大数据分析技术 Project 1 实验报告

16307130086 何畅扬　　16307130138 常朝坤

一、 概述

本次 project 的主题为基于 kdd13_Scalable All-Pairs Similarity Search in Metric Spaces 的 Similarity search，并在真实的分布式集群上进行测试。分治法的思想在本项目中有很好的体现。

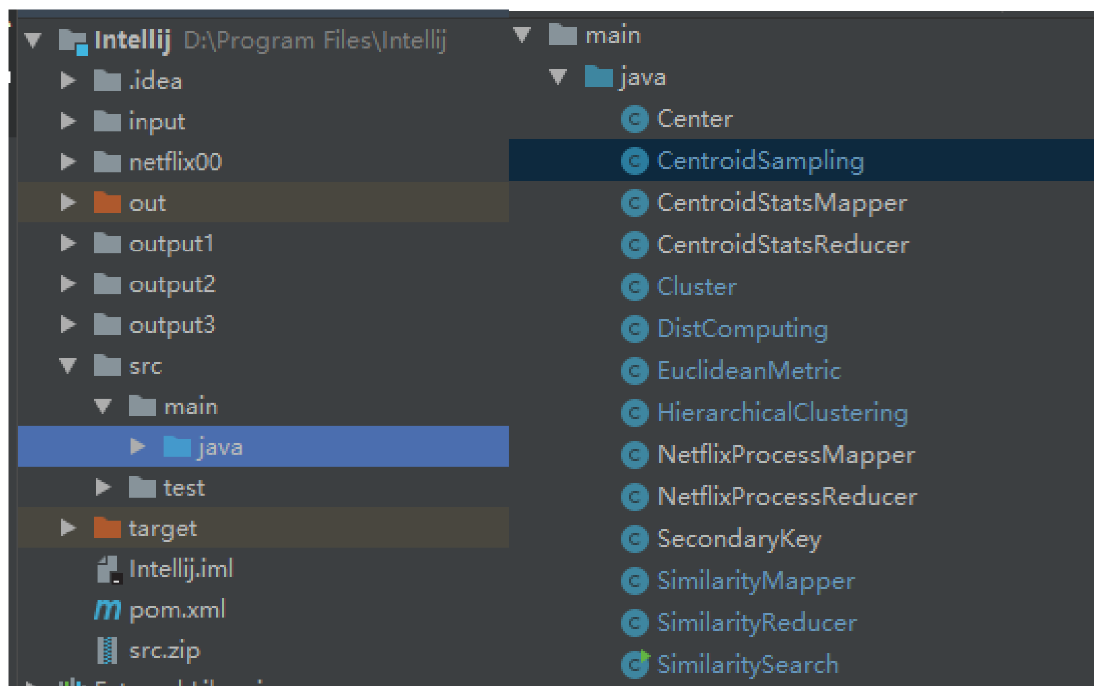
二、 项目目标

从具有 K 个样本点的样本集中，选择样本间距为 r 以内的所有 pairs 并输出。

三、 符号表示

1. P：点
2. C：圆心
3. R：半径
4. S：点集（workset）
5. r：样本间距
6. Inner pairs：由一个点集的 inner 点间计算出的 pairs
7. Inner to Outer pairs：由一个点集的 inner 与 outer 点间计算出的 pairs

四、 项目结构



左图所示为整体的结构，其可用于本地调试。右图展示了源码的架构，各个文件的命名直接与其功能相关，其中 SimilaritySearch 为最终的运行主类。

五、 项目细节

1. Centroid Sampling

首先对一个样本集，我们需要产生 n 个中心点。论文中列举了 random sampling， KMeans++及 seeding approach 等方法，从结果看前两者效率较高，故我们选择了 random sampling 的方式进行中心点的产生。值得注意的是，各个中心点之间的距离需要满足大于 r，否则会产生结果的重叠。

