本文中采用的是一种基于密度的时间序列聚类算法：Density Peaks[17]。

选择Density Peaks算法主要有以下这些原因。

首先，一个好的时间序列聚类算法需要能够忽略某些异常数据点，不仅仅是由于一些异常数据点本身就是无法聚类的，而且这些数据点的存在会以不可预测的方式影响到其他可以聚类的数据点。Density Peaks算法已经被证明能够忽略异常数据点的存在。

其次，Density Peaks算法能够处理那些会聚集成任意形状的数据集，这与假设聚类是空间中的球形的K均值及其相关的算法相反。对于本文中用到的距离度量算法DTW来说，这一点相当重要。

此外，许多聚类算法需要用户设置很多的参数，相比之下，Density Peaks只需要设置两个参数，对于用户的选择不是特别敏感，而且参数相对来说是比较直观的。

Density Peaks算法假设集群中心是由局部密度较低的邻居所包围的，并且与所有较高密度点的距离较远。因此，对于数据集中的每一个数据点，Density Peaks算法会计算两个量：局部密度、与更高密度点的距离。

数据点i的局部密度是指与它距离小于阈值的数据点的个数：

= (2.1)

其中是一个距离阈值，当时，,当时，。

与更高密度点的距离是指数据点i与任意比它密度高的点的最小距离：

(2.2)

对于具有最高密度的数据点，其设置为。

表2.1和表2.2是计算和的算法。

给定全对距离矩阵D和距离阈值，对于数据集中的每一个点i, 的计算方法如表2.1中的1-3行所示。

表2.1 计算局部密度

|  |  |
| --- | --- |
| 输入 | D,全对距离矩阵  ，距离阈值 |
| 输出 | ，数据集中n个数据点的局部密度向量 |
| 1  2  3 | for i=1:n    end |

在表2.2中，对于每一个数据点i，使用表1中计算出来的局部密度ρ，计算出比其密度高的点的列表，如行2所示。在行4中，这个列表以降序排列。从行5到行7，对于降序中每一个数据点，计算它们与更高密度点的距离。对于最高密度点的特殊情况，其计算方法如行8所示。

表2.2 计算与更高密度点的距离

|  |  |
| --- | --- |
| 输入 | D,全对距离矩阵  ，局部密度向量 |
| 输出 | ，与更高密度点的最近距离向量 |
| 1  2  3  4  5  6  7  8 |  |

给定每个点的和，Density Peaks算法计算聚类中心χ，并且基于这些中心进行聚类。聚类中心的确定有一个启发式的方法：那些ρ\*δ 较大的点更有可能是中心点。表2.3给出了聚类中心选择的算法。

表2.3 聚类中心选择算法

|  |  |
| --- | --- |
| 输入 | ，与更高密度点的最近距离向量  ，局部密度向量  k,聚类中心 |
| 输出 | ，聚类中心 |
| 1 |  |

以ρ\*δ的值进行降序，前k个数据被选为聚类中心。k的值可以手动设置。

Density Peaks算法的最后一步是聚类的分配。每一个数据点的标签与它的更高密度点列表中的最近邻的标签相同。表2.4是聚类的分配算法。

表2.4 聚类分配算法

|  |  |
| --- | --- |
| 输入 | ，聚类中心  ，与更高密度点的最近距离向量  ：基于降序排列的数据点的索引 |
| 输出 | ，聚类 |
| 1  2  3  4  5  6  7  8 |  |

1到3行中聚类中心的标签被确定好了。初始化后，如行4到8所示，数据集中的每一个点的标签与它的更高密度点列表中的最近邻数据点的标签相同。注意到这个算法允许聚类有任意的、可能是非凸的形状，不像k均值算法及其变种，仅限于输入空间的Voronoi分区。

图2.1、2.2是对Density Peaks算法的一个简单的说明。

图2.1展示出了二维空间中的18个数据点，密度最大值点是1和9，因此1和9被确定为集群的两个中心点。图2.2显示的是每个数据点的和，称为决策图。很大且较大的数据点会被选择作为一个聚类中心，剩下的数据点的标签与离其最近的更高密度点的标签一样。这样，对于每一个数据点就分配了标签。整个算法所需设置的参数只有两个：距离阈值和聚类个数。

