Efficient Distributed Density Peaks for Clustering Large Data Sets in MapReduce

Yanfeng Zhang, Shimin Chen, and Ge Yu

摘要

Density Peaks(DP) 是最近提出来的聚类算法，它与已存在的其它聚类算法相比有很多优点，已经被广泛利用。但是，由于 DP 算法需要计算输入数据的每两点间的距离，因此带来了平方级别的计算开销，这在大数据集中是不可忍受的。在本文中，我们研究出一种高效的 DP 分布式算法。我们首先展示了原始 MapReduce 方法 (Basic-DDP) 具有很高的通信和计算开销。然后，我们提出了 LSH-DDP 近似算法，它利用 Locality Sensitive Hashing(LSH) 将数据进行分区，执行局部计算，然后聚合局部结果得到近似的最终结果。我们解决了在 DP 中使用 LSH 的几个挑战。我们利用 DP 的特性来解决一些结果值不能在本地分区中直接近似的事实。我们给出了 LSH-DDP 的形式化分析，并表明通过调整 LSH-DDP 的参数可以控制其近似质量和运行时间。实验结果表明，LSH-DDP 比 Basic-DDP 快 1.7-70 倍，比 EDDPC 快 2 倍。跟 K-means 相比，LSH-DDP 还有更好的效果。另外，LSH-DDP 可以通过适当降低精度来获得一个更高的效率。

关键词：密度峰；分布式聚类；MapReduce

1 引言

近年来，随着互联网和信息技术的不断发展，我们在日常生活中产生的数据量正在以指数级的形势在不断增长，个别互联网企业处理的数据甚至达到 PB 的级别。面对如此大规模的数据，使用以往的数据挖掘算法已经难以进行处理并从中获取所需的信息了。而聚类是数据挖掘领域中最为基础的问题之一，可以用来发现数据的簇信息和分布情况。因此如何开发出面向大规模数据的聚类算法成为了一个重要的研究方向。

目前，针对大规模数据的聚类，主要是利用分布式的思想对原来的单机版聚类算法进行改进来实现的。一般来说，首先是要通过某种策略对数据进行分区，然后将每个分区交给集群中对应的节点进行局部计算，最后将局部结果进行聚合，得到最终的全局聚类结果。该论文提出了 LSH-DDP 算法，使用局部敏感哈希 (LSH) 的方法对原始数据集进行划分，对每个分区进行局部近似计算时则使用了一种较新的聚类算法，density peak 密度峰值 (DP) 算法进行聚类，实现了 DP 算法的分布式版本，算法思想比较新颖，具有一定的复现价值。

然而，从该论文所提供的源代码中发现，该算法是使用 Hadoop 中的 MapReduce 运算框架进行实现的，所有的中间结果都需要写入磁盘，且每一步的输入都是通过读取上一步输出的文件得到的，大量的磁盘读写操作导致代码执行效率比较低。针对这一问题，本次课程的论文复现工作通过在 Spark 这一基于内存的计算框架对算法代码进行重构，并优化了中间结果的输出方式，避免了不必要的磁盘读写，提高算法的执行效率。

2 相关工作

目前，面向大规模、分布式数据的聚类算法主要可以分为四种类型，分别是：基于分区的方法[1]、基于网格的方法[2]、基于密度的方法[3]以及基于模型的方法[3]。基于分区的方法一般采用一种迭代策略，根据一些预定义的定量优化标准将数据划分为特定的簇。而基于网格的方法则是将原始数据空间量化为有限个网格，然后根据每个网格中样本的统计特征对网格进行分组，从而实现聚类。基于密度的方法首先估计对象在特征空间中的分布密度，然后搜索由低密度区域分隔的高密度区域，形成聚类。基于模型的方法首先选择一种模型，通过优化模型参数来拟合局部数据的分布，然后收集估计的参数并在中央服务器中使用，形成聚类。以下简单介绍一下有关分布式聚类算法相关的几个工作，分别是 REMOLD 算法、LDSDC 算法和 EDDPC 算法。

2.1 REMOLD 算法

REMOLD[4]是一种面向大规模数据的基于模型的聚类算法。首先，REMODL 采用一种基于局部敏感哈希 (LSH) 的均衡划分方法对大型数据集进行均匀划分。然后，对数据集的每个分区进行局部聚类，并观察到每个局部聚类的密度分布与高斯分布的形状相似，使用高斯模型来表示每个局部聚类。最后，将每个分区的局部高斯模型聚合在服务器中，REMOLD 根据这些局部高斯模型恢复全局集群。更具体地说，就是定义模型连接来测量两个模型之间的密度连通性，并用优化程序合并局部模型。由于只需传递模型参数，REMOLD 需要较低的网络传输成本。

2.2 LDSDC 算法

LDSDC[5]是一种分布式聚类算法，称为局部密度子空间分布式聚类 (LDSDC) 算法，用于聚类大规模高维数据，其动机是高维 (HD) 数据集的局部密集区域通常分布在低维 (LD) 子空间中。LDSDC 采用 local-global-local 处理结构，首先对局部密集区域 (原子簇) 进行分组，然后在每个子站点 (sub-site) 进行子空间高斯模型 (SGM) 拟合 (灵活且可扩展数据维度)，最后根据从全局站点广播的合并结果对每个子站点的原子簇进行合并。此外，该论文还提出了一种快速估计 HD 数据的 SGM 参数的方法，并给出了该方法的收敛性证明。

