文本聚类

本讲内容

- 聚类的概念(What is clustering?)
- 聚类在IR中的应用
- *K*-均值(*K-Means*)聚类算法
- 聚类评价
- 簇(cluster)个数(即聚类的结果类别个数)确定
- 层次聚类

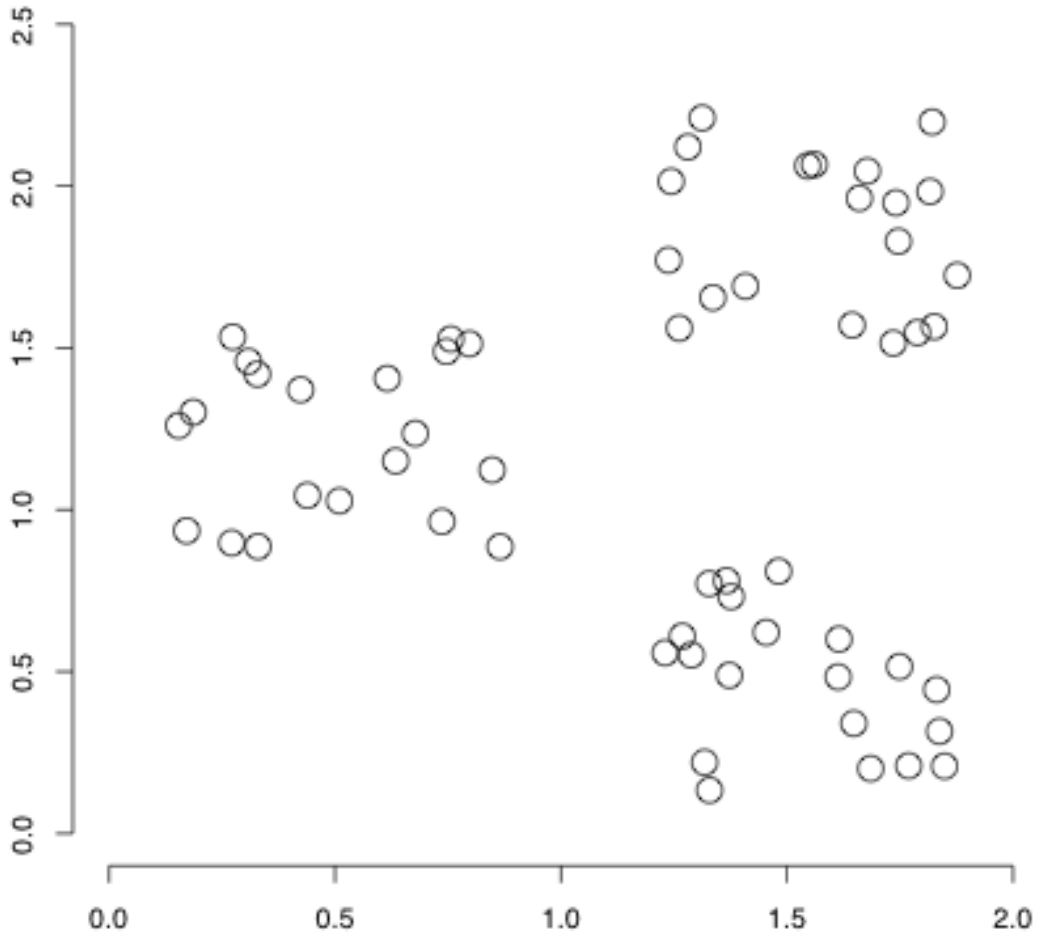
提纲

- 聚类介绍
- 聚类在IR中的应用
- K-均值聚类算法
- 聚类评价
- 簇个数确定
- 层次聚类

聚类(Clustering)的定义

- (文档)聚类是将一系列文档按照相似性聚团成子集或者簇(cluster)的过程
- 簇内文档之间应该彼此相似
- 簇间文档之间相似度不大
- 聚类是一种最常见的无监督学习(unsupervised learning)方法
  - 无监督意味着没有已标注好的数据集

一个具有清晰簇结构的数据集



提出一个算法来寻找该例中的簇结构

分类 vs. 聚类

- 分类：有监督的学习
- 聚类：无监督的学习
- 分类：类别事先人工定义好，并且是学习算法的输入的一部分
- 聚类：簇在没有人工输入的情况下从数据中推理而得
  - 但是，很多因素会影响聚类的输出结果：簇的个数、相似度计算方法、文档的表示方式，等等

提纲

- 聚类介绍
- 聚类在IR中的应用
- K-均值聚类算法
- 聚类评价
- 簇个数确定
- 层次聚类

聚类假设

聚类假设：在考虑文档和信息需求之间的相关性时，同一簇中的文档表现互相类似。

聚类在IR中的所有应用都直接或间接基于上述聚类假设

Van Rijsbergen的原始定义: “closely associated documents tend to be relevant to the same requests” （彼此密切关联的文档和同一信息需求相关）

聚类在IR中的应用

应　　用	聚类对象	优　　点
搜索结果聚类	搜索结果	提供面向用户的更有效的展示
“分散—集中”界面	文档集和文档子集	提供另一种用户界面，即不需要人工输入关键词的搜索界面
文档集聚类	文档集	提供一种面向探索式浏览的有效的信息展示方法
基于语言建模的IR文档集	文档集	提高正确率和/或召回率
基于聚类的检索	文档集	加快搜索速度

文档聚类用于提高召回率

- 为提高搜索召回率
  - 可以实现将文档集中的文档进行聚类
  - 当文档*d*和查询匹配时，也返回包含*d*的簇所包含的其它文档
  - 希望通过上述做法，在输入查询“car“时，也能够返回包含“automobile”的文档
  - 由于聚类算法会把包含 “car”的文档和包含 “automobile”的文档聚在一起
  - 两种文档都包含诸如 “parts”、 “dealer”、 “mercedes”和“road trip”之类的词语

聚类的要求

- 一般目标：将相关文档放到一个簇中，将不相关文档放到不同簇中
  - 如何对上述目标进行形式化？
- 簇的数目应该合适，以便与聚类的数据集相吻合
  - 一开始，假设给定簇的数目为K。
  - 后面会介绍确定K的半自动方法
- 聚类的其它目标
  - 避免非常小和非常大的簇
  - 定义的簇对用户来说很容易理解
  - 其它……

扁平聚类 vs. 层次聚类

- 扁平算法
  - 通过一开始将全部或部分文档随机划分为不同的组
  - 通过迭代方式不断修正
  - 代表算法：K-均值聚类算法
- 层次算法
  - 构建具有层次结构的簇
  - 自底向上(Bottom-up)的算法称为凝聚式(agglomerative)算法
  - 自顶向下(Top-down)的算法称为分裂式(divisive)算法

硬聚类 vs. 软聚类

- 硬聚类(Hard clustering): 每篇文档仅仅属于一个簇
  - 很普遍并且相对容易实现
- 软聚类(Soft clustering): 一篇文档可以属于多个簇
  - 对于诸如浏览目录之类的应用来说很有意义
  - 比如，将 胶底运动鞋 (sneakers) 放到两个簇中：
    - 体育服装(sports apparel)
    - 鞋类(shoes)
  - 只有通过软聚类才能做到这一点
- 本节课关注扁平的硬聚类算法

扁平算法

扁平算法：将N篇文档划分成K个簇

- 给定一个文档集合及聚类结果簇的个数K
- 寻找一个划分将这个文档集合分成K个簇，该结果满足某个最优划分准则

- 全局优化：穷举所有的划分结果，从中选择最优的那个划分结果
  - 无法处理
- 高效的启发式方法: K-均值聚类算法

提纲

- 聚类介绍
- 聚类在IR中的应用
- *K*-均值聚类算法
- 聚类评价
- 簇个数确定
- 层次聚类

K-均值聚类算法

- 或许是最著名的聚类算法
- 算法十分简单，但是在很多情况下效果不错
- 是文档聚类的默认或基准算法

聚类中的文档表示

- 向量空间模型
- 同基于向量空间的分类一样，采用欧氏距离的方法来计算向量之间的相关性. . .
- 欧氏距离与余弦相似度差不多等价(如果两个向量都基于长度归一化，那么欧氏距离和余弦相似度是等价的)
- 然而，质心向量通常都没有基于长度进行归一化

K-均值聚类算法

- *K*-均值聚类算法中的每个簇都定义为其质心向量
- 划分准则：使得所有文档到其所在簇的质心向量的平方和最小
- 质心向量的定义

$$\vec{\mu}(\omega) = \frac{1}{|\omega|} \sum_{\vec{x} \in \omega} \vec{x}$$

其中 ω 代表一个簇

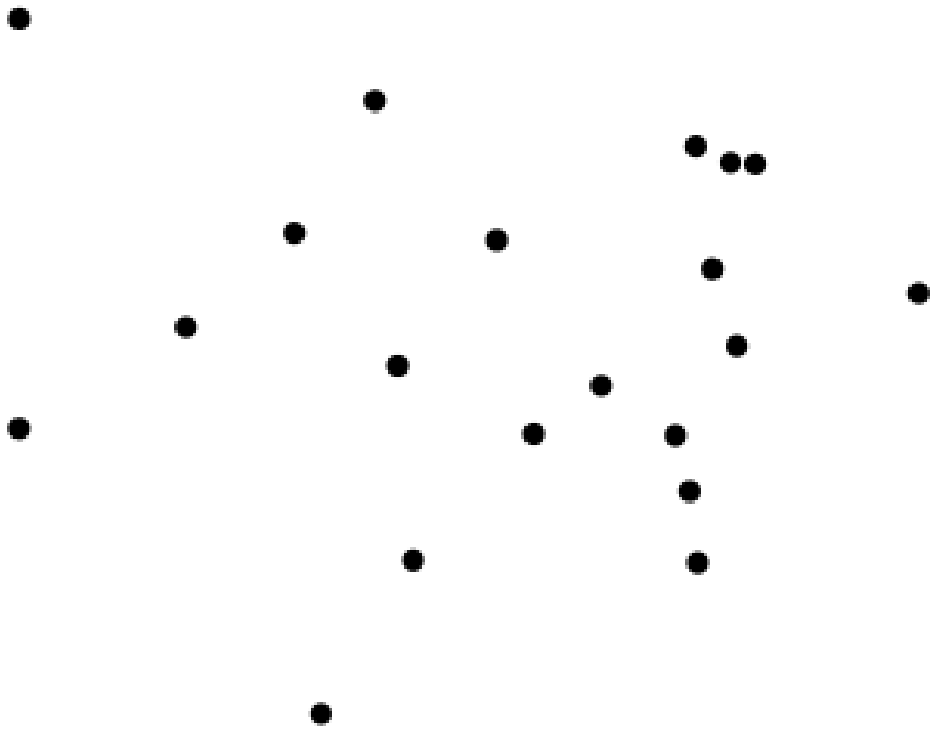
- 通过下列两步来实现目标优化
  - 重分配(reassignment): 将每篇文档分配给离它最近的簇
  - 重计算(recomputation): 重新计算每个簇的质心向量

K-均值聚类算法

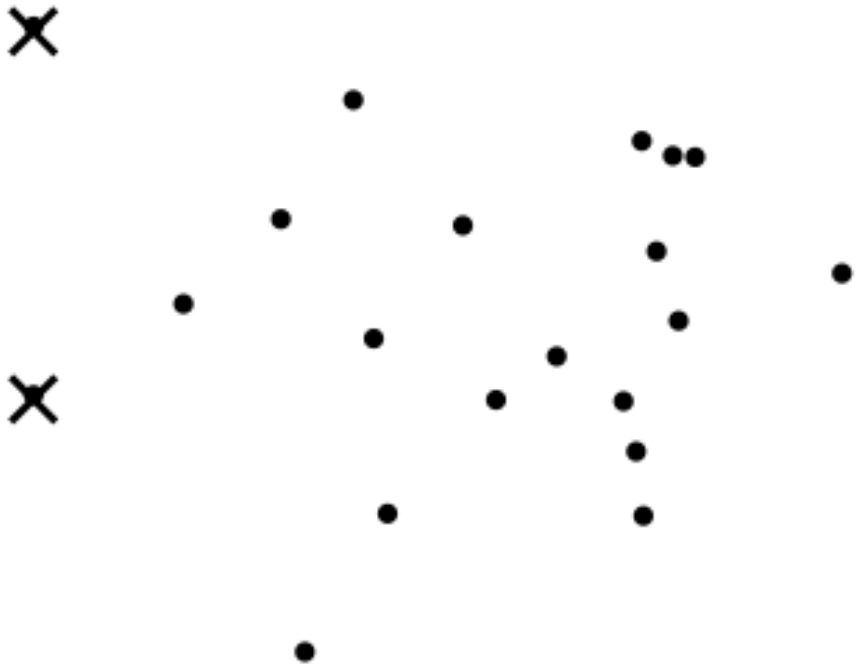
K-MEANS$(\{\vec{x}_1, \ldots, \vec{x}_N\}, K)$

1 $(\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_K) \leftarrow$ SELECTRANDOMSEEDS$(\{\vec{x}_1, \ldots, \vec{x}_N\}, K)$
2 **for** $k \leftarrow 1$ **to** K
3 **do** $\vec{\mu}_k \leftarrow \vec{s}_k$
4 **while** stopping criterion has not been met
5 **do for** $k \leftarrow 1$ **to** K
6 **do** $\omega_k \leftarrow \{\}$
7 **for** $n \leftarrow 1$ **to** N
8 **do** $j \leftarrow \arg\min_{j'} |\vec{\mu}_{j'} - \vec{x}_n|$
9 $\omega_j \leftarrow \omega_j \cup \{\vec{x}_n\}$ *(reassignment of vectors)*
10 **for** $k \leftarrow 1$ **to** K
11 **do** $\vec{\mu}_k \leftarrow \frac{1}{|\omega_k|} \sum_{\vec{x} \in \omega_k} \vec{x}$ *(recomputation of centroids)*
12 **return** $\{\vec{\mu}_1, \ldots, \vec{\mu}_K\}$

例子



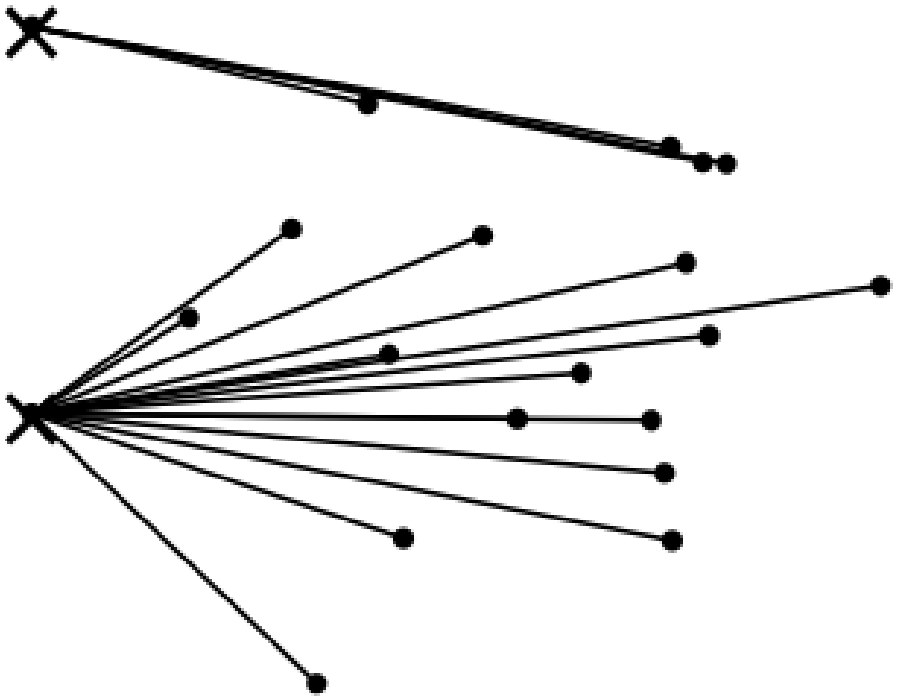
例子：随机选择两个种子(K=2)



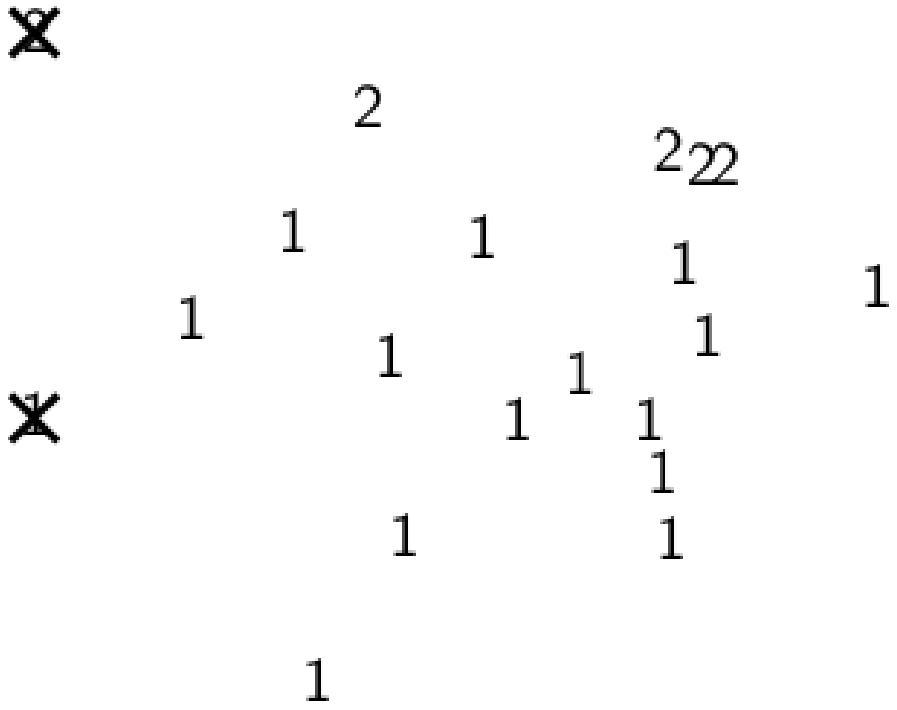
(i) 猜猜最后划分的两个簇是什么？

(ii) 计算簇的质心向量

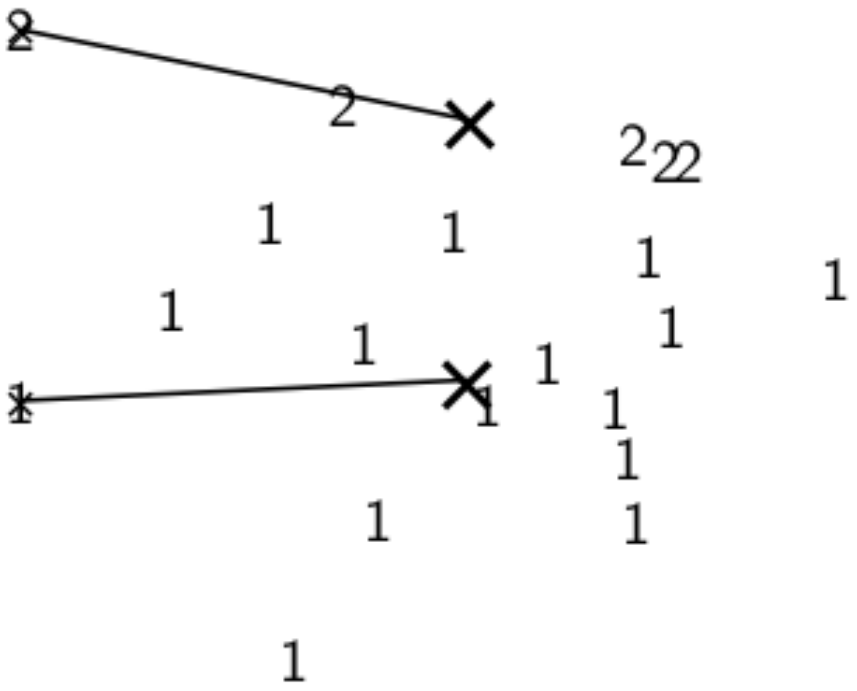
例子：将文档分配给离它最近的质心向量(第一次)



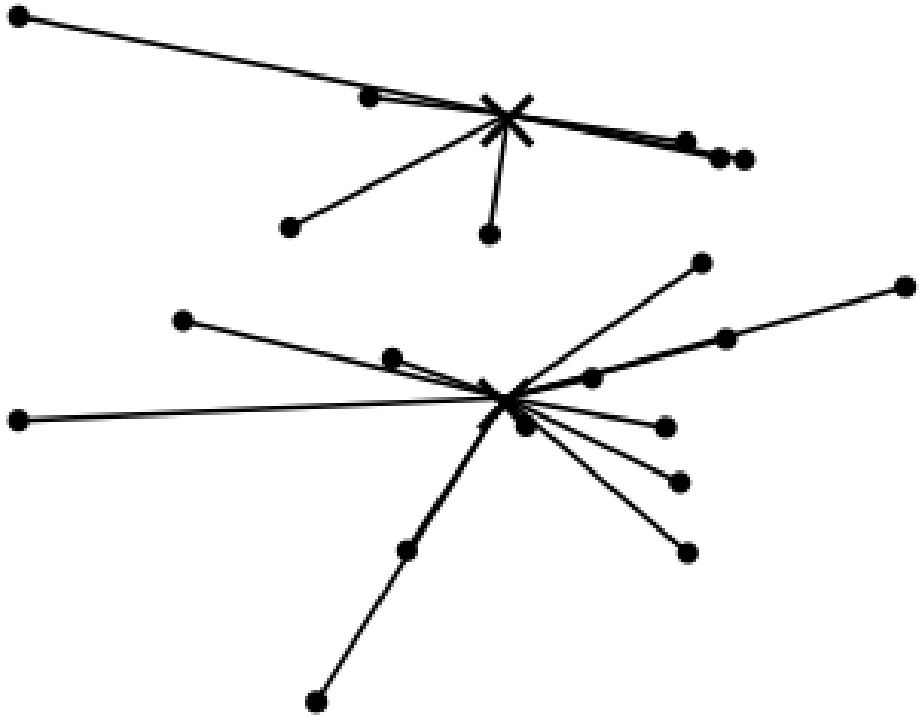
例子：分配后的簇(第一次)



