Apriori Versions Based on MapReduce for Mining Frequent Patterns on Big Data

José María Luna, Francisco Padillo, Mykola Pechenizkiy, and Sebastián Ventura

摘要

模式挖掘是从原始数据中提取有意义和有用信息的最重要任务之一。这项任务的目的是提取代表数据中任何类型的同质性和规律性的项目集。已有学者提出了改进后的运用于 MapReduce 框架上的 Apriori 算法 SPAprioriMR 及 TopAprioriMR，用于处理大规模的模式挖掘任务，本文则结合以上两种算法做了进一步改进，形成 MyApriori 算法。首先，本文所实现算法可以提取数据集中全部频繁项集；其次，本算法可分别展示 k 项集中所存在的频繁项集；最后，优化了 TopAprioriMR 的剪枝策略以及文件读写策略。结果显示，在对规模较大的数据进行处理时本文所提出的算法在各方面都优于 TopAprioriMR 算法。

关键词：大数据；MapReduce；模式挖掘

1 引言

数据分析在商业智能等许多领域都有越来越大的应用，它包括一套将原始数据转化为有意义和有用信息的技术。随着数据在每个应用中的重要性不断增加，需要处理的数据量已经变得无法管理，而这种技术的性能可能会下降。大数据一词越来越多地被用来指代以有效方式处理这种高维数据集所带来的挑战和进步。[1][2]

模式挖掘被认为是数据分析和数据挖掘的一个重要部分。其目的是提取代表数据中任何类型的同质性和规律性的子序列、子结构或项目集，表示内在的和重要的属性。这个问题最初是在购物车分析的背景下提出的，目的是找到一起购买的最频繁的产品。自从它在 90 年代初被正式定义以来，有大量算法被提出。这些算法中的大多数是基于类似 Apriori 的方法，产生一个由任何单项组合形成的候选项目集或模式列表。然而，当这些要组合的单项的数量增加时，模式挖掘问题变成了一项艰巨的任务，需要更有效的策略。

传统的模式挖掘算法并不适合真正的大数据，呈现出两个有待解决的主要挑战。1）计算复杂性和 2）内存要求。在这种情况下，单台机器上的顺序模式挖掘算法可能无法处理整个程序，因此，对其进行调整以适应分布式计算可能是解决该问题的一项有效途径。

Moens 等人首次提出了基于 MapReduce 的方法 BigFIM 来挖掘大数据集上的项目集[3]，Luna 等人则基于 Apriori 提出了一系列运行于 MapReduce 框架上的 Apriori 改进算法 AprioriMR[4]。考虑到已有学者在分布式计算上实现模式挖掘算法，本文的目的则是期望基于原有 Apriori 算法设计一个更有效率且更为强大的大数据模式挖掘算法。

2 相关工作及论文方法

在该部分中我们简要介绍传统的 Apriori 算法与基于 MapReduce 计算框架改进后的 Apriori 算法。

2.1 传统的 Apriori 算法

在这个著名的算法中，Agrawal 等人提出在每个事务中生成所有可行的项目集，并为它们分配一个支持度。然后，该算法检查每个新项目集是否已经由之前的事务生成，如果是，则它们的支持度会统一增加。对每行事务重复相同的过程，产生模式列表 L，该模式列表中含有每个已发掘模式的支持度或出现频率。正因如此，当事务和单例的数量越多，算法计算复杂度和内存需求就越高。值得注意的是，大量事务在 Apriori 算法中意味着大量迭代，因此运行时间会大大增加。同时，事务中大量的单例相互组合将会产生大量的候选项集，对每个候选项集的判断又大大增加了计算复杂性和内存需求。这也为我们之后改进 Apriori 算法指明了方向。

