使用 MapReduce 进行层次密度聚类

吴东彤

摘要

基于层次密度的聚类是探索性数据分析的有力工具，在数据集的理解和组织中发挥着重要作用。但是，由于分层聚类方法的计算复杂度在待聚类对象数量上具有二次下限，其在大型数据集上的适用性受到限制。MapReduce 是一种流行的编程模型，可以加速在大型分布式数据集上运行的数据挖掘和机器学习算法。但分层聚类算法本身就很难与 MapReduce 进行并行化。在论文中，作者初步讨论了一个解决方案，它是基于一个精确的，但非常需要计算的随机块并行化方案。为了能够使用 MapReduce 有效地将基于分层密度的聚类应用于大型数据集，作者提出了一种不同的并行化方案，该方案基于一种更快的递归采样方法计算近似的聚类层次结构。这种方法基于 HDBSCAN*，是最先进的基于分层密度的聚类算法，结合了一种称为数据气泡的数据汇总技术。在大量数据集上，从运行时间和近似质量两个方面对所提方法进行了评估，显示了其有效性和可扩展性。

关键词：密度层次聚类；MapReduce；大数据

1 引言

本课程实验复现的论文是 Joelson 等人写的“Hierarchical Density-Based Clustering Using MapReduce”[1]，该篇论文于 2021 年发表于 IEEE Transactions on Big Data 上。

基于密度的聚类是一类算法，直接或间接地涉及到密度估计问题，这种估计以某种方式将集群的“高密度区域和低密度区域隔开”，其中“高”和“低”密度的概念取决于某种类型的密度阈值[2]。基于分区密度的聚类算法，如 DBSCAN，有基本的要求。它们需要一个用户定义的关键参数——密度阈值，然而通常情况下，不可能通过单一的全局密度阈值来同时检测不同密度的集群，同时，单一的平面聚类无法描述位于不同密度水平上的嵌套簇之间可能存在的层次关系。因此，不同密度水平的嵌套簇只能用基于层次密度的聚类方法[3-5]来描述，这些方法在不同的粒度和分辨率上提供了对数据集更详细的描述。

然而，分层聚类算法通常比分区算法更需要计算。实际上，分层聚类方法的计算复杂度在待聚类对象数量上有一个二次下限。随着在越来越多的实际应用程序场景中生成了非常大的数据库，使用分层聚类分析此类数据变得具有挑战性。在这些情况下，并行化可以提高计算性能并扩大算法的适用性。为了实现这一目标，必须使用允许及时分析大量数据的编程模型，同时随着数据分析需求和数据库规模的增长，确保不发生故障的安全性和可伸缩性。这样一个编程模型 MapReduce 在[6]中被提出，该模型是一种抽象，允许简化分布式和可伸缩的编程。

MapReduce 编程模型要求算法将任务分成 map 和 reduce 两个主要函数，并在它们上面施加一个执行流。这些函数在数据集的分布式部分上独立地执行作业，因此每个作业不能与正在执行的其他作业共享关于其数据部分的信息。这种约束可能成为传统聚类算法的负担，特别是分层聚类算法，因为它们通常基于数据对象之间的成对比较，这可能需要重复映射整个数据集，映射次数与数据集大小的平方成正比。这种计算往往需要大量的计算机能力、网络负载和处理时间，特别是对于大量的数据。

因此，很少人提出基于 MapReduce 的层次聚类技术，且提出的大多是针对 Single-Linkage。然而，这些技术有缺点，即大量的分区和重复计算，使得它们的实现在实践中既具有挑战性又效率低下。

为了使用 MapReduce 有效地将基于分层密度的聚类应用于大型数据集，论文提出了一种替代的并行化方案，称为 MapReduce HDBSCAN* 或 MR-HDBSCAN*，该方案基于[7]中初步介绍的递归采样策略，有效地计算出近似的聚类层次结构。该方法结合了两种可能的数据汇总技术，即随机样本和数据气泡[8]，并将这两种变体与使用随机块方法的精确版本就聚类质量和运行时间两方面进行比较。实验表明，虽然随机块版本提供了精确的结果，但提出的近似变体在聚类质量方面可以获得具有竞争力的结果，同时在分布式环境中速度要快几个数量级。

2 相关工作

2.1 层次密度聚类算法

与传统的分区和分层聚类算法相比，HDBSCAN* 算法有几个优点[5,9]。它结合了基于密度的聚类和基于层次的聚类两个方面，产生了一个完整的基于密度的聚类层次结构，从中可以直接提取出只由最突出的簇组成的层次结构。它的结果可以通过完整的树状图、简化的集群树和其他不需要任何关键参数作为输入的可视化技术来可视化。实际上，HDBSCAN* 的唯一参数 mpts 只是算法执行的非参数密度估计的一个平滑因子，可以嵌套的方式从算法中自动生成，而不需要指定一个特定的阈值作为输入。HDBSCAN* 还非常灵活，用户可以选择它直接分析得到的层次结构和集群树，或者通过该树执行局部切割，自动获得根据给定标准的最优平面 (非层次结构) 解决方案。

对于给定的有 n 个数据对象的数据集 $X = \{x_1, x_2, ..., x_n\}$，$x_i$ 和平滑因子的一个特定值 m_{pts}，HDBSCAN* 使用以下三个定义：

核心距离：一个物体 $x_p \in$ X 关于 m_{pts} 的核心距离 $d_{core}(x_p)$，表示 x_p 到它第 m_{pts} 个最近点（包括 x_p）的距离。ε 邻域内至少有 m_{pts} 个对象的物体称为核心物体，其中 ε 为最小半径（最大密度阈值）。

相互可达距离：两个物体 $x_p, x_q \in$ X 之间关于 m_{pts} 的相互可达距离表示为

$$d_{m_{reach}}(x_p, x_q) = max(d_{core}(x_p), d_{core}(x_q), d(x_p, x_q)) \tag{1}$$

相互可达图：数据集 X 关于 m_{pts} 的互达性图是一个完全加权图 $G_{m_{reach}}$，其中数据对象是顶点，每对顶点之间的边权由对应顶点对之间的相互可达距离给出。

HDBSCAN* 的主要步骤为：（1）计算数据集 X 中每个对象的核心距离；（2）计算 X 中任意两个对象之间的相互可达距离；（3）构建 X 的相互可达图，并计算最小生成树 MST；（4）为每个对象添加以自身核心距离为权值的自边，获得扩展的全局 MST_{ext}；（5）从 MST_{ext} 中提取层次结构。通过 HDBSCAN* 算法提取的集群 C_i 的稳定性与密度函数的模态质量 (相对) 过剩的统计概念有关

$$S(C_i) = \sum_{x_i \in C_i} (\frac{1}{\varepsilon_{min}(x_j, C_i)} - \frac{1}{\varepsilon_{max}(C_i)}) \tag{2}$$

其中 $\varepsilon_{min}(x_j, C_i)$ 为当 $x_i \in C_i$ 是 ε 的最小值，$\varepsilon_{max}(C_i)$ 为集群 C_i 存在时 ε 的最大值。

2.2 MapReduce 编程模型和框架

MapReduce 是一个以大规模数据并行方式处理大规模数据的编程模型[6]。与其他现有的并行处理模型相比，MapReduce 有以下几个优点：程序员不需要知道与数据分布、存储、复制和负载平衡相关