2.3 EDDPC 算法

EDDPC[6]是一种高效的分布式密度峰值聚类算法，是密度峰 Density peak 算法的一种分布式实现，根据每个数据对象的 ρ 值和 δ 值进行聚类。该算法在互动性、非迭代过程、不假设数据分布等方面优于其他传统聚类算法。为了提高效率和可扩展性，该算法利用 Voronoi 图和仔细的数据复制和过滤策略，来减少大量无用的距离计算成本和数据 shuffle 开销，实验结果表明，EDDPC 算法可以显著提高性能 (高达 40 倍，与 naive MapReduce 实现相比)。

3 本文方法

3.1 本文方法概述

本文首先通过 hadoop 的运算框架 MapReduce 实现了分布式密度峰聚类 (DP) 算法的 naive 版本，然后提出了局部敏感哈希 (LSH) 技术对数据进行划分，并采用分层 LSH 划分的方法解决了数据划分倾斜的问题，且提出了分区内部近似计算 ρ 与 δ 和分区间结果聚合的方法，解决边界点计算问题，最后还提出了进一步修正 δ 的策略，进一步提高了计算精度。以上改进方法对 naive 版本的算法进行优

化，得到了新的分布式聚类算法，LSH-DDP 算法。

该算法大致流程如图 1所示：

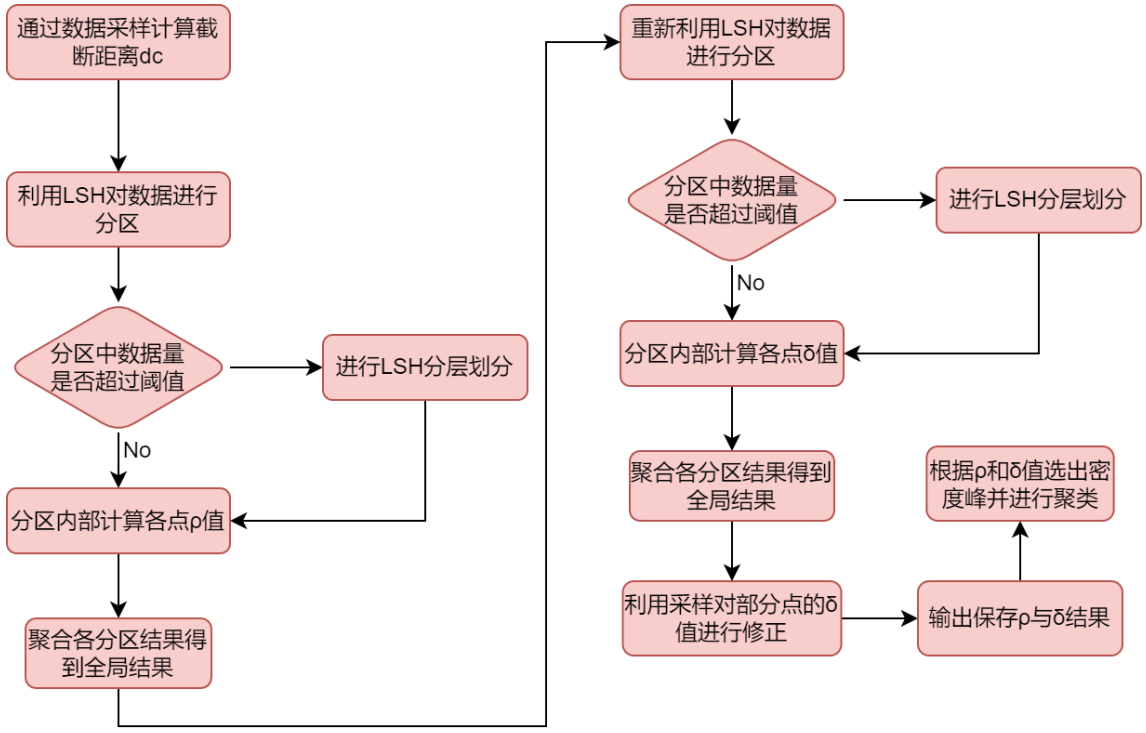

图 1: 算法流程图

3.2 DP 密度峰聚类

密度峰聚类算法 Density Peak 算法[7]是 2014 年发表在 science 上的一种较为新颖的聚类算法，该算法通过计算每个数据点的局部密度 ρ 和聚类高密度点的最短距离 δ 作为筛选密度峰和聚类的依据。

ρ 的计算公式如下：

$$\rho_i = \sum_j \chi\left(d_{ij} - d_c\right)$$

其中，若 $x < 1$，则 $\chi(x) = 1$，否则 $\chi(x) = 0$，d_{ij} 是两点间的距离，d_c 是截断距离。某个点的局部密度 ρ_i 可以理解为离该点截断距离范围以内其它的数据点的数量。

δ 的计算公式如下：

$$\delta_i = \min_{j:\rho_j > \rho_i} \left(d_{ij}\right)$$

某个点的最短距离 δ_i 可以理解为该点离其它局部密度比它大的点的最短距离。规定局部密度最大的点的 δ 距离为无穷大（或者等于第二大的 δ 值）。

我们可以发现一个事实：密度峰具有较高的局部密度，且与其它高密度点距离较远。因此，计算出每个点的 ρ 与 δ 值后，我们可以通过画出每个点关于 ρ 和 δ 的散点图（称为决策图），找出靠近右上角的点来筛选出对应的密度峰，即高密度点。接着每个非密度峰点被分配到离自己最近的密度峰对应的簇中，从而完成聚类过程。

3.3 局部敏感哈希 LSH

Datar 等人[8]对局部敏感哈希进行如下的定义：

对于数据点集的定义域 S，以 D 作为距离度量，一个 LSH 函数族 $\mathcal{H} = \{h : S \to U\}$ 被称为对于 D 是 (r_1, r_2, p_1, p_2) 敏感的，如果对于任意的点 $v, q \in S$：

