哈尔滨工业大学

<<大数据分析>>
大作业报告

(202_年度春季学期)

姓名：	
学号：	
学院：	计算学部
教师：	

目录

第 1 章 需求分析

1.1 研究问题的背景

随着线上电商业务有业务的发展，根据客户已有的购买信息为客户推荐他们可能在未来购买的商品这一需求逐渐成为了数据驱动商业智能的重要趋势；如果做好产品推荐与引导，是评估一个在线电子商务平台发展潜力的重要指标。在做产品推荐时，一个很重要的手段便是对产品进行关联分析；；关联分析做推荐时，主要用于个性化不强的场景。比如根据购买记录，通过关联分析发现群体购买习惯的内在共性，指导超市产品摆放。对于偏个性化场景，比如给目标用户推荐产品，可以先找出购买习惯与目标用户相似的人群，对此特定人群的购买记录进行关联分析，然后将分析出的规则与目标用户的购买记录结合，进行推荐。

1.2 问题的需求分析

通过发掘潜在客户，精准营销，其主要是思路是通过关联分析，发现许多购买 A 的用户还会购买 B，即有规则 A—>B，可通过有购买 B 产品行为的用户，找到 A 产品的潜在意向用户，进行精准营销。

关联分析有个最著名的案例就是：啤酒与尿布的故事：

“原来，在美国，妇女们经常会嘱咐她们的丈夫下班以后给孩子买一点尿布回来，而丈夫在买完尿布后，大都会顺手买回一瓶自己爱喝的啤酒（由此看出美国人爱喝酒）。商家通过对一年多的原始交易记录进行详细的分析，发现了这对神奇的组合。于是就毫不犹豫地将尿布与啤酒摆放在一起售卖，通过它们的关联性，互相促进销售。“啤酒与尿布”的故事一度是营销界的神话”。

输入的待分析关联规则挖掘数据如下记录所示：

示例：

1. 柑橘类水果 人造黄油 即食汤 半成品面包
2. 咖啡 热带水果 酸奶
3. 全脂牛奶
4. 奶油乳酪 肉泥 仁果类水果 酸奶
5. 炼乳 长面包 其他蔬菜 全脂牛奶
6. 腐蚀性清洁剂 黄油 白饭 全脂牛奶 酸奶
7. 面包卷
8. 瓶装啤酒 开胃酒 其他蔬菜 面包卷 超高温杀菌的牛奶
9. 盆栽
10. 谷物 全脂牛奶

因此我们的分析目的便是在一定的正确率要求下，获取某种购物关联规则，例如：瓶装啤酒 —>长面包,全脂牛奶 —>肉泥 等关联规则。

随着大数据时代的到来，海量数据对于以频繁项集挖掘为代表的很多传统数据挖掘算法的有效性和效率提出了挑战，因此我们可以通过大数据计算框架的手

段来解决这些问题。

1.3 研究问题的挑战

频繁项集关联规则挖掘中有多种可供选择的算法，其中一种便是 Apriori 算法，这种算法简单、易理解、数据要求低，但是 Apriori 算法本身也有很多不足之处，对数据库的扫描次数过多，而且扫描后可能产生大量的候选项集，在频繁项目集长度变大的情况下,运算时间显著增加，如何在大数据处理框架解决该问题是一个值得研究的话题。

第二章 系统设计

2.1 apriori 算法概论

Apriori 算法是一种最有影响力的挖掘布尔关联规则的频繁项集算法，它是由 Rakesh Agrawal 和 Ramakrishnan Skrikant 提出的。它使用一种称作逐层搜索的迭代方法，k 项集用于探索（k+1） 项集。首先，找出频繁 1 项集的集合。该集合记作L_1。L_1用于找频繁 2 项集的集合 L_2，而L_2 用于找L_3,….,如此下去，直到不能找到 k项集。每轮寻找L_k 需要进行一次数据库扫描。为提高频繁项集逐层产生的效率，通常采用一种称作 Apriori 性质用于压缩搜索空间。性质具体内容为：一是频繁项集的所有非空子集都必须也是频繁的，二是非频繁项集的所有父集都是非频繁的。

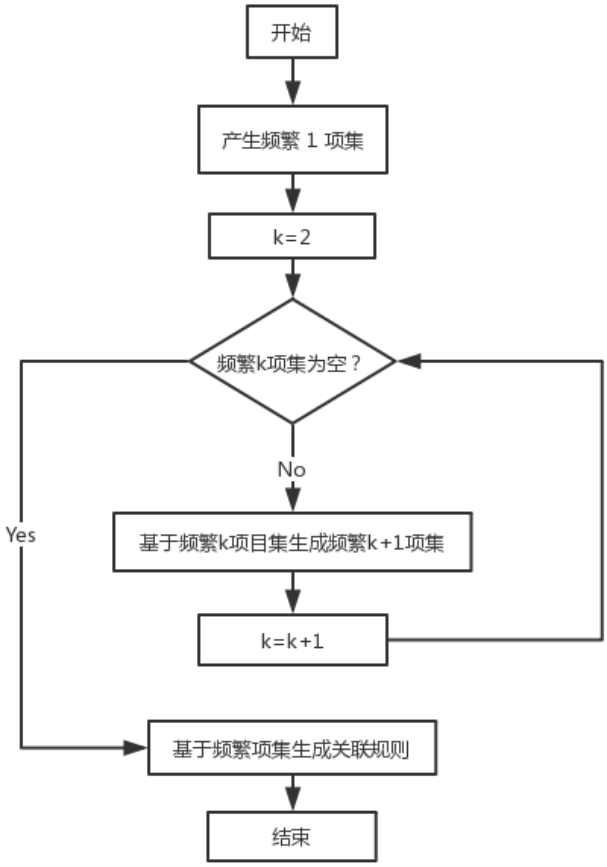
2.2 apriori 算法基本流程

首先找出所有的频集，这些项集出现的频繁性至少不限于设定的的最小支持度。然后由频集产生强关联规则，这些规则必须满足最小支持度和最小可信度。然后使用第 1 步找到的频集产生期望的规则，产生只包含集合的项的所有规则，其中每一条规则的右部只有一项，这里采用的是中规则的定义。一旦这些规则被生成，那么只有那些大于预设的最小可信度的规则才被留下来。