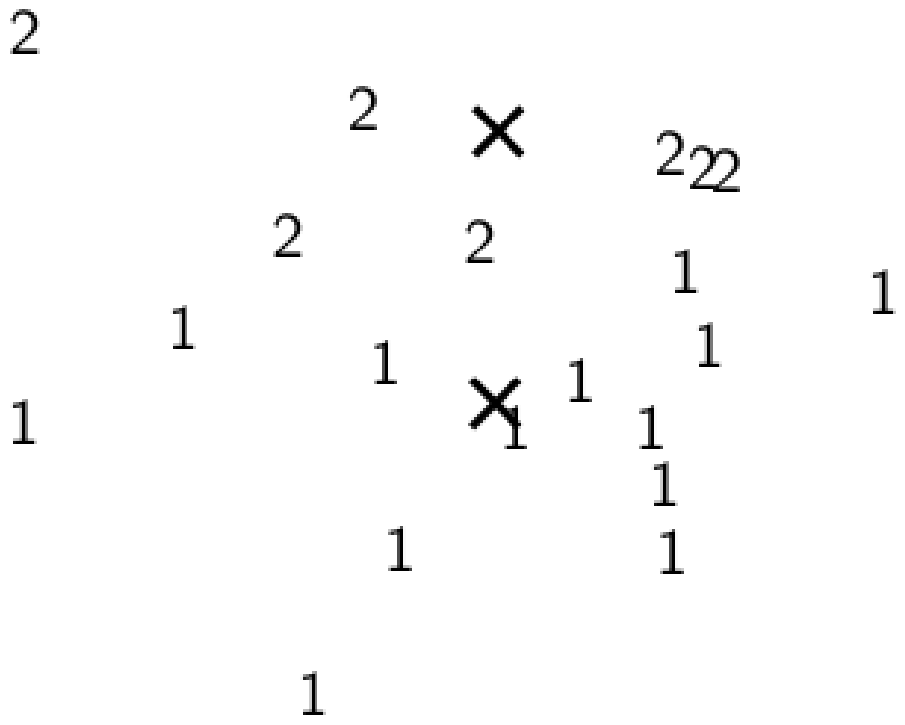
例子：重新计算质心向量



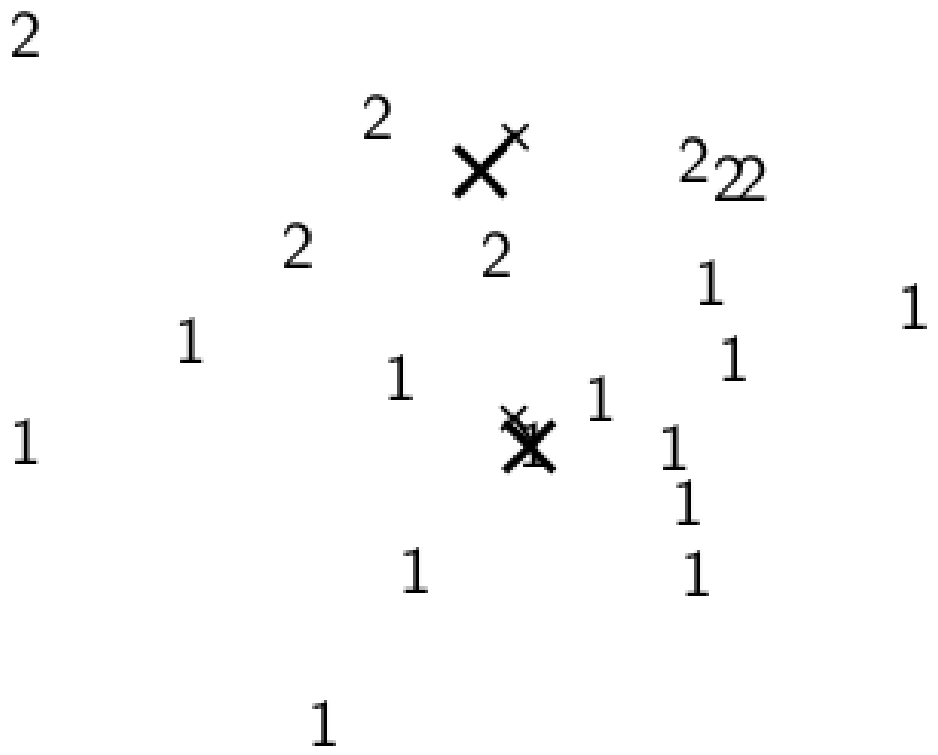
例子：将文档分配给离它最近的质心向量(第二次)



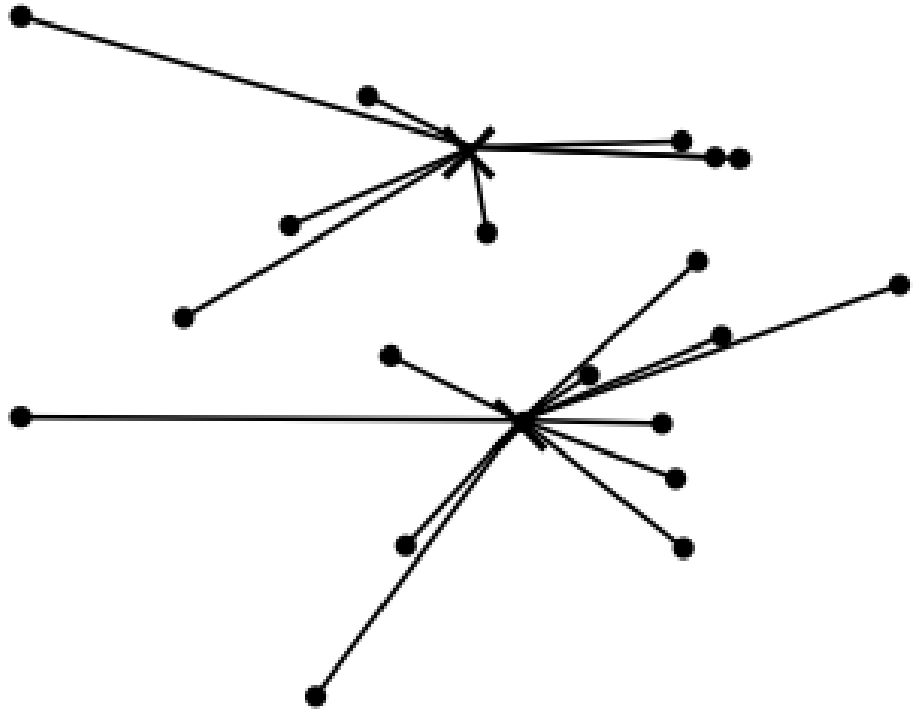
例子：重新分配的结果



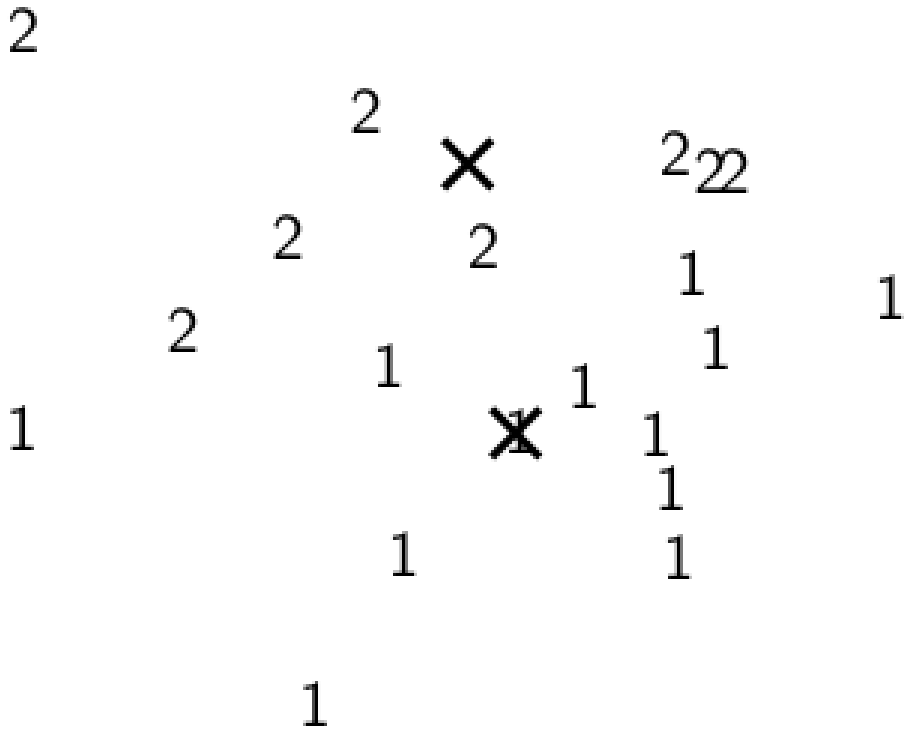
例子：重新计算质心向量



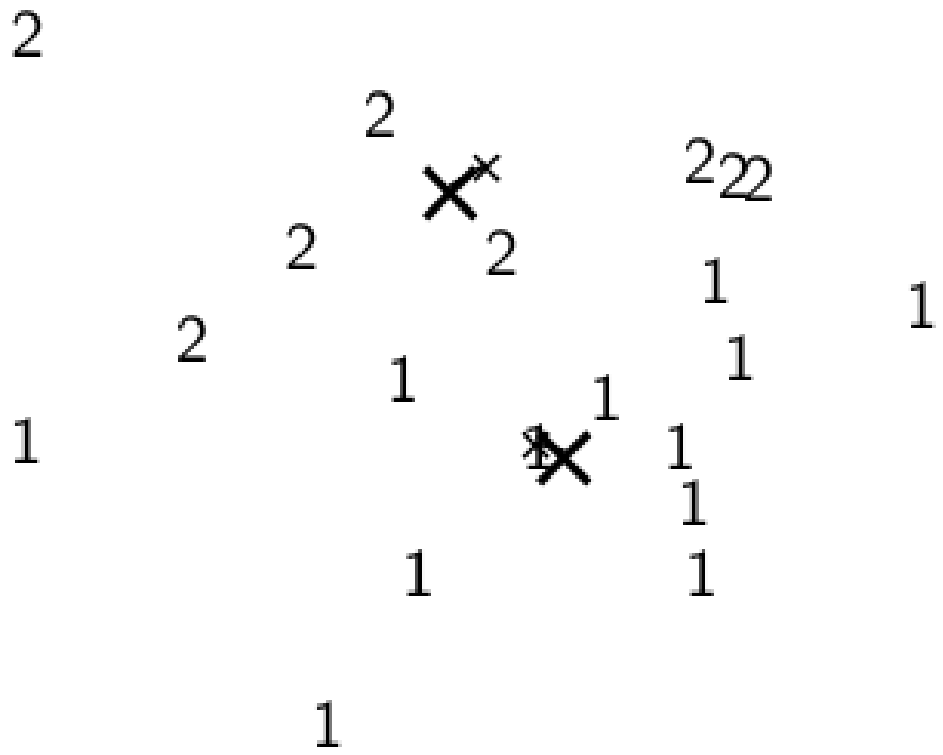
例子：再重新分配(第三次)



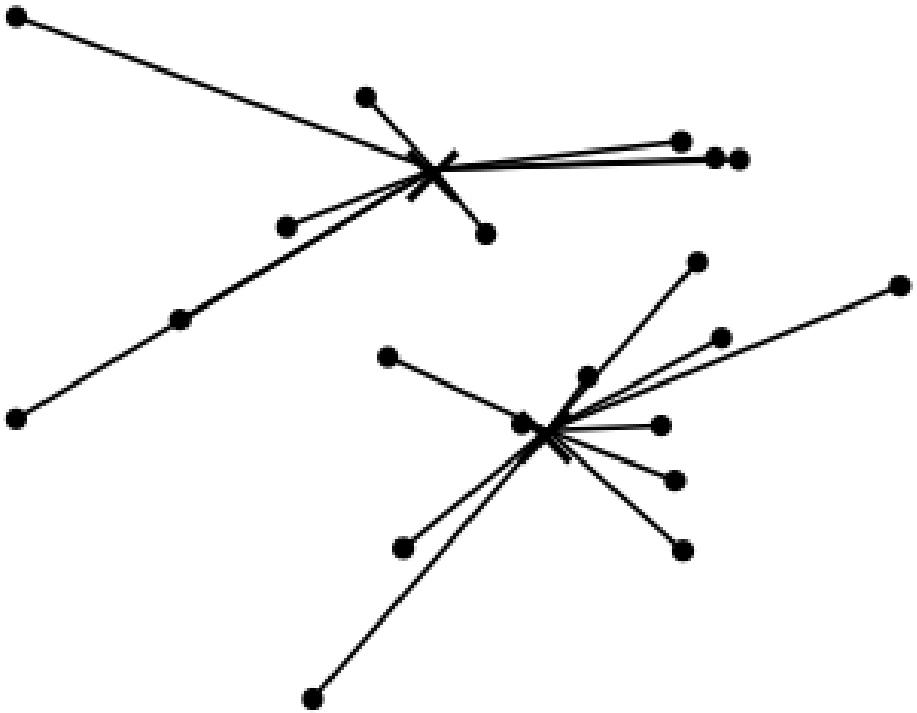
例子：分配结果



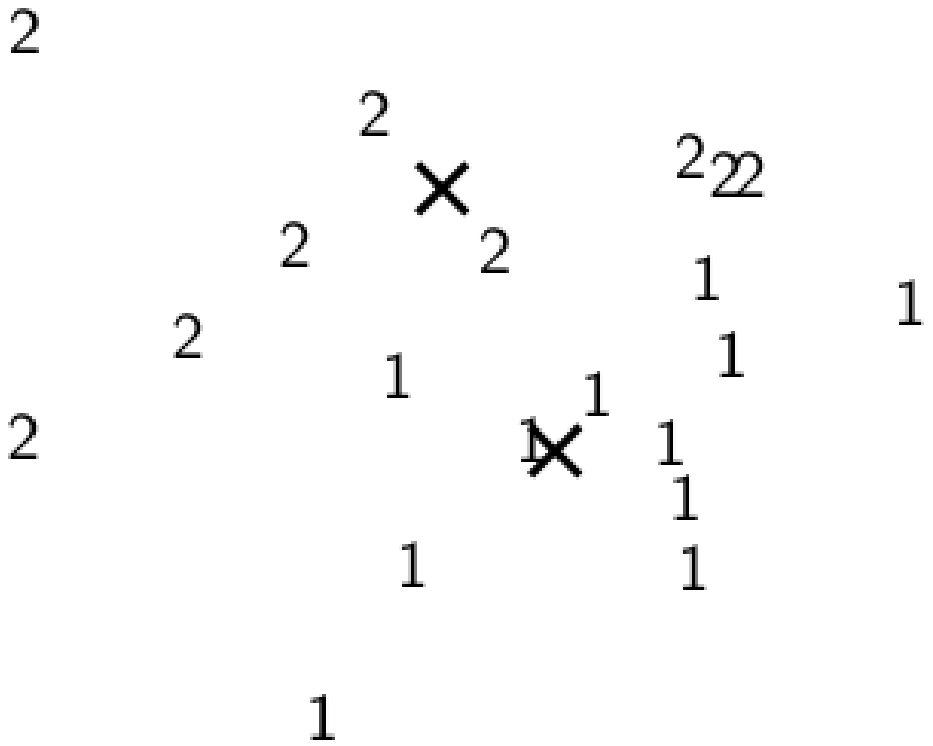
例子：重新计算质心向量



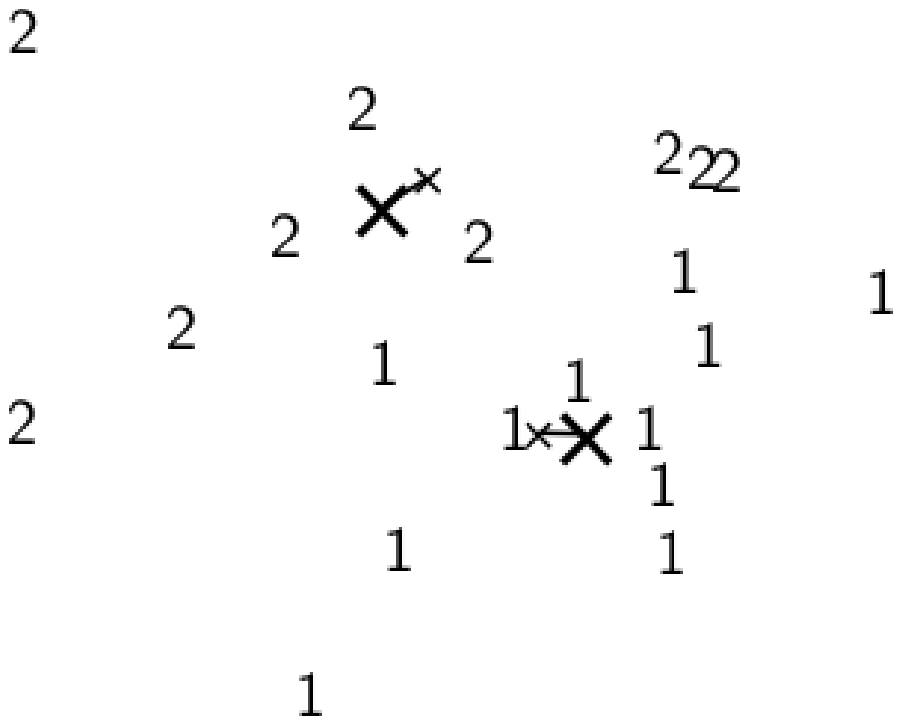
例子：再重新分配(第四次)



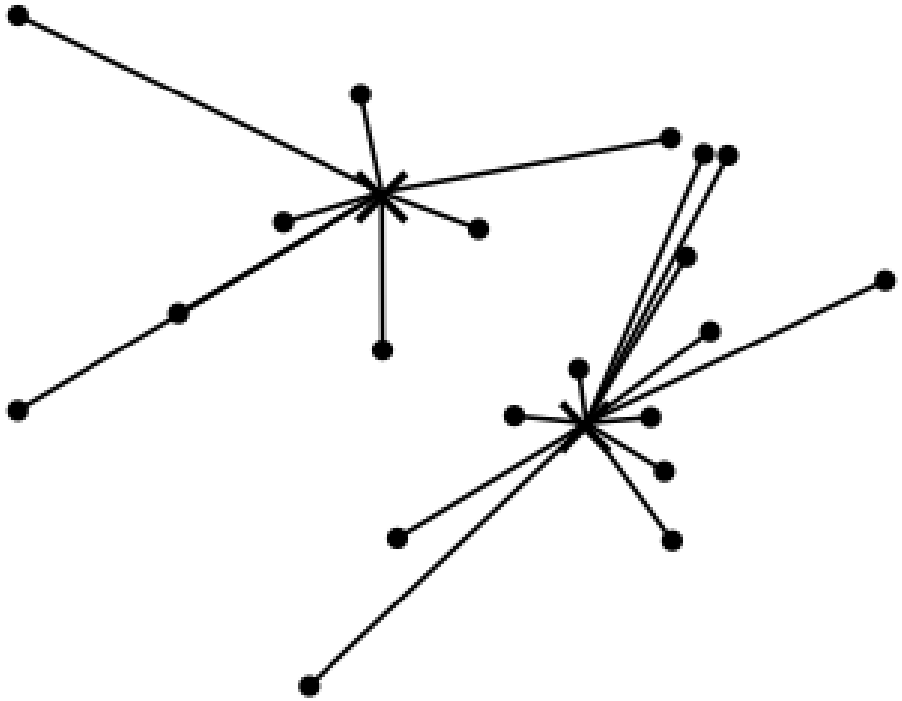
例子：分配结果



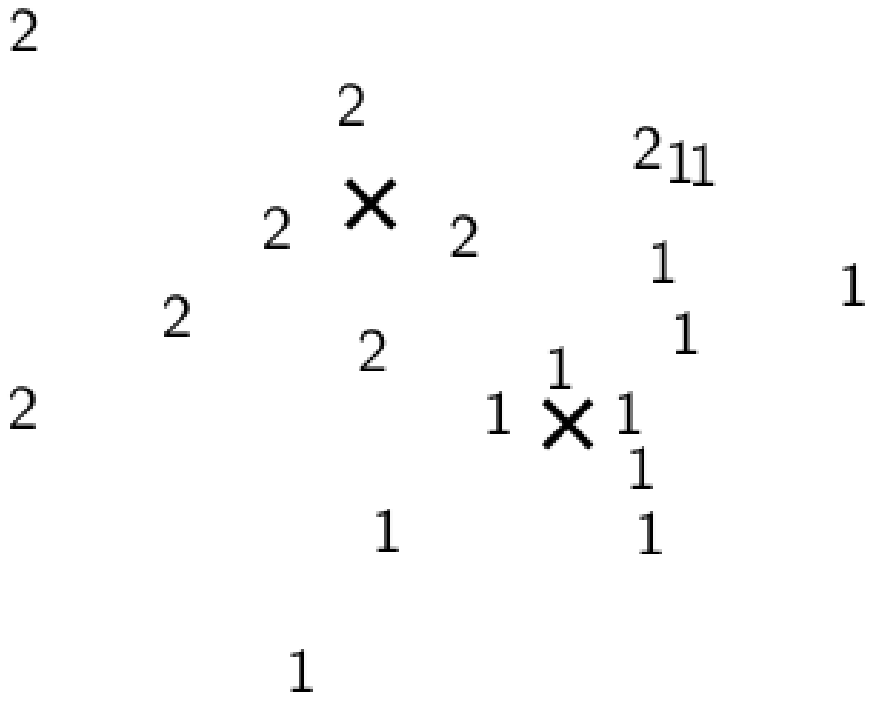
例子：重新计算质心向量



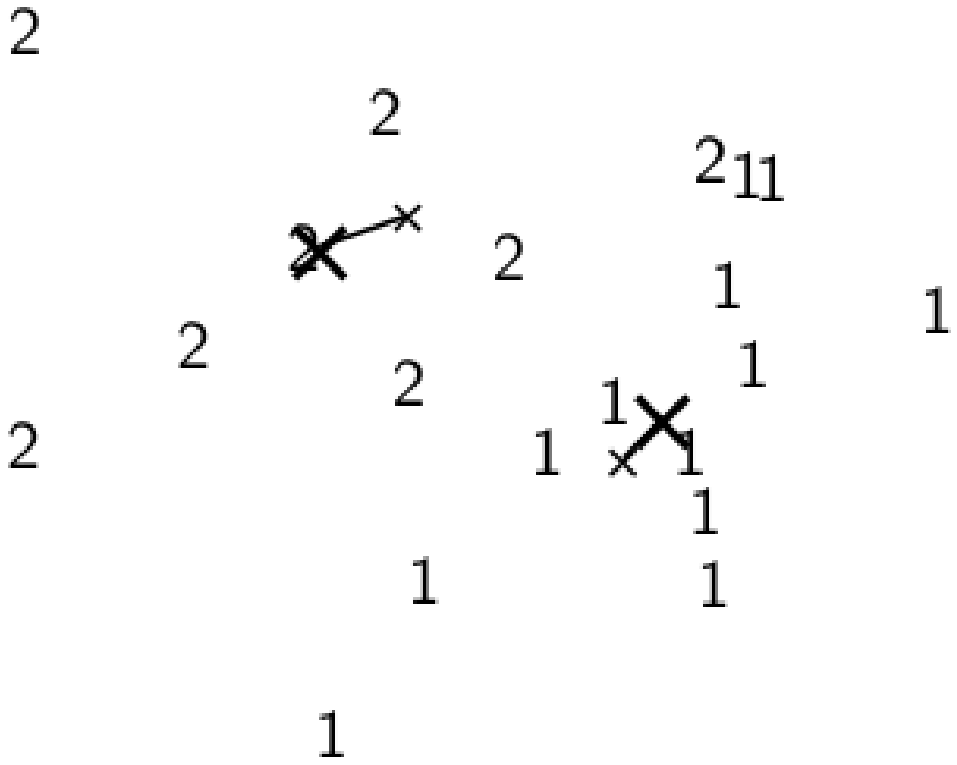
例子：重新分配(第五次)