• 若 $v \in B(q, r_1)$，则有 $Pr_{\mathcal{H}}[h(q) = h(v)] \geqslant p_1$

• 若 $v \notin B(q, r_2)$，则有 $Pr_{\mathcal{H}}[h(q) = h(v)] \leqslant p_2$

其中 $B(q, r)$ 是指以点 q 为中心，r 为半径的球。换句话来说，LSH 函数在两个点足够接近时，得到哈希值相等的概率足够高；反正，当两个点足够远时，得到哈希值相等的概率足够低。局部敏感哈希的本质是将高维数据降维到低维数据，同时还能在一定程度上保持原始数据的相似度不变。

本文使用欧式距离进行数据点间的距离度量，欧式距离下的 LSH 函数族定义如下：

$$h(p) = \lfloor \frac{a \cdot p + b}{w} \rfloor$$

其中，a 是一个随机向量，每一项都是服从 p 稳态分布的随机变量，由于这里使用的是欧式距离，故 $p = 2$，即为标准高斯分布。p 为数据点的向量，w 是 LSH 函数的宽度，而 b 是从 $[0, w]$ 中随机选取的一个随机数。

3.4 LSH 分区策略

显然，DP 需要一种保持局部性的分区策略，因为 ρ 的计算只依赖于距离数据点 dc 半径范围内的数据点，而 δ 的计算则是寻找距离最近的密度更高的点，距离较近的点在计算中起着更重要的作用。利用局部敏感哈希（LSH）可以使相距更近的点有更大的可能被分到同一个分区，具有相同哈希值的数据点被划分在同一个分区中。

然而，根据 LSH 函数的公式，两个相距较远的点也有可能碰巧被哈希到同一个桶中，为了降低这种情况的发生概率，本文使用多个哈希函数对数据点进行哈希，只有两个点所有哈希值都相等时，它们才会被划分在同一个分区中。但是，当增加哈希函数的数量后，两个比较接近的点有可能得到不完全一样的哈希值，从而被划分在不同的分区中，为了降低这种情况发生的概率，本文采用了多组哈希函数进行数据的分区，两个点只要有一组哈希函数得到的哈希值相同，它们就被划分在同一个分区中。多组哈希函数对于多个分区布局 (Partition Layout)，下面的近似计算和聚合算法都是基于这一概念提出的。

通过实验我们发现，虽然 LSH 分区能将位置比较靠近的数据点分配在相同的分区中，然而，如果数据分布比较集中，会出现某些分区的数据点数量过大的问题，即数据划分出现了倾斜，会严重影响算法的执行效率。为了解决这一问题，本文提出了 LayerLSH 分层 LSH 划分的方法，对数据量过大的分区重新利用 LSH 进一步划分成多个小分区，效果如图 2所示。

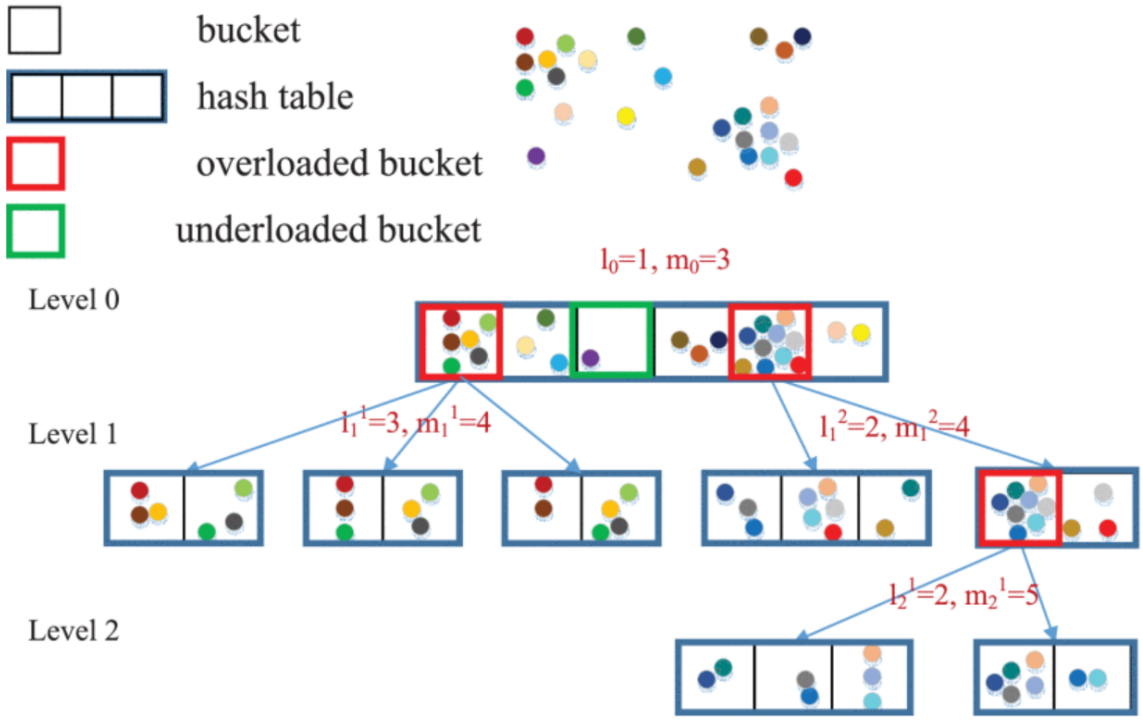

图 2: LayerLSH 示意图

3.5 局部近似计算与结果聚合

将数据通过 LSH 划分为若干个分区后，它们分别被分派到集群中不同的节点进行局部计算，包括计算分区中每个点的局部密度 ρ 和 δ 距离（通过 3.2 中的公式计算）。