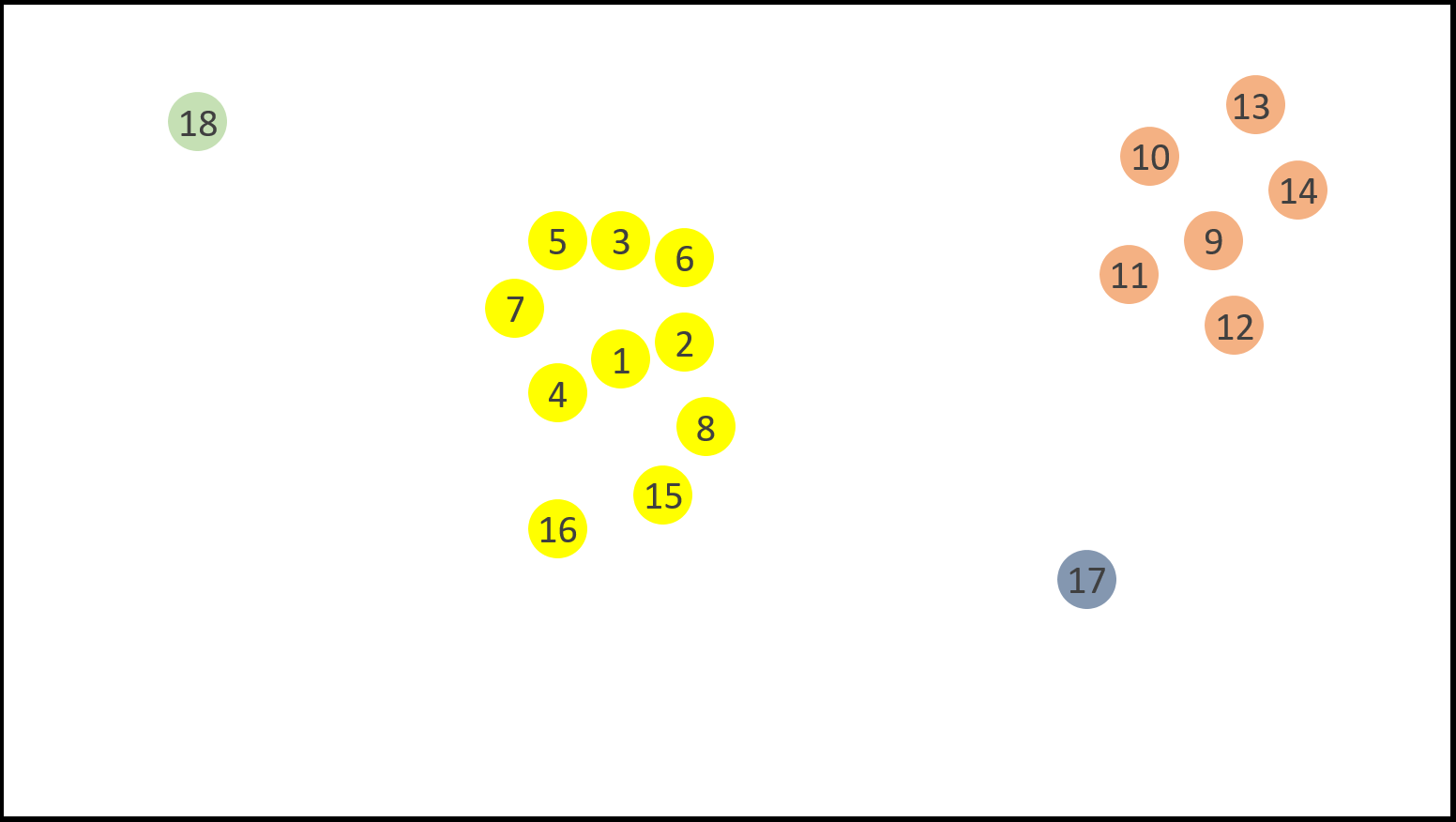


图2.1 图二维数据点的分布，数据点以局部密度降序编号

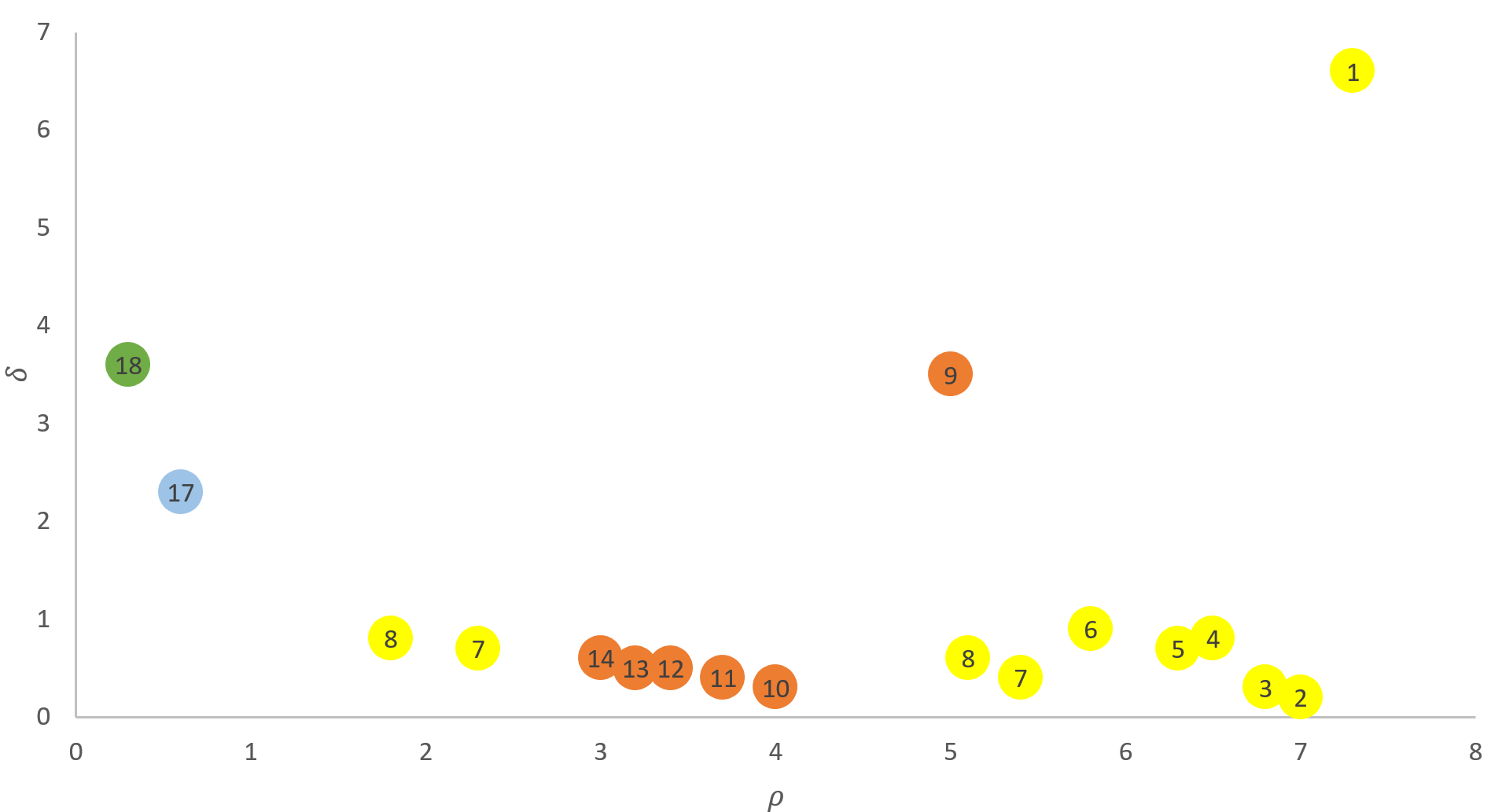


图2.2 决策图，不同的颜色代表不同的集群

**2.2 时间序列相似性度量算法**

在绪论中提到过几种时间序列相似性度量方法中，本文采用时间序列相似性度量的首选方法：动态时间弯曲距离DTW。

DTW沿着时间轴对时间序列数据进行拉伸以获得更有效的匹配，如图2.3所示。

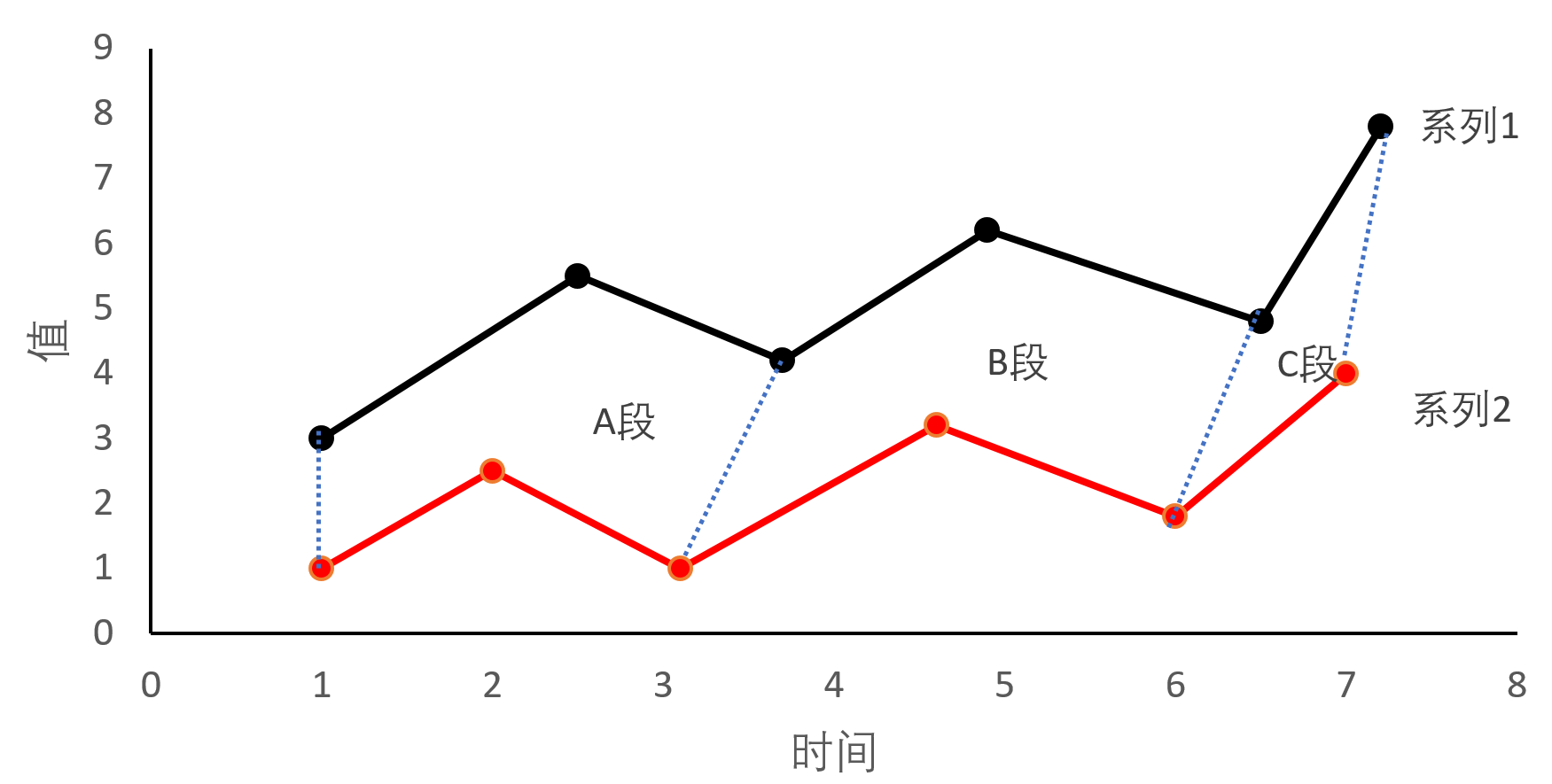


图2.3 DTW示意图

如图2.3所示，两个时间序列在A段、B段、C段上都有着十分相似的形状，但是每个段都需要经过拉伸才得到更好的匹配。DTW来自于语音识别领域，必须要对时间轴进行拉伸才能匹配不同的说话速度。欧氏距离只适用于两条具有相同长度的时间序列，但是DTW可以测量两条不同长度的时间序列之间的距离。欧式距离是在时间戳上一对一进行计算，但是DTW为了实现时间上的弯曲，会进行多对一的计算。这种多对一可以视为在时间序列中重复了某些数据点，可以自动地产生两个长度一样的一对一的时间序列。然后在弯曲后的两个时间序列的距离时可以使用任意的距离度量方法，如欧氏距离。

考虑两个不同长度的时间序列, ，,的长度分别为和。那么，如下递归定义的值：