流程如下：

第一步通过迭代，检索出事务数据库中的所有频繁项集，即支持度不低于用户设定的阈值的项集；

第二步利用频繁项集构造出满足用户最小信任度的规则。



apriori 算法的基本流程图

2.3 Apache Spark 简介

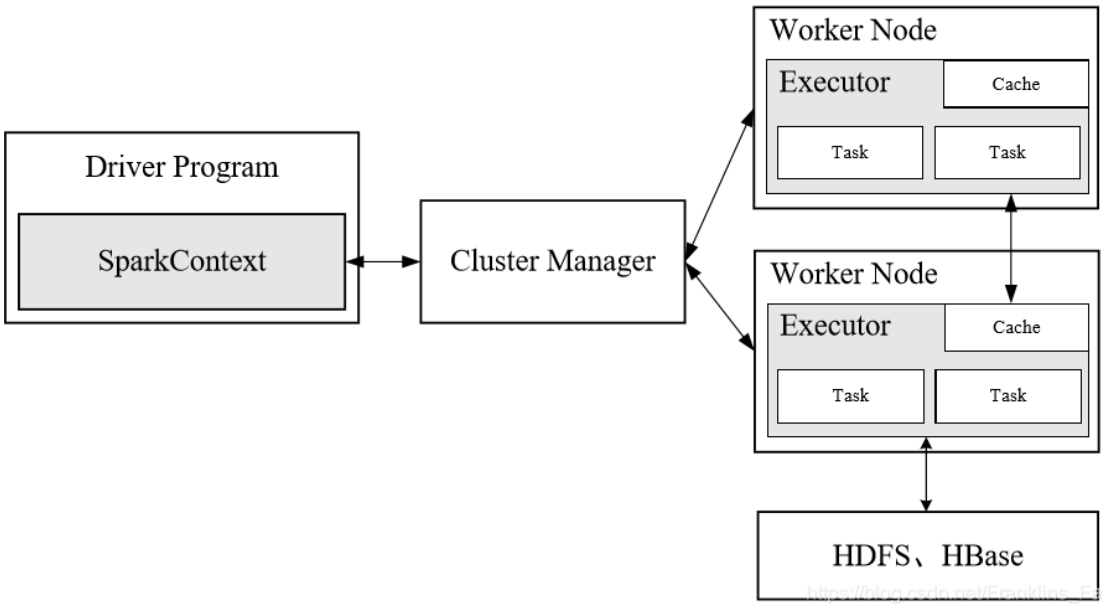
Apache Spark 是专为大规模数据处理而设计的快速通用的计算引擎。Spark 是 UC Berkeley AMP lab (加州大学伯克利分校的 AMP 实验室)所开源的类 Hadoop MapReduce 的通用并行框架，Spark，拥有 Hadoop MapReduce 所具有的优点；但不同于 MapReduce 的是——Job 中间输出结果可以保存在内存中，从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。

Spark 采用 RDD 机制，也就是弹性分布式数据集，spark 中的很多特性都和他有关，所以先谈一个 RDD 是一个分布式对象集合，本质上是一个只读的分区记录集合 ，提供了一个抽象的数据架构，不必担心底层数据的分布式特性，只需将具体的应用逻辑表达为一系列转换处理，一个 RDD 的不同分区可以被保存到集群中不同的节点上， 从而可以在集群中的不同节点上进行并行计算，不同 RDD 之间的转换操作形成依赖关系，可以实现数据流水处理，避免中间数据存储。

RDD 提供了一种高度受限的共享内存模型，其是只读的记录分区的集合，不能直接修改 ，类似软件构造中的不可变类，只能基于稳定的物理存储中的数

据集创建 RDD，或者通过在其他 RDD 上执行确定的转换操作（如 map、join 和 group by）而创建得到新的 RDD ，这里，RDD 提供了一组丰富的操作以支持常见的数据运算，分为 “动作”（Action）和“转换”（Transformation）两种类型

因此 RDD 具有很多的优点：惰性调用、管道化、避免同步等待、不需要保存中间结果、每次操作变得简单，计算的中间结果持久化到内存，数据在内存中的多个 RDD 操作之间进行传递，避免了不必要的读写磁盘开销，放的数据可以是 Java 对象，避免了不必要的对象序列化和反序列化。



Apache Spark 的架构设计图

2.4 系统各部分简介

2.4.1 数据预处理模块

读取来自数据集的购物篮数据，然后对其进行预处理其，包含的过程有数据抽样、数据过滤、数据标准化和数据清洗，对缺失值进行补全等。

2.4.2 传统关联规则挖掘算法模块

本文为了便于与大数据处理框架下运行的算法进行比较，将传统频繁项集挖掘算法 Apriori 基于 scala 进行实现，并期望在单机单线程的环境下运行，作为后续实验分析的 baseline 存在。

2.4.3 基于大数据处理框架的关联规则挖掘模块

在第一部分的基础上，将传统频繁项集挖掘 Apriori 算法在 Apache Spark 大数据处理框架下进行实现，并算法中使用的数据结构使用 RDD 进行重写，并期望在单机伪分布式的环境下运行。

2.4.4 基于大数据处理框架的有限扫描关联规则挖掘模块

在第二部分大数据处理框架下的 Apriori 算法的基础上，面向海量数据的频繁项集改进挖掘算法，通过随机取样的方式，使算法其能够在不损失过多效果的同时，尽可能解决大规模数据带来的问题，期望在第二部分相同的环境下运行并比对效果。

第 3 章 工程实现：传统 Apriori 关联规则挖掘算法

3.1 简介

Apriori 算法是第一个关联规则挖掘算法，也是最经典的算法。它利用逐层搜索的迭代方法找出数据库中项集的关系，以形成规则，其过程由连接（类矩阵运算）与剪枝（去掉那些没必要的中间结果）组成。该算法中项集的概念即为项的集合。包含 K 个项的集合为 k 项集。项集出现的频率是包含项集的事务数，称为项集的频率。如果某项集满足最小支持度，则称它为频繁项集，在本次大作业中，传统 Apriori 算法仅作为评估时的 baseline 作为对照。

3.2 算法思想

该算法的基本思想是：首先找出所有的频集，这些项集出现的频繁性至少和预定义的最小支持度一样。然后由频集产生强关联规则，这些规则必须满足最小支持度和最小可信度。然后使用第 1 步找到的频集产生期望的规则，产生只包含集合的项的所有关联规则。

当寻找频繁项集时,我们首先需要考虑的是频繁的定义，在 Apriori 中与频繁有关的是两个属性：一个是项集的支持度，另一个是置信度；支持度被定义为数据集中包含该项集的记录所占的比例；这个度量是针对项集来说的,因此可以定义一个最小支持度,在进行挖掘时只保留满足最小支持度的项集。置信度是针对一条关联规则来定义的，这条规则的可信度被定义为关联规则整体项集的概率除以关联规则左面项集的概率。

3.3 代码实现

scala 实现：

主要算法描述：该算法有多个迭代组成，每个迭代可以分为两步，第一步首

先检索出事务数据库中的所有频繁项集，即支持度不低于用户设定的阈值的项集；第二步再利用频繁项集构造出满足用户最小信任度的规则，按照这个规则进行多次迭代，直到不能再产生新的频繁项集时终止。

```
def runApriori(minSupport: Double = 0.02, minConfidence: Double = 0.35) = {
  var itemCombs = itemSet.map(word => (Set(word), getSupport(Set(word))))
    .filter(wordSupportPair => (wordSupportPair._2 > minSupport))
  var k: Int = 1
  for (itemComb <- itemCombs) {
    this.toRetItems += (itemComb._1 -> itemComb._2)
  }
  var currentLSet: Set[Set[String]] = itemCombs.map(wordSupportPair => wordSupportPair._1).toSet
  k = k + 1
  while (currentLSet.size > 0) {
    val currentCSet: Set[Set[String]] = currentLSet.map(wordSet => currentLSet.map(wordSet1 => wordSet | wordSet1))
      .reduceRight((set1, set2) => set1 | set2)
      .filter(wordSet => (wordSet.size == k))
    val currentItemCombs = currentCSet.map(wordSet => (wordSet, getSupport(wordSet)))
      .filter(wordSupportPair => (wordSupportPair._2 > minSupport))
    currentLSet = currentItemCombs.map(wordSupportPair => wordSupportPair._1).toSet
    var associationRules: List[(Set[String], Set[String], Double)] = List()
    var CurrentToRetItems : Map[Set[String], Double] = Map()
    for (itemComb <- currentItemCombs) {
      CurrentToRetItems += (itemComb._1 -> itemComb._2)
    }
    this.toRetItems = this.toRetItems ++ CurrentToRetItems
    CurrentToRetItems.keys.foreach(item =>
      item.subsets.filter(wordSet => (wordSet.size < item.size & wordSet.size > 0))
        .foreach(subset => {
          associationRules = associationRules :+ (subset, item diff subset,
            this.toRetItems(item).toDouble / this.toRetItems(subset).toDouble)
        }
        )
    )
    this.associationRules=this.associationRules ++ associationRules.filter(rule => rule._3 > minConfidence)
    k += 1  }
```