2. Centroid Stats

此阶段的主要目的是得到每个中心点的半径。

setup

载入 sampling 阶段产生的中心点位置

实现方式：对于（1）的结果，我们将其包装成一个字符串（每个中心点之间按照\t 隔开），main 函数中得到结果之后将其存入 configuration 中，如下图所示：

```
CentroidSampling cs = new CentroidSampling();
String centroids = cs.sampling();
conf.set(CentroidSampling.FLAG_CENTROID,centroids);
```

在 setup 函数里，通过从 configuration 中读出字符串并解包装，就能够得到所有的中心点。

```
private String[] centroids;
private int centroids_size = 0;
@Override
public void setup(Context context){
    String centroid_string = context.getConfiguration().get(CentroidSampling.FLAG_CENTROID);
    centroids = centroid_string.split("\t");
    centroids_size = centroids.length;
}
```

map

```
input : key:offset value:Px’s position
output: key: Ci value :dist<Ci,Px>
```

从样本集加载点 Px 作为 input，对所有中心点做一遍遍历，寻找最近的中心点 Ci，成为 Si 的 inner 元素，并输出 Px 与 Ci 距离。

reduce

```
Input: key: Ci value:list<dist<Ci,Px>>
output: key:Ci#Ri value :Null
```

此处，我们找到了以 Ci 为中心点的集合 Si 的所有点，这些点距离 Ci 的信息被存储在 value 的 list 中。我们对 list 做一遍遍历，找到集合内距离圆心最远的点离圆心的距离(即 value 的最大值)作为此中心点的半径，并以 Ci，Ri 间距离为输出。

3. Similarity Mapper

setup

读取 CentroidStats 阶段的输出（即各点圆心及半径）。同样地，在上一阶段执行结束后，通过一个函数将圆心及半径信息读入字符串，通过 conf 传递，在此不做赘述。

map

```
input : key:offset value:Px’s position
output: key: Ci#Ri value :Px#pointType#dist<Px,Ci>
```

此阶段，我们主要对每个点做集合的归类，从而得到各集合完善的内部点信息。首先，我们读入点 Px，找到 Px 最近的圆心点 Ci，将 Px 标记为 Ci 的 inner 类型点并输出；再找到所有距离在 Rj+r 范围内的圆心 Cj, 将 Px 标记为 Cj 的 outer 类型点并输出。在对所有点进行一遍遍历之后，我们就能够得到每个以 Ci 为圆心的点集 Si 的所有 inner 与 outer 点。

reduce

```
input : key: Ci#Ri value :list<Px#pointType#dist<Px,Ci>>
output: key: Px,Py value :Null
```

对每个以 Ci 为圆心的点集 Si，我们对所有内部点做如下处理：

Inner 与 inner 逐个匹配，若距离小于 r，则输出。（复杂度 O(n2)）

inner 与 outer 逐个匹配，若距离小于 r，则输出。（复杂度 O(n2)）；

不用对 outer 间做逐一匹配的原因是若两个 outer 在此点集中距离小于 r，它们必然会在另一个点集中互为 inner/inner 或 inner/outer 关系。

这样的做法保证了对所有距离为 r 之内 pairs 的包含，但是具有两大缺点：

- 复杂度高，效率低
- 存在冗余

针对前者，我们在 compression 里进行优化；对于后者，我们在 exploting commutativity 进行优化。

4. Exploting commutativity

这一阶段的优化目标为去除所有冗余的 pairs。

由于各个点集的 inner 点互不重叠，故不存在 Inner pairs 的冗余；但 Inner to Outer pairs 必然存在冗余。该证明在论文中未提及，故在此处特别说明：

证明：设集合 Sa 存在 Inner 点 P1 与 Outer 点 P2，两者构成 Inner to Outer pairs，则两点间距离必小于 r；故对 P2 所在集合 Sb，dist<P1,Cb> <= dist<P2,Cb>+r <=Rb + r，即 P1 必属于 Sb 的 outer 点。故对 Sb 内所有点做运算时，P1 与 P2 也必然作为 Inner to Outer pairs 考虑再其中。