(2.3)

的值有多种计算的方法，取决于应用的领域。例如，对于连续的时间序列数据，可以是欧式距离。对于离散的时间序列数据，可以是类别度量方法。DTW方法主要关注的是弯曲时间轴，对于数值没有影响。由于这个事实，时间弯曲可以很容易地扩展到多维空间中。

DTW算法首先初始化为0，对于，为∞，对于，为∞。算法重复执行上式，计算。通过一个简单的二重循环，将索引和的值分别由1增加到m和由1增加到n即可。该方法需要计算所有的值，因此需要次迭代。最佳的弯曲可以理解为通过m\*n网格的最优路径。

**2.3 DTW下界过滤算法**

如绪论所述，DTW下界过滤的算法包括LB\_Yi，LB\_kim，LB\_keogh等。

LB\_Yi算法是基于这样的观察：时间序列Y上的所有比时间序列X的最大（小）值点还要大（小）的点对于最终的DTW距离至少贡献了其差值的平方。

对于两个给定的时间序列和，首先计算出的最大值点和最小值点。然后计算中比更大的点与的距离之和，同样计算出中比更小的点与的距离之和，以此来构造下界距离。

图2.4是对LB\_Yi算法的一个直观上的展示。但是LB\_Yi算法与真实的DTW距离之间还有着不小的差距。Kim等人引入了更接近真实DTW距离的LB\_Kim下界函数。LB\_Kim算法分别提取两个时间序列的4个特征，包括起始数据点、结束数据点、最大值点和最小值点。然后返回相应特征的差值最大者作为下界距离。