2.2 基于 MapReduce 计算框架的 Apriori 算法

为了减少传统 Apriori 算法的密集计算所需时间，Luna 等人提出了一个基于 MapReduce 的 Apriori 算法。该算法的工作原理是为每个特定的子数据库运行一个 Mapper，每个 Mapper 负责为每个子数据库挖掘完整的项目集。这里，事务 $t_l \in \mathcal{T}$ 中的每个模式 P 以 $\langle k, v\rangle$ 的形式表示，其中键 k 是模式 P，而值 v 是来自第 l 个事务的 P 的支持度。之后，运行 shuffle 和 sort 程序，将不同的 $\langle k, supp(P)_l\rangle$ 分组。由相同的键值 P 配对，并以 $\langle P, \langle \text{supp}(P)_l, \ldots, \text{supp}(P)_m\rangle\rangle$ 的形式输入到 Reducer 中。最后，在 Reducer 中将支持度进行累加得到模式 P 的总支持度，并与支持度阈值进行判断决定是否输出为频繁项集。

3 论文方法

在本文中，我们通过详细学习 Luna 等人所提出的基于 MapReduce 框架改进的 Apriori 算法，同时思考对比其所提出的方法 SPAprioriMR 和 TopAprioriMR 的算法逻辑及其优缺点，在其基本思想不变的条件下对这两种方法加以结合，并结合其不足之处加以改进，旨在在复现出这两种算法的基础上使得代码运行效率相对于上述两种算法在同一现实数据集上能取得更优秀的表现。

3.1 本文方法概述

在所有事务中寻找所有模式的任务是相当具有挑战性的，因为搜索空间随着事务中出现的单项数量呈指数级增长。给定一个含有 n 个单项的数据集，在数据集中可最多找到 $2^n - 1$ 个不同的模式. 此外，这个数据集可能包含大量的事务，计算每个模式频率的过程也是一个很棘手的问题。所有这些都凸显了对搜索空间进行剪枝处理的重要性。有一个重要的剪枝策略是反单调性，其定义为频繁模式的任何子模式也是频繁的，而包括不频繁模式的所有模式将永远不会是频繁的。本文即是以该策略为核心而设计的基于 MapReduce 计算框架的迭代式 Apriori 算法。

3.2 算法介绍

在这里，我们基于 Luna 等人所提出的 SPAprioriMR 和 TopAprioriMR 方法，变换代码逻辑实现了本文的 MyApriori 方法并应用于 MapReduce 计算框架。该算法是一种迭代式的计算方法，通过考虑最小支持度阈值来对搜索空间进行剪枝进而挖掘频繁项集，并从挖掘频繁单项开始，$\{\forall i \in I : |i| = 1 \wedge \text{support}(i) \geqslant threshold\}$。其中 I 表示数据集的单项集。MyApriori 算法随后迭代的挖掘频繁二项集，其二项集候选集从频繁单项集中组合生成，同时二项集候选集的各单项子集均存在于频繁单项集中。这个过程将会迭代多次，在第 s 次迭代中，MyApriori 算法将挖掘出频繁 s 项集，且该 s 项集的任意 $s-1$ 项子集均存在于频繁 $s-1$ 项集中。