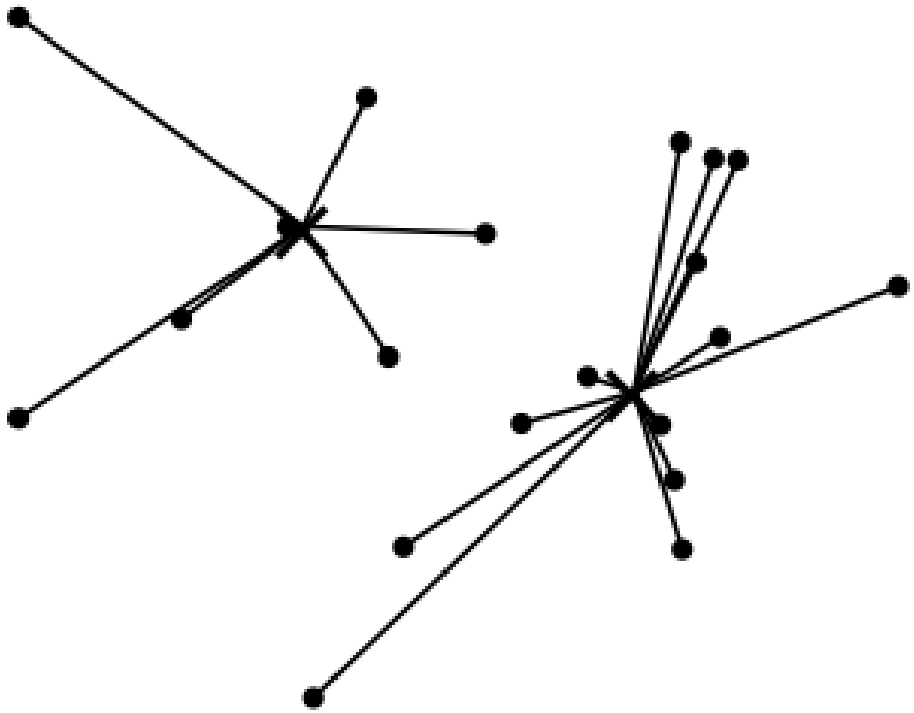
例子：分配结果



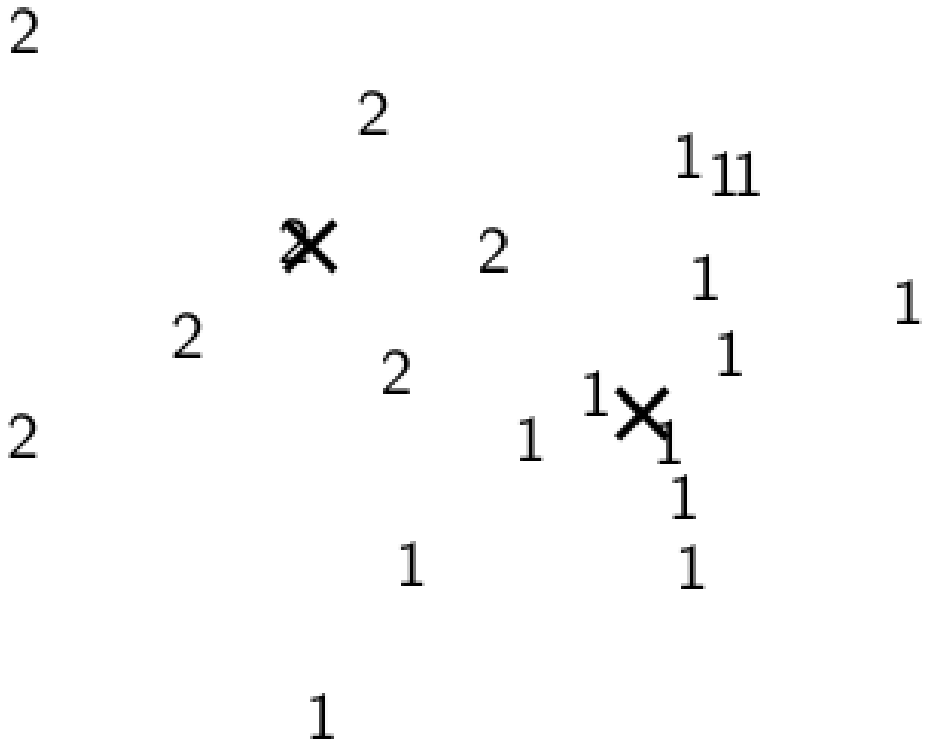
例子：重新计算质心向量



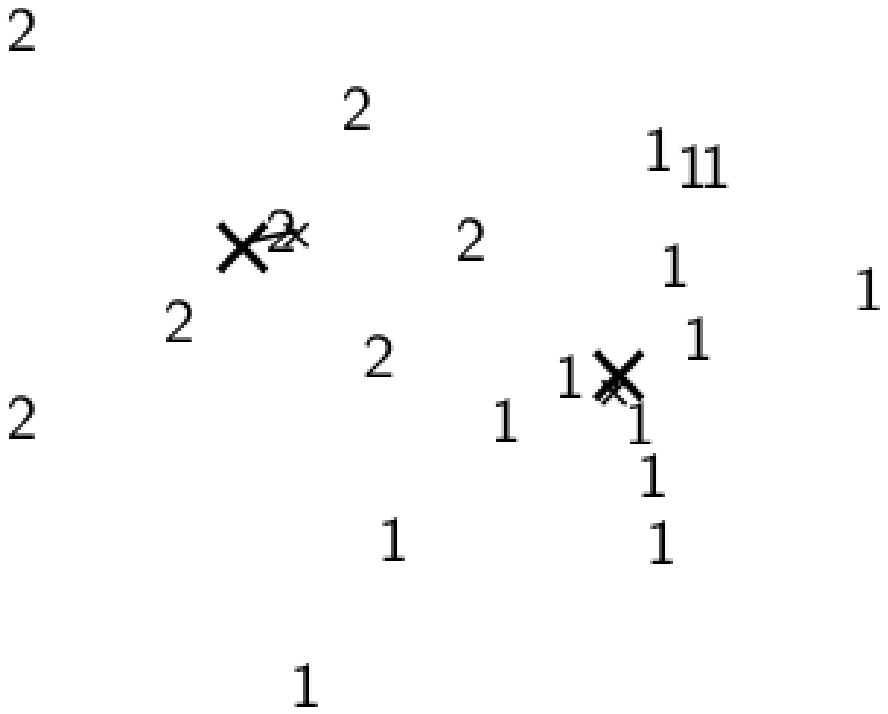
例子：重新分配(第六次)



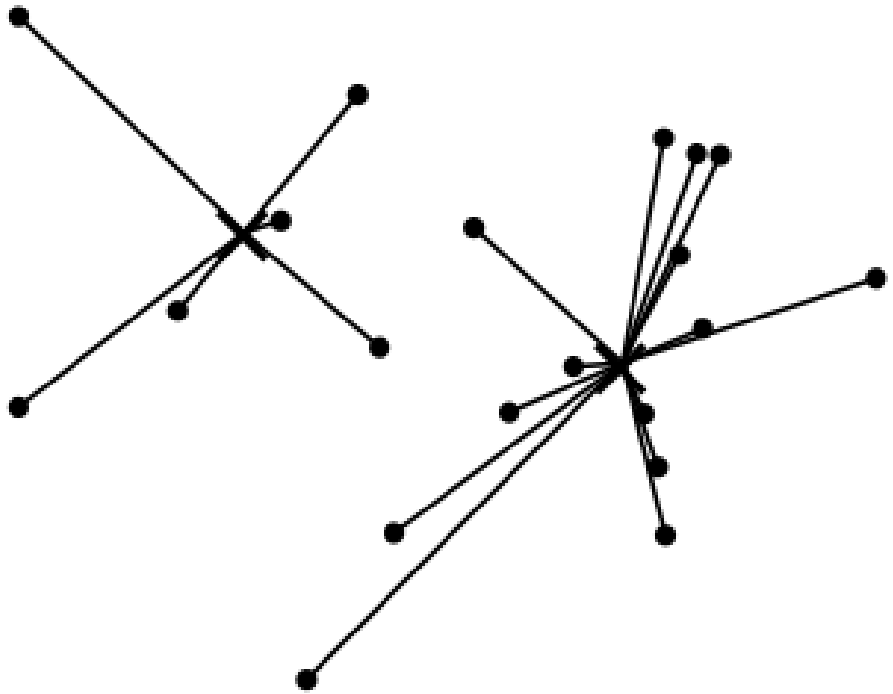
例子：分配结果



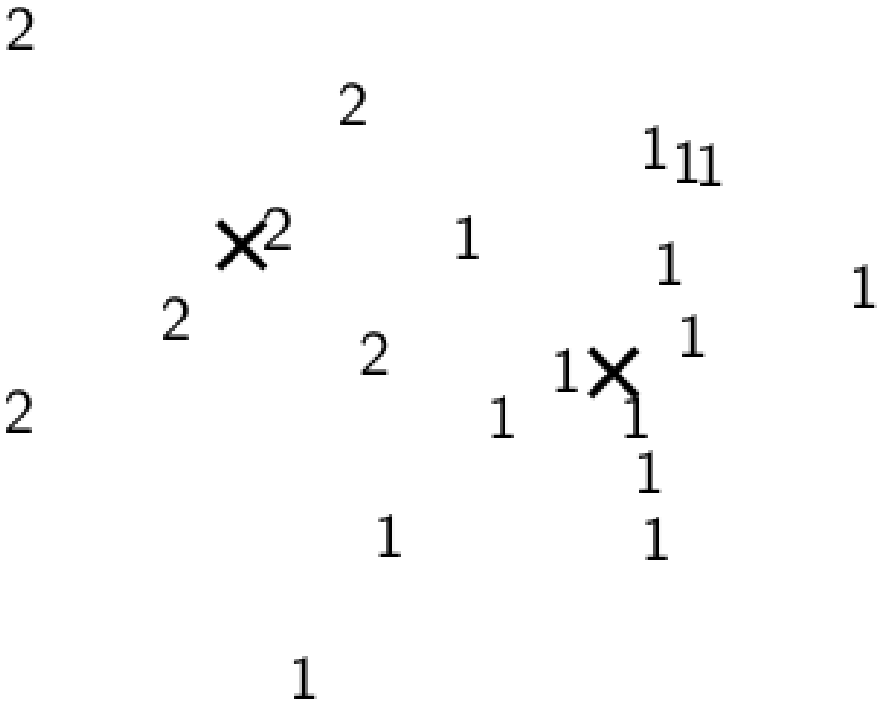
例子：重新计算质心向量



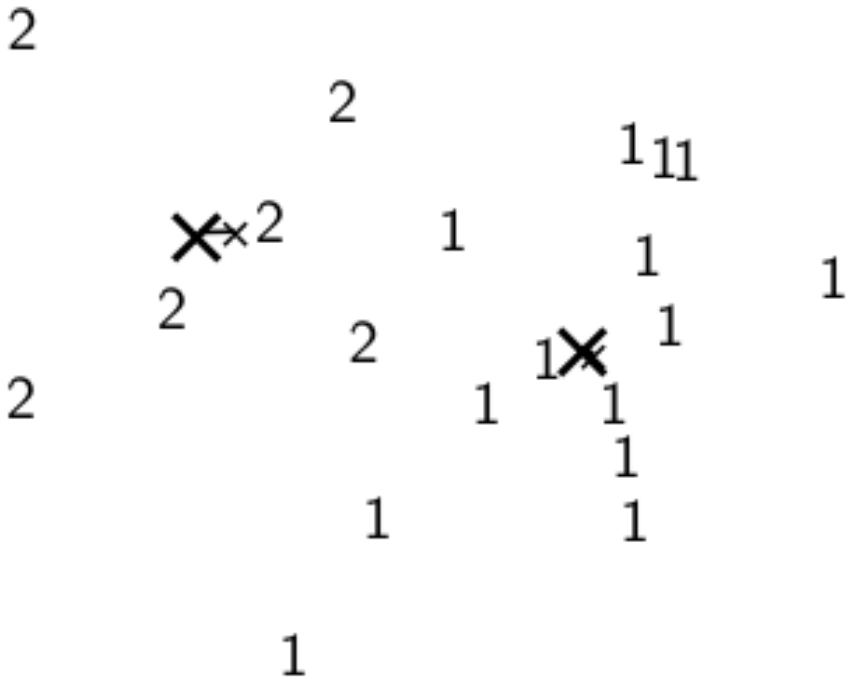
例子：重新分配(第七次)



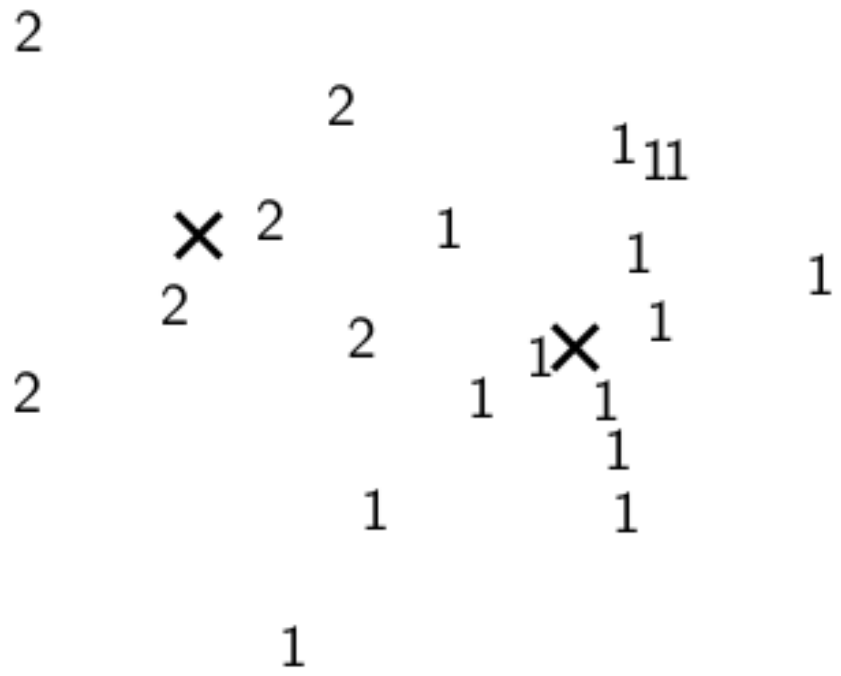
例子：分配结果



例子：重新计算质心向量



质心向量和分配结果最终收敛



K-均值聚类算法一定会收敛: 证明

- **RSS(Residual Sum of Squares，残差平方和) = 所有簇上的文档向量到(最近的)质心向量的距离平方和的总和**
- 每次重新分配之后RSS会下降
  - 这是因为每个向量都被移到离它最近的质心向量所代表的簇中 (只有找到更近的质心才会重新分配)
- 每次重新计算之后RSS也会下降
  - 参见下一页幻灯片
- 可能的聚类结果是有穷的
- 因此：一定会收敛到一个固定点
- 当然，这里有一个假设就是假定出现了等值的情况，算法都采用前后一致的方法来处理(比如，某个向量到两个质心向量的距离相等)

簇的质心 $\vec{\mu}(\omega)=\frac{1}{|\omega|}\sum_{\vec{x}\in\omega}\vec{x}$

所有向量到其质心距离的平方和 $\mathrm{RSS}_k=\sum_{x\in\omega_k}|\vec{x}-\vec{\mu}(\omega_k)|^2$

K个簇的所有向量到其质心距离的平方和 $\mathrm{RSS}=\sum_{k=1}^{K}\mathrm{RSS}_k$

如果能够证明在每次迭代后RSS的值单调递减，那么K-均值算法就会收敛。

x_m和v_m分别是文档向量和簇质心向量的第m个分量

$$\mathrm{RSS}_k(\vec{v})=\sum_{\vec{x}\in\omega_k}|\vec{v}-\vec{x}|^2=\sum_{\vec{x}\in\omega_k}\sum_{m=1}^{M}(v_m-x_m)^2$$

使得每个RSS_k达到最小值 $\frac{\partial\mathrm{RSS}_k(\vec{v})}{\partial v_m}=\sum_{\vec{x}\in\omega_k}2(v_m-x_m)=0$

得到v_m，其刚好基于每个向量分量来计算的质心的定义 $v_m=\frac{1}{|\omega_k|}\sum_{\vec{x}\in\omega_k}x_m$

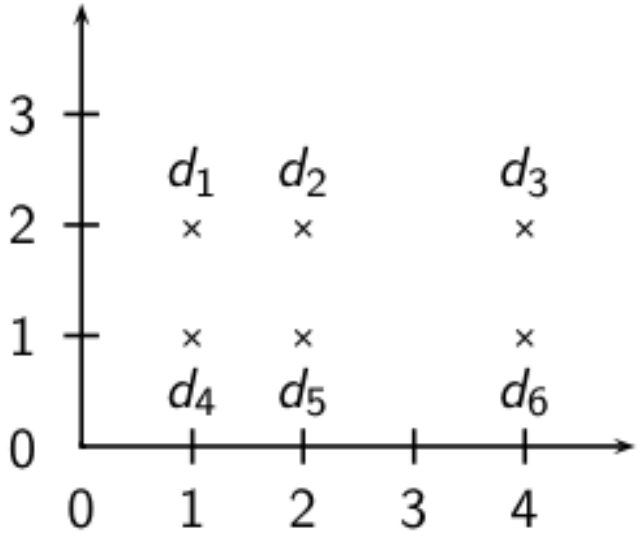
因此，当将旧质心替换为新质心 $\vec{v}$ 时，可以让RSS_k极小化。重新计算之后，作为RSS_k之和的RSS一定也会下降。

K-均值聚类算法一定是收敛的

- 但是不知道达到收敛所需要的时间!
- 如果不太关心少许文档在不同簇之间来回交叉的话，收敛速度通常会很快 (< 10-20次迭代)
- 但是，完全的收敛需要多得多的迭代过程

K-均值聚类算法的最优性

- 收敛并不意味着会达到全局最优的聚类结果!
- 这是K-均值聚类算法的最大缺点之一
- 如果开始的种子选的不好，那么最终的聚类结果可能会非常糟糕



- K=2情况下的最优聚类结果是什么？
- 对于任意的种子d_i、d_j，是否都会收敛于该聚类结果？
- 对于种子d_2和d_5，K-均值算法最后收敛为$\{\{d_1, d_2, d_3\}, \{d_4, d_5, d_6\}\}$
- 对种子d_2和d_3，收敛结果为$\{\{d_1, d_2, d_4, d_5\}, \{d_3, d_6\}\}$，这是$K$=2时的全局最优值

K-均值聚类算法的初始化

- 种子的随机选择只是*K*-均值聚类算法的一种初始化方法之一
- 随机选择不太鲁棒：可能会获得一个次优的聚类结果
- 一些确定初始质心向量的更好办法
  - 非随机地采用某些启发式方法来选择种子(比如，过滤掉一些离群点，或者寻找具有较好文档空间覆盖度的种子集合)
  - 采用层级聚类算法寻找好的种子
  - 选择 i (比如 $i = 10$) 次不同的随机种子集合，对每次产生的随机种子集合运行*K*-均值聚类算法，最后选择具有最小RSS值的聚类结果

K-均值聚类算法的时间复杂度

- 计算两个向量的距离的时间复杂度为$O(M)$
- 重分配过程：$O(KNM)$ (需要计算 KN 个文档-质心的距离)
- 重计算过程：$O(NM)$ (在计算质心向量时，需要累加簇内的文档向量)
- 假定迭代次数的上界是 I
- 整体复杂度: $O(IKNM)$ – 线性
- 但是，上述分析并没有考虑到实际中的最坏情况
- 在一些非正常的情况下，复杂度可能会比线性更糟

提纲

- 聚类介绍
- 聚类在IR中的应用
- K-均值聚类算法
- 聚类评价
- 簇个数确定
- 层次聚类

怎样判断聚类结果的好坏?

- 内部准则(Internal criteria)
  - 一个内部准则的例子： *K*-均值聚类算法的RSS值
- 但是内部准则往往不能评价聚类在应用中的实际效用
- 替代方法：外部准则(External criteria)
  - 按照用户定义的分类结果来评价，
  - 即对一个分好类的数据集进行聚类，将聚类结果和事先的类别情况进行比照，得到最后的评价结果

外部准则

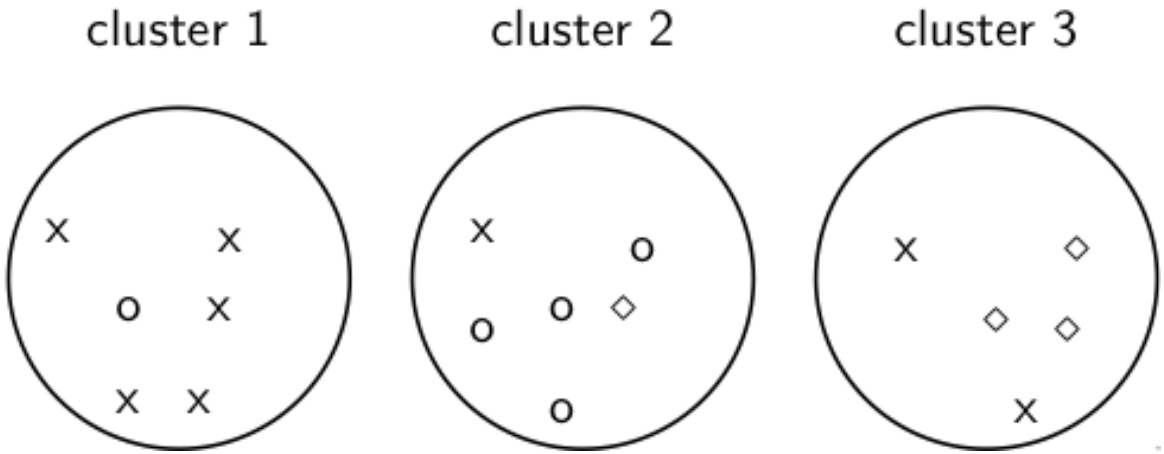
- 基于已有标注的标准数据集(如Reuters语料库)来进行聚类评价
- 目标：聚类结果和给定分类结果一致
- (当然，聚类中并不知道最后每个簇的标签，而只是关注如何将文档聚到不同的组中)
- 评价指标
  - 纯度(purity)
  - NMI (Normalized Mutual Information，归一化互信息)
  - RI (Rand Index，兰德指数)
  - F值 (F measure）

外部准则: 纯度

$$\text{purity}(\Omega, \mathrm{C}) = \frac{1}{N}\sum_{k}\max_{j} | \omega_k \cap c_j |$$

- $\Omega = \{\omega_1, \omega_2, \ldots, \omega_K\}$ 是簇的集合
- $C = \{c_1, c_2, \ldots, c_J\}$ 是类别的集合
- 对每个簇 ω_k：找到一个类别c_j，该类别包含ω_k中的元素最多，为n_{kj}个，也就是说ω_k的元素最多分布在c_j中
- 将所有n_{kj}求和，然后除以所有的文档数目N

纯度计算的例子



计算纯度

$\max_j |\omega_1 \cap c_j| = 5$ (class x, cluster 1);

$\max_j |\omega_2 \cap c_j| = 4$ (class o, cluster 2);

$\max_j |\omega_3 \cap c_j| = 3$ (class ◇, cluster 3)

纯度为 $(5 + 4 + 3) / 17 \approx 0.71$.