在 runApriori()方法中：

currentCSet 代表当前的候选集合

currentItemCombs 代表从候选集合中选出的频繁项到该项支持度的映射

associationRules 是根据频繁项生成的关联规则到置信度的映射

如何设置最小支持度与最小置信度并没有统一的标准，大部分情况下都是根据业务经验和现实需求设置初始值，然后经过多次调整，获取与业务相符的关联规则结果。本文中选取模型的预先输入参数为：最小支持度 0.02、最小置信度 0.35，可以通过实际测试以及考虑预期结果来获得几个比较好的参数。

3.4 程序运行输出

在
java version "1.8.0_251"
scala version 2.12.12 的环境下运行
最小支持度 0.02
最小置信度 0.35
我们选择购物篮数据集进行测试，这个数据集数据总条目为 9835 条，我们取前 10 条展示：

```
1. 柑橘类水果 人造黄油 即食汤 半成品面包
2. 咖啡 热带水果 酸奶
3. 全脂牛奶
4. 奶油乳酪 肉泥 仁果类水果 酸奶
5. 炼乳 长面包 其他蔬菜 全脂牛奶
6. 腐蚀性清洁剂 黄油 白饭 全脂牛奶 酸奶
7. 面包卷
8. 瓶装啤酒 开胃酒 其他蔬菜 面包卷 超高温杀菌的牛奶
9. 盆栽
10. 谷物 全脂牛奶
```

算法运行产生的输出结果：

```
1. (Set(本地蛋类),Set(其他蔬菜),0.35096153846153844)
2. (Set(本地蛋类),Set(全脂牛奶),0.47275641025641024)
3. (Set(仁果类水果),Set(全脂牛奶),0.3978494623655914)
4. (Set(黑面包),Set(全脂牛奶),0.3887147335423197)
5. (Set(酸奶油),Set(其他蔬菜),0.40283687943262414)
6. (Set(柑橘类水果),Set(全脂牛奶),0.36855036855036855)
7. (Set(热带水果),Set(全脂牛奶),0.40310077519379844)
8. (Set(黄油),Set(全脂牛奶),0.4972477064220184)
9. (Set(牛肉),Set(全脂牛奶),0.4050387596899225)
10. (Set(猪肉),Set(其他蔬菜),0.37566137566137564)
11. (Set(根茎类蔬菜),Set(全脂牛奶),0.44869402985074625)
12. (Set(冷冻蔬菜),Set(全脂牛奶),0.4249471458773784)
13. (Set(其他蔬菜),Set(全脂牛奶),0.38675775091960063)
14. (Set(黄油),Set(其他蔬菜),0.3614678899082569)
15. (Set(根茎类蔬菜),Set(其他蔬菜),0.43470149253731344)
16. (Set(猪肉),Set(全脂牛奶),0.3844797178130512)
17. (Set(酸奶),Set(全脂牛奶),0.40160349854227406)
18. (Set(人造黄油),Set(全脂牛奶),0.4131944444444444)
19. (Set(凝乳),Set(全脂牛奶),0.4904580152671756)
20. (Set(酸奶油),Set(全脂牛奶),0.449645390070922)
21. (Set(水果/蔬菜汁),Set(全脂牛奶),0.36849507735583686)
22. (Set(糕点),Set(全脂牛奶),0.3737142857142857)
23. (Set(其他蔬菜, 酸奶),Set(全脂牛奶),0.5128805620608898)
24. (Set(全脂牛奶, 酸奶),Set(其他蔬菜),0.39745916515426494)
25. (Set(其他蔬菜, 根茎类蔬菜),Set(全脂牛奶),0.4892703862660944)
26. (Set(全脂牛奶, 根茎类蔬菜),Set(其他蔬菜),0.47401247401247404)
```

从输出中选择一项来说明，例如(Set(其他蔬菜, 根茎类蔬菜),Set(全脂牛奶),0.4892703862660944)，说明顾客购买其他蔬菜，根茎类蔬菜的前提下，再购买全脂牛奶的概率大约有 48.92%
程序运行时间统计：
运行时间统计：

```
1. 数据总条目 9835
2. 加载总用时: 843ms
3. 生成 k=1 层级的频繁项用时: 481ms
```

```
-——---------
产生 k=2 层级的候选项数目: 1711
生成 k=2 的所有可能的候选项集用时: 27ms
生成 k=2 频繁项用时: 9474ms
产生 k=2 层级的频繁项目: 61 条
生成 k=2 层级的关联规则用时: : 52ms
-——---------
产生 k=3 层级的候选项数目: 477
生成 k=3 的所有可能的候选项集用时: 11ms
生成 k=3 频繁项用时: 2703ms
产生 k=3 层级的频繁项目: 2 条
生成 k=3 层级的关联规则用时: : 1ms
-——---------
产生 k=4 层级的候选项数目: 1
生成 k=4 的所有可能的候选项集用时: 0ms
生成 k=4 频繁项用时: 7ms
产生 k=4 层级的频繁项目: 0 条
生成 k=4 层级的关联规则用时: : 0ms
```

可以看出算法运行中大部分时间用于生成频繁项

3.5 缺点和不足

首先，在每一步产生侯选项目集时循环产生的组合过多，没有排除不应该参与组合的元素；其次，在每次计算项集的支持度时，都对数据集中的全部记录进行了一遍扫描比较，这种扫描会大大增加计算系统的 I/O 开销，代价是随着数据集中记录数目的增加访存时间会呈现出大幅度的增加，因此需要设计更好性能的算法或是使用大数据计算框架进行处理。

第 4 章 工程实现：基于大数据处理框架关联规则挖掘算法

4.1 简介

Spark 框架是一种分布式计算框架，相对于 Hadoop 其具有改进的数据流处理的批处理框架，基于内存计算及处理优化，保证了集群计算的效率，同时 Spark 提供了更多灵活可用的数据操作，比如 filter, union, join, 以及各种对 key value pair 的方便操作，甚至提供了一个通用接口，让用户根据需要开发定制的数据操作，因此我们可以在 Spark 平台上实现 Apriori 频繁项集挖掘的并行化算法。

4.2 算法思想

再次分析 Apriori 算法，我们可以看出它产生频繁项集有两个特点：第一，它是逐层的，即从频繁 1-项集逐个产生到频繁 k-项集；第二，每次迭代后都由前一次产生的频繁项来产生新的候选项，然后对新产生的候选项集进行支持度计数进而得到新的频繁项集。

根据以上的特点，我们可以将算法分为两个阶段：第一个阶段，从 HDFS 上获取原始的数据集，将所有的事务转化为 Spark RDD 并利用 cache()缓存到分布式内存中。然后在该 RDD 上扫描所有的事务，进行支持度计数，产生频繁 1-