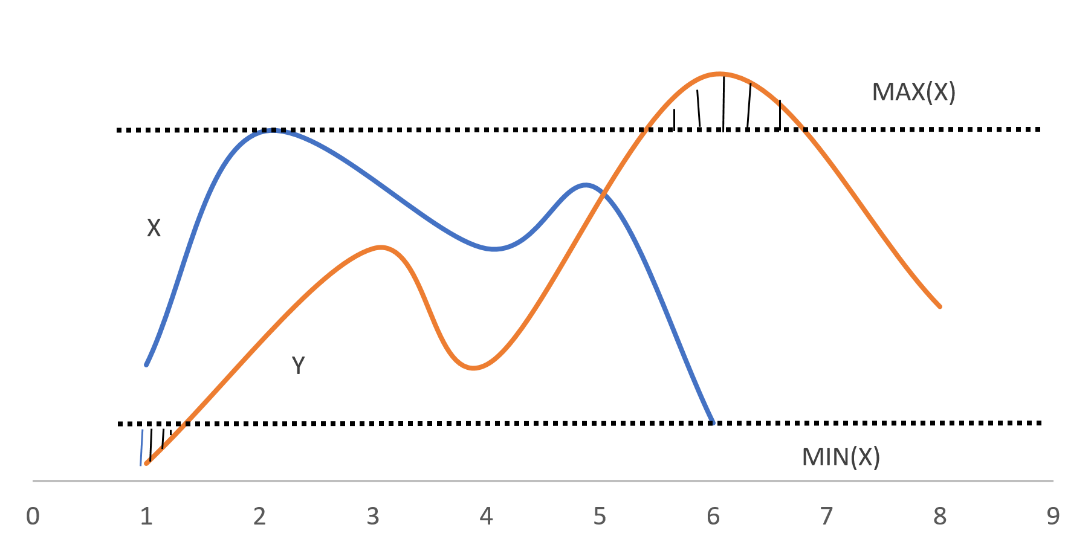


图2.4 LB\_Yi算法

假设有两个时间序列和，其LB\_Kim距离的计算方法如下：

（2.4） 如图2.5是LB\_Kim算法的一个直观的解释。

之后Keogh提出了LB\_Keogh算法，与真实的DTW距离非常接近。这也是本文中所使用的算法。

对于两个给定的时间序列和，以及分段因子r，计算两个新的序列U和L：

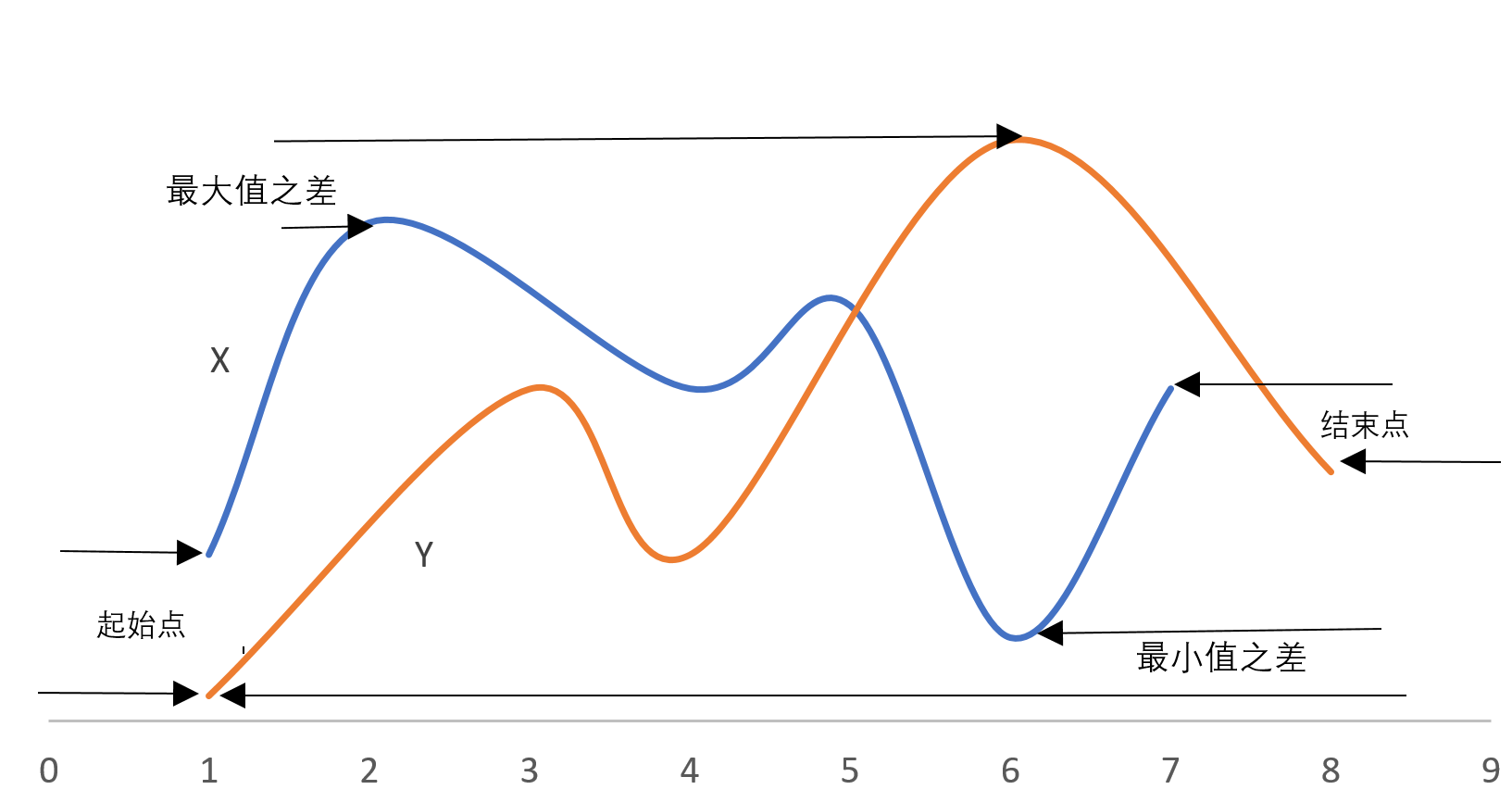


图2.5 LB\_Kim算法

(2.5)

(2.6)

U和L分别代表着上界和下界。他们分别从上面和下面形成了一个边界包裹着X。显然，U和L满足以下的不等式：

定义了U和L之后，接下来可以计算DTW距离了：  
 （2.7）

图2.6是对LB\_Keogh算法的一个直观上的解释。时间序列X被上界U和下界L包围，Y中比U大的点到U的距离与Y中比L小的点到L的距离之和就是LB\_Keogh下界距离。显然，这个距离比LB\_Yi距离和LB\_kim距离都更紧致。

LB\_Keogh算法如何进行索引？可以利用分段常数近似（Piecewise Constant Approximation ，PAA) 来表示时间序列并对其进行索引。

假设已经有一个时间序列X=X1. . .Xn,我们想要索引的空间的维度设为N（1≤N≤n）。为了方便起见，假设N是n的因子。

一个长度为n的时间序列X可以表示成N维空间中的一个向量C’=c1’. . .cN’，C’的第i个元素的计算方法如下：

(2.8)

简单来说，为了将n维的时间序列降到N维，数据被分为N个帧。计算得到落在帧内的数据的平均值形成一个向量，这个向量就是数据的缩减表示。上面这个式子中复杂的下标运算只是为了保证原始序列被分为正确数量的帧。