外部准则:归一化互信息

$$NMI(\Omega,\mathrm{C})=\frac{I(\Omega,\mathrm{C})}{[H(\Omega)+H(\mathrm{C})]/2}$$

$$I(\Omega,\mathrm{C})=\sum_k\sum_j P(\omega_k\cap c_j)\log\frac{P(\omega_k\cap c_j)}{P(\omega_k)P(c_j)}$$

$$=\sum_k\sum_j\frac{|\omega_k\cap c_j|}{N}\log\frac{N|\omega_k\cap c_j|}{|\omega_k||c_j|}$$

$$H(\Omega)=\sum_k P(\omega_k)\log P(\omega_k)$$

$$=\sum_k\frac{|\omega_k|}{N}\log\frac{|\omega_k|}{N}$$

其中，$P(\omega_k)$、$P(c_j)$及$P(\omega_k\cap c_j)$分别是一篇文档属于ω_k、c_j及$\omega_k\cap c_j$ 的概率。

外部准则：兰迪指数/准确率(Rand index)

- 定义

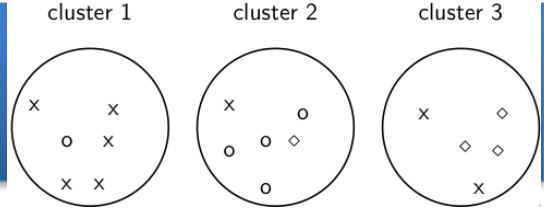
$$RI = \frac{TP+TN}{TP+FP+FN+TN}$$

- 考虑所有两个文档之间(文档对)的关系，可以得到 2x2 的列联表

	same cluster	different clusters
same class	true positives (TP)	false negatives (FN)
different classes	false positives (FP)	true negatives (TN)

- 将聚类看成是一系列的决策过程，即对文档集上所有N(N−1)/2个文档对进行决策。当且仅当两篇文档相似时，将它们归入同一簇中。
- TP(True-positive,真阳性):将两篇相似文档归入一个簇，而TN(True-negative,真阴性)将两篇不相似的文档归入不同的簇。在此过程中会犯两类错误：FP决策会将两篇不相似的文档归入同一簇，而FN决策将两篇相似的文档归入不同簇。
- RI计算的是正确决策的比率，就是在8.3节中提到的精确率

兰迪指数：例子



回到上例，三个簇中分别包含6、6、5个点，因此处于同一簇的文档对的个数为：

$$TP + FP = \binom{6}{2} + \binom{6}{2} + \binom{5}{2} = 40$$

6*5/2

其中， 簇1中的x 对，簇2中的 o 对，簇3中的 ⋄ 对，以及簇3中的 x 对，都是真正例：

$$TP = \binom{5}{2} + \binom{4}{2} + \binom{3}{2} + \binom{2}{2} = 20$$

于是，$FP = 40 - 20 = 20$。类似地，可以计算出FN和TN。

	same cluster	different clusters
same class	TP = 20	FN = 24
different classes	FP = 20	TN = 72

RI is then

$$(20 + 72)/(20 + 20 + 24 + 72) \approx 0.68.$$

外部准则：F值

- 可以使用F 值来度量聚类结果，并通过设置β > 1 以加大对FN 的惩罚，此时实际上也相当于赋予召回率更大的权重

$$P=\frac{TP}{TP+FP} \qquad R=\frac{TP}{TP+FN} \qquad F_\beta=\frac{(\beta^2+1)PR}{\beta^2 P+R}$$

提纲

- 聚类介绍
- 聚类在IR中的应用
- *K*-均值聚类算法
- 聚类评价
- 簇个数确定
- 层次聚类

簇个数确定

- 在很多应用中，簇个数 K 是事先给定的
  - 比如，可能存在对K的外部限制
  - 例子：在“分散-集中”应用中，在显示器上(上世纪90年代)很难显示超过10-20个簇
- 如果没有外部的限制会怎样？是否存在正确的簇个数？
- 一种办法：定义一个优化准则
  - 给定文档，找到达到最优情况的K 值
  - 能够使用的最优准则有哪些？
  - 我们不能使用前面所提到的RSS或到质心的平均平方距离等准则，因为它们会导致$K = N$ 个簇

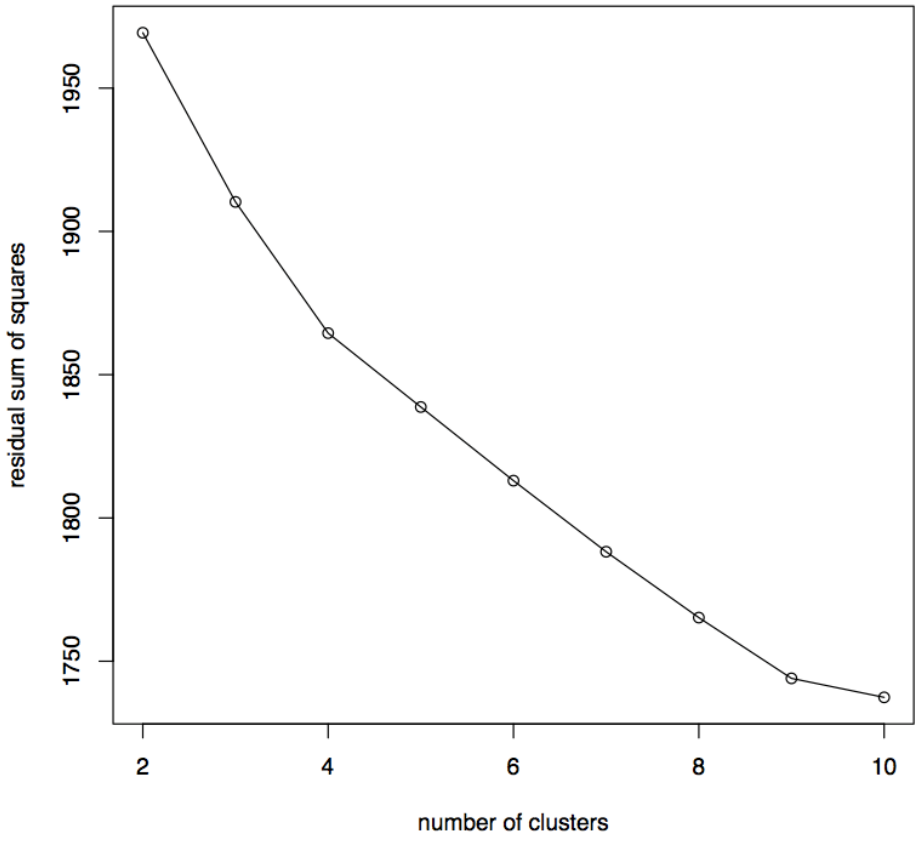
简单的目标函数

- 基本思路
  - 从1个簇开始 ($K = 1$)
  - 不断增加簇 (= 不断增大 K)
  - 对每个新的簇增加一个惩罚项
- 在惩罚项和RSS之间折中
- 选择满足最佳折中条件的 K

- 给定聚类结果，定义文档的代价为其到质心向量的(平方)距离(失真率)
- 定义全部失真率 RSS(K) 为所有文档代价的和
- 然后：对每个簇一个惩罚项 λ
- 于是，对于具有 K 个簇的聚类结果，总的聚类惩罚项为 $K\lambda$
- 定义聚类结果的所有开销为失真率和总聚类惩罚项的和
  - RSS(K) + $K\lambda$
- 选择使得 (RSS(K) + $K\lambda$) 最小的K值
- 当然，还要考虑较好的λ值 . . .

在曲线中寻找拐点

本图中两个拐点：4 和 9



提纲

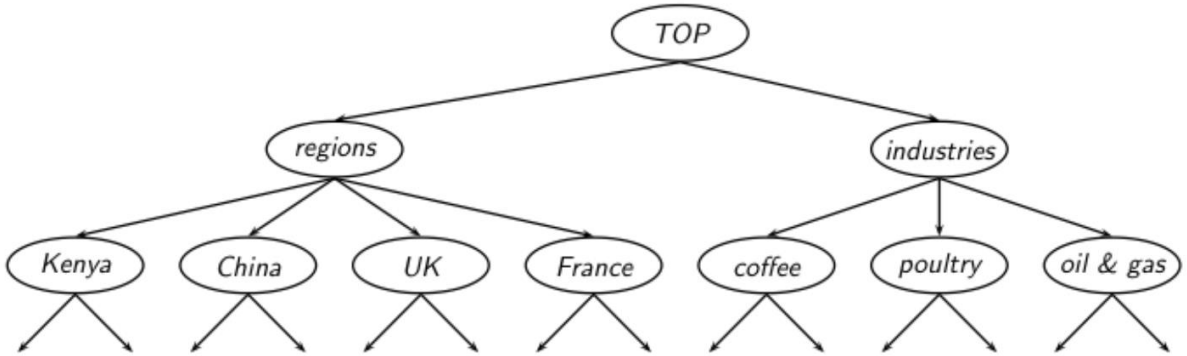
- 聚类介绍
- 聚类在IR中的应用
- *K*-均值聚类算法
- 聚类评价
- 簇个数确定
- 层次聚类

层次聚类 hierarchical clustering

- 扁平聚类
  - 优点：概念简单、速度快
  - 缺点：算法返回的是一个无结构的扁平簇集合，需要预先定义簇的数目，并且聚类结果具有不确定性
- 层次聚类
  - 输出一个具有层次结构的簇集合，因此能够比扁平聚类输出的无结构簇集合提供更丰富的信息。
  - 不需要事先指定簇的数目，并且大部分用于IR中的层次聚类算法都是确定性算法。
  - 在获得这些好处的同时，其代价是效率降低。最普遍的层次聚类算法的时间复杂度至少是文档数目的平方级，而*K*-均值算法的时间复杂度是线性的。
- 当效率因素非常重要时，选择扁平聚类算法。而当扁平算法的问题（如结构信息不足、簇数目需要预先定义、聚类结果非确定性）需要加以考虑时，则采用层次算法

层次聚类的目标

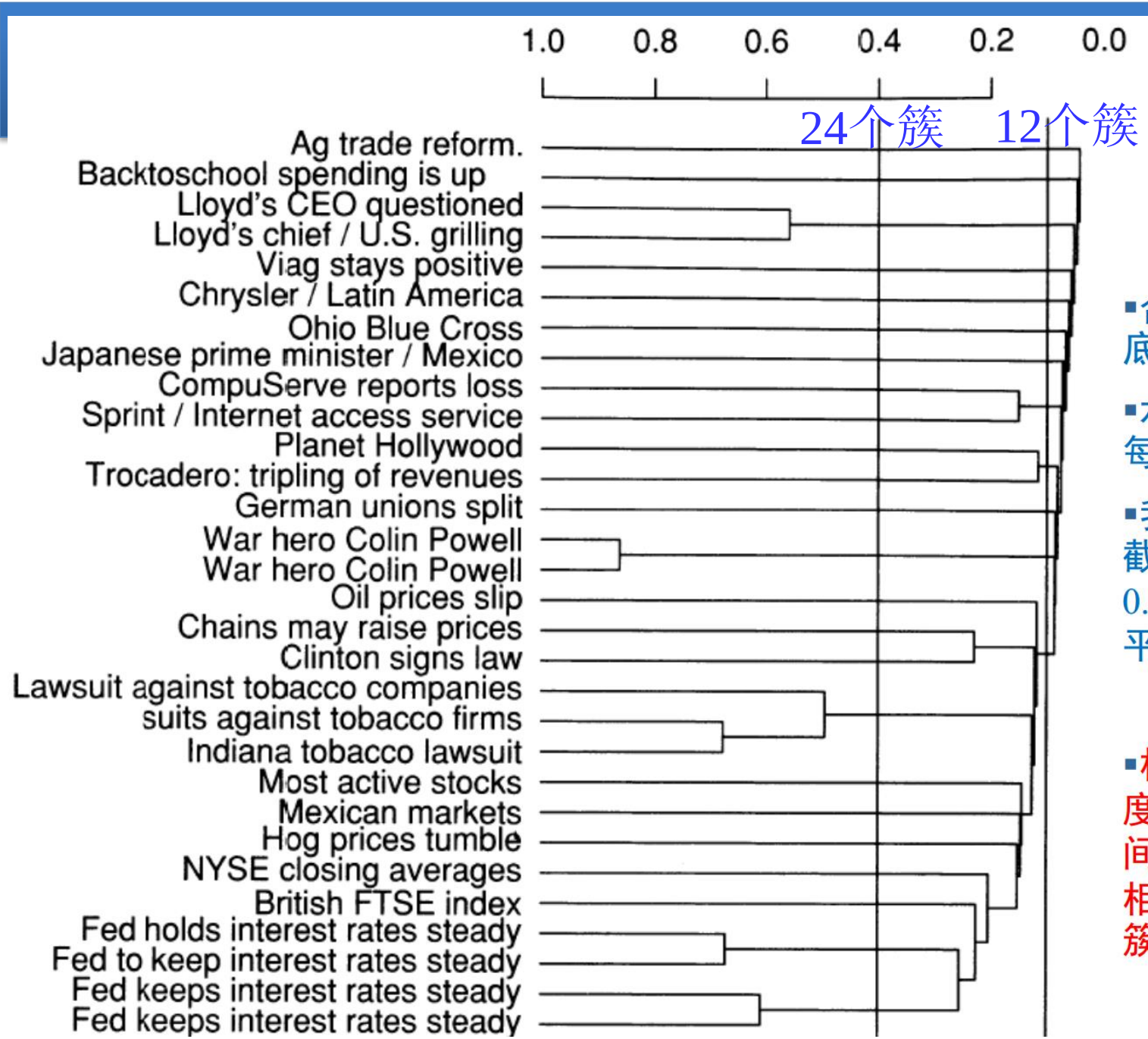
- 目标：生成类似于前面提到的Reuters目录的一个层次结构。



- 这个层次结构是自动创建的，可以通过自顶向下(分裂式divisive)或自底向上(凝聚 agglomerate)的方法实现。
  - 自底向上的算法：一开始将每篇文档都看成是一个簇，然后不断地对簇进行两两合并，直到所有文档都聚成一类为止
  - 而自顶向下的方法：首先将所有文档看成一个簇，然后不断利用某种方法对簇进行分裂直到每篇文档都成为一个簇为止。

层次凝聚式聚类

- 在**IR** 领域， **HAC** 方法的使用比自顶向下方法更普遍。最著名的自底向上的方法是层次凝聚式聚类**(hierarchical agglomerative clustering，HAC)**
  - 一开始每篇文档作为一个独立的簇
  - 然后，将其中最相似的两个簇进行合并
  - 重复上一步直至仅剩一个簇
  - 整个合并的历史构成一个二叉树
  - 一个标准的描述层次聚类合并历史的方法是采用树状图**(dendrogram)**



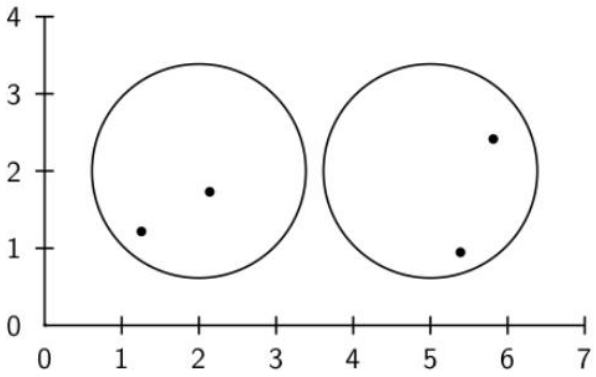
- 合并的历史可以从底往上生成
- 水平线上给出的是每次合并的相似度
- 我们可以在特定点截断 (比如 0.1 或 0.4) 来获得一个扁平的聚类结果

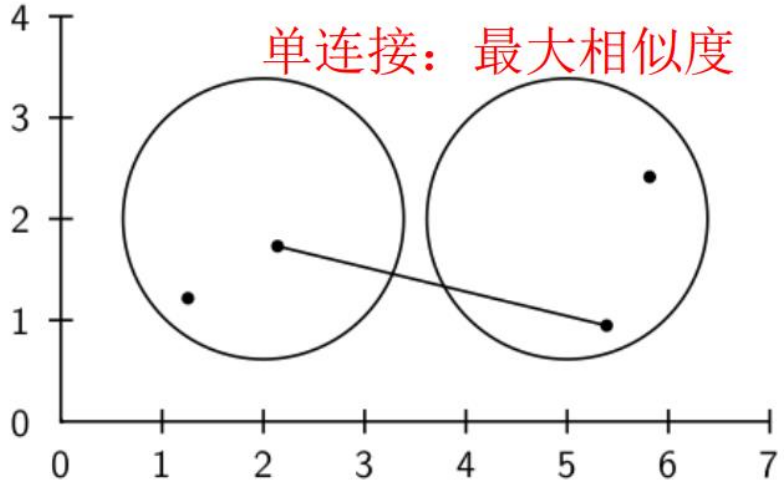
- 相似度：此前相似度都定义在文档之间，现在我们假设相似度定义在两个簇之间

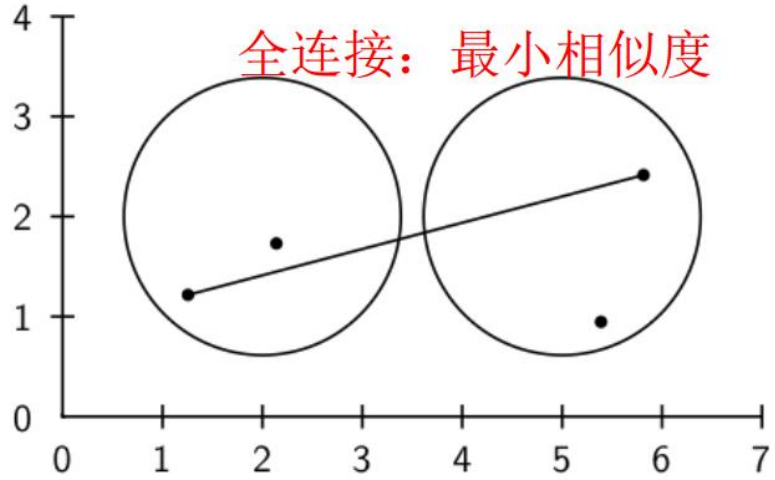
关键问题：如何定义簇间相似度

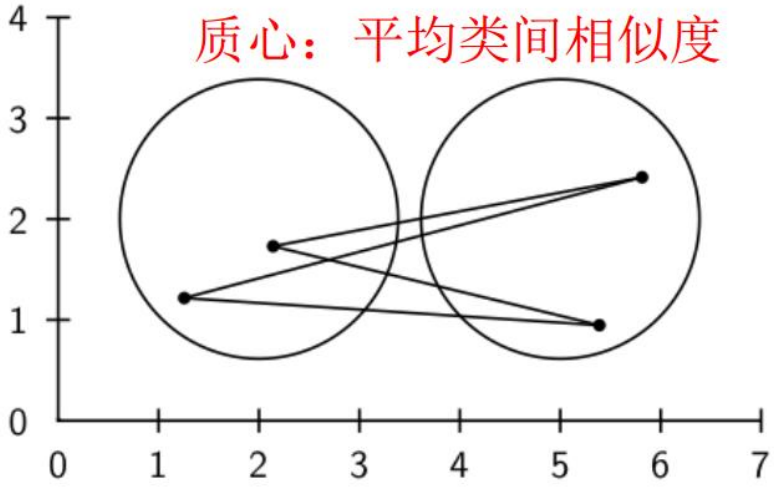
- 单连接(Single-link)
  - 两个最大相似的成员之间的相似度
- 全连接(Complete-link)
  - 两个最不相似的成员之间的相似度
- 质心法: 平均的类间相似度
  - 所有的簇间文档对之间相似度的平均值 (不包括同一个簇内的文档之间的相似度)
  - 这等价于两个簇质心之间的相似度
- 组平均(Group-average): 平均的类内和类间相似度
  - 所有的簇间文档对之间相似度的平均值 (包括同一个簇内的文档之间的相似度)