值得注意的是，分区内进行的只是近似计算，不能保证绝对正确，可以利用不同分区的布局计算结果进行聚合，得到相对准确的结果。局部密度 ρ 的近似计算与结果聚合过程可以通过图 3进行说明。分布于分区内比较靠近中间位置的那些点计算得到的 ρ 数值相对准确，因为所有截断距离范围内的点都在同一个分区中，而位于分区的边缘位置上的那些数据点所计算出来的数值就会有误差，这取决于该点有多少截断距离范围内的点被划分在不同的分区中。

通过观察图 3我们知道，有一些点（例如点 p_2）在某个分区布局中属于边缘点，而在另外一个分区布局中却处于比较中央的位置，此时，通过聚合不同分区布局所计算出来的数值，具体来说就是局部密度取最大值，δ 距离取最小值，就能获得相对准确的计算结果。

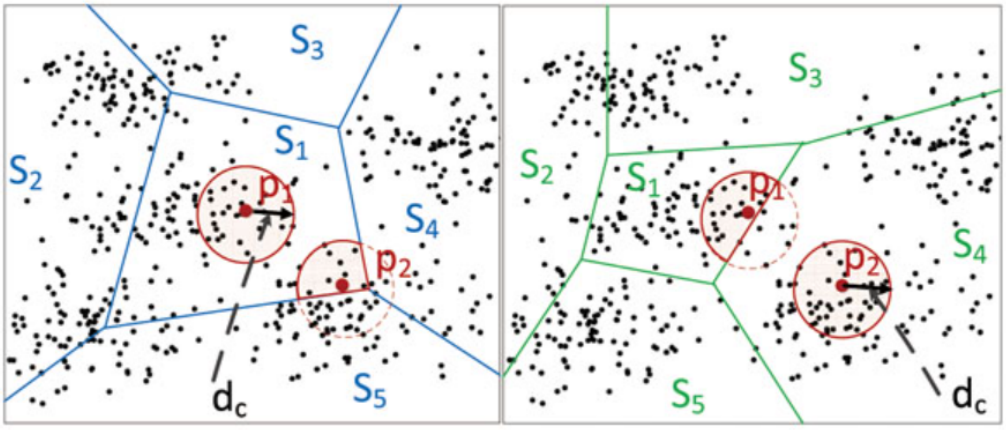

图 3: 分区计算示意图

3.6 进一步修正 δ

原论文通过形式化分析，得到 δ 的近似计算精度如下：

$$Pr\left[\delta_i^m = \delta_i\right] = 1 - \left[1 - P_{\delta_i}\left(d_{iu_i}, w\right)^\pi\right]^M$$

由此我们可以知道，某个点到它的 upslope 点（距离最近的高密度点）距离 d_{iu_i} 越大，即 δ_i 真实值越大，δ 的估计精度就越低。我们可以通过给定一个精度的下界，利用上述公式计算出满足该精度需求的最大的 d_{iu_i} 值，即 δ 的上界 γ，所有 δ 值大于 γ 的数据点，它们的 δ 值很有可能估计不正确。

修正 δ 值时，我们只需要将 δ 值大于 γ 的数据点筛选出来作为需要修正的数据点即可，然后通过抽样的方法把各个分区中的部分高密度点收集起来，重新计算它们与待修正数据点间的距离来修正各数据点的 δ。

4 复现细节

4.1 与已有开源代码对比

本文的实现参考了原论文所提供的 MapReduce 代码，使用 java 语言编写，包括 GetDc、Job1LSHRho、Job2AggregateRho、Job3LSHDelta、Job4AggregateDelta 和 Job5DeltaCorrect 六个 MapReduce 任务组成。本复现采用 Spark 内存计算框架对源代码进行重写，使用 scala 语言进行编写，将第 1、2、3 和 4 号任务进行合并作为一个分区计算与聚合任务，缩短了整体的任务链条，且使用基于内存的计算避免了大量磁盘读写操作，提高了算法整体的执行效率。

4.1.1 计算截断距离

截断距离 d_c 是 DP 算法中一个关键的参数，它指明了 3.2 计算局部密度的公式中“局部”的意义。虽然在 DP 的论文[7]中指出了，不同的 d_c 对最终的聚类结果影响不十分显著，但我们还是需要在没有数据相关先验知识的前提下选择一个相对合适的 d_c 值。由于处理的是大规模数据，自然不能进行全量计算，在原论文的实现中，计算 d_c 是采用抽样的方式实现的，首先采用分布式蓄水池抽样 (Reservoir Sampling) 对原始数据进行抽样，然后计算样本点间的距离，并进行升序排序，选择前 1% 到 2% 的距离值作为最终的截断距离即可。在本复现中，数据抽样不再使用自己定义的抽样算法，而是直接采用 Spark 提供的针对 RDD 的抽样方法，takeSample() 方法对大数据进行指定样本数的无放回抽样，抽样效率更高，可靠性大大提高。本复现获取截断距离 d_c 模块的伪代码如下所示：

Procedure 1 getDc 通过抽样计算获取截断距离 d_c.