项集。第二个阶段则从频繁 1-项集开始，使用频繁 k-项集去产生频繁 k+1 项集，不断迭代下去，直到不能产生新的频繁项集。在算法运行时，主节点每次迭代时需要将候选项集以 broadcast()的形式分发到每个从节点，每个从节点收到之后进行一系列的 MapReduce 操作进而得到新的频繁项集，如此反复直至求得最大频繁项集。

4.3 代码实现

scala on spark

首先配置 sparkContent 与 HDFS，使用伪分布式实现算法

```
//配置 sparkContent
1.     val conf = new SparkConf().setAppName("Apriori").setMaster("local")
2.     val sc = new SparkContext(conf)
3.
4.     //配置 HDFS
5.     val hdfsConf = new Configuration()
6.     val hdfs = FileSystem.get(hdfsConf)
7.
8.     //设置输出文件
9.     val path: Path = new Path(output)
10.    if (hdfs.exists(path)) {
11.      hdfs.delete(path, true)
12. }
```

利用 sparkContent.textFile()读入数据，然后使用将其利用 cache()缓存到分布式内存

```
1.  val transactions = sc.textFile(input)
2.        .map(x => {
3.          val elementSet = x.split("\\s+").toSet
4.          elementSet
5.        }).cache()
6.    val transactions_num = transactions.count().toDouble
```

首先通过支持度计数来计算频繁 1 项集，以用于后续的迭代计算，在每一个事务中执行 flatMap 函数，将其中项集的所有元素都取出来铺平，再在该基础上执行 map 函数，将原有的 Item 映射为：<Set(Item), 1/数据数目>的 key/value 形式，接下来执行 reduceByKey 函数，用于统计每一个候选 1-项集的支持度，并利用支持度阈值进行过滤，将支持度大于支持度阈值的项集生成频繁 1-项集，完成第一阶段，下面是第一阶段的算法伪代码：

```
1.  val L_1 = transactions.flatMap(_.seq)
2.        .map(item => (Set(item), 1 / transactions_num))
3.        .reduceByKey((supportA, supportB) => supportA + supportB)
4.        .filter(item => item._2 >= minSupport)
```

在第二阶段，当频繁 k-项集计算结束后，首先对计算得到分布式 RDD 利用 collect()方法进行收集，再将其转化为< itemSet >的集合形式进行存储 ，在生成候选项时，先通过 map 函数将频繁 k-项集中的项集进行组合，并使用 reduceRight()从末尾向前将集合进行合并，使用 items.size == k 的条件对生成的候选项进行过滤，选出频繁 k+1-项集的候选集合，最后利用 broadcast（）广播到每个工作节点

```
1.   var currentLSet = L_1.collect().map(item => item._1).toSet
2.     //当候选项集不为空时，持续迭代
3.     while (!currentLSet.isEmpty) {
4.
5.       //从之前挖掘生成的频繁项中组合生成候选项
6.       var currentCSet: Set[Set[String]] = currentLSet
```

```
          .map(itemsA => currentLSet.map(itemsB => itemsA | itemsB))
          .reduceRight((set1, set2) => set1 | set2)
          .filter(items => (items.size == k))
        // 将候选项集广播到各个分区
        val broadcastCurrentC = sc.broadcast(currentCSet)
```

接下来，通过 flatMap 函数获取每个候选项集在原始事务集中的支持度，进一步对每个候选项使用 map 函数得到<ItemSet, 1/数据数目>的 key/value 形式，之后通过 reduceBykey 函数对每个事务的最终的支持度进行收集计数，并利用支持度阈值进行过滤，将支持度大于支持度阈值的项集生成频繁(k+1)-项集

```
  //生成新的 k-频繁项组合
        val ItemFreqCombs = transactions
          .flatMap(transaction => broadcastCurrentC.value.filter(Candidate =>
    Candidate.subsetOf(transaction)).map(Candidate => (Candidate, 1 / transactio
    ns_num)))
          .reduceByKey((supportA, supportB) => supportA + supportB)
          .filter(items => items._2 >= minSupport)

        //ItemFreqCombs.saveAsTextFile(output + "/" + "freq" + k)
        //利用计算结果收集新的频繁项集合
        currentLSet = ItemFreqCombs.collect().map(item => item._1).toSet

```

如果计算得到的频繁(k+1)-项集非空的话，需要根据迭代生成的频繁项集来计算关联规则，利用频繁(k+1)-项集建立频繁项索引 toRetItems，并利用 broadcast（）广播将其到每个工作节点，对于每个频繁(k+1)-项集，我们都可以将其划分为一个大小为 k 的频繁项集以及一个频繁项，可以通过遍历对应阶数产生的关联规则左项的长度来构造；这里以频繁项集 A 与繁项集 b 为例，首先需要从数据集中计算 A->b 中左项 A 的频度，再从 toRetItems 中索引 Ab 频繁项的频度即可计算规则 A->b 的置信度，再利用置信度阈值进行过滤，将置信度大于支持度阈值的项集生成需要的关联规则。

```
        //产生的频繁项组合数目>0,计算关联规则
        if (currentLSet.size > 0) {

          var toRetItems: Map[Set[String], Double] = Map()
          for (elems <- ItemFreqCombs.collect()) {
            toRetItems += (elems._1 -> elems._2)
          }
          val broadcastToRetItems = sc.broadcast(toRetItems)

          val TimeFlagD = System.nanoTime()
          //生成关联规则
          val associationRules = transactions
            .flatMap(transaction =>
              for {
                set <- broadcastToRetItems.value.keys.toArray
                i <- 1 until k
                subset <- set.subsets(i)
                if (subset.subsetOf(transaction))
                diff = set.diff(subset)
              } yield ((set -- diff, diff), 1 / transactions_num))
            .reduceByKey(_ + _)
            .map(x => ((x._1._1, x._1._2), broadcastToRetItems.value.get(x._1.
    _1.union(x._1._2)).getOrElse(0.0) / x._2))
            .filter(x => x._2 >= minConfidence)
```

4.4 程序运行与输出

在
java version "1.8.0_251"
hadoop-2.7.7
spark-3.0.1-bin-hadoop2.7
scala version 2.12.12 的环境下运行
最小支持度 0.02
最小置信度 0.35，

同样选择小规模的购物篮数据集进行测试，这个数据集数据总条目为 9835 条

k=2 阶关联规则

```
((Set(凝乳),Set(全脂牛奶)),0.4904580152671819)
((Set(糕点),Set(全脂牛奶)),0.3737142857142852)
((Set(猪肉),Set(全脂牛奶)),0.38447971781305657)
((Set(本地蛋类),Set(全脂牛奶)),0.47275641025641746)
((Set(柑橘类水果),Set(全脂牛奶)),0.36855036855036977)
((Set(黄油),Set(其他蔬菜)),0.3614678899082616)
((Set(猪肉),Set(其他蔬菜)),0.3756613756613809)
((Set(热带水果),Set(全脂牛奶)),0.40310077519379306)
((Set(酸奶油),Set(全脂牛奶)),0.4496453900709258)
((Set(水果/蔬菜汁),Set(全脂牛奶)),0.3684950773558401)
((Set(酸奶油),Set(其他蔬菜)),0.40283687943262786)
((Set(冷冻蔬菜),Set(全脂牛奶)),0.4249471458773828)
((Set(黄油),Set(全脂牛奶)),0.4972477064220252)
((Set(根茎类蔬菜),Set(其他蔬菜)),0.434701492537306)
((Set(根茎类蔬菜),Set(全脂牛奶)),0.4486940298507383)
((Set(黑面包),Set(全脂牛奶)),0.3887147335423251)
((Set(其他蔬菜),Set(全脂牛奶)),0.386757750919609)
((Set(酸奶),Set(全脂牛奶)),0.40160349854226896)
((Set(本地蛋类),Set(其他蔬菜)),0.3509615384615436)
((Set(人造黄油),Set(全脂牛奶)),0.41319444444445047)
((Set(仁果类水果),Set(全脂牛奶)),0.39784946236559415)
((Set(牛肉),Set(全脂牛奶)),0.4050387596899274)
```

k=3 阶关联规则

```
((Set(其他蔬菜, 酸奶),Set(全脂牛奶)),0.5128805620608942)
((Set(全脂牛奶, 酸奶),Set(其他蔬菜)),0.3974591651542704)
((Set(其他蔬菜, 根茎类蔬菜),Set(全脂牛奶)),0.4892703862660995)
((Set(全脂牛奶, 根茎类蔬菜),Set(其他蔬菜)),0.47401247401247926)
```

