面向大数据的高效关联规则推荐算法

易琦*，齐豪†

2019 年 6 月 21 日

目录

1 任务分析 **3**

2 并行化设计思路和方法 **3**

2.1 FP-Growth 介绍 3

2.2 FP-Growth 的并行化设计 4

2.2.1 计算频繁 1-项集 4

2.2.2 重分配条件模式集 5

2.2.3 并行挖掘频繁模式 5

2.3 生成关联规则 5

2.4 生成用户推荐 5

3 详细算法设计与实现 **6**

3.1 FP-Growth 的并行化设计 6

3.1.1 计算频繁 1-项集 6

3.1.2 重分配条件模式基 6

3.1.3 并行挖掘频繁模式 6

3.2 生成关联规则 7

3.3 生成用户推荐 7

*SA18225002,yiqi@mail.ustc.edu.cn

†qihao@mail.ustc.edu.cn

4 实验结果与分析 **8**
4.1 实验结果 8
4.2 实验分析 8

5 程序代码说明 **9**
5.1 项目结构 9
5.2 运行方式 9

1 任务分析

本题目共计一下几个任务：

- A 频繁模式挖掘
- B 关联规则生成
- C 关联规则匹配
- D 推荐分值计算

具体地，可以分为如下两个步骤：

（1）、给定购物篮数据集 D 和活跃用户数据集 U，编写 Spark 程序，以支持度阈值 minsupp=9.2 从数据集 D 中挖掘频繁模式（步骤 A）。

（2）将关联规则与数据集 U 中的用户概貌进行匹配并计算出每个用户的推荐项目（步骤 D）。为简单起见，赛题仅要求给出置信度最大的项（即 Top-1 项）作为推荐结果，如果置信度最大的项有多个，则给出编号最小的项作为结果。如果某用户没能产生推荐项（即没有关联规则与其概貌匹配），则以 0 作为结果。

2 并行化设计思路和方法

2.1 FP-Growth 介绍

FP-Growth 算法是一种高效的关联分析算法。它采取如下分治策略：将提供频繁项集的数据库压缩到一棵频繁模式树（FP-tree），但仍保留项集关联信息。通过构建 FP-Growth 的条件模式基，可以将 FP-Growth 部署到分布式系统上，充分利用分布式系统的算力。因此对于任务（1），我们将采用 FP-Growth 作为频繁模式挖掘的主要工具，并在此基础上进行了一些改进。

FP-Growth 的算法如图 2.1。

Input: *FP - tree* constructed using *DB* and
a minimum support threshold ξ.
Output: The complete set of frequent patterns.
Method: Call FP- growth (*FP - tree*, *null*).
Procedure: FP- growth (*Tree*, α)
{
  if *Tree* contains a single path *P*
  then foreach combination(denoted as β)
  of the nodes in *P* **do**
    generate pattern $\beta \bigcup \alpha$ with *support* =
      minimum support of nodes in β;
  else foreach a_i in the header of *Tree* **do** {
    generate pattern $\beta = a_i \bigcup \alpha$ with
      $support = a_i.support$;
    construct β's conditional pattern base and
      then β's conditional FP-tree $Tree_\beta$;
    if $Tree_\beta \neq \varnothing$ **then**
      call **FP - growth**($Tree_\beta$, β); }
}

图 1: FP-Growth 算法

2.2 FP-Growth 的并行化设计

FP-Growth 的并行化算法主要包含有三个部分：

- 计算频繁 1-项集
- 重分配条件模式基
- 并行挖掘频繁模式

2.2.1 计算频繁 1-项集

在这一步，首先读取数据集作为 RDD，然后计算每一个单项的支持度，将支持度大于支持度阈值的项提取出来，得到频繁 1-项集。同时删去数据

集支持度小于支持度阈值的项，得到精简后的数据集。

2.2.2 重分配条件模式集

假设频繁 1-项集为 L。将 L 按照支持度大小重新排序。对第 1 步中得到的数据集的每一项也按照支持度排序，得到 T。使用 T 中每一项生成若干个条件模式基，将这些基和 L 中的元素相对应。具体地，假设 $T = \{t_1, t_2, ..., t_k\}$，那么将生成 k 个条件模式基：$\{(t_1, ..., t_i) : 1 \leq i \leq k\}$。最后，将这些基按照其最大支持度的元素被映射到 L.size 个集合中（$(t_1, ..., t_i) \rightarrow P(t_i)$），记这些集合为 $P(a_1), P(a_2), ..., P(a_k)$，其中 $\{a_1, a_2, ..., a_k\} = L$。

2.2.3 并行挖掘频繁模式

对于 T 中的一个记录 $\{t_1, t_2, ..., t_m\}$，它的条件模式基被分别映射到了 $P(t_1), P(t_2), ..., P(t_m)$。因此，各个 $P(t_i)$ 可以独立的进行频繁模式挖掘，并且由于 $P(t_i)$ 不存在数据的相关性，通信量也极少。在这一步中，对每一个 $P(t_i)$ 使用 FP-Growth 算法。

2.3 生成关联规则

使用并行化 FP-Growth 生成的频繁模式的同时，可以方便的计算出各个频繁模式的支持度。现在只需要利用这些支持度给出相应的关联规则即可。

假设频繁模式的集合为 F，对应支持度为 S。对 F 中的每一条记录 f，f 所诱导的关联规则为：$\{(f - \{f_i\} \rightarrow f_i, conf = S[f]/S[f_i]) : f_i \in f\}$。这里，不用考虑 f 的子集所诱导的关联规则，因为 f 的子集也必定作为频繁模式包含在了 F 中。

2.4 生成用户推荐

将关联规则按照 conf 排序，得到 R。对于每一条用户的数据，只需在 R 中匹配一个 conf 最大的关联规则即可。这里的匹配是指，关联规则的右侧集合包含在用户数据中。

为了并行化计算，可以将用户数据进行分区，每一个分区单独计算。

3 详细算法设计与实现

由于大部分内容在上一节中已经讨论过了，因此这一节只给出核心的代码及注释。

3.1 FP-Growth 的并行化设计

相应的代码在 AssociationRuleMining/src/main/scala/spark/DFP.scala 中.