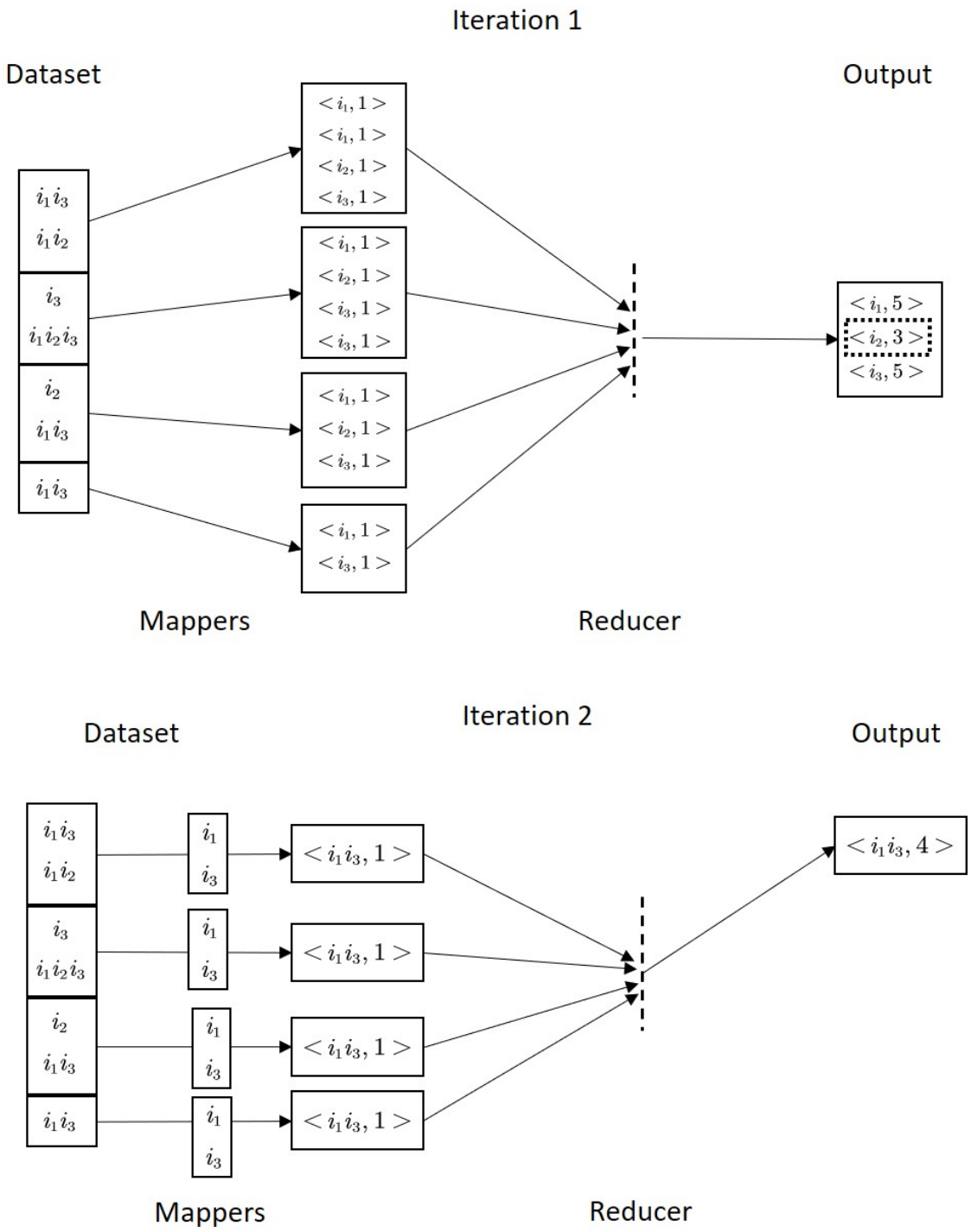

图 1: MyApriori 在处理样本数据集时的迭代方式，最小支持度阈值设定为 4

为了更好理解本算法，图 1演示了 MyApriori 算法对一个样本数据集的处理过程。在本样本案例中，最小支持度阈值为 4，所以任何支持度小于这个值的模式或项目集都被标记为不经常出现（虚线），将不予转发至 Recucer 处理。该算法在第一次迭代中挖掘了所有的单项，发现 i_2 是不经常出现的，因为它只被三个事务满足 (支持度阈值设置为 4)，因此在第一轮迭代中我们丢弃 i_2 项，转而输出满足的频繁一项集集合 $\{i_1, i_3\}$；在下一次迭代中，每个 Mapper 将接受两组参数，第一组参数是初始的数据集，第二组参数是上一轮迭代中所筛选出的 $s-1$ 频繁项集集合，这一轮迭代的各 Mapper 将从 $s-1$ 频繁项集集合中选择两个频繁项集进行组合，形成 s 项集的候选集合，再与数据集中各事务进行对比，从而产生将传入 Reducer 的 s 项集，最终各个 s 项集在 Reducer 中进行新一轮支持度聚合，产生出 s 项频繁项集。一旦达到频繁项集所筛选的最大项数，该算法将停止，此时各轮迭代所产生的输出将是

数据集中所有的频繁项集。

3.3 算法实现伪代码

Procedure 1 MyAprioriMapper(t_l, s)

Input: Dataset t_l, itemsize s

1: L_{fre} = load the list of frequent item-sets of size $s-1$
2: C = { P : generate candidate patterns P of size s from L_{fre} }
3: C = { P : $\forall${subsets of P with size $s-1$} $\in L_{fre}$ }$\cap C$
4: $\forall P \in C$, then $supp(P)_l = 1$
5: **for all** $P \in C$ **do**
6: emit $\langle P, supp(P)_l \rangle$
7: **end for**