Input: 输入路径 *in,* 输出路径 *out,* 样本容量 *size,* 距离百分比 *pc*
Output: 截断距离 *dc*

```
sc = session.sparkContext
 lines = sc.textFile(in)
 sample = lines.takeSample(withReplacement=false, size)
 distSize = (size * size - size) / 2
 distances = new Array[Double](distSize)
 distances ← compute the distance of all points pairs
 distances.sort(order=ascending)
 dc = distances(distSize * pc)
 sc.saveAsTextFile(dc, out)
 return dc
```

4.1.2 局部计算与聚合

局部计算分为计算局部密度 ρ 和 δ 距离两个部分，值得注意的是，这两个过程不能并行进行，因为 δ 值的计算依赖于 ρ 值，因此我们的计算顺序是：第一步进行一次 LSH 分区，在各分区内部近似计算每个点的 ρ 值，第二步将不同分区布局各分区计算出来的属于相同点的 ρ 值进行聚合（这里取最大值）得到最终全局 ρ 值结果，第三步再次进行一次 LSH 分区，在各分区内部近似计算每个点的 δ 值，第四步将不同分区布局各分区计算出来的属于相同点的 δ 值进行聚合（这里取最小值）得到最终全局 δ 值结果。

原文代码将以上四个步骤划分为了四个不同的 MapReduce 任务来分别执行，每一步的中间结果都要输出到分布式文件系统中，输入数据都要从上一步的输出文件中进行读取，增加了大量不必要的磁盘读写开销，且任务链条过长，导致算法执行效率较低。因此，在本复现中，上述四个步骤被合并成一个 GetRhoAndDelta 任务去执行，利用基于内存的 RDD 计算，提高计算速度，减少不必要的磁盘读写操作，大大提升了算法的执行效率。本复现局部计算与聚合模块的伪代码如下：

Procedure 2 getRhoAndDelta 通过 LSH 分区计算每个点的 ρ 和 δ 值.

Input: 输入路径 *in,* 输出路径 *out,* 截断距离 *dc,* 期望精度 *A,* 哈希函数数量 *l,* 哈希函数组数 *L*
Output: ρ 和 δ 结果写入文件 out

```
sc = session.sparkContext
 data = sc.textFile(in)
 w = getW(A, L, l)
 a ← generate L*l random vectors
 b ← generate L*l random values between 0 to w
 partitionedData = data.map(item => LSH(item, a, b, w)).partition()
 rhoResult = data.mapPartition(partition => getRho(partition, dc)).reduceByKey(Max)
 data = data.join(rhoResult)
 partitionedData = data.map(item => LSH(item, a, b, w)).partition()
 rhoAndDeltaResult = data.mapPartition(partition => getDelta(partition)).reduceByKey(Min)
 sc.saveAsTextFile(rhoAndDeltaResult, out)
```

对数据进行 LSH 分区时，利用多组哈希函数（L 组，每组 l 个哈希函数）进行哈希计算并分配分区 ID 的算法伪代码如下：

Procedure 3 LSH() 利用多组哈希函数对数据点做哈希计算并分配分区 ID

Input: 数据点 *point,* 随机向量 *a,* 随机偏移量 *b,* 函数宽度 *w*
Output: *(id, point)*

```
for i in 0 to L do
    key = i + ":"
     for j in 0 to l do
        hashValue = floor((a[i][j]*point+b[i][j])/w)
         key = key + hashValue + ":"
    end
    res ← (key,point)
end
return res
```

分区内部进行 ρ 和 δ 的计算时，需要判断分区内数据量的大小，若超出指定的大小，则需要对分区内的数据继续进行 LayerLSH 分层划分，避免数据倾斜的问题。分区计算 ρ 值的伪代码如下：

Procedure 4 getRho() 分区内部计算每个点的局部密度 ρ

Input: 分区内所有数据点 *partitiion,* 截断距离 *dc*
Output: *(pointId, ρ)*

```
if partitioin.size < limit then
    for point in partition do
        rho = 0
        for other in partition - point do
            if distance(point,other) < dc then
                rho = rho + 1
            end
        end
        res ← (point.id,rho)
    end
end
else
    res ← repartition and compute rho
end
return res
```

分区中 δ 的计算类似，这里不再赘述 getDelta() 的代码。

4.1.3 修正 δ

δ 的修正步包括两个抽样过程，分别是对需要修正 δ 的数据点的抽样以及对密度较高的数据点的抽样。这里涉及到两个问题：什么样的数据才需要对其 δ 值进行修正，以及密度达到多高的点才能被认为是高密度点。

对于第一个问题，原论文中给出了一种解决方案，就是根据 3.6 中得出的 δ 精度估计公式，只要给定一个预期的最低精度下界，以及哈希函数的相关参数，即可利用优化方法（本文使用的是信任区间狗腿法 DogLeg），求解出对应的最短距离 γ。如果观察到某个点的 δ 大于这个 γ，则我们可以认为该点的 δ 近似值是不准确的，需要进一步的修正，以达到预期的精度要求。

对于第二个问题，原文给出了一个粗略的抽样率公式，其概率大小跟数据点本身的密度以及设定好的密度峰最低密度有关，大概思想是密度越高的点被选为高密度点样本的概率越高，但这一思想并没有在源代码中体现，因此本复现工作将其在代码中进行了实现。

修正 δ 的伪代码如下：

Procedure 5 DeltaCorrect 通过抽样修正部分数据点的 δ 距离.

