，设子图的点的集合 \mathbb{A} 和子图的点的集合 \mathbb{B} 都是 \mathbb{V} 的子集，且 $\mathbb{A} \cap \mathbb{B} = \emptyset$ 。

定义 \mathbb{A} 和 \mathbb{B} 之间的切图权重为： $W(\mathbb{A}, \mathbb{B}) = \sum_{i \in \mathbb{A}, j \in \mathbb{B}} w_{i,j}$ 。

即：所有连接 \mathbb{A} 和 \mathbb{B} 之间的边的权重。

2. 对于无向图 $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ ，假设将它切分为 k 个子图：每个子图的点的集合为 $\mathbb{A}_1, \dots, \mathbb{A}_k$ ，满足 $\mathbb{A}_i \cap \mathbb{A}_j = \emptyset, i \neq j$ 且 $\mathbb{A}_1 \cup \dots \cup \mathbb{A}_k = \mathbb{V}$ 。

定义切图 `cut` 为： $cut(\mathbb{A}_1, \dots, \mathbb{A}_k) = \sum_{i=1}^k W(\mathbb{A}_i, \bar{\mathbb{A}}_i)$ ，其中 $\bar{\mathbb{A}}_i$ 为 \mathbb{A}_i 的补集。

5.3.1 最小切图

1. 引入指示向量 $\vec{q}_j = (q_{j,1}, \dots, q_{j,N})^T, j = 1, 2, \dots, k$ ，定义：

$$q_{j,i} = \begin{cases} 0, & i \notin \mathbb{A}_j \\ 1, & i \in \mathbb{A}_j \end{cases}$$

则有：

$$\begin{aligned} \vec{q}_j^T \mathbf{L} \vec{q}_j &= \frac{1}{2} \sum_{m=1}^N \sum_{n=1}^N w_{m,n} (q_{j,m} - q_{j,n})^2 \\ &= \frac{1}{2} \sum_{m \in \mathbb{A}_j} \sum_{n \in \mathbb{A}_j} w_{m,n} (1-1)^2 + \frac{1}{2} \sum_{m \notin \mathbb{A}_j} \sum_{n \notin \mathbb{A}_j} w_{m,n} (0-0)^2 + \\ &\quad \frac{1}{2} \sum_{m \in \mathbb{A}_j} \sum_{n \notin \mathbb{A}_j} w_{m,n} (1-0)^2 + \frac{1}{2} \sum_{m \notin \mathbb{A}_j} \sum_{n \in \mathbb{A}_j} w_{m,n} (0-1)^2 \\ &= \frac{1}{2} \left(\sum_{m \in \mathbb{A}_j} \sum_{n \notin \mathbb{A}_j} w_{m,n} + \sum_{m \notin \mathbb{A}_j} \sum_{n \in \mathbb{A}_j} w_{m,n} \right) \\ &= \frac{1}{2} (cut(\mathbb{A}_j, \bar{\mathbb{A}}_j) + cut(\bar{\mathbb{A}}_j, \mathbb{A}_j)) = cut(\mathbb{A}_j, \bar{\mathbb{A}}_j) \end{aligned}$$

因此 $cut(\mathbb{A}_1, \dots, \mathbb{A}_k) = \sum_{j=1}^k \vec{q}_j^T \mathbf{L} \vec{q}_j = tr(\mathbf{Q}^T \mathbf{L} \mathbf{Q})$ 。其中 $\mathbf{Q} = (\vec{q}_1, \dots, \vec{q}_k)$ ， $tr(\cdot)$ 为矩阵的迹。

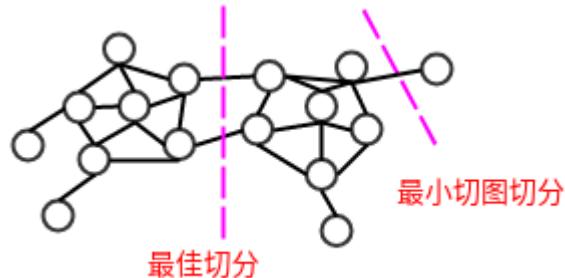
考虑到顶点 i 有且仅位于一个子图中，则有约束条件：

$$q_{j,m} \in \{0, 1\}, \quad \vec{\mathbf{q}}_i \cdot \vec{\mathbf{q}}_j = \begin{cases} 0, & i \neq j \\ |\mathbb{A}|_j, & i = j \end{cases}$$

2. 最小切图算法： $cut(\mathbb{A}_1, \dots, \mathbb{A}_k)$ 最小的切分。即求解：

$$\begin{aligned} & \min_{\mathbf{Q}} \text{tr}(\mathbf{Q}^T \mathbf{L} \mathbf{Q}) \\ & s.t. q_{j,m} \in \{0, 1\}, \quad \vec{\mathbf{q}}_i \cdot \vec{\mathbf{q}}_j = \begin{cases} 0, & i \neq j \\ |\mathbb{A}|_j, & i = j \end{cases} \end{aligned}$$