Procedure 2 MyAprioriReducer(t_s, s)

Output: L_{fre} the list of frequent item-sets of size $s-1$

1: $support = 0$
2: **for all** $supp \in \langle supp(P)_1, supp(P)_2, \ldots, supp(P)_m \rangle$ **do**
3: $support+ = supp$
4: **end for**
5: **if** $support > threshold$ **then**
6: emit $\langle P, support \rangle$
7: **end if**

4 复现细节

4.1 与已有开源代码对比

本次论文复现无源码参考。

在上述提到的伪代码中，Mapper 阶段的候选集由两步生成，第一步则是 Procedure1 中第 2 行，从 $s-1$ 频繁项集中两两组合产生 s 项候选集合，参考代码如下所示：

```
private List<List<Integer>> getCandidateItemsets(List<List<Integer>>
   prevItemsets, int passNum) {
   List<List<Integer>> candidateItemsets = new ArrayList<List<Integer>>();
   for (int i = 0; i < prevItemsets.size(); i++) {
      for (int j = i + 1; j < prevItemsets.size(); j++) {
         List<Integer> outerItems = prevItemsets.get(i);
         List<Integer> innerItems = prevItemsets.get(j);
         List<Integer> newItems = null;
         if (passNum == 1) {
            newItems = new ArrayList<Integer>();
            newItems.add(outerItems.get(0));
            newItems.add(innerItems.get(0));
         } else {
            int nDifferent = 0;
            int index = -1;
            for (int k = 0; k < passNum; k++) {
               if (!innerItems.contains(outerItems.get(k))) {
                  nDifferent++;
                  index = k;
               }
            }
            if (nDifferent == 1) {
               newItems = new ArrayList<Integer>();
               newItems.addAll(innerItems);
               newItems.add(outerItems.get(index));
            }
```

```
            }
            if (newItems == null) continue;
            Collections.sort(newItems);
            if (isCandidate(newItems, prevItemsets) && !candidateItemsets.
               contains(newItems)) {
               candidateItemsets.add(newItems);
               //System.out.println(newItems);
            }

         }
      }
      return candidateItemsets;
   }
```

在 Procedure1 中第 3 行则是产生候选集的第二步，在该步中，由第一步两两组合所产生的 s 项集合将进行进一步剪枝，即将该集合中的所有元素一一取出并判断其 $s-1$ 项子集是否均存在于 $s-1$ 项频繁项集之中，若有一子集不存在，则该 s 项项集不能成为候选集，只有 $s-1$ 项子集均存在 $s-1$ 项频繁项集之中，该项集才能加入到 s 项候选集列表中，该步骤代码演示如下，其中若可作为候选集集合中一员则返回 true，如果不能则返回 false：

```
   private boolean isCandidate(List<Integer> newItems, List<List<Integer>>
      prevItemsets) {
      List<List<Integer>> subsets = getSubsets(newItems);
      for (List<Integer> subset : subsets) {
         if (!prevItemsets.contains(subset)) {
            return false;
         }
      }
      return true;
   }
```

至此 Mapper 阶段的关键步骤代码演示完毕，最终将 Map 操作之后形成的键值对传入 Reducer 阶段进行聚合即可，Reducer 阶段仅需负责计算某一项集所对应的支持度，若该项集支持度大于给定的最小支持度阈值，则将结果写出，若该项集支持度小于或等于给定的最小支持度阈值，则将该项集舍弃，不与操作。Reducer 阶段代码演示如下：

```
   public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
      int sum = 0;
      int minSup = context.getConfiguration().getInt("minSup", 5);
      for (IntWritable val : values) {
         sum += val.get();
      }
      result.set(sum);
      if (sum > minSup) {
         context.write(key, result);
      }
   }
```

4.2 实验环境

本次实验在 Hadoop 集群上运行，该集群确保计算节点充足，每个节点都包含 16 个内核，物理内存为 125.7 吉字节，拥有 8.0 吉字节的交换空间，运行操作系统为 centos 7.5.1804。具体来说，实验在 Hadoop 3.0.0 上运行，每个任务都有 12 个 Map 任务，最终汇总为 1 个 Reduce 任务进行输出。