Input: 输入路径 *in,* 输出路径 *out,* 函数宽度 *w,* 期望精度 *A,* 哈希函数数量 *l,* 哈希函数组数 *L*
Output: δ 修改后的结果写入文件 out

```
sc = session.sparkContext
 datas = sc.textFile(in)
 deltaBound = getDeltaBound(A, L, l, w)
 deltaSample = datas.filter(point.delta > deltaBound)
 higherSample = datas.sample(point => {
 rate = getSampleRate(point, density peak)
 point.sample(rate)})
 correctedRes = deltaSample.map(point => computeDelta(point, higherSample))
 sc.saveAsTextFile(correctedRes, out)
```

4.2 实验环境搭建

本实验使用一台虚拟机进行实验环境的搭建，虚拟机硬件配置如表 1所示。

表 1: 硬件环境信息

设备	配置
内存	12GB
处理器	6 核
硬盘	100GB

本实验通过在 Linux 上搭建 spark 平台进行程序的测试与运行，使用到的软件信息如表 2所示。

表 2: 软件环境信息

系统/软件	版本信息
CentOS	7
java	1.8
scala	2.12.15
Hadoop	3.2.4
Spark	3.2.3

4.3 代码结构说明

本复现的代码主要包括三部分，分别是实现算法主流程的类，算法中使用到的实用类，以及画图的类，代码结构如图 4所示。

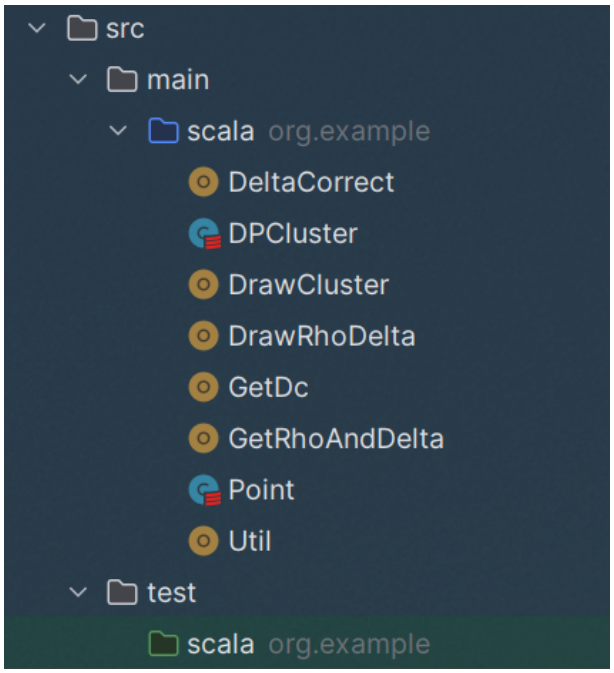

图 4: 代码结构图

GetDC：通过抽样近似计算出指定数据集的截断距离 d_C

GetRhoAndDelta：通过 LSH 对数据进行划分，分布式近似计算每个点的密度 ρ 和距离 δ 指标

DeltaCorrect：通过抽样修正部分数据点的 δ 值

Point：表示数据点的类

DPCluster：实现分区内局部近似计算的类

Util：一些公用的方法

DrawRhoAndDelta：画决策图 DrawCluster：画聚类效果图

4.4 创新点

本复现的创新点主要包括三部分：

• 使用基于内存计算的 Spark 框架对源代码进行重写，避免了 MapReduce 实现中大量不必要的磁盘读写操作，提高了算法执行效率；

• 使用 RDD 的有向无环图对任务编排进行优化，缩短了整体的任务链条，进一步提升了算法的运行效率；

• 在计算截断距离时使用 Spark 自带的抽样方法来代替原来代码里面自己实现的分布式蓄水池抽样算法，提升了可靠性和稳定性。

5 实验结果分析

本实验首先使用 aggregate 数据集进行测试，分为四大模块，分别是计算截断距离 GetDc，LSH 分区计算 GetRhoAndDelta，进一步修正 DeltaCorrect 和完成聚类并画图的模块。

5.1 GetDc 模块

通过 Saprk-submit 提交任务并指定参数，这里指定输入路径和输出路径都为 HDFS 分布式文件系统中的目录，同时还指定抽样数量为 200、数据的维度为 2 维、距离百分比为 4% 还有数据分隔符为制表符：

```
[root@node01 ~]# spark-submit --class org.example.GetDc --master local[2] lsh_ddp-fatjar-1.0.jar
 hdfs://172.31.226.70:9000/dataset/aggregation.txt hdfs://172.31.226.70:9000/output/dc_agg 200 2
 4.0 -d \\t
```

图 5: GetDc 提交任务

运行结果如下，可以看到，程序计算出来的截断距离 d_c 为 2.777，样本数为 200 个，选取的距离值为排序后的 19900 个距离值中第 796 个距离，结果写入文件。

```
, 32840, None)
22/12/08 17:06:31 INFO BlockManager: Initialized BlockManager: Bl
840, None)
sample size: 200
position: 796/19900
dc: 2.7771388153997623
[root@node01 bigdata]#
```

图 6: GetDc 运行结果

查看写入结果文件的目录：

```
[root@node01 bigdata]# hdfs dfs -ls /output/dc_agg
Found 3 items
-rw-r--r--   3 root supergroup          0 2022-12-08 17:06 /output/dc_agg/_SUCCESS
-rw-r--r--   3 root supergroup          0 2022-12-08 17:06 /output/dc_agg/part-00000
-rw-r--r--   3 root supergroup         22 2022-12-08 17:06 /output/dc_agg/part-00001
```

图 7: GetDc 结果写入文件

5.2 GetRhoAndDelta 模块

通过 Saprk-submit 提交任务并指定参数，这里指定输入输出路径、数据维度、截断距离以及分隔符：

```
[root@node01 bigdata]# spark-submit --class org.example.GetRhoAndDelta --master local[2] lsh_ddp-
fatjar-1.0.jar hdfs://172.31.226.70:9000/dataset/aggregation.txt hdfs://172.31.226.70:9000/output
/rho_delta_agg 2 2.7771388153997623 -d \\t
```