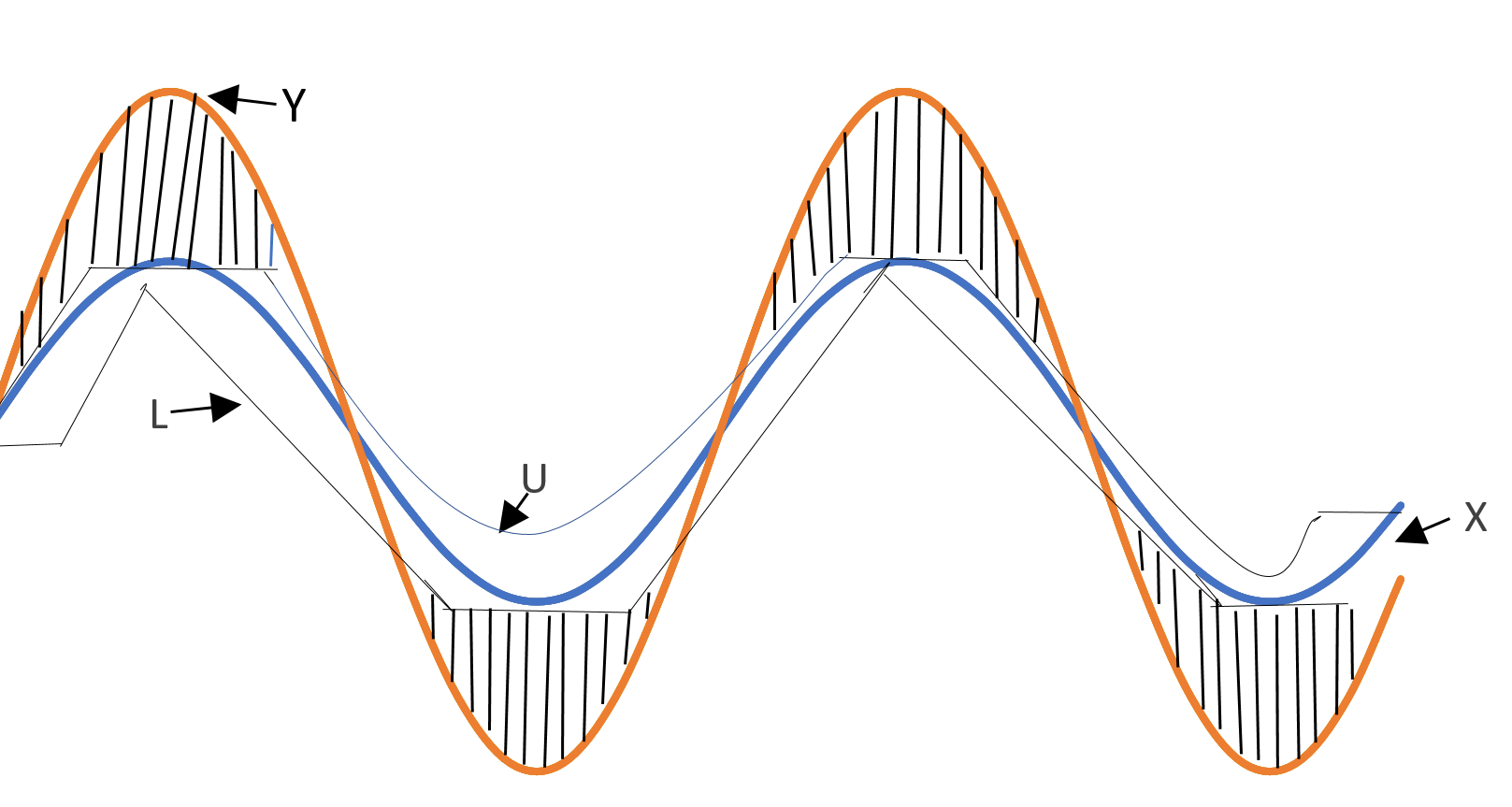


图2.6 LB\_Keogh算法

图2.7是对PAA的一个简单的说明。

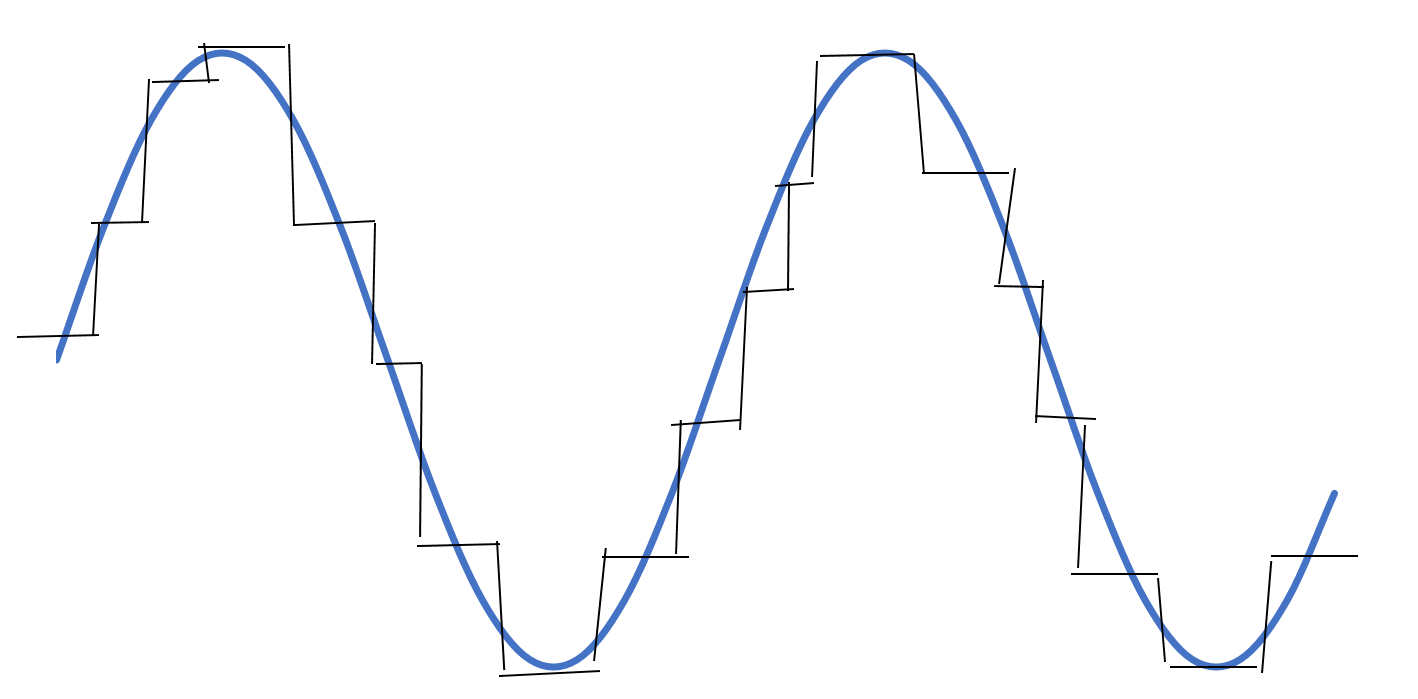


图2.7 PAA算法

给定两个原始的时间序列X和Y，我们可以利用PAA将它们转换成X’和Y’。它们的欧氏距离计算方法如下：  
 (2.9)

计算LB\_Keogh距离需要n个值，由于n可能的数量级是成百上千。然而多维索引结构在维度超过16的时候就开始快速退化，因此需要一种能够产生一个较低的N维序列的方法，其中N是一个能够被多维索引结构有效处理的值。