3.1.1 计算频繁 1-项集

如算法 1所示。

Algorithm 1: 计算频繁 1-项集

```
1 singletons = transactions
2       .flatMap(identity)
3       .map(item => (item, 1))
4       .reduceByKey(_ +_)
5       .filter(_._2 >= support)
```

3.1.2 重分配条件模式基

如算法 2 所示。

Algorithm 2: 重分配条件模式基

```
1 retrans = transactions
2       .map(t => pruneAndSort(t, sortedSingletons))
3       .flatMap(buildConditionalPatternsBase)
4       .groupByKey(sortedSingletons.length)
```

3.1.3 并行挖掘频繁模式

如算法 3。

Algorithm 3: 并行挖掘频繁模式

```
1 result = retrans
2       .flatMap(t => minePatternFragment(t._1, t._2.toList,
    minSupport))
3       .collect().toList ++ sortedSingletons.map(List(_))
```

3.2 生成关联规则

相应的代码在 AssociationRuleMining/src/main/scala/spark/ calculateSupport.scala 中。如算法 4 所示。

Algorithm 4: 生成关联规则

```
1  freq_supp.keys.foreach(freq_item=>{
2       if (freq_item.size > 1) {
3            freq_item.foreach(item=>{
4                 rules = rules :+ ((remove(freq_item, item),
     List(item), freq_supp.get(freq_item).get.toDouble /
     freq_supp.get(List(item)).get.toDouble))
5            })
6       }
7       }
8  })
9  sorted_rules = rules.map(t=>(t._1.sorted, t._2,
     t._3)).sortWith((a, b)=>le(a, b))
10 sorted_rules
```

3.3 生成用户推荐

如算法 5 所示。

Algorithm 5: 生成用户推荐

```
trans_pattern.map(item=>
    findrecom(rules, item)).collect().toList
```

4 实验结果与分析

4.1 实验结果

在 AssociationRuleMining/output 中保存有一次运行的所有输出。其中，各个文件的含义如下：

freqitem.txt	使用 FP-Growth 挖掘出的频繁模式
freqsupp.txt	频繁模式对应的支持度
rules.txt	生成的关联规则
result.txt	用户推荐的结果

表 1: 输出文件

本次实验中，由于硬件平台的限制，只在测试样例中运行了所设计的算法。相应的输入为 AssociationRuleMining/dataset/test_data/目录下的 trans_10W.txt 和 test_2W.txt。

运行的硬件平台为一台 6 核 Itel i7 的计算机。虽然没有使用分布式系统（因为我们只有一台电脑），但是却充分利用了多核的并行性，CPU 的利用率一般可达 400+%。在该平台上，各个部分的时间统计如表 2 所示

FP-Growth 挖掘	88min
生成关联规则	<1 min
生成用户推荐	23min

表 2: 时间统计

4.2 实验分析

从表 2 中可以发现，占主要时间的是并行 FP-Growth 挖掘。FP-Growth 计算时间主要来自于两个方面：

- 计算频繁 1-项集和重分配条件模式基。这一部分时间耗费较少，只需要几分钟就可以得到结果。
- 并行挖掘频繁模式。虽然样本数据经过了裁剪，但是还是十分大，导致计算量较大。此外，硬件平台的计算能力也十分有限，导致算法的执行时间较大。

此外，算法运行的内存开销也比较大，这是因为在 FP-Growth 中生成条件模式基时，会将数据重复赋值的原因。这虽然避免了多个进程之间的通信，但会引入额外的存储开销。在本例中，算法大部分时间都会占用 6G-7G 的内存。

5 程序代码说明

项目已经上传到了 gitee[1]上，可以通过这个连接访问：
https://gitee.com/albertcity/AssociationRuleMining/tree/master

5.1 项目结构

项目的主要的代码在 AssociationRuleMining/src/main/scala 目录下。在该目录中，sequential 文件夹中含有普通 FP-Growth 的实现，spark 文件夹下有分布式 FP-Growth 以及关联规则的挖掘等的实现，是本实验的主要工作。

5.2 运行方式

可以采用单机模式运行。分为以下几个步骤：

- 安装 java 8。更高版本的 java 与 spark 不兼容
- 安装并配置 sbt。记得将 java 的内存设置的大一点，否则会溢出。可以参考 https://blog.csdn.net/a532672728/article/details/72477591
- 在项目目录下，运行 sbt run。此命令会自动根据 build.sbt 中的依赖下载相应的环境，但由于我们已经把下载的依赖包一并上传了，因此会直接开始编译并执行

[1]一个国内的 github 镜像网站，下载和上传速度更快

- 选择 Main Object 执行。

此外，也可以像普通 spark 工程那样在集群上运行。这里就不多叙述了。

值得注意的是，为方便起见，已经将输入文件的路径写死在了程序里。可以通过更改 AssociationRuleMining/src/main/scala/Mail.scala 文件来使用任意的输入。