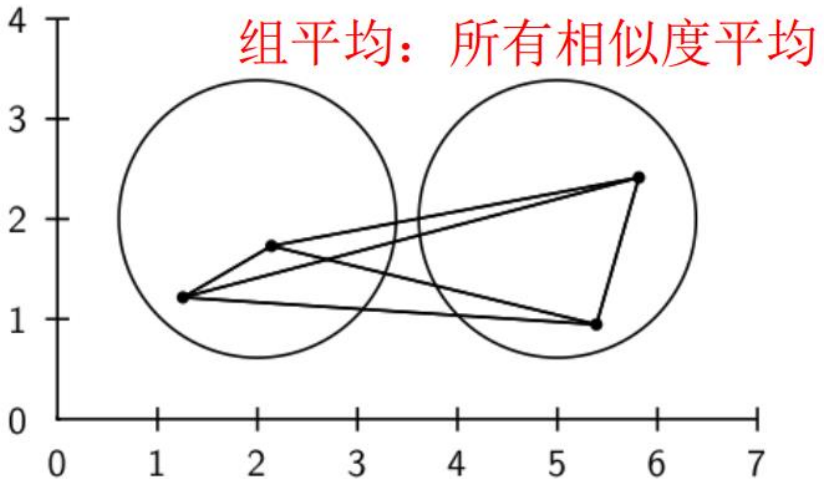
四种簇相似度示例1


4
3
2
1
0
0 1 2 3 4 5 6 7

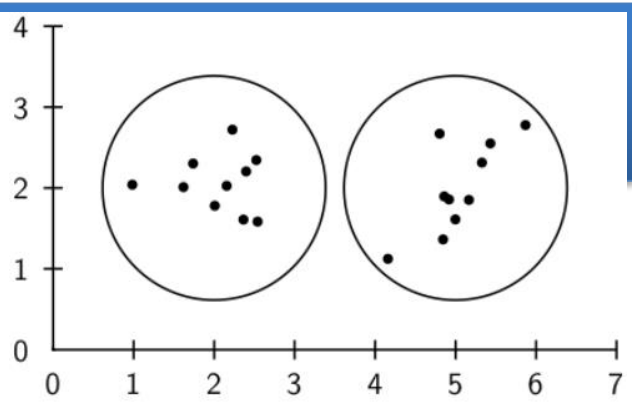

单连接：最大相似度
4
3
2
1
0
0 1 2 3 4 5 6 7


全连接：最小相似度
4
3
2
1
0
0 1 2 3 4 5 6 7

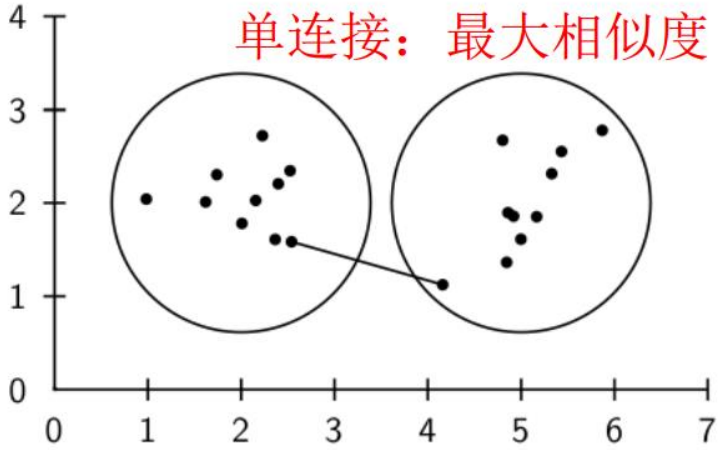

质心：平均类间相似度
4
3
2
1
0
0 1 2 3 4 5 6 7


组平均：所有相似度平均
4
3
2
1
0
0 1 2 3 4 5 6 7

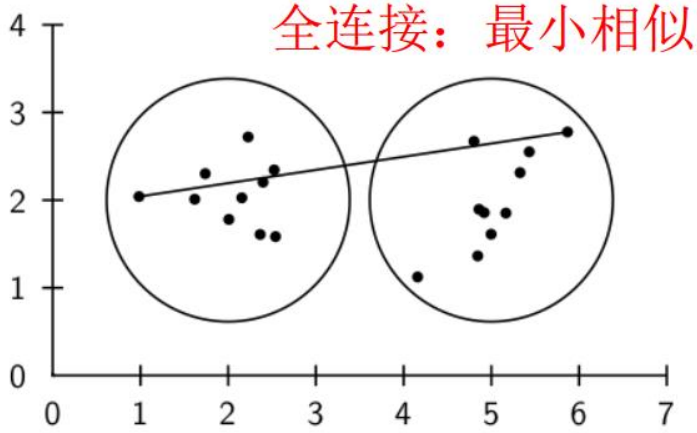
四种簇相似度示例2



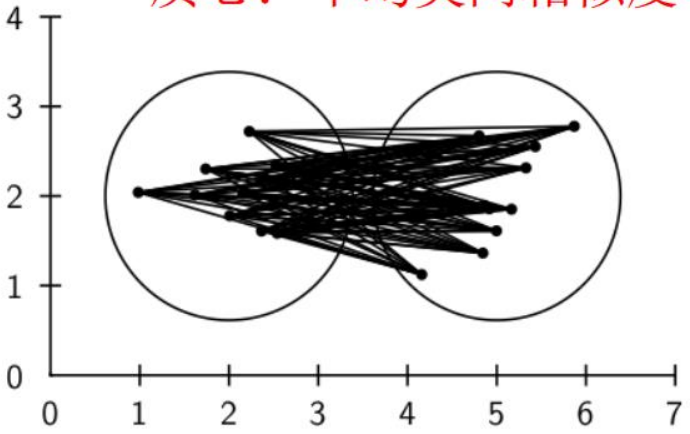
单连接：最大相似度



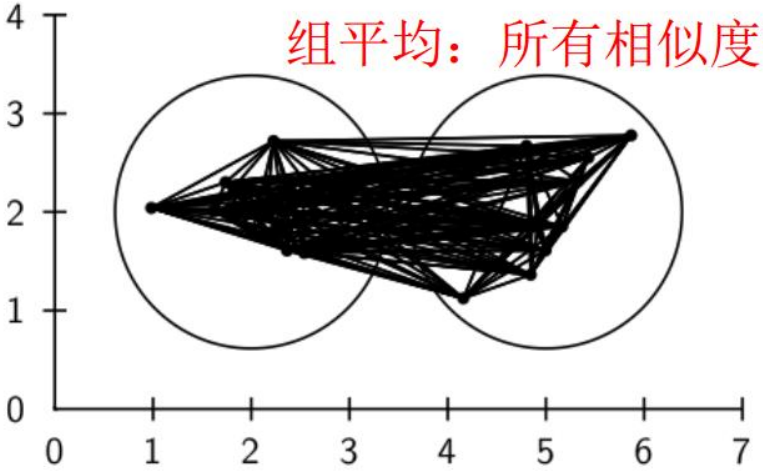
全连接：最小相似度



质心：平均类间相似度



组平均：所有相似度平均

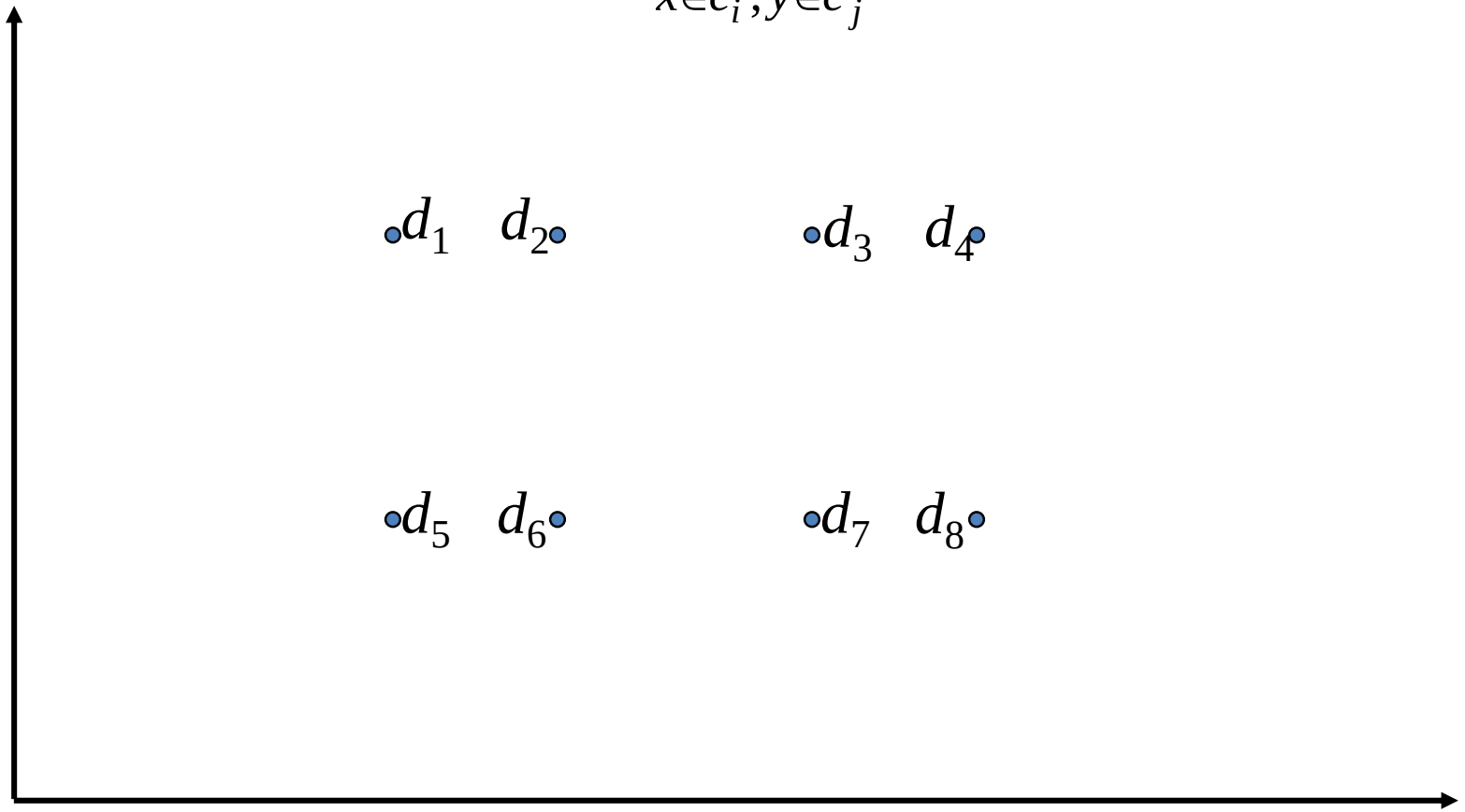


单连接/全连接算法

- 单连接聚类(**single-link clustering**)
  - 两个簇之间的相似度定义为两个最相似(最近)的成员之间的相似度。
  - 这种单连接的合并准则是局部的，即仅仅关注两个簇互相邻近的区域，而不考虑簇中更远的区域和簇的总体结构。
- 全连接聚类(**complete-link clustering**)
  - 两个簇之间的相似度定义为两个最不相似的成员之间的相似度，这也相当于选择两个簇进行聚类，使得合并结果具有最短直径。
  - 全连接聚类准则是非局部的，聚类结果中的整体结构信息会影响合并的结果。这种聚类实际上相当于优先考虑具有较短直径的紧凑簇，而不是具有长直径的松散簇
  - 这种做法可能对离群点较为敏感，比如某个远离中心的文档会显著增加候选簇的直径从而完全改变最后的聚类结果

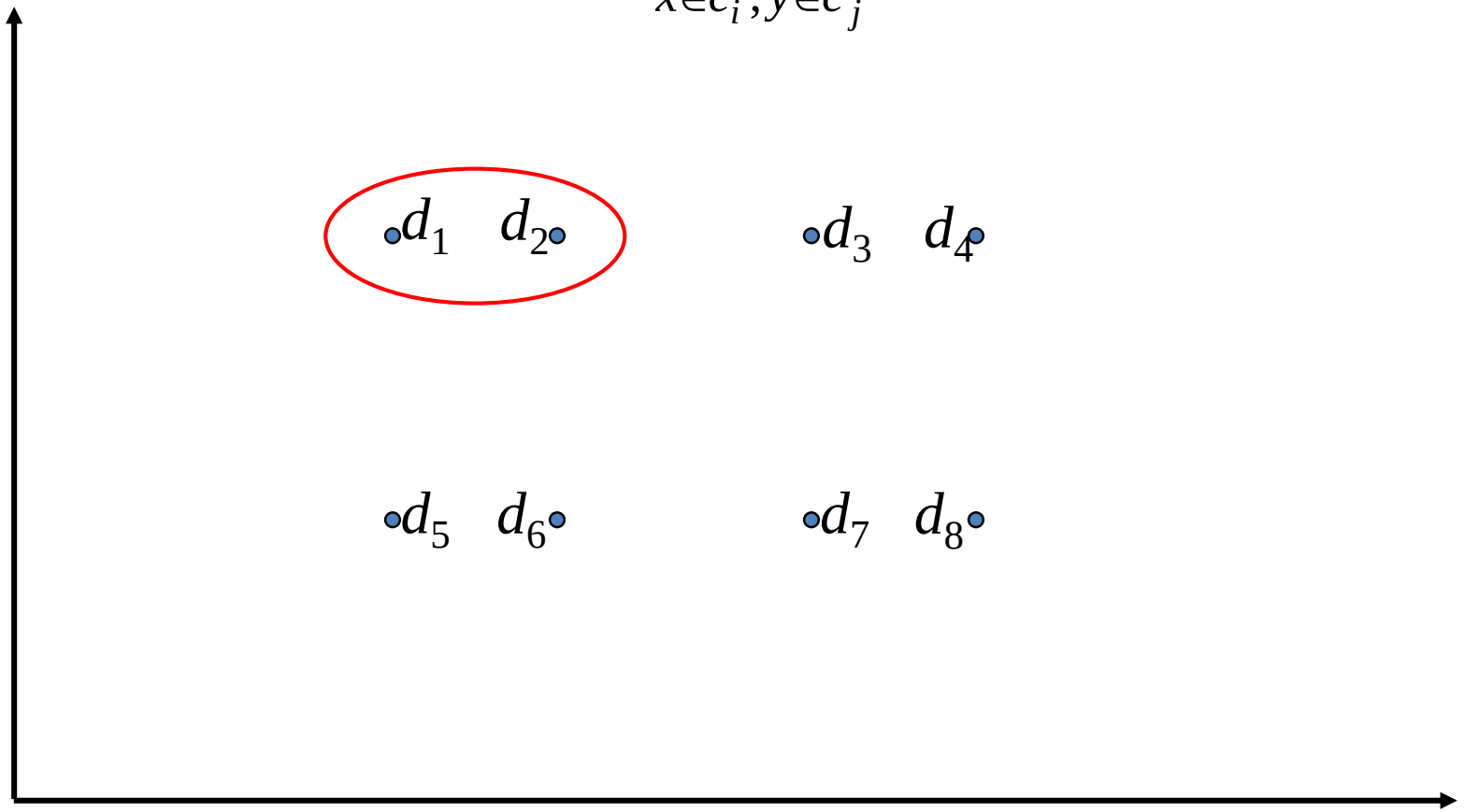
Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



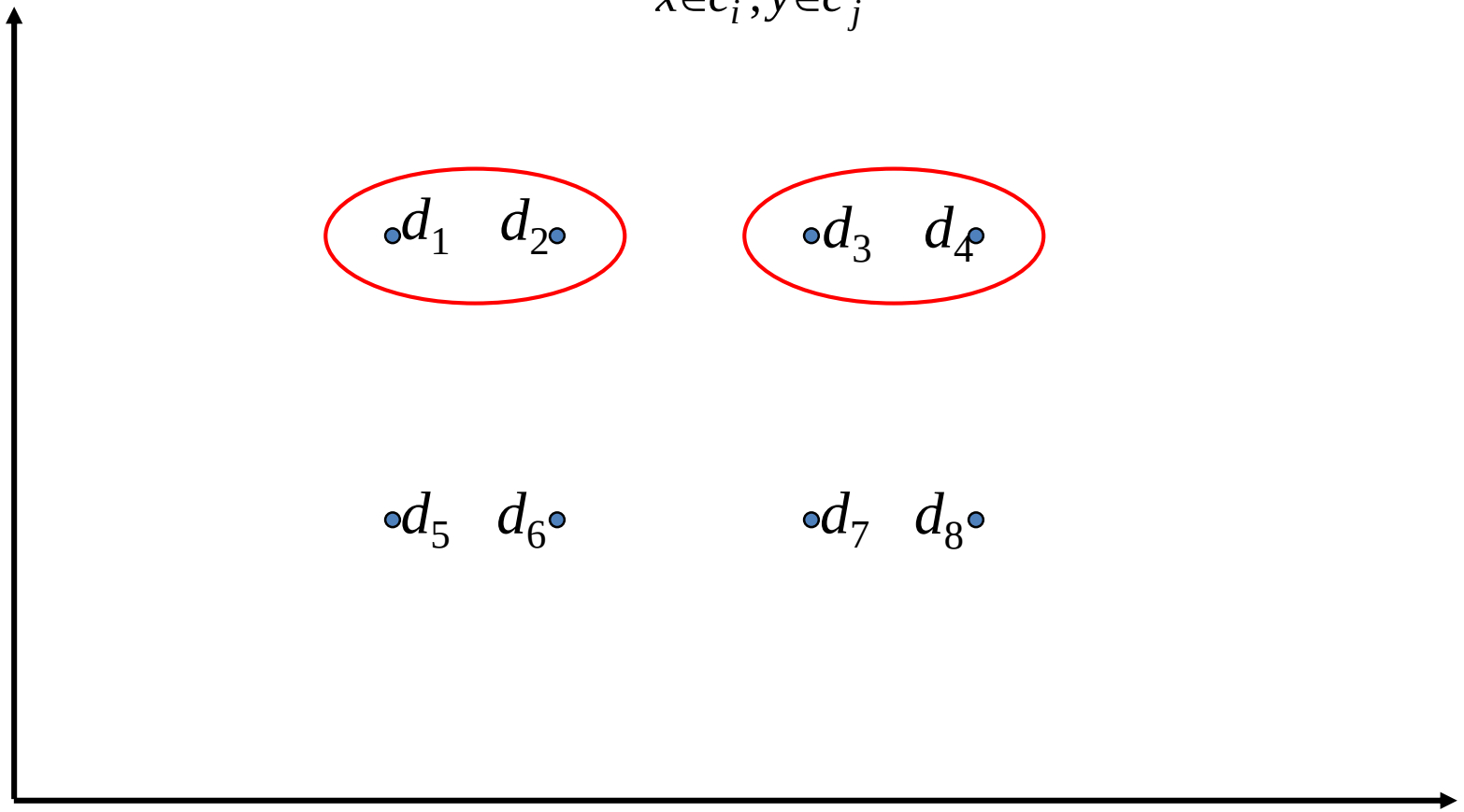
Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



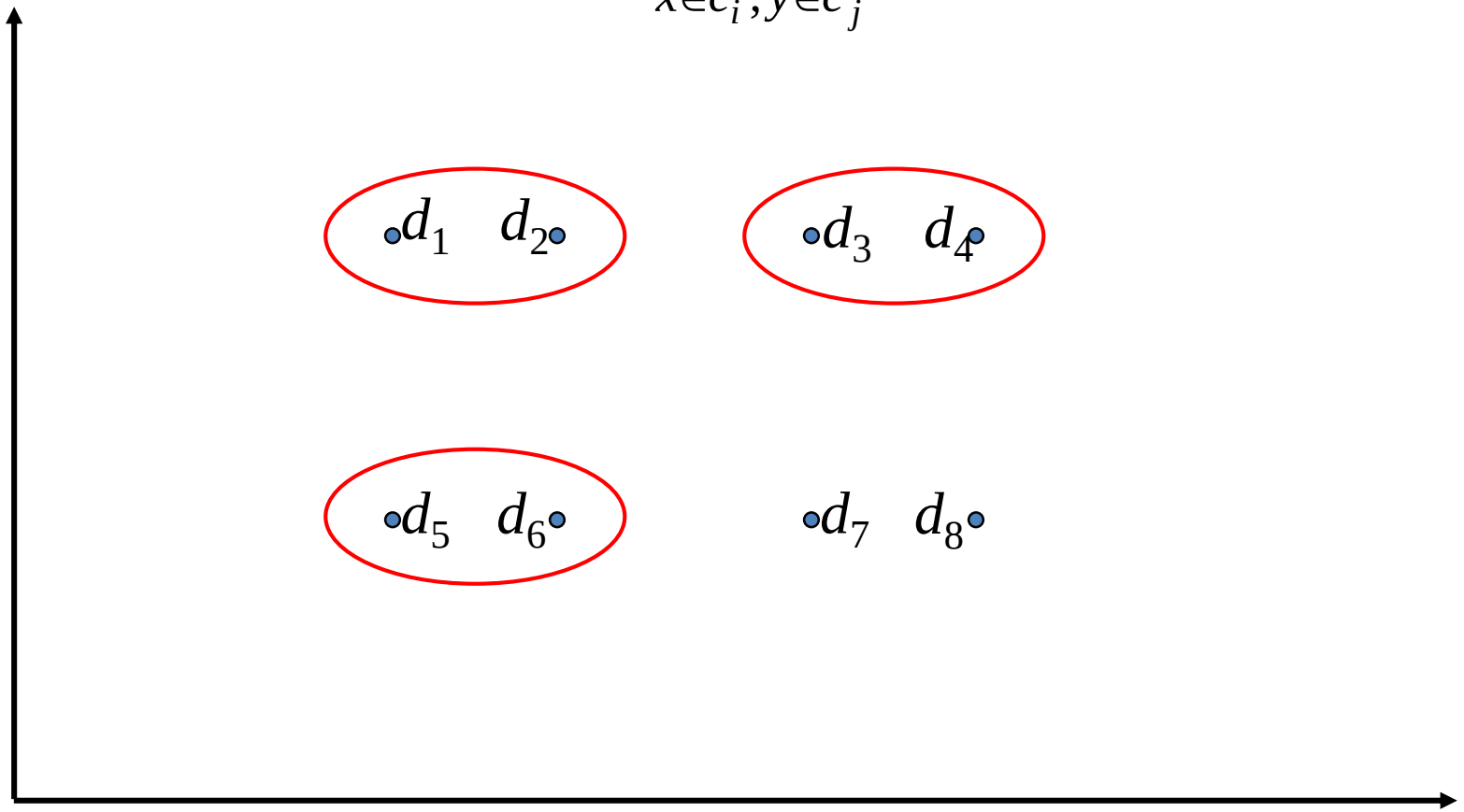
Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



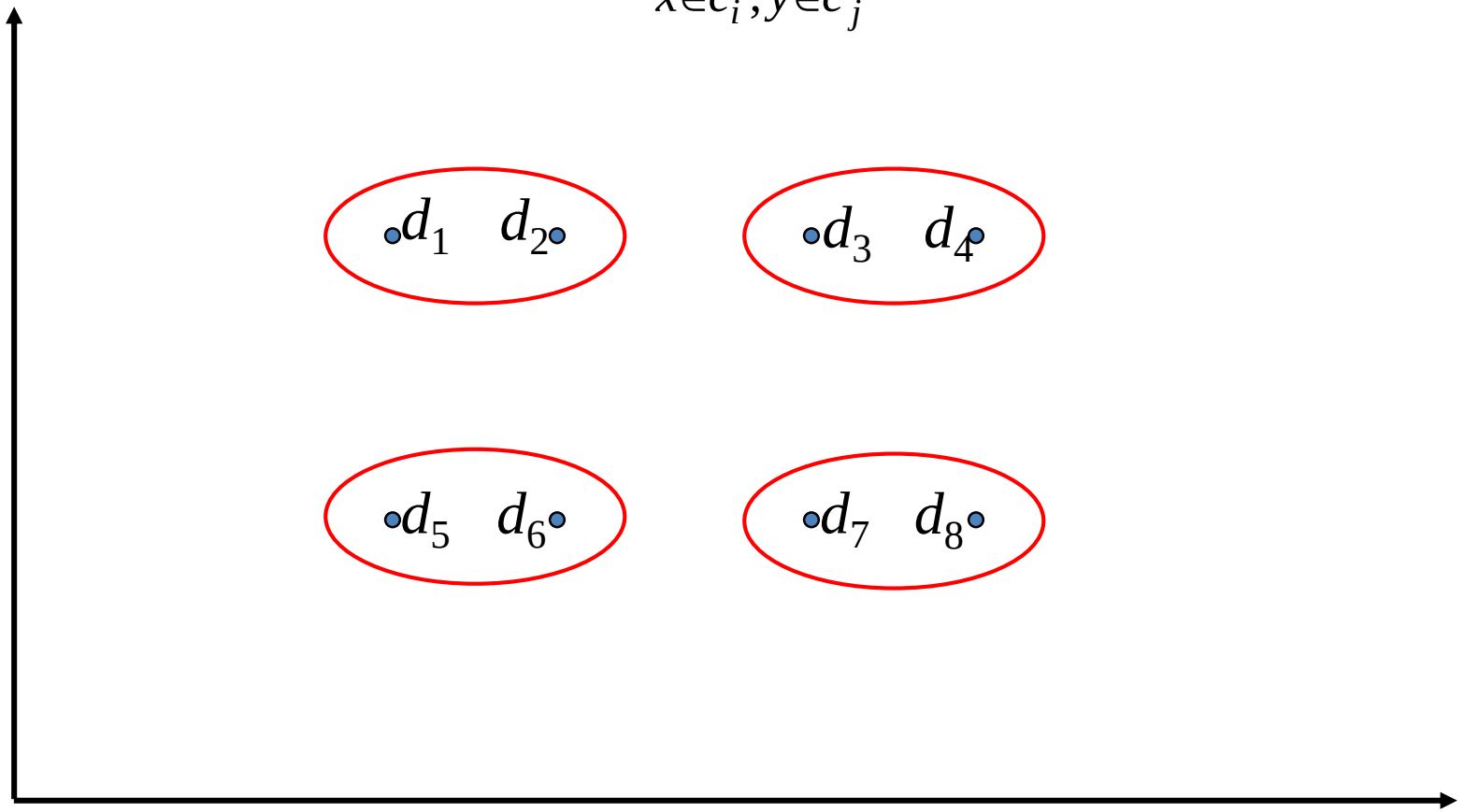
Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