的细节，因此隐藏了实现细节，允许程序员开发更多地关注处理策略的应用程序/算法。程序员需要指定两个函数：map 和 reduce，大致步骤如下：（1）map 阶段忽略输入文件格式，转换文件内容并输出<key/value> 键值对；（2）shuffle 阶段根据键值对的 key 值，转换 map 阶段的输出到 reduce 阶段的输入；（3）reduce 阶段处理收到的键值对并输出最终结果。

由于其可扩展性和简单性，MapReduce 已经成为大规模数据分析的广泛工具，已经创建了一些框架来处理这个模型并促进它的实现。例如，Apache Hadoop，它自创建以来迅速传播。另外一个较新的框架，Apache Spark，除了可以使用 MapReduce 外，它还有额外的优势，因为它使用了一个称为弹性分布式数据集 (RDD) 的抽象。这种抽象允许可伸缩的程序对主存中的持久数据进行操作，具有较高的容错能力，特别适合在迭代算法中使用。这两个框架都穿插了顺序和并行计算，由几轮计算组成，每一轮在分布式系统 (节点) 之间传输信息，这些节点使用本地可用的数据执行所需的计算。根据应用程序的需求，这些计算的输出要么组合起来形成最终结果，要么发送到另一轮计算。然而，MapReduce 工作流受到了限制，因为在每个功能中应用的计算必须独立进行，并且在相同功能的作业之间的并行执行过程中不能交换信息。

2.3 基于随机块方法的 MR-HDBSCAN*

一种跨多个分布式节点并行计算精确 HDBSCAN* 层次结构的简单方法称为随机块方法，它是由我们的一位作者在[7]中初步提出的，本次复现的论文将其描述为比较的基线。它从本质上调整 CLUMP[10]来并行计算扩展最小生成树 MST_{ext}，从中提取 HDBSCAN* 层次结构。为了以与[10]相同的方法并行化和分发 MST，随机块方法将完整的数据集划分为 k 个“基本分区”，然后将它们组合成所谓的“数据块”，因此要求对于任何一对对象 (p，q)，p 和 q 之间的边权可以在一个数据块中被精确确定，必须总共生成 $\binom{k}{2}$ 个数据块 (选择 K 个基础分区的任意 2 个成对组合)。因此，MST 可以在数据块中独立计算或在不同的处理单元中并行地计算，然后这些独立计算的局部 MST 可以组合起来获得完整数据集的精确 MST。

然而，HDBSCAN* 不是基于在原始距离空间中计算的 MST。相反，它是建立在互达距离转换空间中计算的 MST，即边缘权值对应的是相互可达距离而不是原始距离。通过前面的定义我们知道，为了确定两个物体 p 和 q 之间的相互可达距离，我们需要能够计算出 p 和 q 关于 m_{pts} 的核心距离，以及 p 和 q 之间的距离。因此，如果对象的邻域不共享任何对象，在单个数据块中必须存在 $2 \times m_{pts}$ 个不同的对象。在最坏的情况下，这些 $2 \times m_{pts}$ 不同的对象属于不同的基本分区。所以，给定基本分区个数 k 和平滑因子 m_{pts}，必须生成 $\binom{k}{2\times m_{pts}}$ 个唯一的数据块，并将其分布到不同的处理单元。

在实践中，这将导致不可接受的计算负担，特别是对于较大的 m_{pts} 值 (为了实际应用，m_{pts} 值必须大于 2)。这种负担对于适合单机内存的小型数据集可能是可以容忍的，但对于需要跨多个节点分布和处理的大型数据集则是不允许的。

3 本文方法

论文提出的在 MapReduce 框架内并行计算 HDBSCAN* 的方法基于递归采样。与 MST 计算的并行化不同，递归采样方法消除了在多个处理单元上对重叠数据集的重复处理过程。这是通过一个“知情的”数据细分来实现的，该细分根据数据本身的结构，将数据划分为要在不同的处理单元处理的数

据。

3.1 简单递归采样

图 1显示了递归采样方法的流程。对于不能在单个节点中完全处理的数据，捕获最突出的聚类的子样本作为数据集中包含的结构的非常粗略的表示。这些集群构成了集群层次结构的更高级别。首先使用 HDBSCAN* 对采样的样本进行聚类，然后使用 FOSC 框架提取主要的集群，从而捕获样本中最突出的集群。根据提取的样本分区，将整个数据集的分区归纳到相对应的样本分区获得新的子集，具体做法就是将每个数据对象分配到与其最近的样本代表相同的集群，然后将每个子集发送到不同的处理单元进行进一步细化。根据处理单元的容量，这些子集将通过应用相同的策略递归地划分为较小的子集，这些子集代表集群层次结构的下一层，直到数据块的大小可以由处理单元完全处理，以获得数据子集的精确 MST。在递归分区过程中，确定分层连接这些局部 MST 的边，以便在最后得到整个数据集的生成树，对于小到可以在单个节点上处理的子集，该生成树是“局部最小的”。

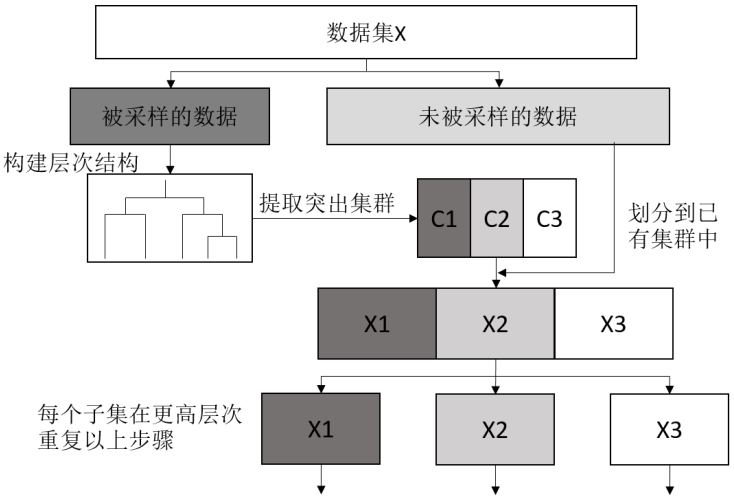


图 1: 递归采样示意图

3.2 数据气泡

由于“代表性”样本对象之间的距离很难代表了它们所代表的对象集之间的距离，且仅使用一个数据样本的密度很难近似估计整个数据集的密度，因此当样本量与整个数据集相比较小时，采用简单递归采样的方法可能会导致较差的结果。为了缓解这一问题，论文[11]中引入了“数据气泡”的概念。数据气泡，简单理解就是计算一组数据的统计量，用统计量的值来代表这组数据。在论文[11]中已经表明，数据气泡允许从极低的样本中恢复大型数据集的聚类结构。

在对整个数据集的一次连续扫描中，对于每一个子集 X_b，计算形式为 (n, LS, SS) 的统计量，其中 n 是子集 X_b 中对象的数量，$LS = \sum_{x_i \in X_b} \overrightarrow{x_i}$ 是 X_b 的对象的线性和，$SS = \sum_{x_i \in X_b} \overrightarrow{x_i}^2$ 是 X_b 的对象的平方和。