首先计算分段常数近似下的U和L，记做U’和L’。尽管它们也是分段常数近似，但是与之前提到的球平均值的方法是不一样的，它们的计算方法如下：  
 (2.10)

(2.11)

现在我们可以定义低维的下界距离，LB\_PAA：  
 (2.12)

**2.4 有标签聚类的评价指标**

如果有了类别标签，那么聚类也可以像其他的算法，如分类算法，计算其“准确率”。对于有标签的聚类，评级指标有Jaccard Coefficient，Fowlkes \_Mallows Index[18]， RandIndex[19]等。

给定一个n个元素的数据集,被划分成两个部分： 和。分别定义TP,TN，FP，FN：TP是相似的数据点分配到相同的簇的点对数；TN是相似的数据点分配到不同的点对数；FP是不相似的数据分配到相同的簇的点对数；FN是不相似的数据分配到不相同的簇的点对数。

Jaccard Coefficient的计算方法如下：

(2.13)

Fowlkes\_Mallows Index的计算方法如下：

(2.14)

RandIndex的计算方法如下：

(2.15)

在本文中采用的是RandIndex算法。

**2.5 Spark分布式计算框架**

Spark是一个轻量级快速处理的框架。它具有支持非循环数据流和内存计算的高级DAG执行引擎，在内存中的运行速度比Hadoop的MapReduce快100倍，就算是在磁盘上也会快10倍。Spark还提供了一堆的库，包括SQL、DataFrames、MLlib、GraphX和Spark Streaming，程序员可以在应用中无缝组合这些库。Spark集群中包括两类节点：Matser、Worker。Mater节点负责管理任务，Worker节点负责执行任务。Matser和Worker上运行的都是一个JVM进程。Spark编程中，最重要的数据结构是弹性分布式数据集RDD(Resilient Distributed Datasets)。RDD是一种只读的数据块，可以从本地文件、HDFS、内存集合等得到。RDD的操作有两种，分别是转化(transformation)和行动(action)。下表是RDD的这两种操作的基本API：

表2.5 RDD的操作

|  |  |  |
| --- | --- | --- |
| 操作 | 名字 | 作用 |
| transformation | map(func) | 根据参数中的函数作用于每一个RDD，返回新的RDD |
| flatMap(func) | 与map相似，但是输入的每个元素可以映射为一个至多个元素 |
| filter(func) | 根据参数中函数过滤掉不符合的数据，返回新的RDD |
| distinct() | 除去重复的元素 |
| union() | 求并集 |
| intersection() | 求交集 |
| subtract() | 求差集 |
| cartesian() | 求笛卡尔积 |
| action | collect() | 把数据集中所有的元素作为一个数组返回到驱动程序中 |
| count() | 返回数据集中记录的数目 |
| countByValue() | 计算各个元素在数据集中出现的次数 |
| reduce(func) | 使用函数来聚合数据 |
| foreach(func) | 对于数据集中的每一个元素执行func |

**3 时间序列聚类优化模型**

根据前面所述，为了利用基于DTW距离的Density Peaks算法对数据集进行聚类，我们需要知道全对距离矩阵。计算这些全对DTW距离矩阵所需的代价是很高的。为了降低这种计算的代价，我们需要进行加速。TADPole模型是Begum在2015年SIGKDD上提出来的一种加速模型[20]。它利用DTW的上下限修剪了不必要的距离计算，能够产生至少一个数量级的加速。对于那些仍然不能满足时间上要求的大型数据集来说，可以采用一个简单的启发式的方法：将那些不可避免的计算以其“有用度”降序排列。因此，这种方法能够快速地产生一个很好的结果。

**3.1 TADPole模型**

TADPole算法的输入是所有数据点的DTW距离的上界距离矩阵和下界距离矩阵。计算这两个矩阵的时间大约占整个聚类时间1%以下，因此是无关紧要的。本文中DTW的上界使用欧式距离，下界使用LB\_keogh距离。

这个模型中唯一需要的参数是距离阈值和聚类数量k。注意，使用上界距离矩阵和下界距离矩阵增加了200%的空间复杂度。但是这是一个无关紧要的问题，因为基于DTW的Density Peaks算法主要是时间复杂度较高，我们关注的是时间上的加速。如果真的需要减少空间的话，可以大大减少这个空间，因为下限距离矩阵中有很多元素是零，可以进行稀疏编码。

接下来，将分几个部分介绍TADPole算法的具体步骤。在3.2.1至3.2.4节中，介绍如何在计算和时，通过可靠的距离修剪，来加速TADPole算法。在3.2.5节展示如何重新排序这些计算，从而得到anytime算法的召回率递减的性质。

3.1.1在计算局部密度时剪枝

考虑在图3.1中显示的4种情况。在TADPole算法的这一步，输入是完全计算好的下界距离矩阵LBMatrix和上界距离矩阵UBMatrix。对于每一个点对(i,j)，当计算他们的局部密度时，我们通过图3.1中显示的4种情进行距离Dij计算的剪枝：

情况A：i和j是同一的

两个同一的对象i,j之间的DTW距离，与他们的欧式距离相等。只需要简单地查找上界距离矩阵即可，不需要实际的DTW计算。这种情况逻辑上是会出现的，但是极少。（见图3-1.A）。