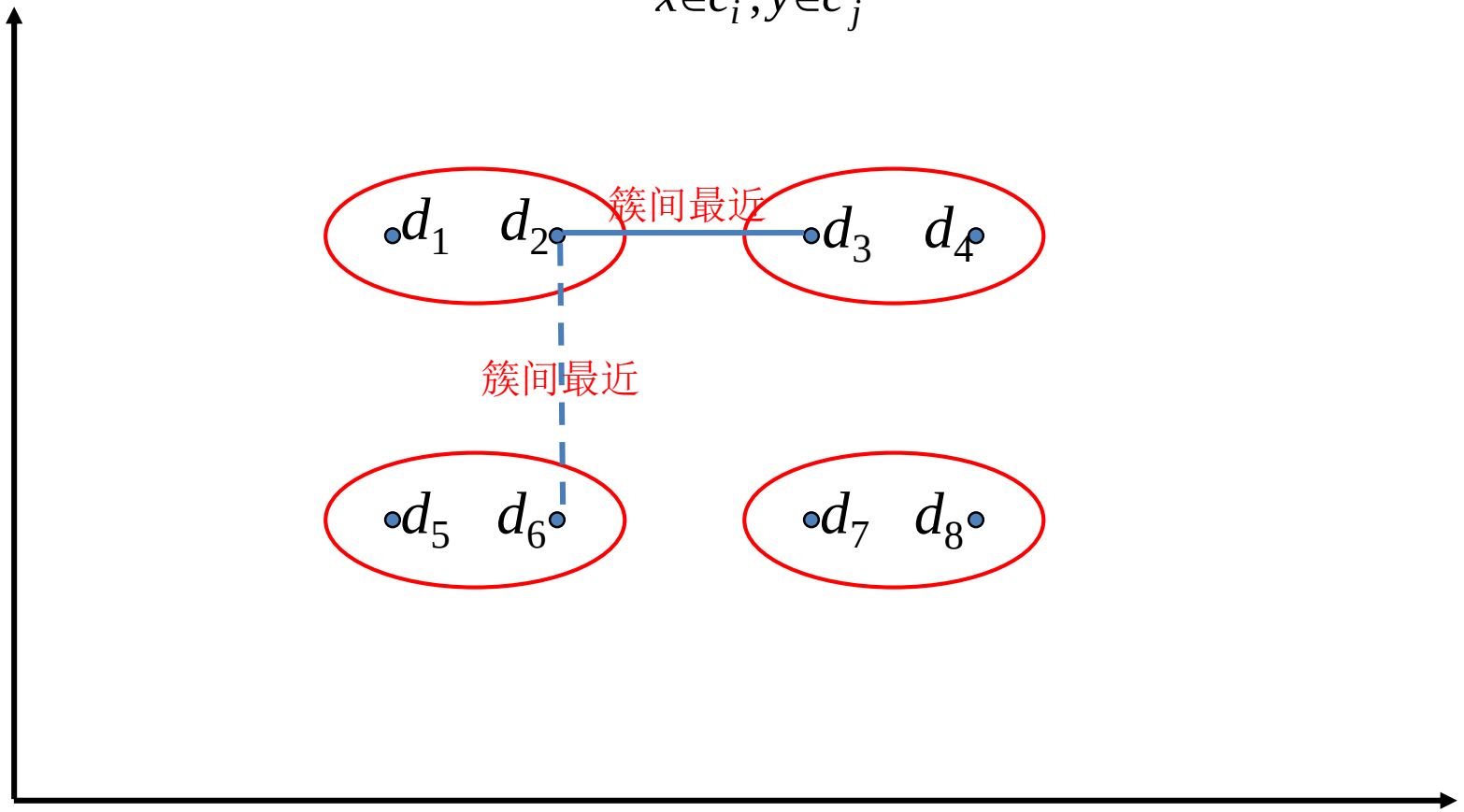
Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



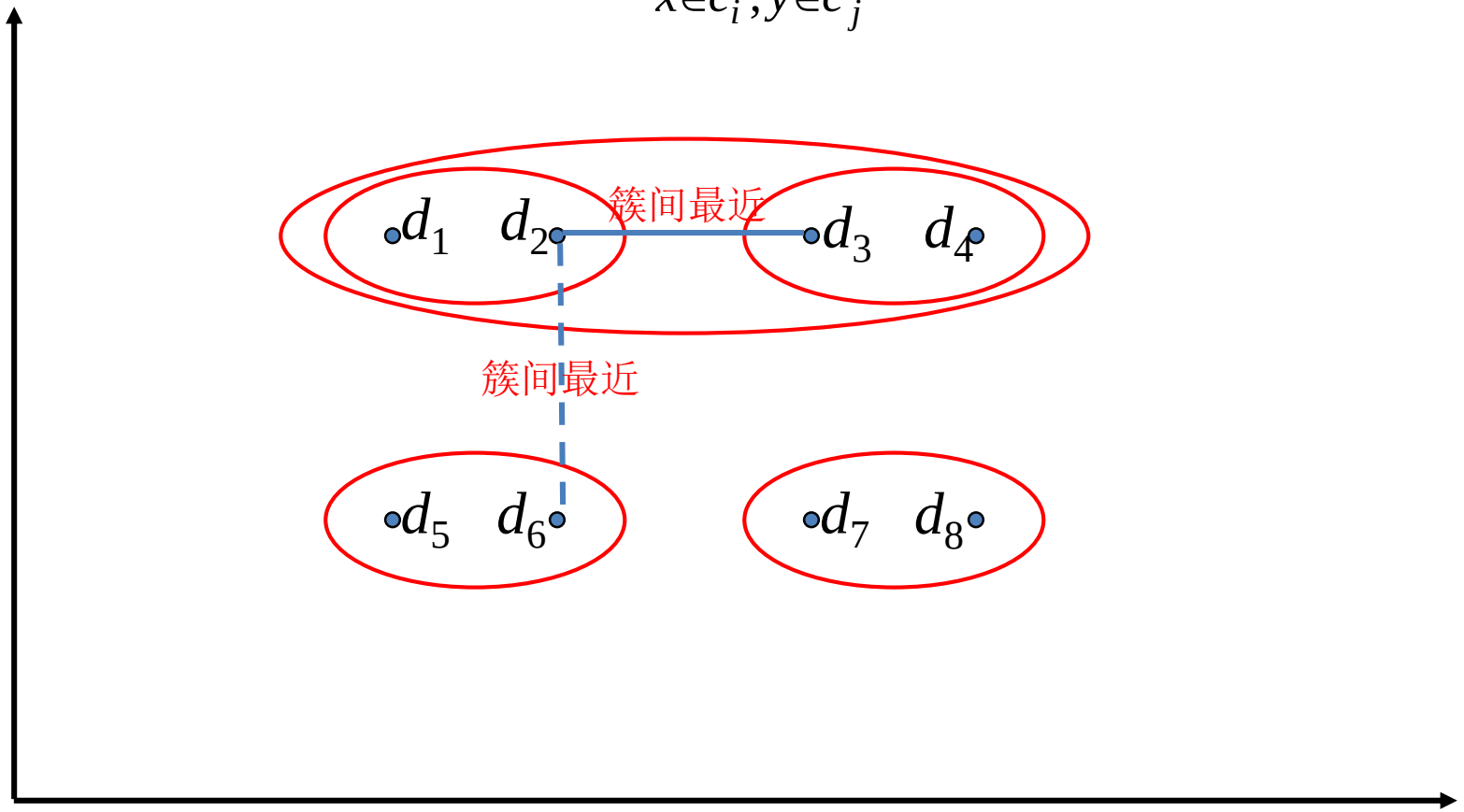
Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



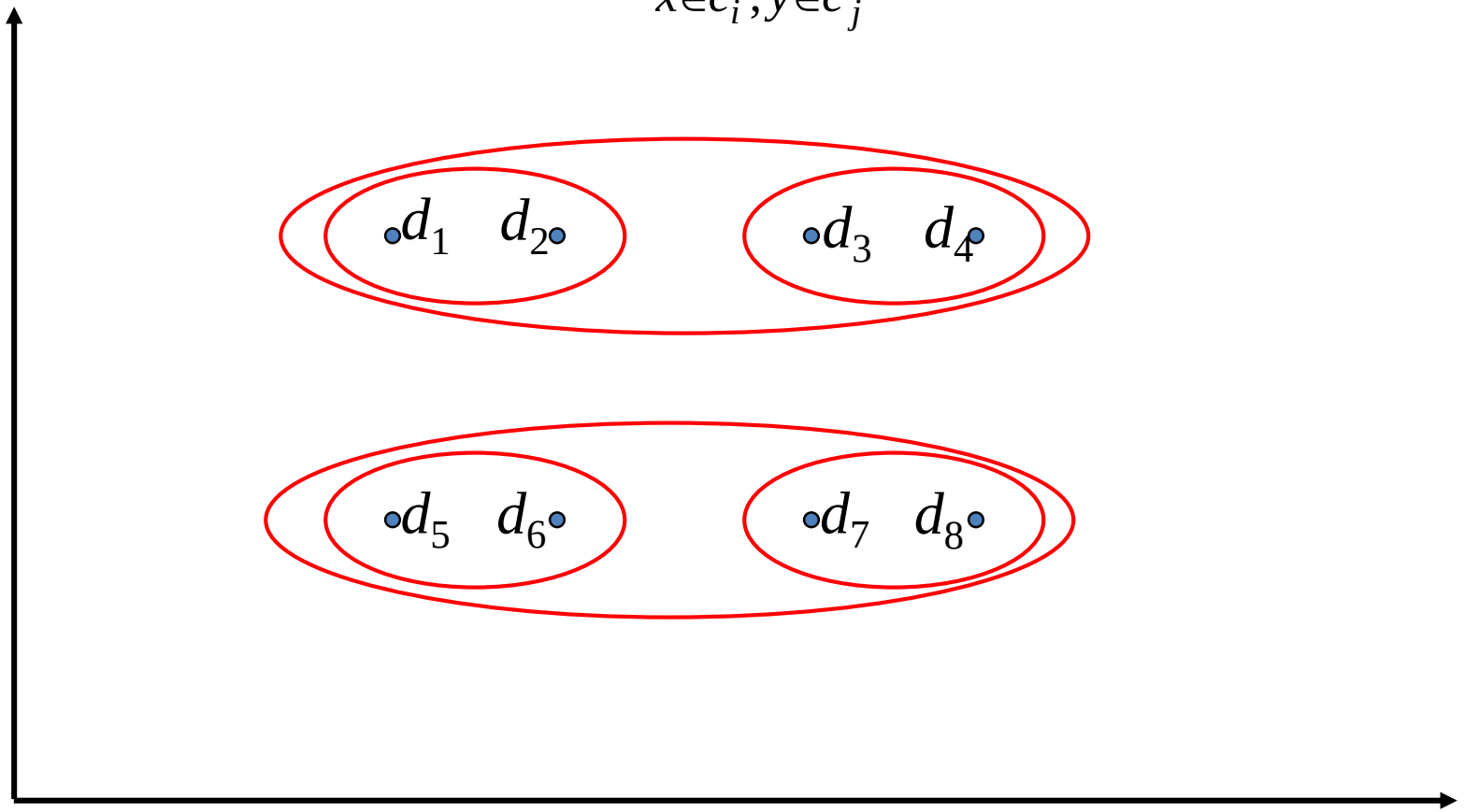
Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



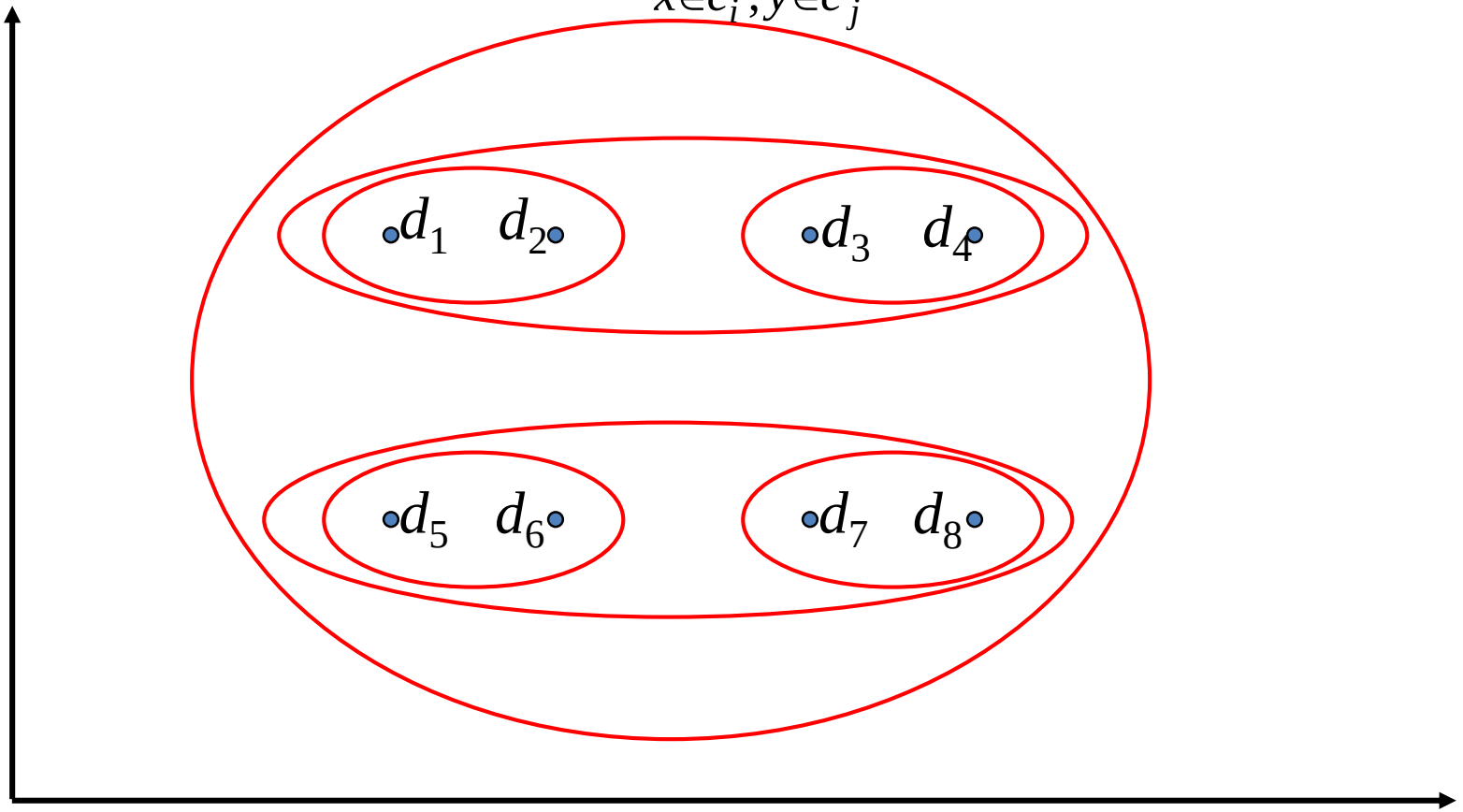
Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



Single Link Example —1

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



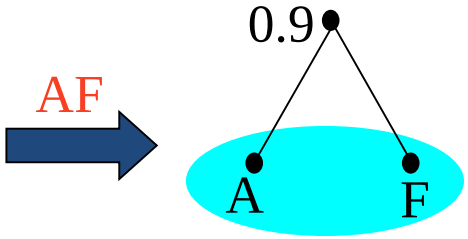
Single Link Example —2

	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

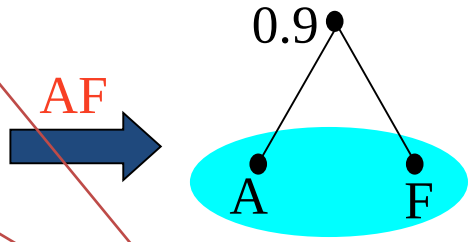
	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

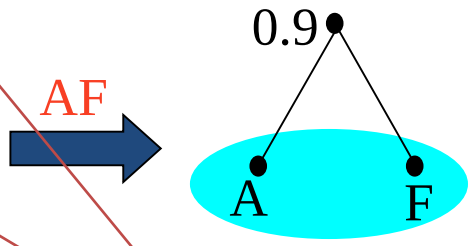
$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$



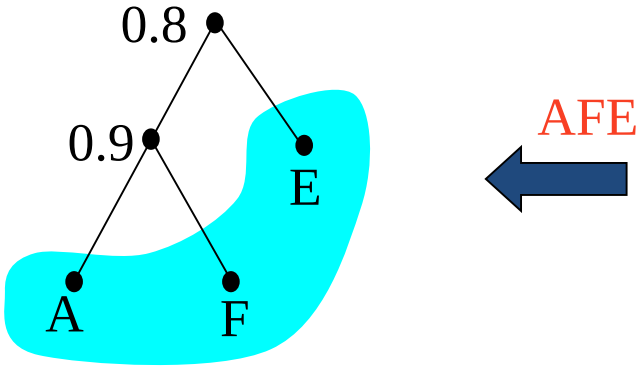
	AF	B	C	D	E
AF	-	0.7	0.5	0.6	0.8
B	0.7	-	0.4	0.5	0.7
C	0.5	0.4	-	0.3	0.5
D	0.6	0.5	0.3	-	0.4
E	0.8	0.7	0.5	0.4	-