这种情况下，优化方式显而易见：二选一。论文通过 (((Pi.id + Pj.id) is odd) XOR (Pi.id < Pj.id))为真，则将 Pi 归入 Sj 的 outer，不将 Pj 归入 Si 的 outer 的方式实现。该方法在“二选一”的正确性上显然。

5. Compression of pairs

在寻找相似 pairs 的过程中，对于比较集中的数据集或数据的子集，我们实际上可以换一种描述方式表示集合中的 pairs。论文中提到了一种高效的描述方式——基于拓扑的描述方式，本次项目以其为基础进行了实现。

本项目中选取了三种拓扑结构进行压缩，分别是 Clique，Biclique，Hub。其中 clique 拓扑的压缩率时最大的，biclique 次之，本项目中的 hub 等同于无压缩输出（这与选取的粒度相关）。

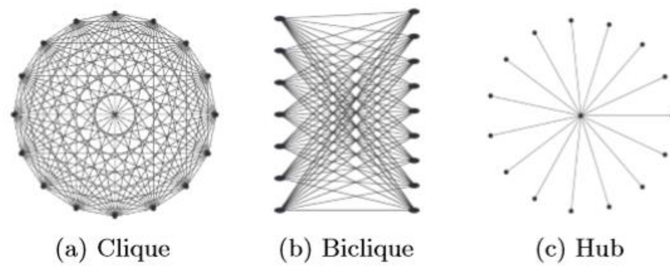
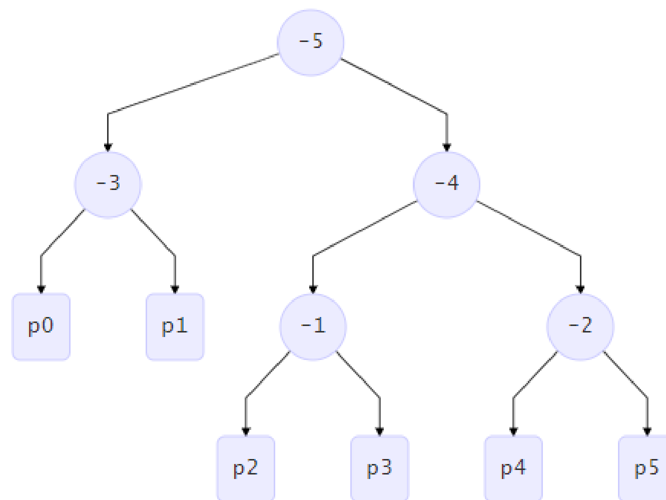


Figure 5: Compression Helps Exhibit the Community Structure

为了构建三种拓扑结构，项目设计过程中采用了层次聚类的方法，层级聚类的实现使用了二叉树结构。层次聚类的聚类条件是 Flexible 的，本项目中将距离最近的两个点集进行聚类。对于两个点集距离的衡量，项目使用了质心作为代表进行运算。有关层次聚类建立的原理和过程此处不做详细介绍，详情可参考代码。



二叉树建立好后，递归遍历每一个非叶子节点，检查该节点的两个子节点能否组成一对压缩结构。检查的过程也是个递归过程，如果该层不能构成，则检查其子集是否可能构建起压缩结构（即考察叶子节点和叶子节点，叶子节点和父节点的兄弟节点之间的关系），一直遍历到叶子节点上，直接输出一个 pairs。可以证明，这并不会有重复的计算。

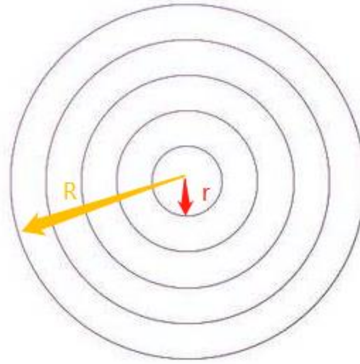
需要注意的是，由于层次聚类树的建立过程中没有考虑 inner 和 outer 的区别，所以此处会造成结果的较大冗余，严重的情况下，甚至会导致压缩后的结果比压缩前还要大。可选的解决方案是，支队 inner 的点建立层次聚类结构，inner 和 outer 之间仍然按照之前的方式计算和输出。