可以看出运行结果和单机算法是近乎相同的

程序运行时间统计

```
spark 启动总用时: 7065ms
数据总条目 9835
加载总用时: 1592ms
生成 k=1 层级的频繁项用时: 866ms
------------
产生 k=2 层级的候选项数目: 1711
生成 k=2 层级的所有可能的候选项集用时: 39ms
生成 k=2 频繁项用时: 1917ms
产生 k=2 层级的频繁项目: 61 条
生成 k=2 层级的关联规则用时: : 12ms
生成的关联规则数目:  22
------------
```

```
产生 k=3 层级的候选项数目: 477
生成 k=3 层级的所有可能的候选项集用时: 15ms
生成 k=3 频繁项用时: 725ms
产生 k=3 层级的频繁项目: 2 条
生成 k=3 层级的关联规则用时: : 8
ms
生成的关联规则数目:  4
------------
产生 k=4 层级的候选项数目: 1
生成 k=4 层级的所有可能的候选项集用时: 2ms
生成 k=4 频繁项用时: 319ms
产生 k=4 层级的频繁项目: 0 条

进程已结束,退出代码 0

```

可以看出相较于单机版的算法，在相同的机器配置与运行环境下，基于大数据处理框架（即时是伪分布式）的挖掘算法在运行时间方面显著优于前者。

4.5 缺点和不足

Apriori 算法本身对数据库的扫描次数过多，而且扫描后可能产生大量的候选项集，在频繁项目集长度变大的情况下,运算时间显著增加，这个问题并不能仅凭大数据处理框架解决，如果数据量过大超过了主存的大小，频繁的换入换出造成的时间浪费同样也是不可忽视的。同时有许多应用并不需要发现所有的频繁项：比方说在某些情况下，我们只要找到大部分的销售频繁关联项就够了，而不必找出所有的频繁项。

第 5 章 工程实现：大数据框架下有限扫描关联挖掘算法

5.1 简介

基于大数据处理框架的有限扫描挖掘算法通过选择原始数据的一个样本，在这个样本上再使用 Apriori 算法进行频繁的挖掘，以牺牲精确度的代价来减少算法开销，为了提高效率，样本大小应该以可以放在内存中为宜，可以适当降低最小支持度来减少遗漏的频繁模式，尽可能发现全部或大部分的频繁项集。即先使用从全体数据集中抽取出来的采样得到一些在整个数据库中可能成立的规则，然后对数据库的剩余部分验证这个结果。

5.2 算法思想

从给定数据集上选取随机样本 S，然后在 S 而不是整个数据集中执行 Apriori 算法搜索频繁项集。用这种方法是牺牲了一些精度换取有效性。样本 S 的大小选取使得频繁项目集产生的数目大大减少。这样算法不需要扫描整个数据集中的事务。由于算法只是搜索随机样本中的数据，因此可能会丢失一些全局频繁项集。为了减少这样的情况，使用比最小支持度更低的支持度阈值来找出局部于 S 的频

繁项集(记做 pS)。然后，原数据集的其余部分用于计算 pS 中每个项集的实际频率。使用一种机制来确定是否所有的频繁项集都包含在 pS 中。如果 pS 实际包含了整个数据集中的所有频繁项集，则只需扫描一次数据集。否则，需要进行重新采样，继续重复上面的步骤，直到出现满足 pS 实际包含了整个数据集中的所有频繁项集。

我们定义一个概念：Negative border（非频繁项边界，或者说是反例边界）：是采样集中的一个非频繁项的集合，当且仅当这些项集中去掉任意一个项后得到的所有直接子集是在采样集中都是频繁集。

使用 Negative border 的目的是充当一个类似于金丝雀的边界探测。Negative border 中的项在整个数据集中没有一个是频繁的，因为我们在采样样本中选择了明显低于比例阈值的支持度阈值，如果项集在采样样本中不是频繁项集，那么其在全体数据集上也就不可能是频繁项集。

但是，如果一个或多个 Negative border 中的项集在全体数据集中被证明是频繁的，那么我们就认为该样本相对于全体数据集不具有代表性，我们需要进行重新采样。

因此，我们可以得到有限抽样算法的流程：

首先在样本上计算频繁项和 Negative border 项，得到的 Negative border 项我们再将其在全体数据集上验证其是否也是非频繁项。

1、 如果 Negative border 中所有项集在整个数据集上计算为都为频繁项集。这种情况下，样本中的频繁项集可以近似正确的频繁项集。

2、 如果存在 Negative border 中的项集在整个数据集中是频繁项集。此时，我们不能确定是否存在更大的项集，这个项集既不在样本的 Negative border 中，又不在样本的的频繁项集中，但是在整个数据集是频繁项集。这样，我们在此次的抽样中得不到结果，算法只能在重新抽样，继续重复上面的步骤，直到出现 Negative border 中没有一个项集在整个数据集上计算为频繁项集的情况停止。

其余在局部样本上从 k 阶频繁项集生成 k+1 阶候选项集，k+1 阶候选项集计算频繁项集，生成关联规则均与第四章中的算法思想相同。

然后我们对算法产生的结果的有效性进行分析

有限扫描挖掘算法不会产生 False Negative，就是在也是就是在全体数据集上为频繁项的数据在抽样后表现为非频繁项，下面我们给出证明：

首先，在每轮的样本采样和频繁项计算中，会出现三种结果：

1. 项集在样本中是频繁项
2. 项集在样本中不是频繁项，而在 Negative border 中
3. 项集在样本中是不是频繁项，也不在 Negative border 中。

如果在全体数据集上为频繁项的数据在抽样后表现为非频繁项，那么它就有上面提到的 2，3 两者可能，从算法的流程可以看到:我们保证所有 Negative border 中的集合在整个数据集都是不频繁的，那么不可能有某个项集满足:它在整个数据集上是频繁的，而且现在样本采样的 Negative border 中，因此 2 的情况可以排除。接下来再看 3 的情况，如果项集在样本中是不是频繁项，也不在 Negative border 中，而在全体数据集上是频繁项，我们可以通过证明否定这种结果：

假设集合 S 在全体数据集上是频繁项集，但是不是样本的频繁项集，并且不在 Negative border 中，根据 Apriori 算法的性质：1.是频繁项集的所有非空子集都必须也是频繁的，2.是非频繁项集的所有父集都是非频繁的。S 在样本上是非频繁集，假设 T 是 S 所有子集中在样本数据上非频繁项的最小子集，那么 T