	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

$$sim(c_i, c_j) = \max_{x \in c_i, y \in c_j} sim(x, y)$$

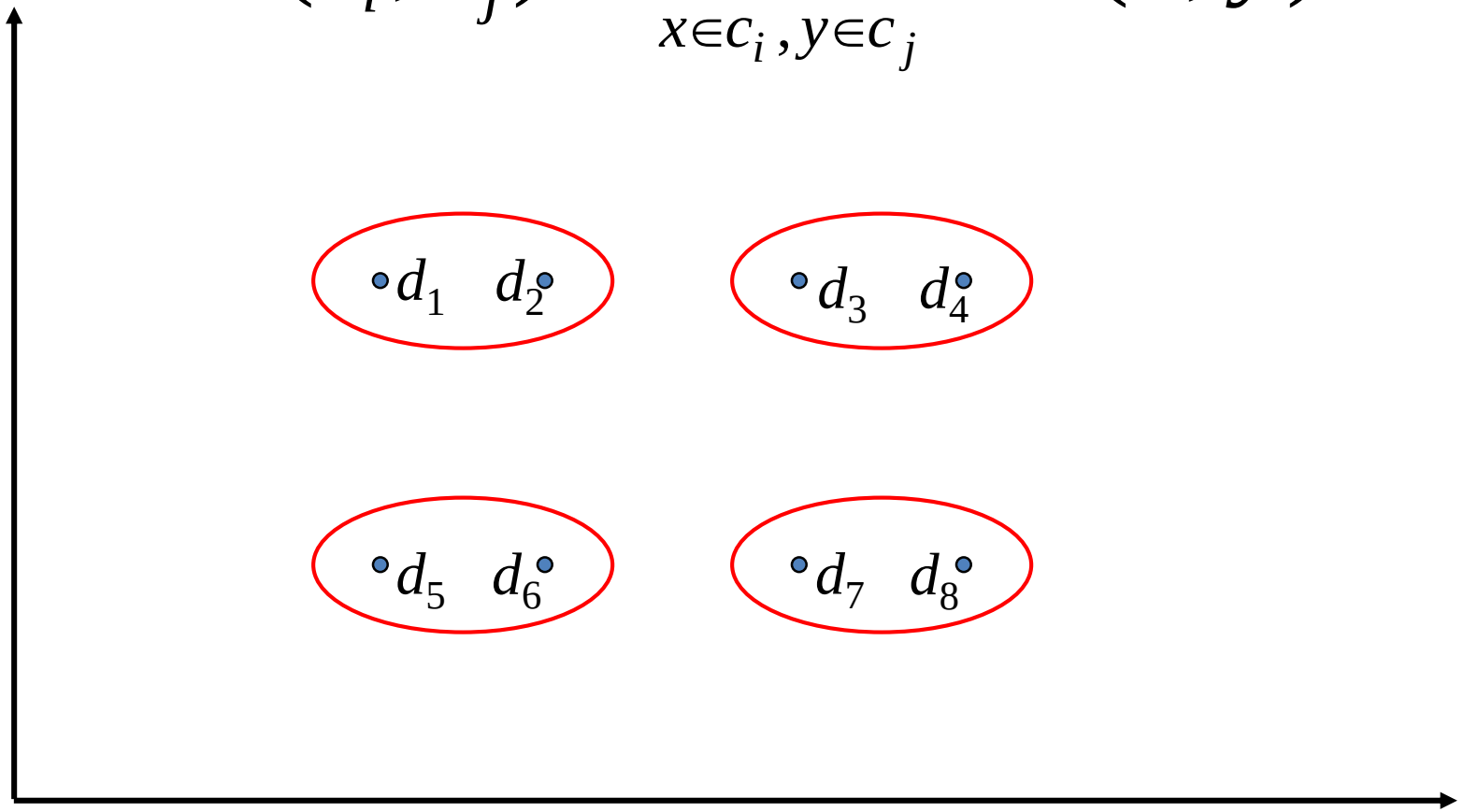


	AF	B	C	D	E
AF	-	0.7	0.5	0.6	0.8
B	0.7	-	0.4	0.5	0.7
C	0.5	0.4	-	0.3	0.5
D	0.6	0.5	0.3	-	0.4
E	0.8	0.7	0.5	0.4	-

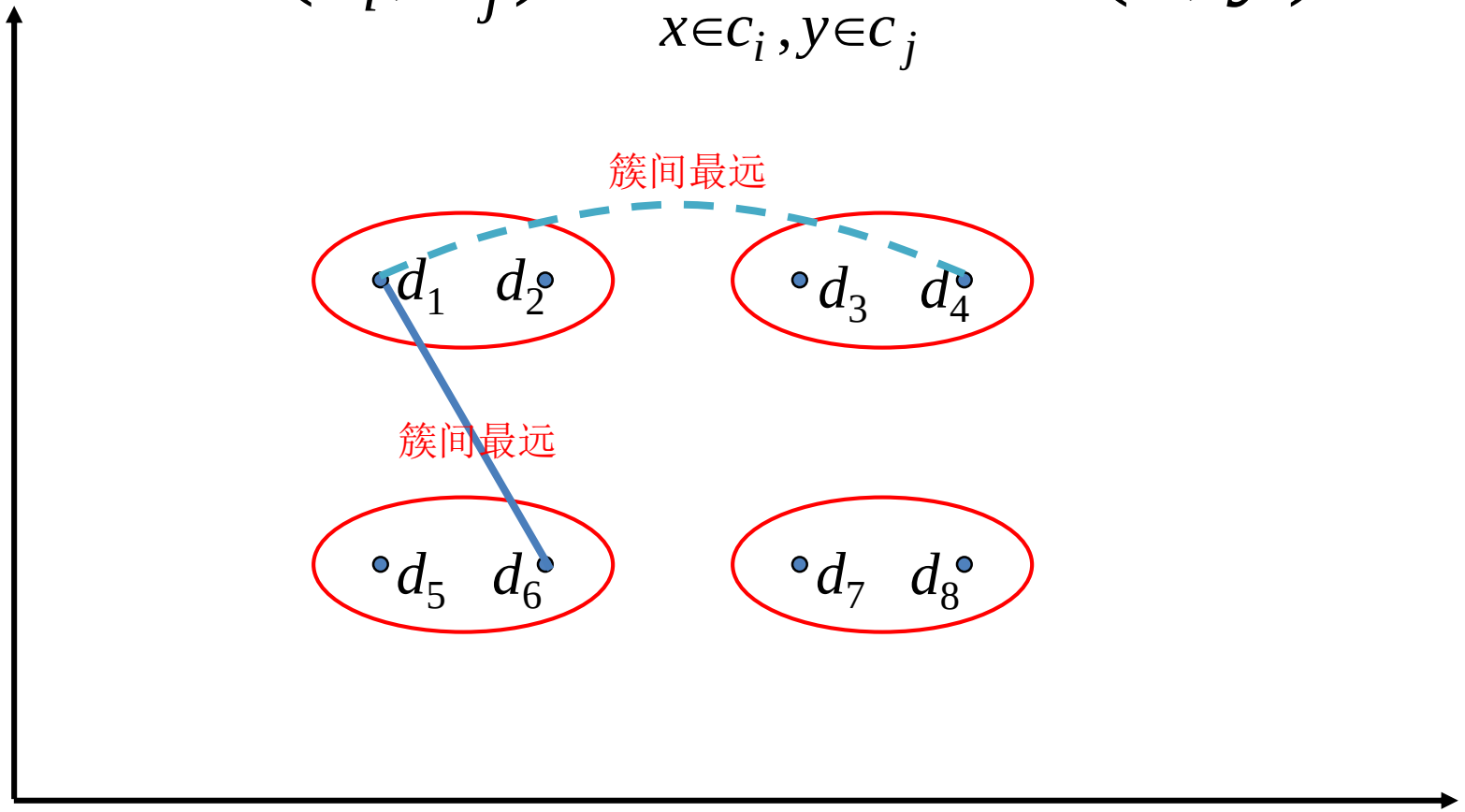


Complete Link Example —1

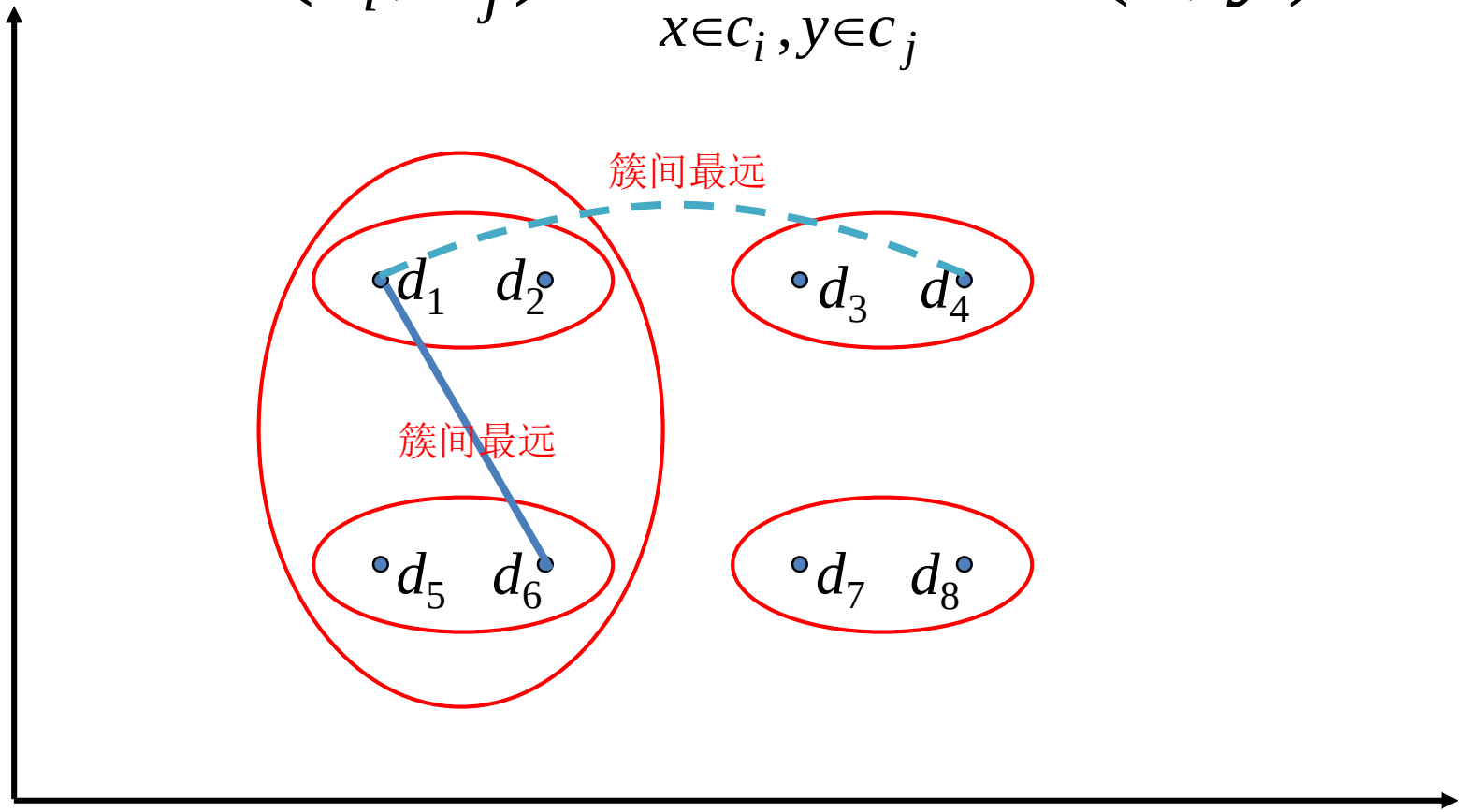
$$sim(c_i, c_j) = \min_{x \in c_i, y \in c_j} sim(x, y)$$



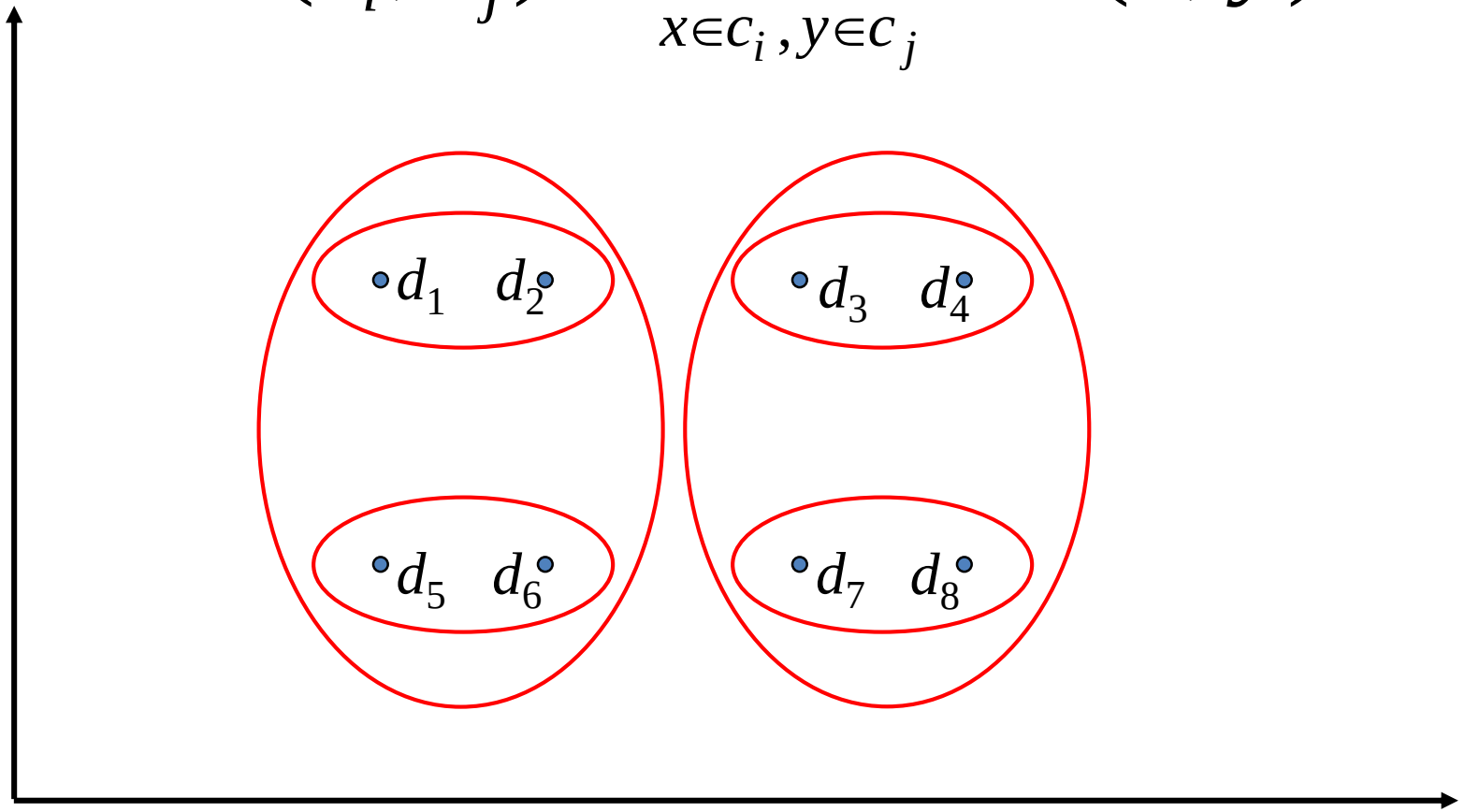
$$sim(c_i, c_j) = \min_{x \in c_i, y \in c_j} sim(x, y)$$



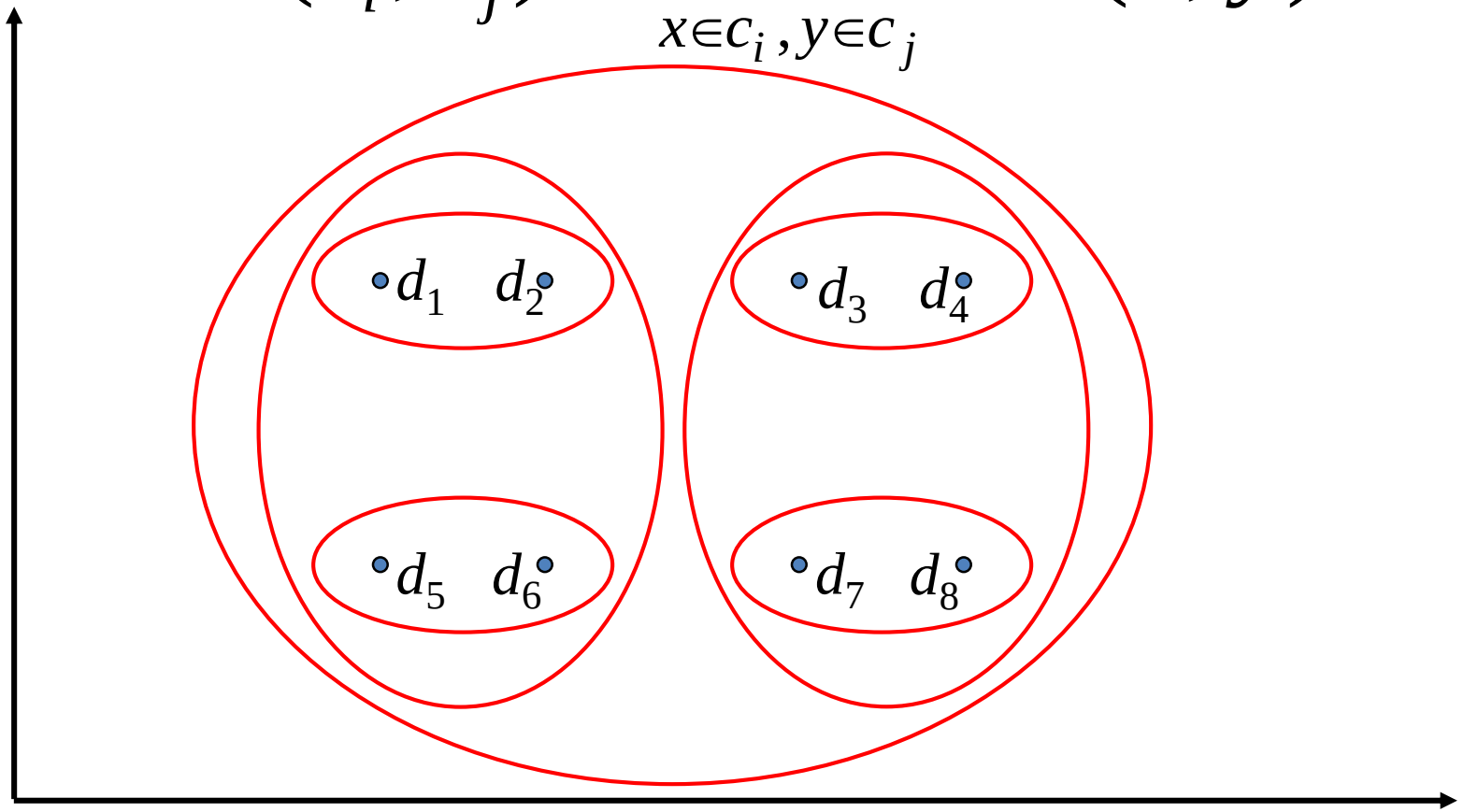
$$sim(c_i, c_j) = \min_{x \in c_i, y \in c_j} sim(x, y)$$



$$sim(c_i, c_j) = \min_{x \in c_i, y \in c_j} sim(x, y)$$



$$sim(c_i, c_j) = \min_{x \in c_i, y \in c_j} sim(x, y)$$



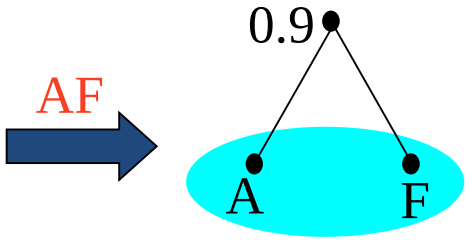
Complete Link Example —2

	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

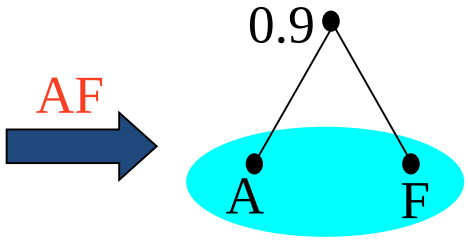
Complete Link Example —2

	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

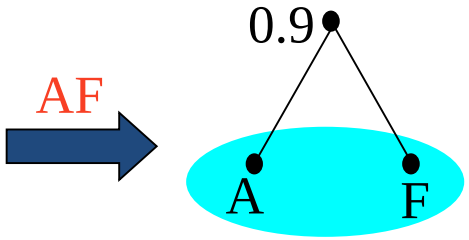


	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-



	AF	B	C	D	E
AF	-	?	?	?	?
B	?	-	0.4	0.5	0.7
C	?	0.4	-	0.3	0.5
D	?	0.5	0.3	-	0.4
E	?	0.7	0.5	0.4	-

	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.7
C	0.5	0.4	-	0.3	0.5	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.5	0.4	-	0.3
F	0.9	0.7	0.2	0.1	0.3	-

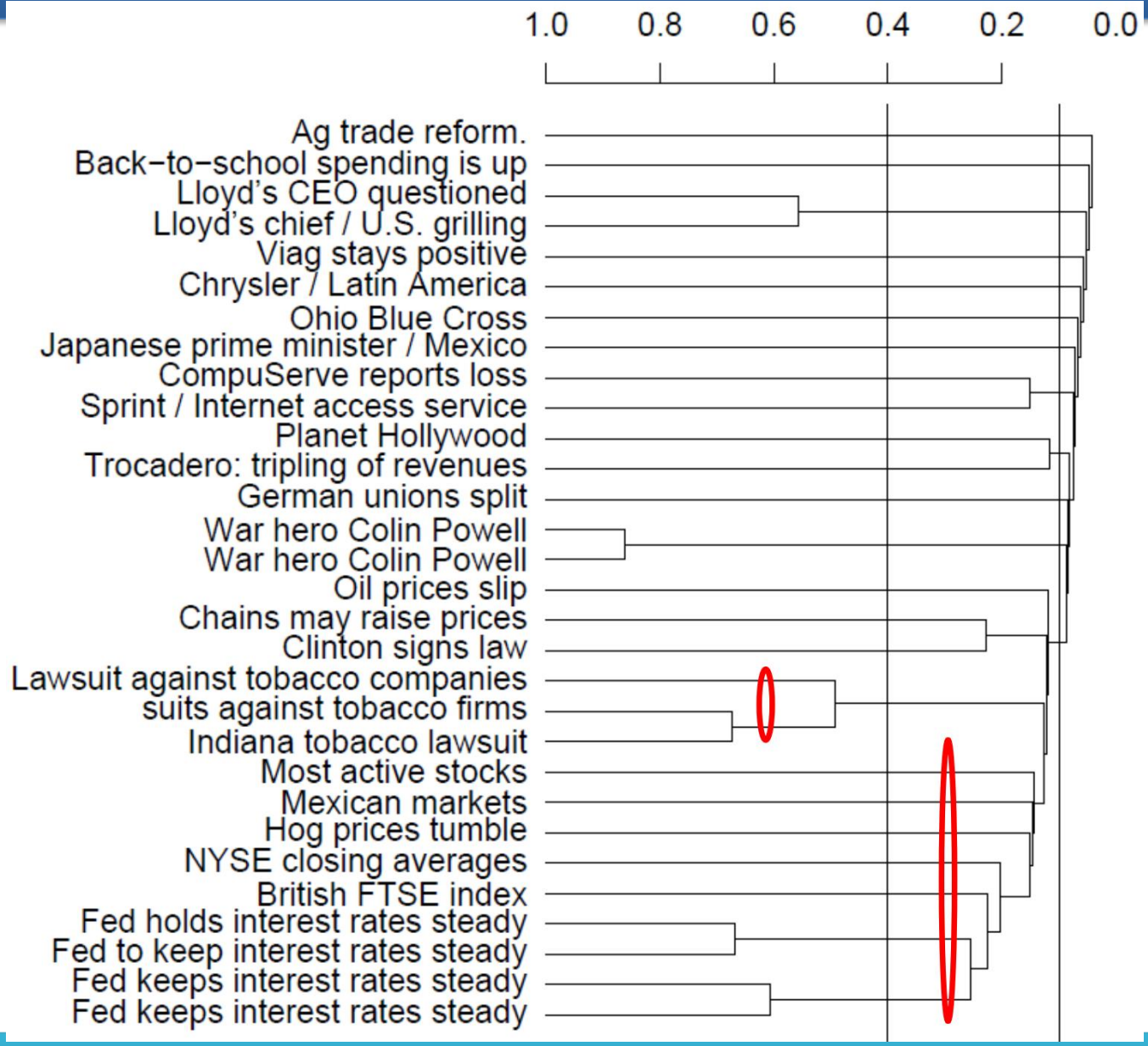


$$sim(c_i, c_j) = \min_{x \in c_i, y \in c_j} sim(x, y)$$

	AF	B	C	D	E
AF	-	?	?	?	?
B	?	-	0.4	0.5	0.7
C	?	0.4	-	0.3	0.5
D	?	0.5	0.3	-	0.4
E	?	0.7	0.5	0.4	-

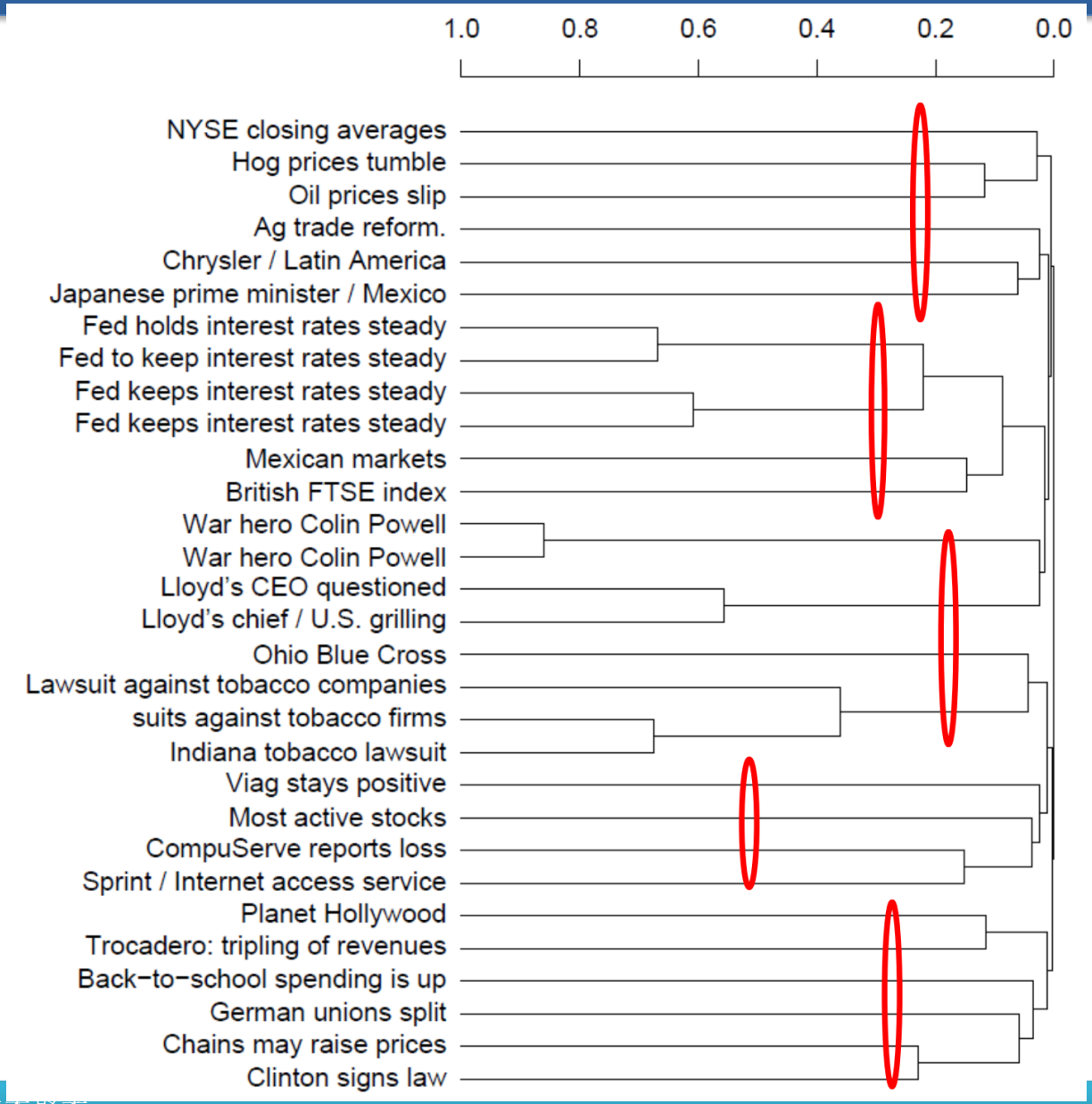
	AF	B	C	D	E
AF	-	0.3	0.2	0.1	0.3
B	0.3	-	0.4	0.5	0.7
C	0.2	0.4	-	0.3	0.5
D	0.1	0.5	0.3	-	0.4
E	0.3	0.7	0.5	0.4	-