对于聚类，每个子集 X_b 可以由一个数据气泡表示，即 $B_{X_b} = (rep, n, extent, nnDist)$，其中 $rep = LS/n$ 是 X_b 中对象的平均值，n 是 X_b 中对象的个数，$extent = \sqrt{\frac{\sum_{i=1,...,n} \sum_{j=1,...,n} (x_i - x_j)^2}{n(n-1)}} = \sqrt{\frac{2 \times n \times SS - 2 \times LS^2}{n(n-1)}}$ 是子集 X_b 的半径，$nnDist$ 是对 X_b 中 $k = 1, ..., m_{pts}$ 的平均 k 近邻距离的估计。假设在子集 X_b 内是均匀分布，则 $nnDist(k) = (\frac{k}{n})^{\frac{1}{d}} \times extent$，其中 d 是数据维度。

3.3 基于 MapReduce 实现 HDBSCAN*

在这里，论文作者将其实现称为 MapReduce HDBSCAN* 或 MR-HDBSCAN*。MR-HDBSCAN* 由三个主要步骤组成：首先，对数据进行分区，直到每个分区都能容纳单个处理单元容量 (用 τ 表示)，并为每个部分计算局部 MST；然后，将所有局部 MST 和集群间边缘合并为单个全局 MST$_{ext}$；最后，从 MST$_{ext}$ 中提取 HDBSCAN* 层次结构作为树状图。

3.3.1 第一步：对数据分区

MR-HDBSCAN* 的第一步是对数据进行分区，直到它适合单个处理单元。该步骤分为四个部分：MapPartitionsToPair 函数，用于在相互可达距离的空间上构建局部 MST，或者，如果数据太大而不能在本地处理，计算每个数据对象的最近的样本代表；Combining 函数，应用在被采样的样本上，创建数据气泡和更新信息；ReduceByKey 函数，负责用 HDBSCAN* 和 FOSC 对数据气泡/样本进行分区；以及 Mapper 函数，它将气泡/样本所表示的数据对象映射到先前获得的分区。MR-HDBSCAN* 算法第一步的流程图如图 2所示。它假设数据位于 MapReduce 框架的分布式文件系统 (DFS) 中。

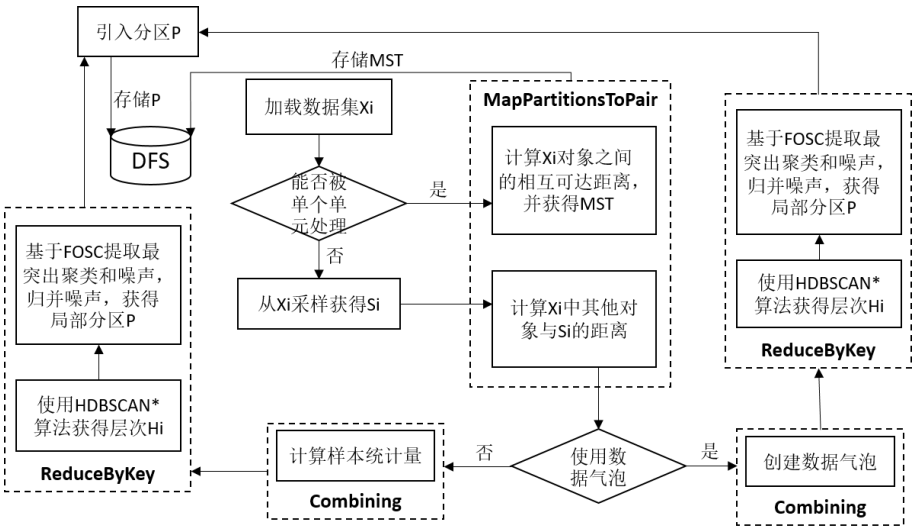


图 2: MR-HDBSCAN* 第一步的流程图

在每次迭代中，DFS 中的每个集群都被加载，由 MapPartitionsToPair 函数独立地并行处理。这个函数接收数据的一个分区，然后将子集 Xi 的每个对象映射到 <key/value> 对。如果 Xi 足够小，可以被单个处理单元进行处理，函数就会计算 Xi 的 MST，用自边扩展它，并将 MST$_{e}xt$ 持久化存储到 DFS 中。如果 Xi 太大，无法适合单个处理单元，则从 Xi 中提取一个样本 Si，计算 Xi 中剩余对象与 Si 的采样对象之间的距离。这些距离被发送到 Combining 函数，该函数计算每个采样对象的统计数据 (n, LS, SS)，或者选择在样本上构建数据气泡。

Combine 函数将样本的统计信息或数据气泡发送给 ReduceByKey 函数，ReduceByKey 函数应用 HDBSCAN* 来构造一个 MST，并从该 MST 构建集群层次结构 Hi。其次，基于以质量过剩 (EOM) 为质量度量的 FOSC 提取最显著的聚类 $(C_{i1}, \ldots, C_{ik})$ 和噪声 (N_i)。将 N_i 的每个噪声样本/气泡插入到最近的突出聚类中，得到局部分区 P，并将其发送给 Mapper 函数。

Mapper 函数根据 ReduceByKey 函数提取的局部集群 P 对 X_i 中的对象划分集群，分配每个对象到代表性数据气泡，若数据气泡未被采纳时则分配到最近的样本代表。这个过程将对层次结构 H_i 的

每个级别进行迭代重复。最后，Mapper 将映射的分区 P_i 的相关信息，包括集群间和集群内边缘，持久化存储到 DFS 中。

3.3.2 第二步：合并 MST

MR-HDBSCAN* 的第二步是将存储的所有局部 MST 的边和集群间边缘合并成单个的全局扩展生成树 (MST_{ext})。每条边都用向量 $[u, v, d_{reach}(u, v)]$ 表示，其中 u 和 v 是边连接的两个顶点，而 $d_{reach}(u, v)$ 是两个顶点之间的相互可达距离。上述部分是由 ReduceByKey 函数完成的。边缘按权重 (d_{reach}) 降序组合，并存储在 DFS 中。

3.3.3 第三步：构建层次结构

构建 MST_{ext} 后，MR-HDBSCAN* 在数据集的所有对象上构建一个层次结构 (树状图)。这一步可以使用两种方法来实现：一个是自顶向下方法[9]，或者基于 Union-Find 算法[12]的自底向上方法。

自顶向下方法的核心思想就是按边的权值大小依次分裂集群。假设最初所有对象都是同一集群的成员，然后通过分裂集群迭代构建下一个层次结构级别，直到所有数据都被认为是噪声。通过从 MST_{ext} 中删除具有最高权重 (d_{reach}) 的边，并在层次结构的新级别上将边连接的组件检测为集群，从而分离集群。虽然自顶向下的方法遵循了 HDBSCAN*[9]的原始建议，但是使用 MapReduce 根据这种方法构造每个层次级别可能计算开销很大。

自底向上 (或聚合) 方法的一般思想是将分布式 MST_{ext} 的所有顶点视为独立的 (子) 树，并根据它们连接边的权重 (d_{reach}) 的递增顺序将它们合并。每次合并都会在更高的层次级别上产生一个新的集群。该过程迭代执行，直到在层次结构的顶部获得包含每个对象的单个集群。在这项工作中，自底向上方法的 MapReduce 实现基于 ReduceByKey 函数。