4.3 创新点

1) 不同于 Luna 等人所提出的 TopAprioriMR 算法只能挖掘出最频繁项集，本文所实现算法可以提取数据集中全部频繁项集；

2) 不同于 Luna 等人所提出的 TopAprioriMR 算法，该算法在每一步迭代过程中都会对输入数据进行处理并写出，而在大数据环境下，输入的数据集通常会非常庞大，而这一步操作将大大增加算法的 IO 时间，使得文件 IO 时间将在很大程度上影响算法的总体运行时常。在本次所改进的算法中，我们优化了 TopAprioriMR 的剪枝策略以及文件读写策略，我们在每一轮迭代时舍弃输入不频繁项集与输出更新后的数据集的操作，转而读入上一轮迭代所产生的频繁项集，大大缩小了文件的 IO 时间对总体运行时间所造成的影响。

5 实验结果分析

同 Luna 等人论文中一样，我们将算法应用于 FIM 社区中可用的最大数据集 Webdocs 数据集。这个数据集包含 5267656 个单项，这些单项分布在 1692082 条事务记录之中。该数据集将产生 $2^{5267656}$ 大小的搜索空间。论文中所提出算法 TopAprioriMR 与本文所复现算法 MyApriori 算法在该数据集上的实验结果见表 1 中，表格展示了针对不同的最小支持度阈值所发现的模式数量以及算法的运行时间。

表 1 TopAprioriMR 与 MyApriori 算法在 Webdocs 数据集上运行时间与挖掘的频繁项集

算法	支持度阈值	运行时常 (min)	频繁项集个数
TopAprioriMR	338416（0.20）	192.68	319
TopAprioriMR	846041（0.50）	117.76	9
MyApriori	338416（0.20）	14.23	453
MyApriori	846041（0.50）	6.38	10

6 总结与展望

本部分对整个文档的内容进行归纳并分析目前实现过程中的不足以及未来可进一步进行研究的方向。本次论文复现中我们学习了 Luna 等人所提出的适用于 MapReduce 框架的 Apriori 算法，并结合其所提出的两种 SPAprioriMR 及 TopAprioriMR 进行了一定改进，形成了 MyApriori 算法，该算法优化了上述两种算法的剪枝策略以及文件读写策略，使得算法运行时间能相较于原算法有所提高，且能挖掘出数据集所有的频繁项集。

但基于 MapReduce 计算框架的算法存在一个问题，即该算法不适用于小规模数据集，在小规模数据集上其运行时间相较于串行算法会更低。因此之后的改进方向可将该复现的算法在更多维度上与其他基于 MapReduce 算法改进的模式挖掘算法进行对比，例如节点个数、事务数量等，同时也与串行算法在以数据规模为变量的条件下进行比较，从而得出在大规模数据上挖掘频繁项集最有效的方法，同时得到一个介于串行算法和分布式算法的阈值，即数据规模若超过该阈值即使用分布式计算来挖掘频繁项集，若低于该阈值即使用串行算法来挖掘频繁项集。

参考文献

[1] HAN J, CHENG H, XIN D, et al. Frequent pattern mining: Current status and future directions[J]. Data Min. Knowl. Discov., 2007, 15: 55-86. DOI: 10.1007/s10618-006-0059-1.

[2] AGRAWAL R, IMIELINSKI T, SWAMI A. Database mining: a performance perspective[J]. IEEE Transactions on Knowledge and Data Engineering, 1993, 5(6): 914-925. DOI: 10.1109/69.250074.

[3] MOENS S, AKSEHIRLI E, GOETHALS B. Frequent Itemset Mining for Big Data[C]//2013 IEEE International Conference on Big Data. 2013: 111-118. DOI: 10.1109/BigData.2013.6691742.

[4] LUNA J M, PADILLO F, PECHENIZKIY M, et al. Apriori Versions Based on MapReduce for Mining Frequent Patterns on Big Data[J]. IEEE TRANSACTIONS ON CYBERNETICS, 2018, 48(10): 2851-2865.