图 8: GetRhoAndDelta 提交任务

运行结果如下，可以看到，程序利用 LSH 将数据划分为多个分区，每个分区只包含元素数据集中的一部分数据，并在分区中各自进行指标的计算，接着进行聚合，得到最终的 ρ 和 δ 结果，并把结果写入指定的目录中。

```
22/12/08 17:12:51 INFO BlockManager: Initialized BlockManagerId(driver, node01, 330
33, None)
w: 28.51981910253541
generating a and b on rho step...
assigning partition ID on rho step...
LSH partition on rho step:
Map(2:0:2:0 -> 231, 0:1:2:0 -> 16, 1:0:0:0 -> 177, 0:2:1:1 -> 19, 2:0:2:1 -> 43, 2:0:1:0 -> 221,
0:1:1:0 -> 226, 0:1:0:0 -> 10, 2:0:3:0 -> 218, 0:1:1:1 -> 155, 1:-1:1:0 -> 224, 1:-1:0:0 -> 35, 1
:0:1:0 -> 131, 0:0:0:1 -> 18, 2:0:1:1 -> 75, 0:1:0:1 -> 323, 1:-2:1:0 -> 221, 0:0:0:0 -> 21)
local computing rho...
get the finalRho!
generating a and b on delta step...
joining tow RDD to form (pid, (maxRho, point))...
assign partition ID on delta step...
LSH partition on delta step:
Map(1:0:0:0 -> 679, 2:0:0:0 -> 207, 0:2:0:0 -> 337, 0:3:0:0 -> 44, 0:1:0:0 -> 381, 2:0:0:1 -> 581
, 1:0:-1:0 -> 109, 0:0:0:0 -> 26)
local computing delta...
get the final delta!
saving the result...
```

图 9: GetRhoAndDelta 运行结果

5.3 DeltaCorrect 模块

通过 Saprk-submit 提交任务并指定参数：

```
[root@node01 bigdata]# spark-submit --class org.example.DeltaCorrect --master local[2] lsh_ddp-
fatjar-1.0.jar hdfs://172.31.226.70:9000/dataset/aggregation.txt hdfs://172.31.226.70:9000/outp
ut/rho_delta_agg hdfs://172.31.226.70:9000/output/correct_agg 2 2.7771388153997623 -d \\t
```

图 10: DeltaCorrect 提交任务

运行结果如下，可以看到，程序计算得到的需要修正的数据点 δ 下界为 2.237，高密度点抽样的最小密度为 180.0，修正后的结果写入文件。

```
w: 28.51981910253541
delta_bound: 2.237255919802498
min_rho: 180.0
loading data...
sample the high density points...
filtering and correct the delta...
corrected the all the delta!
saving the result...
```

图 11: DeltaCorrect 运行结果

查看写入结果文件的目录：

```
[root@node01 bigdata]# hdfs dfs -ls /output/correct_agg
Found 9 items
-rw-r--r--   3 root supergroup          0 2022-12-08 17:20 /output/correct_agg/_SUCCESS
-rw-r--r--   3 root supergroup       4409 2022-12-08 17:20 /output/correct_agg/part-00000
-rw-r--r--   3 root supergroup       4434 2022-12-08 17:20 /output/correct_agg/part-00001
-rw-r--r--   3 root supergroup       4470 2022-12-08 17:20 /output/correct_agg/part-00002
```