4 复现细节

4.1 与已有开源代码对比

本次复现主要参考了论文附带的源代码（http://www2.dc.ufscar.br/ naldi/source），具体使用过程就是先将对应的数据集上传到集群的 HDFS 系统上，然后将源代码打包放到我们搭建的 spark 集群环境中运行。以下是源代码中一些核心步骤的功能以及实现。

下面两张图是 MR-HDBSCAN* 算法的第一步中 MapPartitionsToPair 函数对应的代码实现。可以看到，程序会先判断当前子集的大小是否小于等于单个处理单元的容量大小。如果是，表明可以直接计算该子集的局部 MST，如图 3所示；如果子集不能被单个处理单元所容纳，则计算当前对象的最近邻居对象，并计算对象的统计量，如图 4所示。

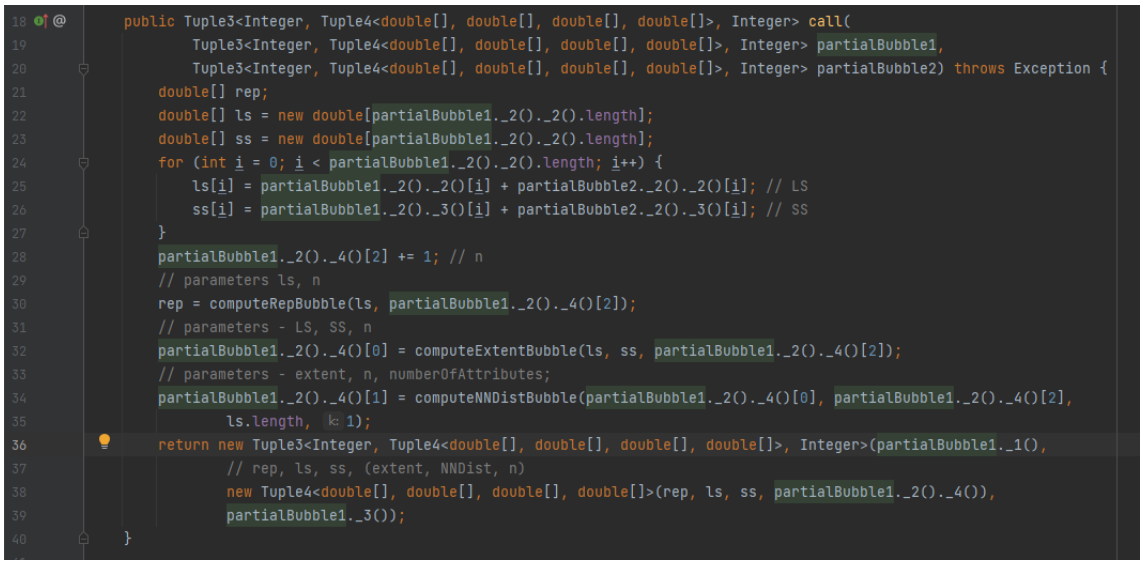
```
if ((dataset != null) && (type._2() <= this.processingUnits)) { // compute local MST from subset
    double[] coreDistances = model.calculateCoreDistances(dataset, this.mpts, this.distanceFunction);
    UndirectedGraph mst = model.constructMST(dataset, coreDistances, selfEdges: true, distanceFunction, indices,
            indices.length);
    for (int i = 0; i < mst.getNumEdges(); i++) {
        // 1, v, dmreach, u
        double[] dmreach = new double[1];
        dmreach[0] = mst.getEdgeWeightAtIndex(i);
        subset.add(
                new Tuple2<Integer, Tuple3<Integer, Tuple4<double[], double[], double[], double[]>, Integer>>(
                        -1,
                        new Tuple3<Integer, Tuple4<double[], double[], double[], double[]>, Integer>(
                                mst.getFirstVertexAtIndex(i),
                                new Tuple4<double[], double[], double[], double[]>(dmreach, null, null, null),
                                mst.getSecondVertexAtIndex(i))));
    }
}
```

图 3: 子集大小小于处理单元容量

```
} else { // compute the nearest neighbor of the first object
    double minDistance = Double.MAX_VALUE;
    int nearestNeighbor = 0;
    for (int neighbor = 0; neighbor < this.bSamples.size(); neighbor++) {
        // compute distances
        double dist = this.distanceFunction.computeDistance(record._2()._2(),
                this.bSamples.get(neighbor)._2()._2());
        if (dist < minDistance) {
            minDistance = dist;
            nearestNeighbor = this.bSamples.get(neighbor)._2()._1().intValue();
        }
    }
    // idSample, idObject, object, id(subset)
    double[] infoBubbles = new double[3];
    infoBubbles[2] = 1;
    double[] ss = new double[record._2()._2().length];
    double[] ls = new double[record._2()._2().length];
    for (int i = 0; i < ss.length; i++) {
        ss[i] = record._2()._2()[i] * record._2()._2()[i];
        ls[i] = record._2()._2()[i];
    }
    subset.add(
            new Tuple2<Integer, Tuple3<Integer, Tuple4<double[], double[], double[], double[]>, Integer>>(
                    nearestNeighbor,
                    new Tuple3<Integer, Tuple4<double[], double[], double[], double[]>, Integer>(
                            record._2()._1(),
                            new Tuple4<double[], double[], double[], double[]>(ls, ls, ss, infoBubbles),
                            record._1())));
}
```

图 4: 子集大小大于处理单元容量

图 5展示了创建数据气泡的相关代码，即 Combining 函数，分别计算数据集的 rep、LS、SS 以及 nnDist 值。



```
public Tuple3<Integer, Tuple4<double[], double[], double[], double[]>, Integer> call(
        Tuple3<Integer, Tuple4<double[], double[], double[], double[]>, Integer> partialBubble1,
        Tuple3<Integer, Tuple4<double[], double[], double[], double[]>, Integer> partialBubble2) throws Exception {
    double[] rep;
    double[] ls = new double[partialBubble1._2()._2().length];
    double[] ss = new double[partialBubble1._2()._2().length];
    for (int i = 0; i < partialBubble1._2()._2().length; i++) {
        ls[i] = partialBubble1._2()._2()[i] + partialBubble2._2()._2()[i]; // LS
        ss[i] = partialBubble1._2()._3()[i] + partialBubble2._2()._3()[i]; // SS
    }
    partialBubble1._2()._4()[2] += 1; // n
    // parameters ls, n
    rep = computeRepBubble(ls, partialBubble1._2()._4()[2]);
    // parameters - LS, SS, n
    partialBubble1._2()._4()[0] = computeExtentBubble(ls, ss, partialBubble1._2()._4()[2]);
    // parameters - extent, n, numberOfAttributes;
    partialBubble1._2()._4()[1] = computeNNDistBubble(partialBubble1._2()._4()[0], partialBubble1._2()._4()[2],
            ls.length, k: 1);
    return new Tuple3<Integer, Tuple4<double[], double[], double[], double[]>, Integer>(partialBubble1._1(),
            // rep, ls, ss, (extent, NNDist, n)
            new Tuple4<double[], double[], double[], double[]>(rep, ls, ss, partialBubble1._2()._4()),
            partialBubble1._3());
}
```

图 5: 创建数据气泡

从第 3 章可以知道，MR-HDBSCAN* 算法的第一步的另外一个核心函数是 ReduceByKey，它的主要作用是提取层次结构，以及提取突出集群或者噪声点，获得局部分区。图 6展示了提取层次结构