情况B：UBMatrix(i,j)<

如果i和j之间的上界距离小于阈值dc，那么i和j之间的距离肯定小于dc（见图3-1.B），因此，我们可以在这两个物体之间进行DTW距离的剪枝。

情况C：LBMatrix（i,j）>

如果i和j的下界比截距dc大，那么这两个对象之间的距离一定在dc之外（见图3-1.C）。我们可以减少一次DTW距离的计算。

情况D：LBMatrix<且UBMatrix(i,j)>

在这种情况下，我们无法判别i和j的实际距离是否在之内，因此，只有这种情况下我们需要计算Dij（见图3.1.D）。

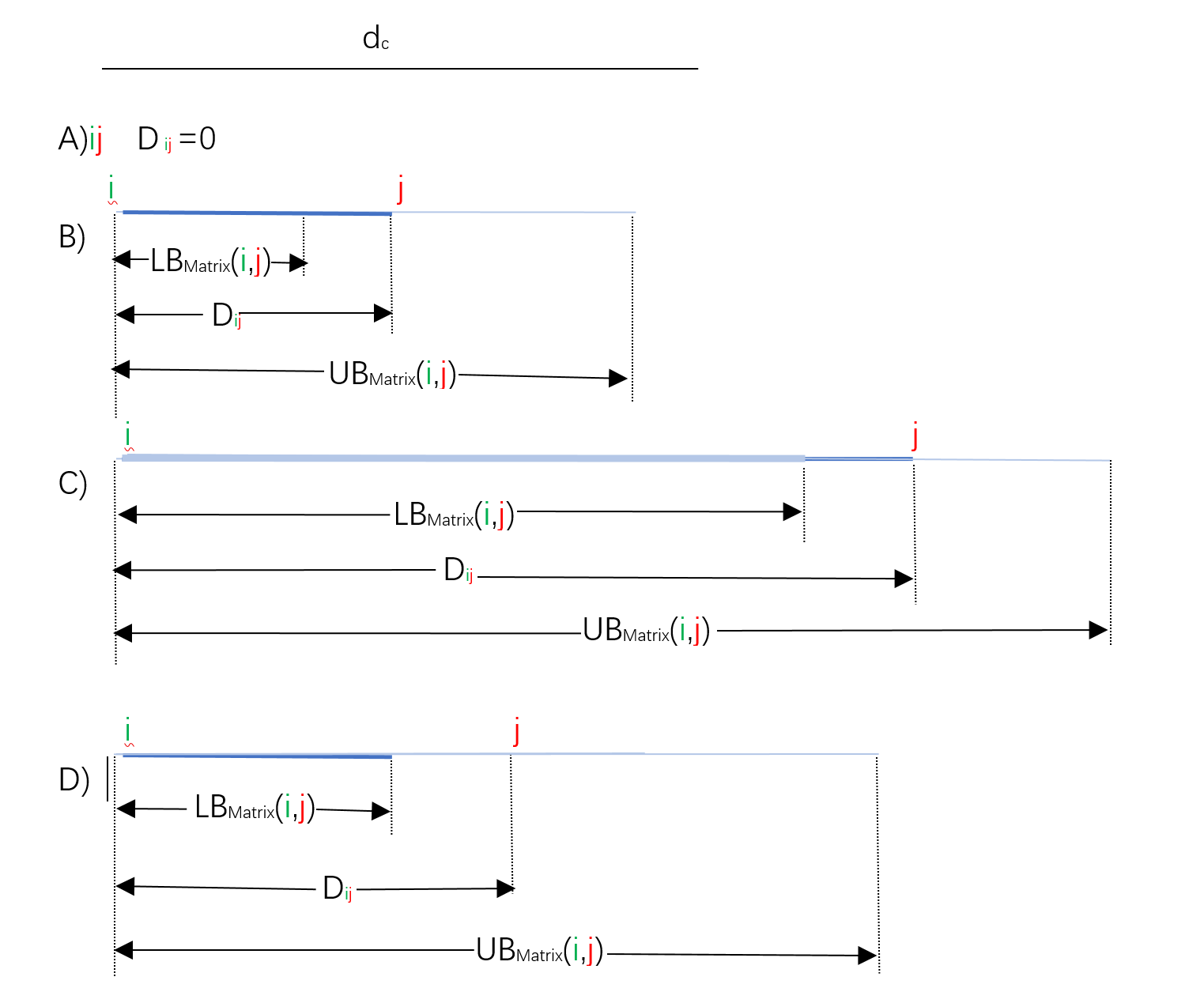


图3.1 在计算局部密度时的4种情况

基于这种想法提出了在计算局部密度时进行剪枝的算法，见表3.1:

表3.1 在计算局部密度时进行剪枝

|  |  |
| --- | --- |
| 输入 | LBMatrix，下界距离矩阵  UBMatrix，上界距离矩阵  Data，数据集  ，距离阈值 |
| 输出 | ，局部密度向量  DSparse，部分计算的DTW距离矩阵 |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | DSparse =empty  for i=1:size(Data)  objectWithin\_Dc=empty  for j=1:size(Data)  if i==j  continue;  else  if LBMatrix(i,j)== UBMatrix(i,j) //情况A  continue  else if UBMatrix(i,j)< //情况B  objectsWithin\_dc=[objectsWithin\_dc j]  else if LBMatrix(i,j)> //情况C  continue  else if LBMatrix(i,j)< and UBMatrix(i,j)> //情况D  DSparse（i,j）=calculateDist (Data(i),Data(j))  if DSparse（i，j）<  objectsWithin\_dc = [objectsWithin\_dc j]  end if  end if  end if  end for  (i)= length(objectsWithin\_dc)  end for |

可以从表3.1看出，在第5行和21行，对于数据集中所有的点对，TADPole算法检查这4种情况适用哪一种，从而判断这些对象之间的距离是否在截距之内。

情况B说明对象显然是在dc之内的，因此无需计算昂贵的DTW距离。情况A和C说明对象对之间的距离不在dc之间。只有情况D说明需要计算DTW距离。

在TADPole算法的最后一部分，对于每一个对象都计算出了局部密度。

3.1.2在计算与更高密度点的最近邻距离时剪枝

在这一步的修剪策略有两个阶段。首先，对于每个对象，我们从比它密度高的列表中找到它的最近邻距离的上限。在第二阶段，基于这些上限，我们执行修剪。当对于数据集中的所有对象，我们都已经找到他们的实际最近邻距离时，TADPole算法终止。

阶段1：上界计算

给定DSparse，和每个对象的，我们初始化其到较高密度列表的最近邻距离的上限ubi为inf。每个对象i，对于比它高的密度列表中的j，我们要么实际已经计算出DTW距离（Dij），要么已经有其上界距离（UBMatrix（i，j））。我们扫描i的较高密度的列表，如果当前的ubi> Dij，或ubi> UBMatrix（i，j），我们将当前的ubi更新为Dij（如果已经计算出来的话）或者UBMatrix（i，j）。所以我们可以保证对象i的最近邻距离不大于ubi。我们给出了这个上界计算的一个示意图，如图3.2所示。