3. 最小切图切分使得不同子图之间的边的权重尽可能的低，但是容易产生分割出只包含几个顶点的较小子图的歪斜分割现象。



5.3.2 RatioCut 算法

1. **RatioCut** 切图不仅考虑最小化 $cut(\mathbb{A}_1, \dots, \mathbb{A}_k)$ ，它还考虑最大化每个子图的点的个数。即：

$$\text{RatioCut}(\mathbb{A}_1, \dots, \mathbb{A}_k) = \sum_{i=1}^k \frac{W(\mathbb{A}_i, \bar{\mathbb{A}}_i)}{|\mathbb{A}_i|}$$

- 最小化 $cut(\mathbb{A}_1, \dots, \mathbb{A}_k)$ ：使得不同子图之间的边的权重尽可能的低。
- 最大化每个子图的点的个数：使得各子图尽可能的大。

2. 引入指示向量 $\vec{\mathbf{h}}_j = (h_{j,1}, \dots, h_{j,N})^T, j = 1, 2, \dots, k$ ，定义：

$$h_{j,i} = \begin{cases} 0, & i \notin \mathbb{A}_j \\ \frac{1}{\sqrt{|\mathbb{A}_j|}}, & i \in \mathbb{A}_j \end{cases}$$

则有：

$$\begin{aligned} \vec{\mathbf{h}}_j^T \mathbf{L} \vec{\mathbf{h}}_j &= \frac{1}{2} \sum_{m=1}^N \sum_{n=1}^N w_{m,n} (h_{j,m} - h_{j,n})^2 \\ &= \frac{1}{2} \sum_{m \in \mathbb{A}_j} \sum_{n \notin \mathbb{A}_j} w_{m,n} \left(\frac{1}{\sqrt{|\mathbb{A}_j|}} - 0 \right)^2 + \frac{1}{2} \sum_{m \notin \mathbb{A}_j} \sum_{n \in \mathbb{A}_j} w_{m,n} \left(0 - \frac{1}{\sqrt{|\mathbb{A}_j|}} \right)^2 \\ &= \frac{1}{2} \left(\sum_{m \in \mathbb{A}_j} \sum_{n \notin \mathbb{A}_j} \frac{w_{m,n}}{|\mathbb{A}_j|} + \sum_{m \notin \mathbb{A}_j} \sum_{n \in \mathbb{A}_j} \frac{w_{m,n}}{|\mathbb{A}_j|} \right) \\ &= \frac{1}{2} \times \frac{1}{|\mathbb{A}_j|} (cut(\mathbb{A}_j, \bar{\mathbb{A}}_j) + cut(\bar{\mathbb{A}}_j, \mathbb{A}_j)) = \text{RatioCut}(\mathbb{A}_j, \bar{\mathbb{A}}_j) \end{aligned}$$

因此 $\text{RatioCut}(\mathbb{A}_1, \dots, \mathbb{A}_k) = \sum_{j=1}^k \vec{\mathbf{h}}_j^T \mathbf{L} \vec{\mathbf{h}}_j = \text{tr}(\mathbf{H}^T \mathbf{L} \mathbf{H})$ 。其中 $\mathbf{H} = (\vec{\mathbf{h}}_1, \dots, \vec{\mathbf{h}}_k)$ ， $\text{tr}(\cdot)$ 为矩阵的迹。

考虑到顶点 i 有且仅位于一个子图中，则有约束条件：

$$\vec{\mathbf{h}}_i \cdot \vec{\mathbf{h}}_j = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$$

3. **RatioCut 算法**： $\text{RatioCut}(\mathbb{A}_1, \dots, \mathbb{A}_k)$ 最小的切分。即求解：

$$\begin{aligned} & \min_{\mathbf{H}} \text{tr}(\mathbf{H}^T \mathbf{L} \mathbf{H}) \\ & s.t. \mathbf{H}^T \mathbf{H} = \mathbf{I} \end{aligned}$$

因此只需要求解 \mathbf{L} 最小的 k 个特征值，求得对应的 k 个特征向量组成 \mathbf{H} 。

通常在求解得到 \mathbf{H} 之后，还需要对行进行标准化： $h_{i,j}^* = \frac{h_{i,j}}{\sqrt{\sum_{t=1}^k h_{i,t}^2}}$

4. 事实上这样解得的 \mathbf{H} 不能完全满足指示向量的定义。因此在得到 \mathbf{H} 之后，还需要对每一行进行一次传统的聚类（如：**k-means** 聚类）。