的具体实现过程；图 7是划定新分区的相关代码。

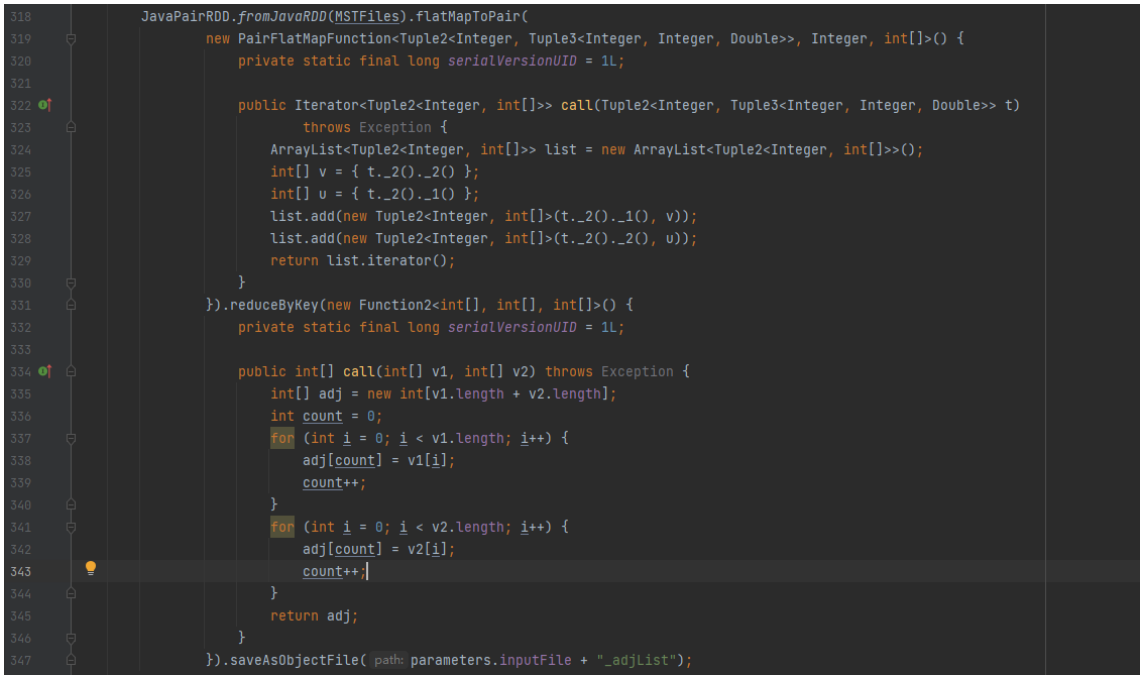
```
// computing hierarchy;
//System.out.println(nB.length + " s: " + size);
System.out.println("nB.length="+nB.length+",size="+size);
if (nB.length >= size) {
    HdbscanDataBubbles model = new HdbscanDataBubbles();
    double[] coreDistances = model.calculateCoreDistancesBubbles(data, nB, eB, nnDistB, this.mpts,
            this.distanceFunction);
    UndirectedGraph mst = model.constructMSTBubbles(data, nB, eB, nnDistB, idBubbles, coreDistances, selfEdges: true,
            this.distanceFunction);
    mst.quicksortByEdgeWeight();
    model.constructClusterTree(mst, this.mclSize, nB);
    flatPartition = model.findProminentClustersAndClassificationNoiseBubbles(model.getClusters(), data, nB, eB,
            nnDistB, this.distanceFunction, idBubbles);
    model.findInterClusterEdges(mst, flatPartition);
    int[] objects = new int[flatPartition.length];
    int[] labels = new int[flatPartition.length];
    for (int i = 0; i < labels.length; i++) {
        objects[i] = flatPartition[i][0];
        labels[i] = flatPartition[i][1];
    }
    return new Tuple3<Tuple3<double[][], double[][], int[]>, Tuple3<int[], int[], Integer>, Tuple3<int[], int[], double[]>>(
            new Tuple3<double[][], double[][], int[]>(data, infoBubbles, idBubbles),
            new Tuple3<int[], int[], Integer>(objects, labels, id),
            new Tuple3<int[], int[], double[]>(model.getVertice1(), model.getVertice2(), model.getDmreach()));
} else {
    return new Tuple3<Tuple3<double[][], double[][], int[]>, Tuple3<int[], int[], Integer>, Tuple3<int[], int[], double[]>>(
            new Tuple3<double[][], double[][], int[]>(data, infoBubbles, idBubbles),
            new Tuple3<int[], int[], Integer>(null, null, id), null);
}
}
```

图 6: 提取层次结构

```
public Tuple2<Integer, Tuple2<Integer, double[]>> call(Tuple2<Integer, Tuple3<Integer, Integer, double[]>> t)
        throws Exception {
    int newId = -2;
    for (int i = 0; i < this.model.size(); i++) {
        // same subset
        // System.out.println("Node: " + this.model.get(i)._2()._2()._3().intValue() + "
        // node2: " + t._2()._1() + " object: " + t._2()._2());
        if (this.model.get(i)._2()._2()._3().intValue() == t._2()._1().intValue()) {
            if (this.model.get(i)._2()._2()._1() != null)
                for (int j = 0; j < this.model.get(i)._2()._2()._1().length; j++) {
                    if (this.model.get(i)._2()._2()._1()[j] == t._1().intValue()) {
                        newId = this.model.get(i)._2()._2()._2()[j];
                    }
                }
        }
    }
    return new Tuple2<Integer, Tuple2<Integer, double[]>>(newId,
            new Tuple2<Integer, double[]>(t._2()._2(), t._2()._3()));
}
```

图 7: 获得局部分区

MR-HDBSCAN* 算法的第二步是合并局部 MST，具体代码如图 8所示。根据前面第一步获得的局部 MST 和各层次之间的边缘，扩展自边，按照边权值降序组合，合成单个全局 MST_{ext}。