6. Repartition

Repartition 主要针对两种情况设立：

- 点集分配不均匀的短板效应，某些点集跑很久而其它点集很快跑完，影响总效率
- 某些点集很大，占用大量的内存空间

因此，我们需要在预估计算次数上限，来判断是否需要进行 repartition



如上图所示，我们在 SimilarityMapper 阶段将每个半径为 R 的 Workset 划分为半径为 r 的圆环。若两个点构成距离为 r 以内的 pairs，它们或在同一圆环内，或在相邻圆环上；因此，统计每个圆环内的点数，并对各圆环点数平方及相邻圆环点数积求和，即可预估比较次数的上限。 若上限大于某一阈值，则将 key: Ci value :list<Px#pointType#dist<Px,Ci>>原封不动输出到待处理文件中，并将 Ci，Ri 添加到缓存文件；否则，按原步骤处理。

进入 Repartition 后，对缓存文件中每个 Ci 对应的工作集 Si 以类似 CentroidSampling 阶段产生一系列小中心点 CCi。接着，对原工作集 Wi(半径为 Ri)中的 inner 点 P 找最近的中心点 CCi，算作 CCi 的 inner；找距离在 R1+t 的点 CCj,算作 CCj 的 outer。后续介绍的 SecondaryKey 的引入，保证了所有 inner 点可以在 outer 点之前进入，这样在处理 outer 时，我们已经确定了每个 CCi 的半径。对所有的 outer 点，我们找最近的中心点 CCi，算作 CCi 的 inner；找距离在 Rj+t 内的点 CCj,算作 CCj 的 outer。至此，新的 workset 已经建立。

Repartition 的 reduce 阶段，按类似于 SimilarityMapper 中的 reduce 阶段类似进行，不做赘述。

7. Secondary Key

SecondaryKey 对于 Reuce 负载的调节具有重要作用，其主要用途是保证 shuffle 阶段的分组按照用户自定义的标准进行，并且保证 Reduce 的输入按照用户所想要的顺序进行排列。本次项目中，通过 SecondaryKey 的设计，保证

了 inner 和 outer 进入同一个 reduce 的同时，保证所有的 inner 点出现在 outer 点之前（这里的“所有”所指的范围为一个 workset）。实现方法如下：

- 构造自定义数据类 BigKey：包含两个成员变量 `firstKey 和 secondKey`
- 重写 Partitioner 方法，按照 fisrtKey 分区
- 重写 WritableComparator 方法，按照 firstKey 比较

六、 集群搭建

集群配置:

- Linux 版本：Ret Hat4.8.5
- jdk 版本：jdk1.8.0_191
- hadoop 版本：hadoop-2.8.5
- zookeeper 版本：zookeeper.3.5.4-beta

本项目在 5 台虚拟机上搭建了 Hadoop 分布式集群，用于测试前述所实现工程。集群通过 zookeeper 等设置解决了单点故障问题，增强了集群的容错性。由于虚拟机自身问题，节点 data4 运行速度极慢，为保证集群稳定性，该节点被手动移除。主从节点服务如下表：

主节点	Secondary Name Node	从属节点
[hadoop@master ~]$ jps 32258 QuorumPeerMain 7477 NameNode 14678 ResourceManager 30599 Jps 7738 DataNode 31467 DFSZKFailoverController 763 JournalNode	[hadoop@data1 ~]$ jps 21313 QuorumPeerMain 24596 NodeManager 26583 Jps 21480 JournalNode 20972 DFSZKFailoverController 23132 DataNode	[hadoop@data2 ~]$ jps 29522 DataNode 28965 QuorumPeerMain 29141 JournalNode 30253 NodeManager 2863 Jps

七、 结果与分析

1. 数据

数据 1：随机点（2 维，3 维，7 维，20 维等），最大的测试数据大小维 2GB 的 7 维数据。

数据 2：处理后的 Netflix price 数据。使用 mapreduce 进行了数据的清洗，得到了 7GB 的数据。数据内容为一个用户对所有电影的观看情况(有评分记录则认为看过，没有评分记录则认为没有看过)