一定在 Negative border 中，首先 T 在样本数据集上是非频繁的，其次 T 的所有子集在样本数据集上是频繁的，否则就与 T 是 S 所有子集中在样本数据上非频繁项的**最小**子集的假设相矛盾；又因为 S 在全体数据集上是频繁项集，所以 T 作为 S 的子集也是频繁项集（基于 Apriori 算法的性质 1）

我们可以发现，T 即在 Negative border，又在全体数据集上是频繁集，所以随机取样算法不能产生结果，需要重新运行采样；因此可以证明：项集在样本中是不是频繁项，也不在 Negative border 中的情况不会出现。

综上述可知：项集要么在样本数据集中是频繁项 ，要么项集在样本中不是频繁项，而在 Negative border 中，此时在全体数据集中一定不会是频繁项，所以说有限扫描挖掘算法会不会产生 False Negative。

5.3 代码实现

配置 spark 和 HDFS 等步骤均与第四章中实现相同。

利用 sparkContent.textFile()读入数据，然后使用将其利用 cache()缓存到分布式内存也与与第四章中实现相同。

为了实现算法思想中的重复采样的需求，利用一个 while 循环实现，并维护一个布尔型遍历 SampleValidity ，表示采样是否有效，当运行全部的频繁项计算且满足有限扫码的采样条件时，将其置为真，算法结束。

利用 spark 中的 RDD. sample()方法进行采样，置换采样设置为 false，fractionOfTransactionUsed 为预先设置的采样比例，设置采样随机种子为 scala.util.Random.nextInt(1000)

```
1.      var SampleValidity : Boolean = false
2.      val loop = new Breaks;
3.      while(SampleValidity!=true){
4.        if (hdfs.exists(path)) {
5.          hdfs.delete(path, true)
6.        }
7.        loop.breakable {
8.         val sample = transactions.sample(false, fractionOfTransactionUsed, sca
la.util.Random.nextInt(1000))
```

生成 1-频繁项和第四章中相同，除了要生成频繁项之外，还需要计算 Negative border 项集 ，再和全数据集上的频繁项进行对照。

```
1.  //抽样下的 1-非频繁项,又称为 NegativeBorder
2.          val NegativeBorderItem = FrequentCountMap.filter(item => item._2 < m
    inSupport).map(item=>item._1)
3.          //计算全数据集的 1-非频繁项
4.          val NegativeItems = transactions.flatMap(_.seq)
5.            .map(item => (Set(item), 1 / transactions_num))
6.            .reduceByKey((supportA, supportB) => supportA + supportB)
7.            .filter(item => item._2 < minSupport).map(item=>item._1)
```

和全数据集上的频繁项进行对照采用先取全数据集上的非频繁项集，再与 Negative border 项集取差集，如果得到差集中有元素的话，说明存在 Negative border 中的项集在全数据集上是频繁项，这时候就需要重新运行采样算法，重新采样了。

```
1.   if( NegativeItems.subtract(NegativeBorderItem).count()>0) {
2.
3.            println("重新运行采样算法 k="+k+"\n"+"-"*12+"\n"+"-"*12)
4.            //NegativeBorderItem.subtract(NegativeItems).foreach(print)
5.            //println("-----")
6.            NegativeItems.subtract(NegativeBorderItem).foreach(print)
```

```
7.
8.             loop.break()
9.               //重新采样运行算法
10.          }
```

除此之外，因为我们选择了一个抽样样本，所有的采样与频繁项计算都需要在这个样本之上，而不是在全数据集 transactions 上

对于由 k 项集生成（k+1）项集时，生成时不仅要计算频繁项集 randItemFreqCombs，还需要计算 Negative border 项集 randNegativeBorderItems，计算这个项集的思路是先将 sampleCountMap 中不满足 items._2 （支持度）>= minSupport 的项集提取出来，再用 filter 方法进行筛选，选出项集中去掉任意一个项后得到的所有直接子集是在采样集中都是频繁集的项集加入 Negative border 中。

```
1.   sampleCountMap = sample
2.           .flatMap(transaction => broadcastCurrentC.value.filter(Candidate =>
     Candidate.subsetOf(transaction)).map(Candidate => (Candidate, 1 / sample_nu
    m)))
3.           .reduceByKey((supportA, supportB) => supportA + supportB)
4.         val randItemFreqCombs = sampleCountMap
5.           .filter(items => items._2 >= minSupport)
6.         randItemFreqCombs.saveAsTextFile(output + "/" + "randFreq" + k)
7.         //利用计算结果收集新的频繁项集合
8.         currentLSet = randItemFreqCombs.collect().map(item => item._1).toSet

9.
10.       //计算 Negative border：是样品的一个非频繁项集合，并且这些项集去掉任意一个项
    后的直接子集是频繁集。
11.         val randNegativeBorderItems = sampleCountMap.filter(items => items._2
     < minSupport).
12.           map(x=>x._1).filter(items=>items.subsets.filter(item=>(item.size==k
    -1))
13.           .forall(x=>currentLSet.contains(x)))
14.         //将 randNegativeBorderItems 广播到所有工作节点
15.         val broadcastRNI = sc.broadcast(randNegativeBorderItems.collect())
16.         //验证 negative border 在全部数据集上也是非频繁项集
17.         val existFalsePositive = transactions
18.         .flatMap(transaction=>broadcastRNI.value.filter(rNI=>rNI.subsetOf(tra
    nsaction)).map(rNI=>(rNI,1 / transactions_num)))
19.           .reduceByKey((supportA, supportB) => supportA + supportB)
20.           .collect().forall(items => items._2 < minSupport)
21.         if(existFalsePositive == false){
22.           println("重新运行采样算法 k="+k+"\n"+"-"*12+"\n"+"-"*12)
23.           loop.break()
24.           //重新采样运行算法
25. }
```

计算关联规则的方法和第四章中相同，只不过将全数据集集合替换为样本集即可，这里不再赘述。

5.4 程序运行与输出

同样选择小规模的购物篮数据集进行测试，这个数据集数据总条目为 9835 条

在第四章相同的环境下运行：

最小支持度 0.02

最小置信度 0.35，

抽样大小 0.5

样本最小支持度：0.9*最小支持度

k=2 阶关联规则

```
((Set(凝乳),Set(全脂牛奶)),0.4922480620155052)
((Set(糕点),Set(全脂牛奶)),0.37354988399071676)
((Set(猪肉),Set(全脂牛奶)),0.373188405797103)
((Set(香肠),Set(面包卷)),0.3675213675213643)
((Set(本地蛋类),Set(全脂牛奶)),0.4478527607361969)
((Set(柑橘类水果),Set(全脂牛奶)),0.36231884057970815)
((Set(猪肉),Set(其他蔬菜)),0.36594202898550876)
((Set(热带水果),Set(全脂牛奶)),0.39525691699604304)
((Set(酸奶油),Set(全脂牛奶)),0.428954423592492)
((Set(酸奶油),Set(其他蔬菜)),0.4182305630026797)
((Set(餐巾),Set(全脂牛奶)),0.39926739926740074)
((Set(冷冻蔬菜),Set(全脂牛奶)),0.422764227642278)
((Set(黄油),Set(全脂牛奶)),0.4726562500000014)
((Set(根茎类蔬菜),Set(其他蔬菜)),0.42418426103646323)
((Set(根茎类蔬菜),Set(全脂牛奶)),0.443378119001914)
((Set(黑面包),Set(全脂牛奶)),0.39755351681957257)
((Set(其他蔬菜),Set(全脂牛奶)),0.3640167364016669)
((Set(酸奶),Set(全脂牛奶)),0.39410939691443897)
((Set(人造黄油),Set(全脂牛奶)),0.4137931034482772)
((Set(仁果类水果),Set(全脂牛奶)),0.3977272727272724)
((Set(牛肉),Set(全脂牛奶)),0.4078431372549035)
```

k=3 阶关联规则

```
((Set(酸奶, 其他蔬菜),Set(全脂牛奶)),0.509174311926607)
((Set(全脂牛奶, 酸奶),Set(其他蔬菜)),0.3950177935943075)
((Set(其他蔬菜, 根茎类蔬菜),Set(全脂牛奶)),0.4615384615384632)
((Set(根茎类蔬菜, 全脂牛奶),Set(其他蔬菜)),0.44155844155844315)
```