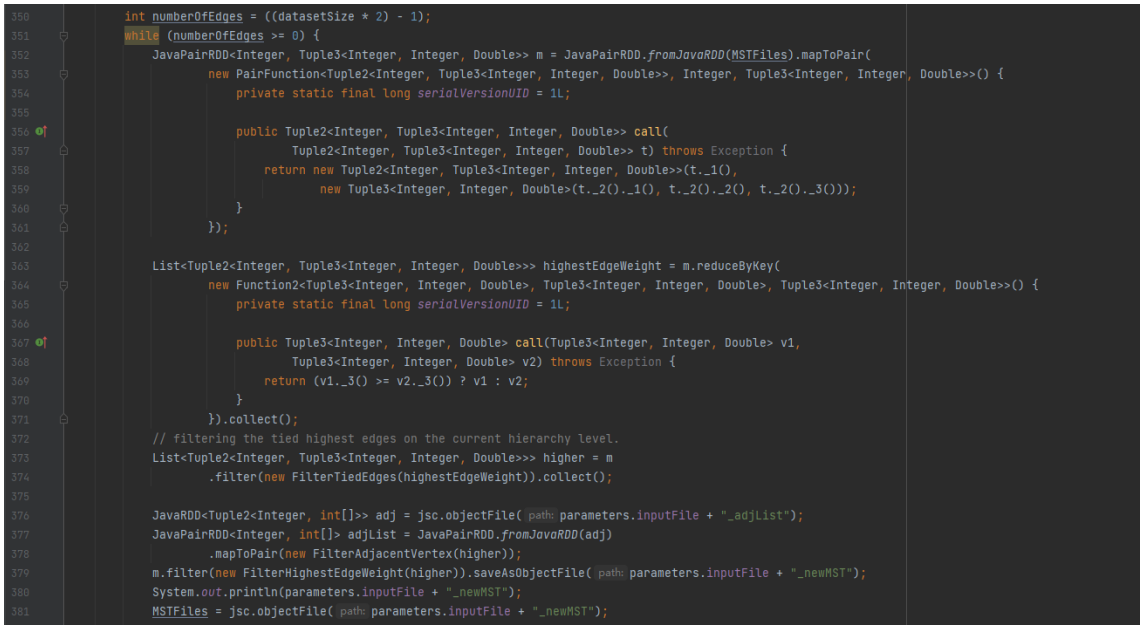
```
JavaPairRDD.fromJavaRDD(MSTFiles).flatMapToPair(
        new PairFlatMapFunction<Tuple2<Integer, Tuple3<Integer, Integer, Double>>, Integer, int[]>() {
            private static final long serialVersionUID = 1L;

            public Iterator<Tuple2<Integer, int[]>> call(Tuple2<Integer, Tuple3<Integer, Integer, Double>> t)
                    throws Exception {
                ArrayList<Tuple2<Integer, int[]>> list = new ArrayList<Tuple2<Integer, int[]>>();
                int[] v = { t._2()._2() };
                int[] u = { t._2()._1() };
                list.add(new Tuple2<Integer, int[]>(t._2()._1(), v));
                list.add(new Tuple2<Integer, int[]>(t._2()._2(), u));
                return list.iterator();
            }
        }).reduceByKey(new Function2<int[], int[], int[]>() {
            private static final long serialVersionUID = 1L;

            public int[] call(int[] v1, int[] v2) throws Exception {
                int[] adj = new int[v1.length + v2.length];
                int count = 0;
                for (int i = 0; i < v1.length; i++) {
                    adj[count] = v1[i];
                    count++;
                }
                for (int i = 0; i < v2.length; i++) {
                    adj[count] = v2[i];
                    count++;
                }
                return adj;
            }
        }).saveAsObjectFile( path: parameters.inputFile + "_adjList");
```

图 8: 合并 MST

MR-HDBSCAN* 算法的第三步是构建层次结构，具体代码如图所示。论文所给的源代码是按照自顶向下的方法实现的，也就是依次去除 MST_{ext} 中最高权值的边，并给边连接的两个组件贴上新标签，作为层次结构的下一层。



```
int numberOfEdges = ((datasetSize * 2) - 1);
while (numberOfEdges >= 0) {
    JavaPairRDD<Integer, Tuple3<Integer, Integer, Double>> m = JavaPairRDD.fromJavaRDD(MSTFiles).mapToPair(
            new PairFunction<Tuple2<Integer, Tuple3<Integer, Integer, Double>>, Integer, Tuple3<Integer, Integer, Double>>() {
                private static final long serialVersionUID = 1L;

                public Tuple2<Integer, Tuple3<Integer, Integer, Double>> call(
                        Tuple2<Integer, Tuple3<Integer, Integer, Double>> t) throws Exception {
                    return new Tuple2<Integer, Tuple3<Integer, Integer, Double>>(t._1(),
                            new Tuple3<Integer, Integer, Double>(t._2()._1(), t._2()._2(), t._2()._3()));
                }
            });

    List<Tuple2<Integer, Tuple3<Integer, Integer, Double>>> highestEdgeWeight = m.reduceByKey(
            new Function2<Tuple3<Integer, Integer, Double>, Tuple3<Integer, Integer, Double>, Tuple3<Integer, Integer, Double>>() {
                private static final long serialVersionUID = 1L;

                public Tuple3<Integer, Integer, Double> call(Tuple3<Integer, Integer, Double> v1,
                        Tuple3<Integer, Integer, Double> v2) throws Exception {
                    return (v1._3() >= v2._3()) ? v1 : v2;
                }
            }).collect();
    // filtering the tied highest edges on the current hierarchy level.
    List<Tuple2<Integer, Tuple3<Integer, Integer, Double>>> higher = m
            .filter(new FilterTiedEdges(highestEdgeWeight)).collect();

    JavaRDD<Tuple2<Integer, int[]>> adj = jsc.objectFile( path: parameters.inputFile + "_adjList");
    JavaPairRDD<Integer, int[]> adjList = JavaPairRDD.fromJavaRDD(adj)
            .mapToPair(new FilterAdjacentVertex(higher));
    m.filter(new FilterHighestEdgeWeight(higher)).saveAsObjectFile( path: parameters.inputFile + "_newMST");
    System.out.println(parameters.inputFile + "_newMST");
    MSTFiles = jsc.objectFile( path: parameters.inputFile + "_newMST");
```

图 9: 去除权重最高的边

```
        System.out.println(" size: " + numberOfEdges);
        // finding connected components
        JavaPairRDD<Integer, int[]> subcomponents = adjList;
        do {
            newIteration = jsc.sc().longAccumulator(); // init with 0
            System.out.println("New iteration - 1:  " + newIteration.value());
            subcomponents = subcomponents.flatMapToPair(new findConnectedComponentsOnMST())
                    .reduceByKey(new Function2<int[], int[], int[]>() {
                        private static final long serialVersionUID = 1L;

                        public int[] call(int[] v1, int[] v2) throws Exception {
                            int[] newArray = new int[v1.length + v2.length];
                            for (int i = 0; i < v1.length; i++) {
                                newArray[i] = v1[i];
                            }
                            for (int i = 0; i < v2.length; i++) {
                                newArray[v1.length + i] = v2[i];
                            }
                            return newArray;
                        }
                    });
            subcomponents.saveAsTextFile( path: parameters.inputFile + "_comp");
            System.out.println(parameters.inputFile + "_comp");
            System.out.println("New iteration? " + newIteration.value());
        } while (newIteration.value() != 0);
        System.exit( status: 1);

        numberOfEdges -= higher.size();
```