2. 效果

对于 MB 级别的数据，算法模型表现良好，运算时间大概在 1-2h 级(最主要的限制在于第一步中的分区过程计算量较大)。而对于 GB 级的数据，5 个节点的集群已经无法在可接受的时间内处理。

3. 压缩

本项目所设计的压缩策略会随着数据量的增大而更好，由于测试的数据集比较小，所以压缩效果不是很显著，但是可以看到，压缩确实可以减小输出的条数。

八、 问题分析

1. 随机点生成瓶颈

对数据进行测试时，发现随机生成中心点对程序性能有一定的限制，由于随机生成是在单机上运行的，一旦生成的随机点个数过多，就会导致运行速度过慢，在 5 个单位大小的 7 维空间中生成 10000 个满足条件（间隔不小于 0.01）的随机点集需要花费 6min 左右。其原因在于随机点个数一旦增多，发生“碰撞”概率就会增大，算法为了避免碰撞而花费了巨大的代价。如果数据量更大，那么生成的速度会更慢，且速度的降低是非线性的。如果将随机点个数调小一些，那么就有可能会造成分区过大，造成 reduce 阶段溢出，需要引入 repartition 机制。

提高随机点生成速度的方法，可以采用分区法，类似于寻找 pairs 的思想，将随机点生成的任务分配到若干相互不会影响的子区间内生成，以减少碰撞的概率。

中心点

另外，中心点过多也导致了分区过程中的运算量增大，对程序的运行有一定影响。

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unh Nc
1499	0	3	1496	54	210 GB	394.81 GB	0 B	54	128	0	16	0	2	1

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking
application_1542799593765_1600	u16307130138	CentroidStats	MAPREDUCE	default	Thu, 20 Dec 2018 00:22:11 GMT	N/A	RUNNING	UNDEFINED		Application
application_1542799593765_1599	u16307130138	CentroidStats	MAPREDUCE	default	Thu, 20 Dec 2018 00:18:08 GMT	N/A	RUNNING	UNDEFINED		Application
application_1542799593765_1598	u16307130138	CentroidStats	MAPREDUCE	default	Wed, 19 Dec 2018	N/A	RUNNING	UNDEFINED		Application

上图三个任务从上到下依次时 100、1000、10000 个中心点，截图的时间为最

上面的 Job 开始后 2min，可以看到他们的速度差异时积极显著的（越下面的 job 开始时间越早）。

2. 阈值的设置问题

阈值的设置和随机点的生成以及后续的聚类和判断都有关系，阈值对算法的影响也比较大。除去论文对阈值的分析外，本项目针对阈值也进行了简单的测试。阈值的设置如果过大，会导致中心点生成数据无法保证，而如果过小，又会造成生成的 pairs 过多。

3. 优化方向

1. 首先是随机点生成的优化，可以采用类似于 similarity search 的分区的思想，前面已经提过
2. 其次是分区时可以改进原本的 Niave 分区算法，使用最近邻搜索算法或 Hash 算法，找到某一个点所属的分区。一种比较合适的算法时 GeoHash 算法，它在地图搜索中有大量应用，与本项目面对的场景很像。
3. 再者就是 Compress 可以通过调节粒度以及分类标准获得更佳的效果。

九、 项目分工

- 集群搭建 常朝坤
- 项目架构设计 何畅扬
- 基础算法实现 何畅扬
- SecondaryKey 设计 前期何畅扬，后期常朝坤
- Repartition 何畅扬 曾瑞莹
- Compress 优化 常朝坤
- 数据搜集与整理 常朝坤
- 项目运行与测试 常朝坤
- 报告书写 何畅扬 常朝坤
- PPT 制作 何畅扬

十、 参考文献

1. Scalable All-Pairs Similarity Search in Metric Spaces

2. https://www.cnblogs.com/datacloud/p/3584640.html

3. https://blog.csdn.net/c_son/article/details/43900503

4. http://www.aboutyun.com/thread-9353-1-1.html