图 12: DeltaCorrect 结果写入文件

5.4 聚类与画图模块

首先根据 DeltaCorrect 模块得到的修正后的 ρ 和 δ 结果画出决策图，即每个数据点以局部密度 ρ 作为横坐标，δ 聚类作为纵坐标画出来的散点图：

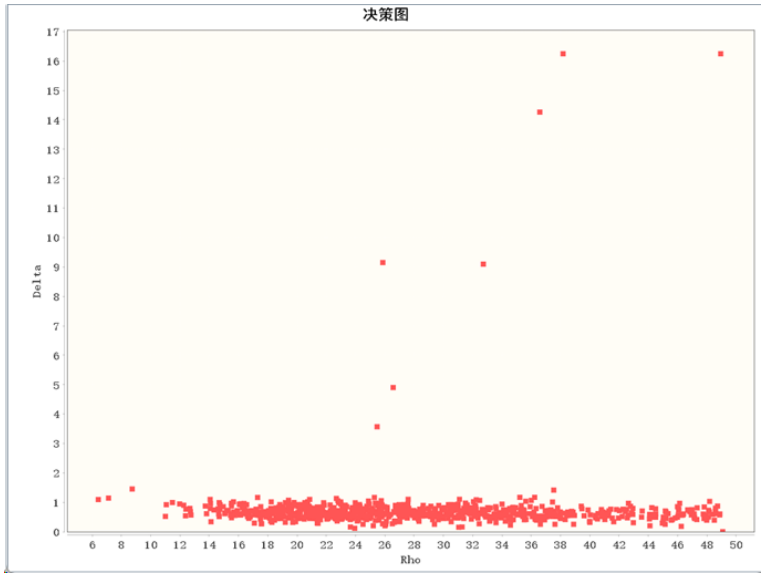

图 13: 决策图

通过决策图选出所有的密度峰后（通常位于决策图的右上方），每个密度峰作为一个簇中心分配簇 id，然后对所有非密度峰的点进行簇的分配，得到以下聚类结果：

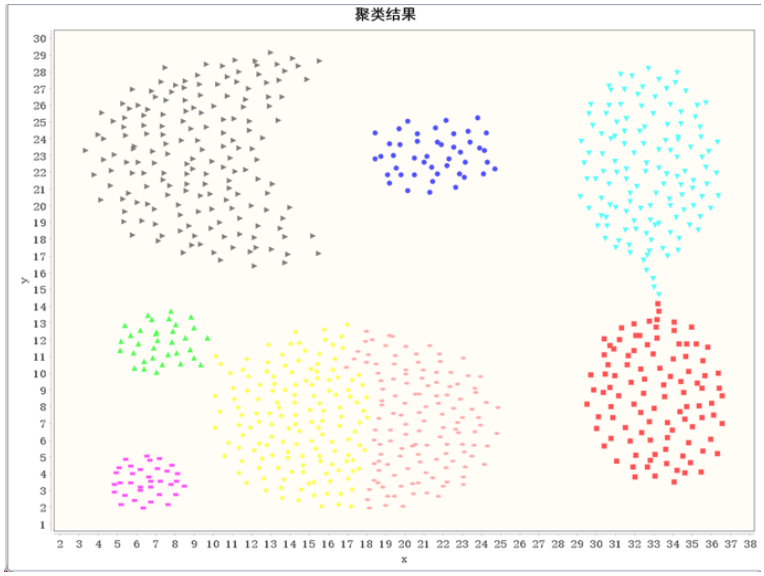

图 14: 聚类结果图

S2 数据集聚类结果如下：

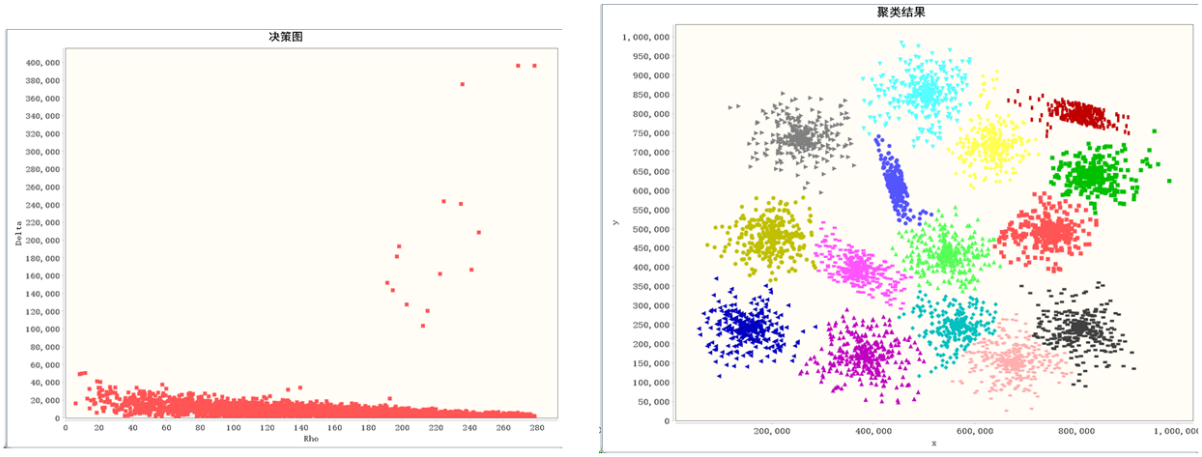

图 15: S2 数据集聚类结果

可以看到，本复现对于数据集的聚类效果比较好，能达到预期的功能。

6 总结与展望

本次论文复现工作预期完成的内容有四个方面：1. 通过 spark 框架改写源代码（基于 MapReduce 框架），实现分布式的 density peak 密度峰聚类算法。2. 实现 LayerLSH 分层 LSH 分区算法，解决数据倾斜带来的效率问题。3. 实现超参的自动学习和调优算法。4. 通过实验对复现效果进行测试，与原论文的结果进行比较。实际完成了第 1、2、4 点（第 3 点由于难度较大，时间有限尚未完成），基本完成预期计划，且经过测试表明，复现算法的聚类效果表现优秀，且执行效率有所提高，

然而，经过测试发现本复现工作还存在一些问题，例如在处理一些维度和数据量比较大的数据集时（例如跑 KDDCUP04Bio.txt 数据集，74 dimensions, 145,751 instances ），会出现运行不完或者报内存溢出错误的情况。通过分析，初步判断是由于本实验使用单机跑 spark 资源不足，且算法设计问题，资源利用率低导致的。

下一步，需要继续优化算法提高资源利用率，实现参数的自动学习和调优，避免人工干预，并在真正的集群上重复实验，继续寻找新的改进点对算法进行优化。

参考文献

[1] GAN W, LIN J C W, CHAO H C, et al. Data mining in distributed environment: a survey[J]. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2017, 7(6): e1216.

[2] BERKHIN P. A survey of clustering data mining techniques[G]//Grouping multidimensional data. Springer, 2006: 25-71.

[3] ROSATO A, ALTILIO R, PANELLA M. Recent advances on distributed unsupervised learning[C]// International Workshop on Neural Networks. 2015: 77-86.

[4] LIANG M, LI Q, GENG Y A, et al. REMOLD: an efficient model-based clustering algorithm for large datasets with spark[C]//2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS). 2017: 376-383.

[5] GENG Y A, LI Q, LIANG M, et al. Local-density subspace distributed clustering for high-dimensional data[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(8): 1799-1814.

[6] GONG S, ZHANG Y. EDDPC: An efficient distributed density peaks clustering algorithm[J]. Journal of Computer Research and Development, 2016, 53(6): 1400-1409.

[7] RODRIGUEZ A, LAIO A. Clustering by fast search and find of density peaks[J]. science, 2014, 344(6191): 1492-1496.

[8] DATAR M, IMMORLICA N, INDYK P, et al. Locality-sensitive hashing scheme based on p-stable distributions[C]//Proceedings of the twentieth annual symposium on Computational geometry. 2004: 253-262.