图 10: 寻找连接的组件

4.2 实验环境搭建

在实验室服务器上搭建了 3 台虚拟机，并在每台主机上下载 ClouderaManager，构建一个小的 spark 集群，如下图所示。

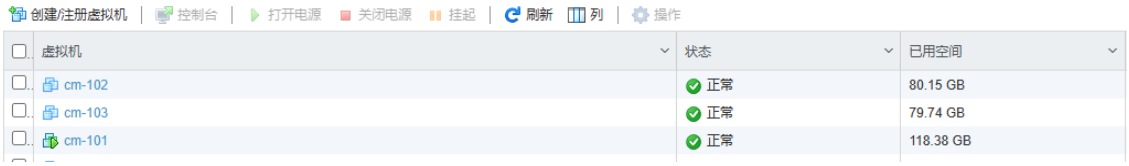


图 11: 创建 3 台虚拟机

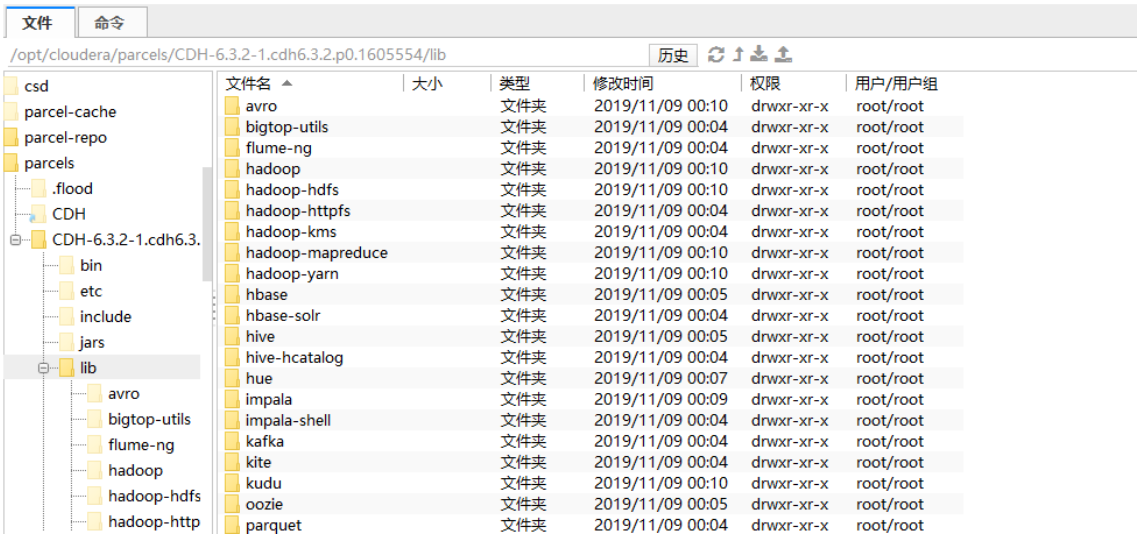
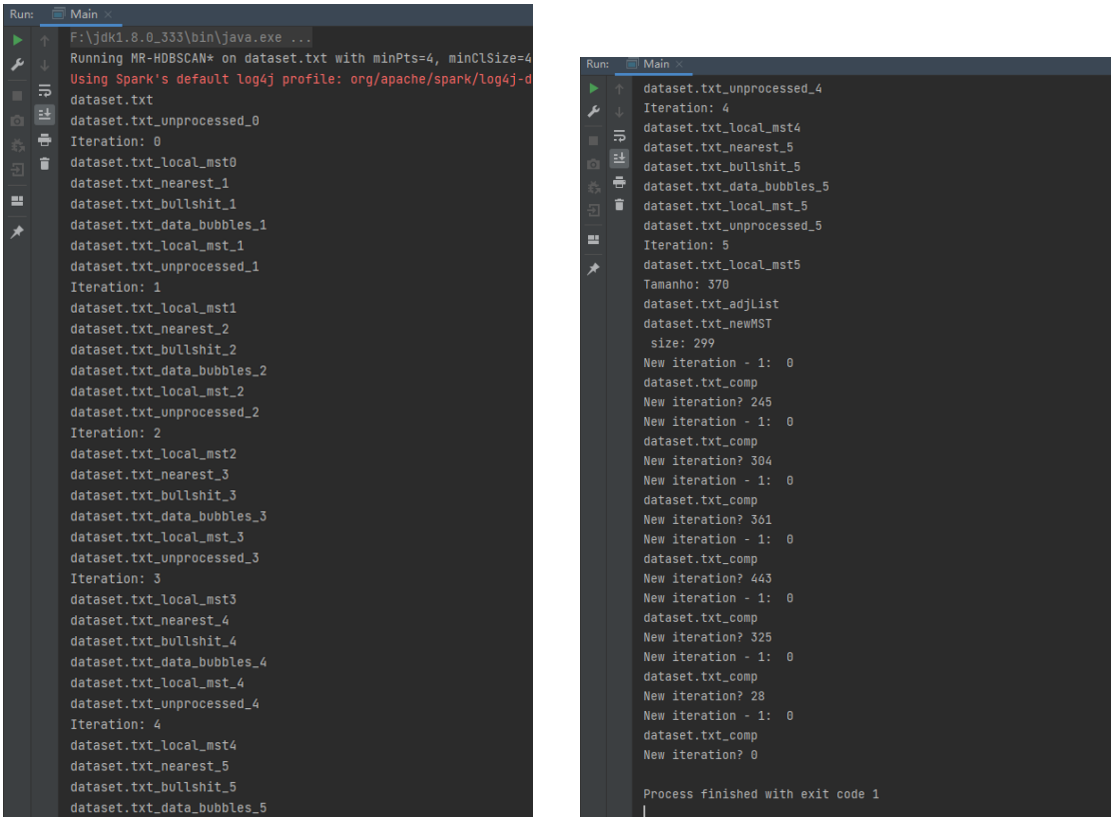


图 12: 下载 ClouderaManager

图 13: ClouderaManager 界面

5 实验结果分析

图 14是已有源代码在我本地电脑上跑出的结果截图，运行的数据集是附带的小数据集，只有 150 个对象。



(a) 结果 1　　(b) 结果 2

图 14: 本地运行结果

5.1 数据集

原论文中使用了三个人工数据集和八个真实数据集。表 1总结了数据集的主要信息。由于配置和性能限制，在我的电脑上无法跑出这些数据集的聚类结果，因此后续列出的 API 值和运行时间都是原作者在论文中给出的。

数据集	对象数目	属性数目	集群数目
Gauss1	1×10^6	10	20
Gauss2	3×10^6	10	30
Gauss3	5×10^6	10	50
YearPrediction	515,345	90	89
Poker	1,025,010	11	10
HT Sensor	919,438	11	3
Skin	245,057	4	2
Yellow1	1,600,000	17	6
Yellow2	8,000,000	17	6
HEPMASS	10,500,000	28	2
HIGGS	11,000,000	28	2

表 1: 各数据集的信息

5.2 评估标准

为了将提取的解与 ground truth 分区进行比较，我们使用了著名的 Adjusted Rand Index(ARI)[13]。在评估的聚类解决方案中，未聚类的噪声对象在 ARI 计算时被视为单对象。只有评估结果与 ground truth 划分一致时，ARI 才达到最大值 1。随机解的 ARI 的期望值为 0。随机块是 HDBSCAN* 的 MapReduce 精确版本，它具有确定性的行为，可以给定数据集的单一层次聚类。但是，MR-HDBSCAN* 简单递归采样版本和 MR-HDBSCAN* 数据气泡版本是基于数据的随机采样，这可能会导致结果的变化。由于这个原因，这些近似算法为每个数据集运行 45 次，并从相应的聚类层次结构中提取出 45 个平面分区。

5.3 API 结果

随机块分区的 ARI 值以及 MR-HDBSCAN* 简单递归采样版本和 MR-HDBSCAN* 数据气泡版本的 ARI 评估的平均值和标准差 (括号内) 如表 2所示。

数据集	随机块版本 API	采样版本 API	数据气泡版本 API
Gauss1	0.881	0.690(0.001)	0.864(0.000)
Gauss2	0.820	0.588(0.001)	0.759(0.002)
Gauss3	0.801	0.602(0.006)	0.777(0.002)
YearPrediction	0.403	0.301(0.005)	0.388(0.004)
Poker	0.310	0.196(0.005)	0.297(0.001)
HT Sensor	0.359	0.287(0.001)	0.330(0.000)
Skin	0.441	0.360(0.004)	0.425(0.002)
Yellow1	0.328	0.250(0.011)	0.290(0.014)
Yellow2	0.426	0.362(0.024)	0.407(0.011)
HEPMASS	0.546	0.408(0.005)	0.529(0.000)
HIGGS	0.235	0.210(0.008)	0.227(0.006)