5. **RatioCut 算法**：

◦ 输入：

- 数据集 $\mathbb{D} = \{\vec{x}_1, \dots, \vec{x}_N\}$
- 降维的维度 k_1
- 二次聚类算法
- 二次聚类的维度 k_2

◦ 输出：聚类簇 $\mathcal{C} = \{\mathbb{C}_1, \dots, \mathbb{C}_{k_2}\}$

◦ 算法步骤：

- 根据 \mathbb{D} 构建相似度矩阵 \mathbf{S} 。
- 根据相似度矩阵构建邻接矩阵 \mathbf{W} 、度矩阵 \mathbf{D} ，计算拉普拉斯矩阵 $\mathbf{L} = \mathbf{D} - \mathbf{W}$ 。
- 计算 \mathbf{L} 最小的 k_1 个特征值，以及对应的特征向量 $\vec{v}_1, \dots, \vec{v}_{k_1}$ ，构建矩阵 $\mathbf{H} = (\vec{v}_1, \dots, \vec{v}_{k_1})$ 。
- 对 \mathbf{H} 按照行进行标准化： $h_{i,j}^* = \frac{h_{i,j}}{\sqrt{\sum_{t=1}^k h_{i,t}^2}}$ ，得到 \mathbf{H}^* 。
- 将 \mathbf{H}^* 中每一行作为一个 k_1 维的样本，一共 N 个样本，利用二次聚类算法来聚类，二次聚类的维度为 k_2 。

最终得到簇划分 $\mathcal{C} = \{\mathbb{C}_1, \dots, \mathbb{C}_{k_2}\}$ 。

5.3.3 Ncut 算法

1. **Ncut** 切图不仅考虑最小化 $\text{cut}(\mathbb{A}_1, \dots, \mathbb{A}_k)$ ，它还考虑最大化每个子图的边的权重。即：

$$\text{Ncut}(\mathbb{A}_1, \dots, \mathbb{A}_k) = \sum_{i=1}^k \frac{W(\mathbb{A}_i, \bar{\mathbb{A}}_i)}{\text{vol}(\mathbb{A}_i)}$$

- 最小化 $\text{cut}(\mathbb{A}_1, \dots, \mathbb{A}_k)$ ：使得不同子图之间的边的权重尽可能的低。
- 最大化每个子图的边的权重：使得各子图内的边的权重尽可能的高。

2. 引入指示向量 $\vec{\mathbf{h}}_j = (h_{j,1}, \dots, h_{j,N})^T, j = 1, 2, \dots, k$ ，定义：

$$h_{j,i} = \begin{cases} 0, & i \notin \mathbb{A}_j \\ \frac{1}{\sqrt{\text{vol}(\mathbb{A}_j)}}, & i \in \mathbb{A}_j \end{cases}$$

则有：

$$\begin{aligned}
\vec{\mathbf{h}}_j^T \mathbf{L} \vec{\mathbf{h}}_j &= \frac{1}{2} \sum_{m=1}^N \sum_{n=1}^N w_{m,n} (h_{j,m} - h_{j,n})^2 \\
&= \frac{1}{2} \sum_{m \in \mathbb{A}_j} \sum_{n \notin \mathbb{A}_j} w_{m,n} \left(\frac{1}{\sqrt{vol(\mathbb{A}_j)}} - 0 \right)^2 + \frac{1}{2} \sum_{m \notin \mathbb{A}_j} \sum_{n \in \mathbb{A}_j} w_{m,n} \left(0 - \frac{1}{\sqrt{vol(\mathbb{A}_j)}} \right)^2 \\
&= \frac{1}{2} \left(\sum_{m \in \mathbb{A}_j} \sum_{n \notin \mathbb{A}_j} \frac{w_{m,n}}{vol(\mathbb{A}_j)} + \sum_{m \notin \mathbb{A}_j} \sum_{n \in \mathbb{A}_j} \frac{w_{m,n}}{vol(\mathbb{A}_j)} \right) \\
&= \frac{1}{2} \times \frac{1}{vol(\mathbb{A}_j)} (cut(\mathbb{A}_j, \bar{\mathbb{A}}_j) + cut(\bar{\mathbb{A}}_j, \mathbb{A}_j)) = Ncut(\mathbb{A}_j, \bar{\mathbb{A}}_j)
\end{aligned}$$

因此 $Ncut(\mathbb{A}_1, \dots, \mathbb{A}_k) = \sum_{j=1}^k \vec{\mathbf{h}}_j^T \mathbf{L} \vec{\mathbf{h}}_j = tr(\mathbf{H}^T \mathbf{L} \mathbf{H})$ 。其中 $\mathbf{H} = (\vec{\mathbf{h}}_1, \dots, \vec{\mathbf{h}}_k)$ ， $tr(\cdot)$ 为矩阵的迹。