对关联规则进行比较，k=3 阶关联规则与精确算法生成的关联规则是完全相同的，而 k=2 阶的关联规则与精确算法的存在一些误差，准确率大概在 85%左右。

运行时间：

```
spark 启动总用时: 7422ms
数据总条目 9835
加载总用时: 1909ms
抽样总条目 4926
生成 k=1 层级的频繁项用时: 401ms
重新运行采样算法 k=1
------------
------------
Set(防腐用品)抽样总条目 5020
生成 k=1 层级的频繁项用时: 93ms
------------
产生 k=2 层级的候选项数目: 595
生成 k=2 层级的所有可能的候选项集用时: 31ms
生成 k=2 频繁项用时: 588ms
产生 k=2 层级的频繁项目: 59 条
生成 k=2 层级的关联规则用时: : 8ms
生成的关联规则数目:  21
------------
产生 k=3 层级的候选项数目: 447
生成 k=3 层级的所有可能的候选项集用时: 13ms
生成 k=3 频繁项用时: 606ms
产生 k=3 层级的频繁项目: 2 条
生成 k=3 层级的关联规则用时: : 6ms
生成的关联规则数目:  4
------------
```

```
产生 k=4 层级的候选项数目: 1
生成 k=4 层级的所有可能的候选项集用时: 13ms
生成 k=4 频繁项用时: 323ms
产生 k=4 层级的频繁项目: 0 条

进程已结束,退出代码 0

```

可以看出和全数据集上的频繁项挖掘算法对比，生成候选项和频繁条目的用时显著的减小了，这是因为我们在抽样集上进行挖掘，生成候选项的数目大大地减少了，同时对数据集的遍历次数也减少了。

5.5 缺点和不足

随机采样算法存在一个很大的缺点就是产生的结果不精确，即存在所谓的数据扭曲，因为在一些实际情况下，分布在同一页面上的数据时常是高度相关的，对这些数据的采样可能不能表示整个数据集中模式与潜在规则的分布，由此而导致的是多次重复采样数据所花费的代价有可能是较大的。

第 6 章 实验评估与分析

6.1 实验设计

6.1.1 实验环境

本次实验的软件环境使用 java version "1.8.0_251", scala version 2.12.12,hadoop-2.7.7,spark-3.0.1-bin-hadoop2.7,运行在 Microsoft Windows 10 专业版，10.0.19042 Build 19042，在 IntelliJ IDEA 2020.3.1 (Ultimate Edition)下运行，其中 spark 运行在 StandAlone 模式

硬件环境安装有 16,258 MB 物理内存，使用 Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (8 CPUs), ~2.3GHz 处理器

6.1.2 数据集介绍

本次实验共采用三个规模不同数据集：

1. data_act_combat5529_Retail，

示例：

```
柑橘类水果 人造黄油 即食汤 半成品面包
咖啡 热带水果 酸奶
全脂牛奶
奶油乳酪 肉泥 仁果类水果 酸奶
炼乳 长面包 其他蔬菜 全脂牛奶
腐蚀性清洁剂 黄油 白饭 全脂牛奶 酸奶
面包卷
瓶装啤酒 开胃酒 其他蔬菜 面包卷 超高温杀菌的牛奶
盆栽
```

```
20. 谷物 全脂牛奶
```

2. Retail_Market_Basket_Data_Set 数据集由 Tom Brijs 提供，包含来自匿名比利时零售商店的零售市场篮子数据，共有 88163 条购物篮数据，所有购物篮共计 1076195 件商品，购物篮的平均大小为 13，且大部分购物篮的大小都在 7-11 的范围之间

示例：

```
30 31 32
33 34 35
36 37 38 39 40 41 42 43 44 45 46
38 39 47 48
38 39 48 49 50 51 52 53 54 55 56 57 58
32 41 59 60 61 62
3 39 48
63 64 65 66 67 68
32 69
48 70 71 72
39 73 74 75 76 77 78 79
36 38 39 41 48 79 80 81
82 83 84
41 85 86 87 88
39 48 89 90 91 92 93 94 95 96 97 98 99 100 101
```

3. IBM_Almaden_Quest T40I10D100K 数据集，是使用 IBM Almaden Quest research group 的生成器生成的数据，共有 100000 条，数据集中涉及 1000 种不同的挖掘项类型

示例：

```
36 69 115 226 278 343 345 358 368 370 401 450 489 494 573 577 581 583 610 682 692 705 722 832 862 886 908 923 932 960 977
8 51 55 73 78 117 140 175 187 229 266 295 304 366 381 413 424 429 501 512 523 529 538 572 575 576 593 675 676 688 735 758 785
53 55 98 159 192 322 332 402 412 413 424 430 450 480 526 538 569 571 572 598 666 672 694 701 797 809 820 826 897 904 928 943 9
48 203 205 227 279 294 320 335 359 414 509 525 529 780 814 850 871 885 922 928 950 984
58 69 73 82 120 146 166 178 217 220 244 256 334 357 378 462 470 487 498 528 538 601 625 658 720 749 763 790 795 800 820 868 88
10 18 75 120 185 194 205 206 321 339 397 477 494 509 544 593 598 618 623 680 720 809 840 876 893 900 913 937 949 956 964 989
38 175 243 329 371 378 385 440 450 541 554 581 597 622 658 663 722 777 788 800 830 843 849 883 887 888 913 947 956 966 981 989
20 31 35 38 71 73 98 102 120 127 132 140 204 217 242 243 257 265 294 340 362 368 387 401 419 470 489 509 514 530 533 548 573 6
1 4 49 72 107 108 115 124 142 146 162 204 215 225 230 242 266 279 296 314 315 316 318 331 347 509 541 576 605 612 624 688 692
21 38 59 71 72 104 122 137 180 196 204 207 256 266 279 334 338 339 421 438 480 523 541 577 617 634 684 741 765 771 830 871 885
28 54 55 70 82 115 204 205 225 245 296 333 350 356 366 388 401 411 460 472 480 525 529 598 620 641 657 675 742 753 758 781 800
```