表 2: 各数据集的 API 结果

结果显示，API（随机块）>API（数据气泡）>API（简单递归采样）。随机块版本获得了最好的质量结果，这并不奇怪，因为它是原始 HDBSCAN* 的精确版本。相反，使用简单递归抽样和汇总时，就会发生信息丢失。然而，使用数据气泡作为一种更复杂的数据汇总技术部分地缓解了这个问题，产生的损失明显要小得多。MR-HDBSCAN* 简单递归采样与 MR-HDBSCAN* 数据气泡相比，后者保留了集群大部分的信息和结构，它也引起了较少的结果可变性，这表现为大多数数据集较小的标准偏差值。从绝对意义上讲，MR-HDBSCAN* 的两个近似版本观测到的结果的变异性非常小，简单采样 ARI 值的标准差低于 0.025，数据气泡 ARI 值的标准差低于 0.015。

5.4 运行时间结果

表 3展示了执行随机块版本所需的计算时间以及简单采样版本和数据气泡版本执行的平均计算时间 (括号里是标准差)。

数据集	随机块版本运行时间	采样版本运行时间	数据气泡版本时间
Gauss1	13312.24	67.35(18.11)	82.75(31.48)
Gauss2	28393.65	162.07(30.50)	225.40(23.45)
Gauss3	1+ month	115.39(0.076)	182.05(0.077)
YearPrediction	11622.61	106.57(17.62)	109.89(21.30)
Poker	28955.89	52.81(5.84)	97.06(35.32)
HT Sensor	31450.89	42.54(4.74)	82.07(25.89)
Skin	1743.93	21.14(4.99)	60.19(26.00)
Yellow1	1+ month	106.44(8.44)	265.50(17.15)
Yellow2	1+ month	474.69(43.26)	776.67(7.40)
HEPMASS	1+ month	385.67(26.62)	695.29(28.01)
HIGGS	1+ month	752.17(48.05)	887.54(46.18)

表 3: 各数据集的运行时间

简单采样和数据气泡这两种近似算法的运行速度比随机块更快，最高 700 倍，平均 240 倍。MR-HDBSCAN* 简单采样版本的平均执行时间最低，紧随其后的是 MR-HDBSCAN* 数据气泡版本，运行最慢的是 MR-HDBSCAN* 随机块版本。使用随机块版本运行部分数据量较大的数据集时，运行时间甚至高达一个多月。

基于递归采样和数据气泡的 MR-HDBSCAN* 算法虽然较精确版本的算法（随机块方法）损失了一点点准确率，但是却大大提高了运行效率。因此，对于大规模的分布式框架和数据集来说，基于数据气泡和 MapReduce 的 HDBSCAN* 算法的近似版本是一个可行的选择，它代表了质量和计算性能之间的最佳权衡。

6 总结与展望

在论文中，作者描述和提出了三种使用 MapReduce 实现 HDBSCAN* 算法的方法。第一种是基于随机块的 HDBSCAN* 算法的 MapReduce 精确版本，该方法的准确度较高，但运行效率很低，不适合在大型数据集上应用。第二种是单纯地采用递归采样的策略，这种方法获得的样本具有随机性和偶然性，因此计算结果不稳定，聚类的准确率与精确版本相比较低，但它运行时间最短。第三种是作者提出的引入数据气泡作为汇总技术的方法，通过计算相关统计值来保留数据集的分布信息和结构信息，允许用较少的样本近似恢复大型数据集的聚类结构，因此这种方法的结果相对稳定，准确率仅次于精确版本，该近似版本的 API 值只比随机块精确版本的 API 值低了 0.01 左右，但运行时间却大大缩短了几百倍。

总的来看，对于大规模的分布式框架和数据集来说，基于递归采样和数据气泡相结合的 MR-HDBSCAN* 聚类算法具有可行性，虽然只是近似结果，牺牲了一点点准确率换来明显提高的运行效率，是一个相对合理的选择。

当然对于该论文，还是有存在不足，比如输入参数 m_{pts} 的值大小的控制，比如初始分区个数 k 和单个处理单元容量的大小控制，又比如当数据集是真实的或者规模过大时，聚类结果并不是很理想，API 值普遍低于 0.5..... 对于这些问题，后续可以通过使用遗传算法优化获得最佳输入参数值，可以优

化分层聚类过程中的层次结构，等等。对此，我还需要学习更多的知识，提高自己的能力，以便更好地解决上述问题。

参考文献

[1] SANTOS J A D, SYED T I, NALDI M C, et al. Hierarchical Density-Based Clustering Using MapReduce[J]. IEEE Transactions on Big Data, 2021, 7(1): 102-114.

[2] H.-P. KRIEGEL J S, P. Krger, ZIMEK A. Density-based clustering[J]. Wiley Interdisciplinary Rev.: Data Mining Knowl. Discovery, 2011, 1(3): 231-240.

[3] ANKERST M, BREUNIG M M, KRIEGEL H P, et al. OPTICS: Ordering Points to Identify the Clustering Structure[J]. ACM SIGMOD, 1999, 28(2): 49-60.

[4] BRECHEISEN S, KRIEGEL H P, KRÖGER P, et al. Visually Mining Through Cluster Hierarchies[J]. Proceedings of the 2004 SIAM International Conference on Data Mining (SDM), 2004: 400-411.

[5] CAMPELLO R J G B, MOULAVI D, ZIMEK A, et al. Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection[J]. ACM Trans. Knowl. Discov. Data, 2015, 10(1): 5:1-5:51.

[6] DEAN J, GHEMAWAT S. MapReduce: Simplified Data Processing on Large Clusters[J]. Commun. ACM, 2008, 51(1): 107-113.

[7] SYED T I. Parallelization of Hierarchical Density-Based Clustering using MapReduce[J]., 2014.

[8] ZHOU J, SANDER J. Data bubbles for non-vector data: Speeding-up hierarchical clustering in arbitrary metric spaces[J]. Very Large Data Bases 30th, 2003: 452-463.

[9] RICARDO J. G. B. CAMPELLO J S, Davoud Moulavi. Density-based clustering based on hierarchical density estimates[J]. Advances in Knowledge Discovery and Data Mining, 2013: 160-172.

[10] OLMAN V, MAO F, WU H, et al. Parallel clustering algorithm for large data sets with applications in bioinformatics[J]. IEEE/ACM Transactions on Computational Biology and Bioinformatic, 2009, 6(2): 344-352.

[11] BREUNIG M M, KRIEGEL H P, KRÖGER P, et al. Data Bubbles: Quality Preserving Performance Boosting for Hierarchical Clustering[J]. ACM SIGMOD, 2001, 30(2): 79-90.

[12] GALIL Z, ITALIANO G F. Data structures and algorithms for disjoint set union problems[J]. ACM Comput. Surv., 1991, 23(3): 319-344.

[13] LAWRENCE HUBERT P A. Comparing partitions[J]. Journal of Classification, 1985, 2(1): 193-218.