考虑到顶点 i 有且仅位于一个子图中，则有约束条件：

$$\vec{\mathbf{h}}_i \cdot \vec{\mathbf{h}}_j = \begin{cases} 0, & i \neq j \\ \frac{1}{vol(\mathbb{A}_j)}, & i = j \end{cases}$$

3. **Ncut** 算法： $Ncut(\mathbb{A}_1, \dots, \mathbb{A}_k)$ 最小的切分。即求解

$$\begin{aligned}
&\min_{\mathbf{H}} tr(\mathbf{H}^T \mathbf{L} \mathbf{H}) \\
&s.t. \mathbf{H}^T \mathbf{D} \mathbf{H} = \mathbf{I}
\end{aligned}$$

4. 令 $\mathbf{H} = \mathbf{D}^{-1/2} \mathbf{F}$ ，则有：

$$\begin{aligned}
\mathbf{H}^T \mathbf{L} \mathbf{H} &= \mathbf{F}^T \mathbf{D}^{-1/2} \mathbf{L} \mathbf{D}^{-1/2} \mathbf{F} \\
\mathbf{H}^T \mathbf{D} \mathbf{H} &= \mathbf{F}^T \mathbf{F} = \mathbf{I}
\end{aligned}$$

令 $\mathbf{L}' = \mathbf{D}^{-1/2} \mathbf{L} \mathbf{D}^{-1/2}$ ，则最优化目标变成：

$$\begin{aligned}
&\min_{\mathbf{H}} tr(\mathbf{F}^T \mathbf{L}' \mathbf{F}) \\
&s.t. \mathbf{F}^T \mathbf{F} = \mathbf{I}
\end{aligned}$$

因此只需要求解 \mathbf{L}' 最小的 k 个特征值，求得对应的 k 个特征向量组成 \mathbf{F} 。然后对行进行标准化：

$$f_{i,j}^* = \frac{f_{i,j}}{\sqrt{\sum_{t=1}^k f_{i,t}^2}}。$$

与 **RatioCut** 类似，**Ncut** 也需要对 \mathbf{F} 的每一行进行一次传统的聚类（如：**k-means** 聚类）。

5. 事实上 $\mathbf{D}^{-1/2} \mathbf{L} \mathbf{D}^{-1/2}$ 相当于对拉普拉斯矩阵 \mathbf{L} 进行了一次标准化： $l'_{i,j} = \frac{l_{i,j}}{d_i \times d_j}$ 。

6. **Ncut** 算法：

- 输入：
 - 数据集 $\mathbb{D} = \{\vec{\mathbf{x}}_1, \dots, \vec{\mathbf{x}}_N\}$
 - 降维的维度 k_1
 - 二次聚类算法
 - 二次聚类的维度 k_2
- 输出：聚类簇 $\mathcal{C} = \{\mathbb{C}_1, \dots, \mathbb{C}_{k_2}\}$
- 算法步骤：
 - 根据 \mathbb{D} 构建相似度矩阵 \mathbf{S} 。
 - 根据相似度矩阵构建邻接矩阵 \mathbf{W} 、度矩阵 \mathbf{D} ，计算拉普拉斯矩阵 $\mathbf{L} = \mathbf{D} - \mathbf{W}$ 。

- 构建标准化之后的拉普拉斯矩阵 $\mathbf{L}' = \mathbf{D}^{-1/2} \mathbf{L} \mathbf{D}^{-1/2}$ 。
 - 计算 \mathbf{L}' 最小的 k_1 个特征值，以及对应的特征向量 $\vec{\mathbf{v}}_1, \dots, \vec{\mathbf{v}}_{k_1}$ ，构建矩阵 $\mathbf{F} = (\vec{\mathbf{v}}_1, \dots, \vec{\mathbf{v}}_{k_1})$ 。
 - 对 \mathbf{F} 按照行进行标准化： $f_{i,j}^* = \frac{f_{i,j}}{\sqrt{\sum_{t=1}^k f_{i,t}^2}}$ ，得到 \mathbf{F}^* 。
 - 将 \mathbf{F}^* 中每一行作为一个 k_1 维的样本，一共 N 个样本，利用二次聚类算法来聚类，二次聚类的维度为 k_2 。
- 最终得到簇划分 $\mathcal{C} = \{\mathbb{C}_1, \dots, \mathbb{C}_{k_2}\}$ 。

5.4 性质

1. 谱聚类算法优点：

- 只需要数据之间的相似度矩阵，因此处理稀疏数据时很有效。
- 由于使用了降维，因此在处理高维数据聚类时效果较好。

2. 谱聚类算法缺点：

- 如果最终聚类的维度非常高，则由于降维的幅度不够，则谱聚类的运行速度和最后聚类的效果均不好。
- 聚类效果依赖于相似度矩阵，不同相似度矩阵得到的最终聚类效果可能不同。