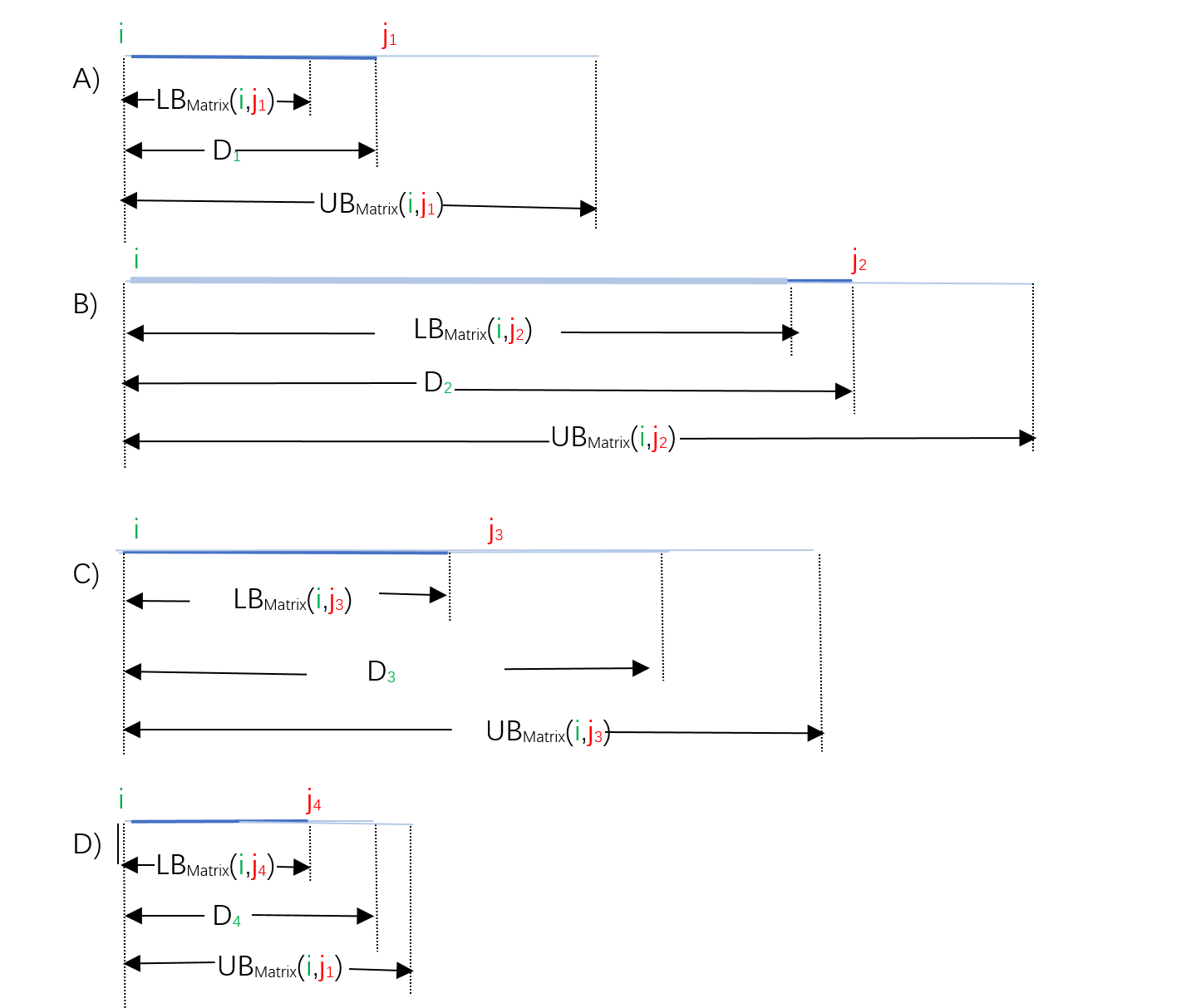


图3.2 在计算最近邻距离时的4种情况

在图3.2中，对象i的更高密度点的元素是j1-j4。假设我们只知道对象i与对象j1和j3的DTW距离，不知道D2和D4。当第一阶段开始时，,ubi被初始化为inf。现在TADPole算法扫描对象j1并且更新ubi为D1.因为UBMatrix(i,j2)和D3都比ubi大，我们不需要更新ubi。在最后一步，假设UBMatrix（i，j4)<ubi，把ubi更新为UBMatrix（i，j4)。这个ubi是对象i与更高密度点的最近邻距离距离的上界。

在表3.2中给出了最近邻距离计算的上界计算算法。首先初始化最近邻距离的上界向量ub为inf。下一步，考虑对象i的更高密度点列表中的每一个对象，我们检查是否i当前的上界可以被缩小。在第5-8行，我们查看是否i和δ\_listi(j)的实际距离已经被计算出来了，然后看这个距离是否可以缩小为ubi。如果这个距离尚未被计算出来，那么在10-12行我们查看是否可以通过用i和δ\_listi(j)的上界距离来取代ubi。

表3.2 计算最近邻距离的上界

|  |  |
| --- | --- |
| 输入 | UBMatrix，上界距离矩阵  Data，数据集  DSparse，部分计算的DTW距离矩阵  δ\_list，更高密度点的列表 |
| 输出 | ub，与更高密度点的最近邻距离的上界向量 |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | ub=inf(size(Data))  for i=1:size(Data)  for j=1:size(δ\_listi)  highDensityItem=δ\_listi (j)  if DSparse(i, highDensityItem)≠empty  if ubi> DSparse(i, highDensityItem)  ubi= DSparse(i, highDensityItem)  endif  else  if ubi> UBMatrix (i, highDensityItem)  ubi= UBMatrix (i, highDensityItem)  endif  endif  endfor  endfor |

在TADPole算法的最后，我们得到了ub，最近邻距离的上界向量。现在我们可以描述在进行与更高点的距离计算时，利用ub来进行剪枝的方法。

阶段2：剪枝

在表3.3中给出了在计算最近邻距离时进行剪枝的算法。

我们开始扫描每个对象的与比其具有更高局部密度的点的距离。在表3.3的第5行，对于每一个对象i，我们测试LBMatrix（i，δ\_listi(j)）是否比在表3.2中计算的ubi大，如果是的话，那么我们对δ\_listi(j)进行剪枝。否则，如果i和δ\_listi(j)的真实距离已经被计算出来了，那么我们考虑i的更高密度点中某个最近邻距离作为距离。如果真实距离还不知道的话，那么我们只能计算它了。最终，对于所有的对象，我们计算出了一个最近邻距离的向量。

在图3.2中，我们发现LBMatrix(i,j2)和LBMatrix(i,j3)都比ubi大。因此，我们可以不计算D2和D3。在这个例子中，我们假设我们已经知道了D1，因此，在第二个阶段的剪枝之后，我们只需要计算D4。

在TADPole的这个阶段之后，对于每一个对象，我们都有了其与更高密度点的最近邻距离。基于这种情况，对于每一个对象，给定其和 ，TADPole算法利用表2.3，计算聚类中心 ，并且基于这些聚类中心，利用表2.4中的方法进行聚类。

表3.3 计算最近邻距离时进行剪枝

|  |  |
| --- | --- |
| 输入 | LBMatrix，下界距离矩阵  Data，数据集  DSparse，部分计算的DTW距离矩阵  δ\_list，更高密度点的列表  ub，与更高密度点的最近邻距离的上界向量 |
| 输出 | δ，与更高密度点的最近邻距离向量 |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | for i=1:size(Data)  temp\_δ=empty  for j=1:size(δ\_listi)  highDensityItem= δ\_listi (j)  if LBMatrix(i, highDensityItem)>ubi  continue  else  if DSparse(i, highDensityItem) ≠empty  temp\_δ=[ temp\_δ DSparse(i, highDensityItem)]  else  DSparse(i,highDensityItem)=  calculateDist(Data(i),Data(highDensityItem))  temp\_δ=[ temp\_δ DSparse(i, highDensityItem)]  endif  endif  endfor  δ(i)=min(temp\_δ)  endfor |