单连接算法产生的树状图



- 注意：很多很小的簇(1 或 2 个成员) 加入到一个大的主簇上面去
- 不存在2个簇或者3个簇的非常均衡的结果

全连接算法产生的树状图



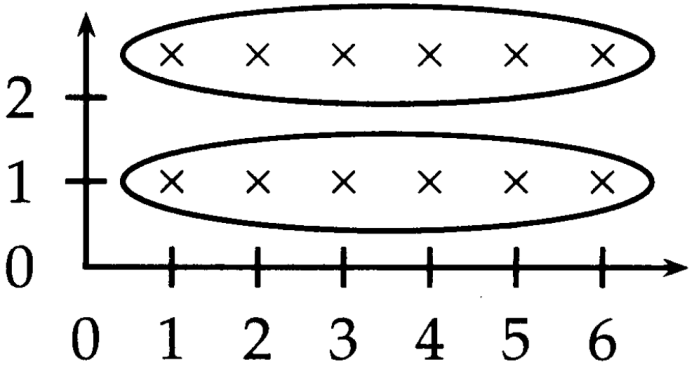
- 比单连接算法产生的树状图均衡得多
- 可以生成一个2个簇的结果，每个簇大小基本相当

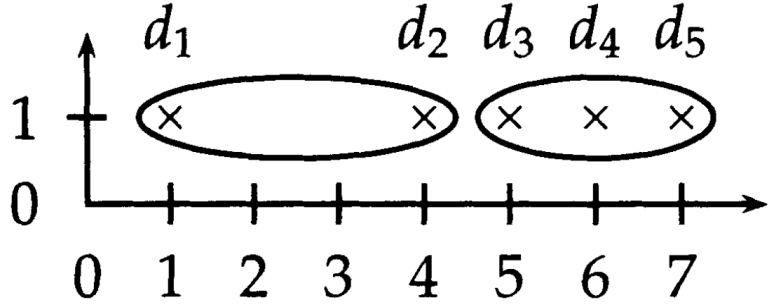
单连接和全连接聚类方法的缺点

- 单连接和全连接聚类方法将簇质量的计算过程简化成两个文档的单一相似度计算，其中
  - 单连接方法中计算的是两篇最相似的文档之间的相似度，
  - 而全连接方法中计算的是两篇最不相似的文档之间的相似度。
- 仅仅根据两篇文档来计算显然不能完全反映出簇中的文档分布情况，因此，这两种聚类方法产生的结果簇往往不是非常理想。

单连接和全连接聚类方法缺点示例

- 单连接方法的链化(Chaining)现象
  - 单连接聚类算法往往产生长的、凌乱的簇结构。对大部分应用来说，这些簇结构并不是所期望的。
- 全连接法:对离群点非常敏感
  - 全连接聚类将d_2和它的正确邻居分开----这显然不是我们所需要的
  - 出现上述结果的最主要原因是存在离群点 d_1
  - 这也表明单个离群点的存在会对全连接聚类的结果起负面影响
  - 单连接聚类能够较好地处理这种情况





组平均凝聚式算法(GAAC)

- GAAC(Group-average Agglomerative Clustering)通过计算所有文档之间(文档对)的相似度来对簇的质量进行计算，因此可以避免在单连接和全连接准则中只计算一对文档相似度的缺陷。
- GAAC也被称为组平均聚类(group-average clustering)或平均连接聚类(average-link clustering)。
- GAAC可以计算所有文档之间相似度的平均值SIM-GA，其中也包括来自同一簇的文档。当然，这种自相似度($d_n=d_m$)在这里并没有使用。
- 计算公式如下：

$$\text{SIM-GA}(\omega_i,\omega_j)=\frac{1}{(N_i+N_j)(N_i+N_j-1)}\sum_{d_m\in\omega_i\cup\omega_j}\sum_{d_n\in\omega_i\cup\omega_j,d_n\neq d_m}\vec{d}_m\cdot\vec{d}_n$$

- $\vec{d}$ 是文档d的长度归一化向量，$\cdot$是内积运算符，N_i 和N_j分别是ω_i和ω_j中的文档数目。

质心法HAC

- 簇相似度为所有簇间文档对之间相似度的平均值
- 一个原始的粗糙实现方法效率不高($O(N^2)$)，但是上述定义相当于计算两个簇质心之间的相似度：

$$\text{SIM-CENT}(\omega_i, \omega_j) = \vec{\mu}(\omega_i) \cdot \vec{\mu}(\omega_j)$$

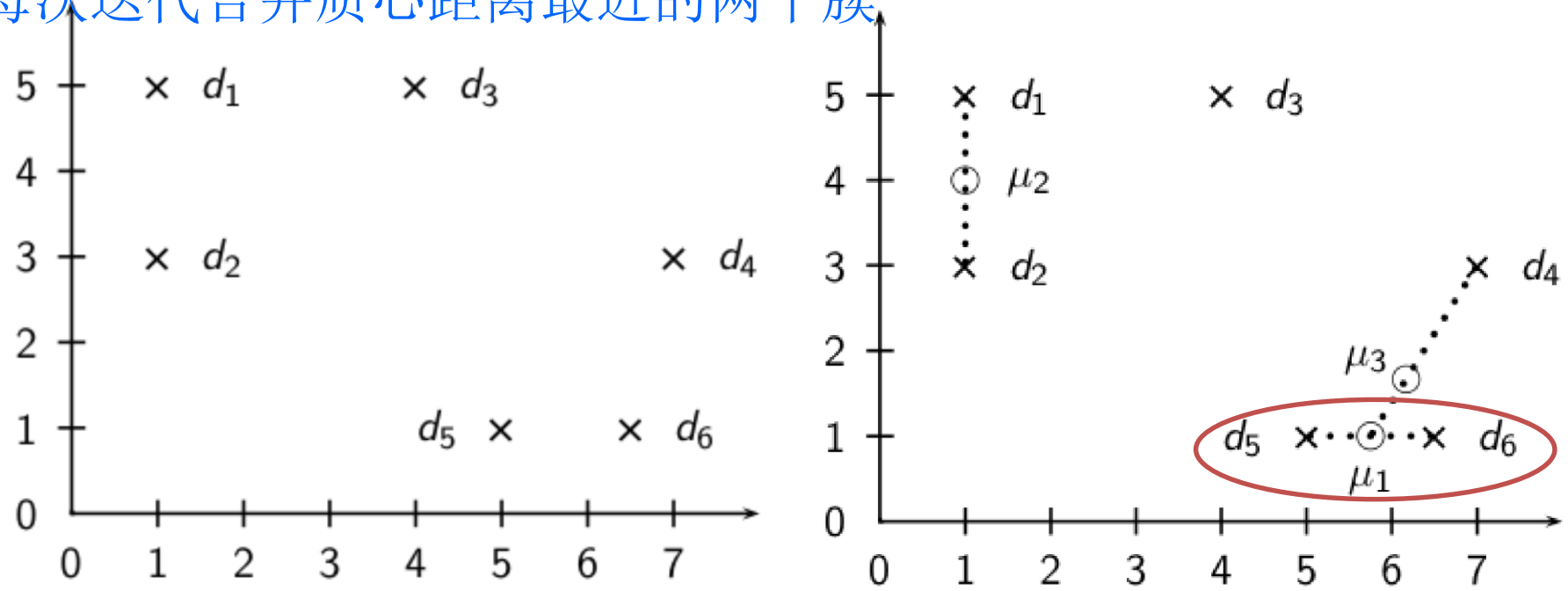
$$= \left(\frac{1}{N_i} \sum_{d_m \in \omega_i} \vec{d}_m \right) \cdot \left(\frac{1}{N_j} \sum_{d_n \in \omega_j} \vec{d}_n \right)$$

$$= \frac{1}{N_i N_j} \sum_{d_m \in \omega_i} \sum_{d_n \in \omega_j} \vec{d}_m \cdot \vec{d}_n$$

- 这也是质心HAC名称的由来
- 注意：这里是内积计算，而非余弦相似度

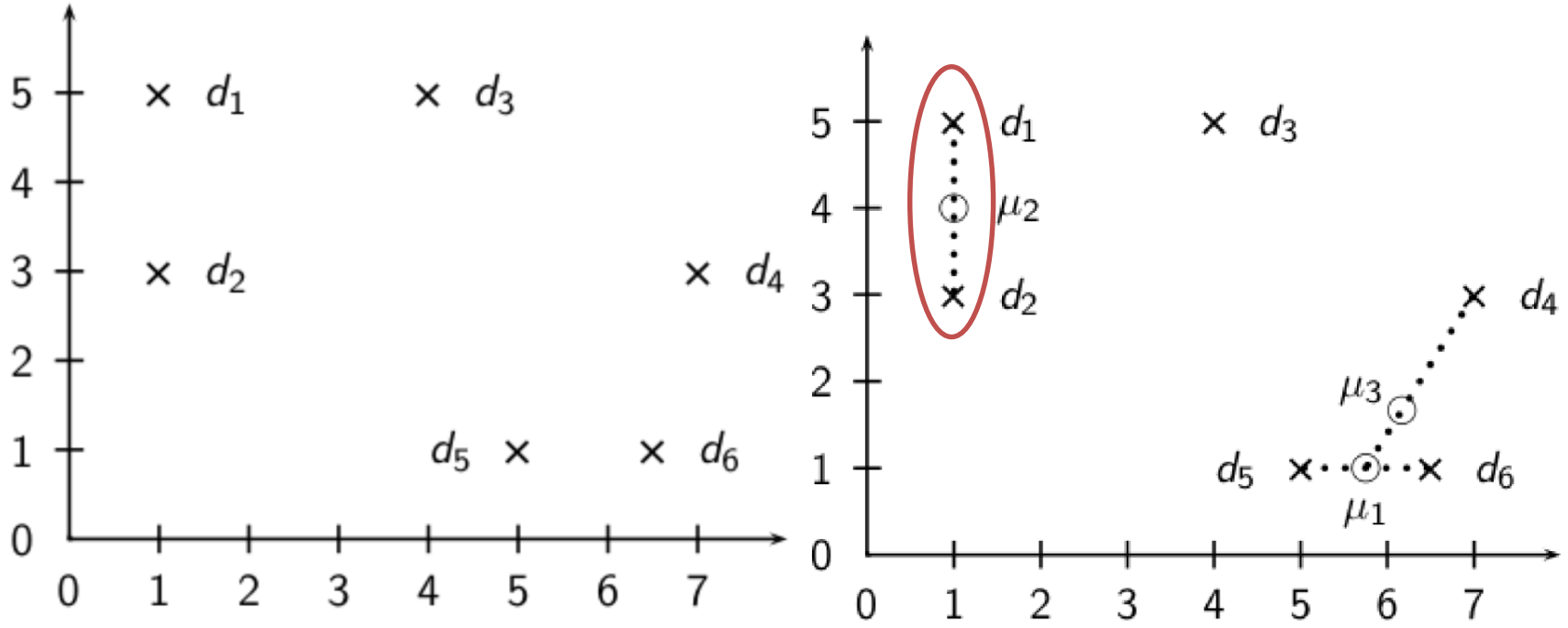
采用质心法进行聚类

每次迭代合并质心距离最近的两个簇



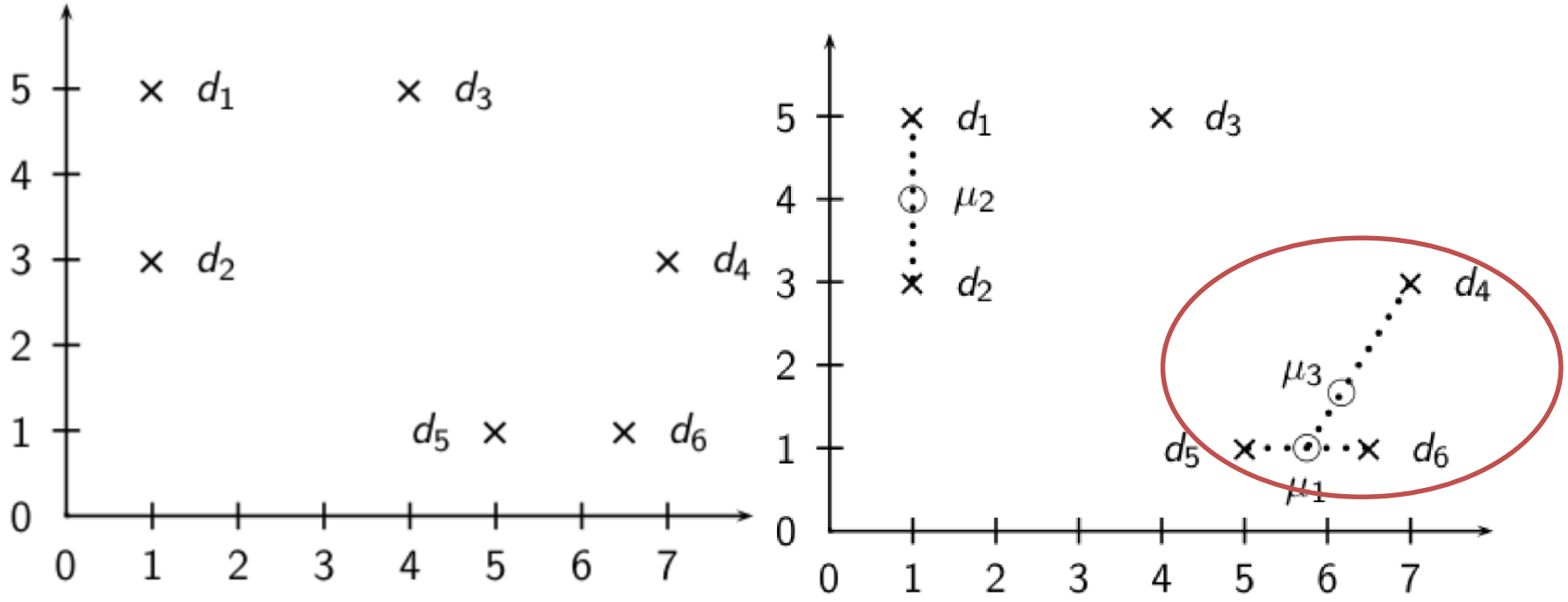
- 第1次迭代中，由于<d_5,d_6>具有最高的质心相似度，所以迭代后形成质心为μ_1的簇{d_5,d_6}

每次迭代合并质心距离最近的两个簇



- 第2次迭代中，由于<d_1,d_2>具有最高的质心相似度，所以迭代后形成质心为μ_2的簇{d_1,d_2}。

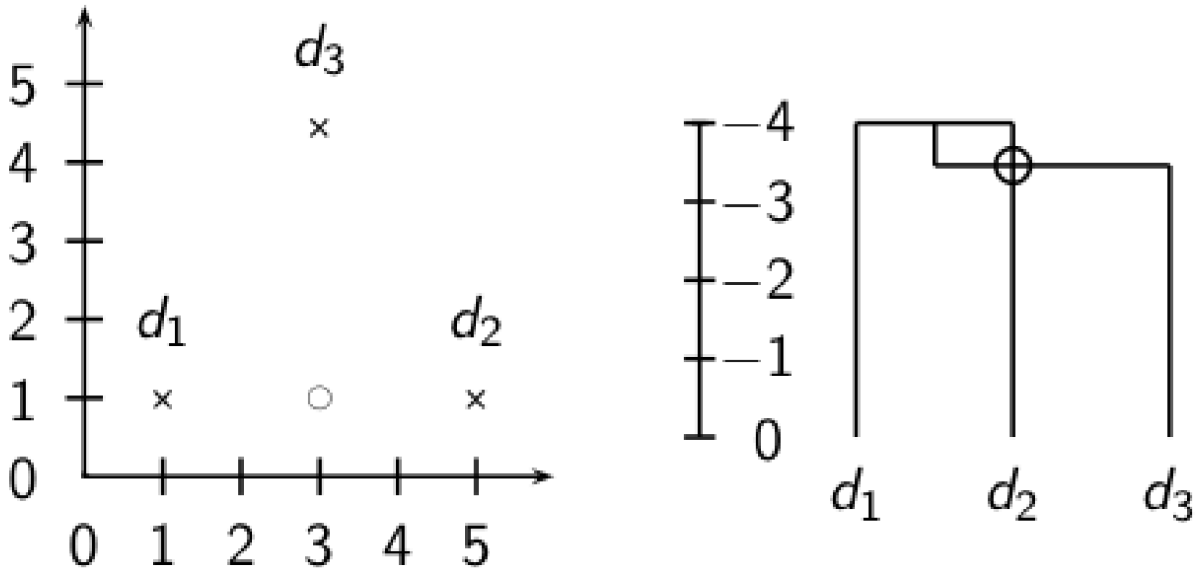
每次迭代合并质心距离最近的两个簇



- 在第3次迭代中，最高的质心相似度在μ_1和d_4之间，因此产生以μ_3为质心的簇$\{d_4,d_5,d_6\}$

相似度颠倒现象

- 与其他三种HAC算法相比，质心聚类方法不是单调的，可能会发生相似度的颠倒现象。也就是说聚类过程中相似度值有可能会下降。



- 在相似度颠倒过程中，合并过程中相似度会增加，导致“颠倒”的树状图。图中，第一次合并 $(d_1 \cup d_2)$ 的相似度是-4.0，第二次合并的相似度 $((d_1 \cup d_2) \cup d_3) \approx -3.5$.

到底使用哪一个HAC聚类算法？

- 由于存在相似度颠倒，不使用质心法
- 由于组平均**GAAC**不会受限于链化，并且对离群点不敏感，所以大部分情况下，**GAAC**都是最佳选择
- 然而，**GAAC**只能基于向量表示来计算
- 对于其他文档表示方法(或者如果仅仅提供了文档对之间的相似度)时，使用全连接方法
- 有些应用中适合用单链算法（比如，**Web**搜索中的重复性检测，判断一组文档重复并不受那些离它们较远的文档所影响)

四种HAC算法的比较

方法	结合相似度	时间复杂度	是否最优?	备注
单连接	簇间文档的最大相似度	$\Theta(N^2)$	yes	链化效应
全连接	簇间文档的最小相似度	$\Theta(N^2 \log N)$	no	对离群点敏感
组平均	所有文档相似度的平均值	$\Theta(N^2 \log N)$	no	大部分应用中的最佳选择
质心法	所有簇间相似度的平均值	$\Theta(N^2 \log N)$	no	相似度颠倒

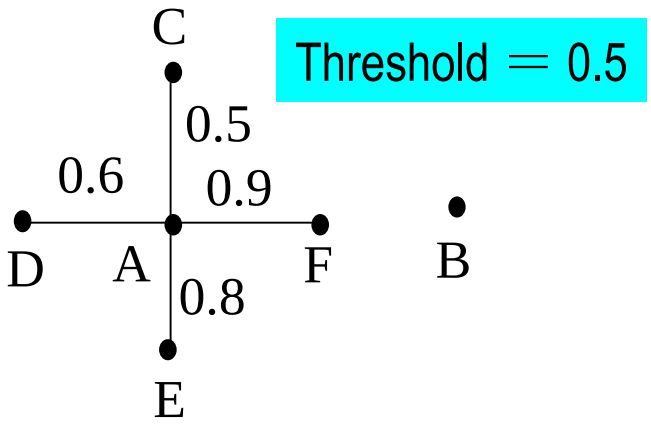
单趟聚类算法(Single-Pass Clustering)

1. 初始化处理：给定一个相似度阈值x，任取一个文档，不失一般性，记该文档为d_1，$C_1 = \{d_1\}$，$C = \{C_1\}$，$D = D - \{d_1\}$，其中C表示已经生成的聚类集。
2. 任取$d' \in D$，对所有的$C_i \in C$，计算d'与C_i的相似度S_i。
3. 找到与d'相似度最大的聚类C_j，即 $S_j > S_i$，对任何$i \neq j$。
4. 如果$S_j >= x$，则$C_j = C_j \cup \{d'\}$；//合并
5. 否则$s = |C| + 1$，$C_s = \{d'\}$，$C = C \cup \{C_s\}$。 //新增
6. $D = D - \{d'\}$。如果$D = \varnothing$，则结束计算，输出C；否则转入步骤2。

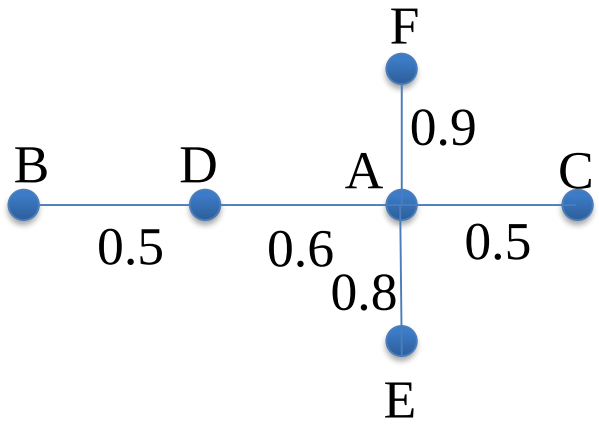
Single-Pass Clustering Example

	A	B	C	D	E	F
A	-	0.3	0.5	0.6	0.8	0.9
B	0.3	-	0.4	0.5	0.7	0.8
C	0.5	0.4	-	0.3	0.4	0.2
D	0.6	0.5	0.3	-	0.4	0.1
E	0.8	0.7	0.4	0.4	-	0.3
F	0.9	0.8	0.2	0.1	0.3	-

- 假定处理次序为 A, B, C, D, E, F
  - 单链 (即每次选择最大相似度)



- 问题：文档处理顺序影响最后聚类结果
  - 处理次序为 B, D, A, C, E, F?



本讲小结

- 聚类的概念(What is clustering?)
- 聚类在IR中的应用
- *K*-均值(*K-Means*)聚类算法
- 聚类评价
- 簇(cluster)个数(即聚类的结果类别个数)确定
- 层次聚类