6.1.3 实验内容

分布在多个不同规模的数据集上，运行大数据处理框架的挖掘算法和有限扫描挖掘算法，对大数据计算框架下的随机与精确算法的性能与正确率进行对照与分析。

统计算法性能方式：对程序运行进行计时

统计正确率的方式：对生成的关联规则与精确算法的进行比较

6.2 对比实验

本次实验选择的对比实验是运行传统 Apriori 算法，记录传统频繁项集挖掘算法面对大规模数据时的表现和限制。

6.3 实验结果

6.3.1 数据集 1 上实验

首先在数据集 1 上进行实验
最小支持度 0.02
最小置信度 0.35，
抽样大小 0.5
样本最小支持度：0.9*最小支持度
运行结果：

单位：ms	传统挖掘算法	spark+精确算法	spark+随机抽样
k=1频繁项	438	930	44
k=2候选项集	29	36	41
k=2频繁项	9297	1721	534
k=2关联规则	51	18	10
k=3候选项集	7	19	14
k=3频繁项	2691	721	565
k=3关联规则	2	9	6
k=4候选项集	1	2	2
k=4频繁项	7	407	314
k=4关联规则	1	1	1
总时间	12524	3864	1531
准确率	100%	100%	85%

数据集 1 的规模较小，尚在常规算法接受的范围内
可以看出大数据计算框架在计算频繁项时能够节省较多的时间，而频繁项计算占用了算法运行大部分时间，相比于精确算法，随机抽样能够节省更多时间，这与抽样的比例有密切的关系。

6.3.2 数据集 2 上实验

最小支持度 0.02
最小置信度 0.35，
抽样大小 0.5
样本最小支持度：0.9*最小支持度
运行结果：

单位：ms	传统挖掘算法	spark+精确算法	spark+随机抽样
k=1频繁项	229409	1602	433
k=2候选项集	13	33	39
k=2频繁项	9384	1418	542
k=2关联规则	39	12	8
k=3候选项集	4	13	14
k=3频繁项	4327	935	493
k=3关联规则	7	7	5
k=4候选项集	1	3	2
k=4频繁项	1361	477	568
k=4关联规则	1	5	1
k=5候选项集	4	2	2
k=5频繁项	1	341	314
k=5关联规则	1	1	1
总时间	244546	4505	2105
准确率	100%	100%	87%

随着数据集中数据的增多，传统挖掘算法需要的时间会显著增加，几乎是基于 spark 上算法的数倍。

从前两个实验中我们可以看出，算法花费的主要时间是在频繁项搜索上，因此之后我们只统计频繁项搜索花费的时间来确定算法的时间代价

6.3.3 数据集 3 上实验

最小支持度 0.02
最小置信度 0.35，
抽样大小 0.5
样本最小支持度：0.9*最小支持度

由于数据集 3 数据量极大，每个篮中包含的项目较多，因此传统挖掘算法几乎无法完成挖掘任务，我们从 100k（100 000）的数据集中选择 10k 条进行实验

单位：ms	传统挖掘算法	spark+精确算法	spark+随机抽样
k=1频繁项	2958	1387	645
k=2频繁项	1055077	131700	21214
k=3频繁项	546350	4593	50766
k=4频繁项	60	338	390
总时间	1604445	138018	73015
准确率	100%	100%	81%

可以发现基于大数据处理框架的算法可以大幅度地较少处理时间（结果约 90%），再进行随机采样可以进一步减少时间

然后我们扩大数据集选取 50%的数据

很可惜，选取 50%的数据，传统挖掘算法显然不能在有效的时间内完成挖掘任务，因此我们跳过对其的测试

单位：ms	传统挖掘算法	spark+精确算法	spark+随机抽样
k=1频繁项		2307	722
k=2频繁项		442524	98529
k=3频繁项		550934	232661
k=4频繁项		357	319
总时间		996122	332231
准确率		100%	70%

可以看出，因为数据量的增多，即使是使用了 spark 计算框架，也需要消耗大量的时间，而随机抽样的方法可以在一定程度上减少时间的消耗。

6.4 实验受参数的影响

6.4.1 支持度

由于在三种算法中，和支持度有关的操作（判断是否为频繁项）都是类似的，仅对传统挖掘算法的实验也具有代表性，因此该实验在传统挖掘算法上执行，运行数据集.

最小支持度 0.01 0.02 0.05 0.07 0.1 0.2

最小置信度 0.35

单位：ms	0.01	0.02	0.05	0.07	0.1	0.2
k=1频繁项计算时间	430	438	391	453	433	436
k=2频繁项计算时间	20321	9297	2408	1032	200	7
k=3频繁项计算时间	16087	2691	25	4	0	0
k=4频繁项计算时间	999	7	0	0	0	0
产生关联规则数目	91	26	2	1	0	0

实验的结果是可以理解的，较大的支持度下会产生更小的频繁项集，进而导致在下一轮迭代计算中产生更小的候选集，更小的候选集在较大的支持度的作用下就会又产生更小的频繁项集，更小的候选集和频繁项集都会减少算法运行消耗的时间。

较大的支持度下会产生更小的频繁项集会产生更少的关联规则，这也是很容易理解的。

6.4.2 置信度

因为置信度涉及关联规则的计算，这次选择 spark 下的频繁项挖掘算法 ，在能产生更多关联项集的数据集 2 上运行

最小支持度 0.02

最小置信度 0.1 0.2 0.35 0.5 0.7

实验结果：

	置信度					
单位：ms/条	0.1	0.2	0.35	0.5	0.7	
k=1频繁项计算时间	1791	1765	1949	1448	1860	
k=2频繁项计算时间	1774	1385	1431	1521	1415	
k=2关联规则计算时间	13	11	14	9	13	
k=2层关联规则数目	35	27	20	20	8	
k=3频繁项计算时间	1069	1227	970	938	1352	
k=3关联规则计算时间	5	6	7	7	13	
k=3层关联规则数目	59	38	26	22	9	
k=4频繁项计算时间	531	480	535	705	494	
k=4关联规则计算时间	11	6	8	6	6	
k=4层关联规则数目	11	7	3	3	1	
k=5频繁项计算时间	409	340	345	685	387	
k=5关联规则计算时间	0	0	0	0	0	
k=5层关联规则数目	0	0	0	0	0	

从结果可以看出，置信度这个参数和频繁项计算时间与关联规则计算时间没有太多的关联，仅仅对产生关联规则数目有影响，这是因为最小置信度仅在生成关联规则时用到，不涉及其他的计算。

6.4.3 取样比例

选择 spark 下的频繁项挖掘算法与随机抽样挖掘抽样算法进行比较

最小支持度 0.02

最小置信度 0.35

抽样比例：0.1 0.2 0.3 0.4 0.5 0.8 1

显然抽样比例为 1 即退化为全数据集上的频繁项挖掘算法

选择从数据集 3 ：100k（100 000）的数据集中选择 10k 条进行实验，因为这个数据集规模较大，更容易体现出随机抽样算法的效果。

	取样比例						
单位：ms/条	0.1	0.2	0.3	0.4	0.5	0.8	1
k=1频繁项计算时间	688	428	668	666	866	683	889
k=2频繁项计算时间	395	759	4244	17754	22016	59130	92937
k=3频繁项计算时间	640	8069	31924	36990	43885	51865	104572
k=4频繁项计算时间	414	284	269	402	302	310	324
	2137	9540	37105	55812	67069	111988	198722
正确率	70%	75%	79%	81%	85%	89%	100%

从结果中可以看出，不同的抽样大小会导致在抽样上花费的时间也不一样，对于一个较小的抽样，在小样本集上遍历生成频繁项所需要的代价就会更小，所需要的运算时间也就越小，进而性能表现也会更好。

但是图表中并没有对小样本导致的样本相对于全体数据集不具有代表性（例如存在 Negative border 中的项集在整个数据集中是频繁项集）的情况进行体现，只对成功计算采样的情况进行了性能分析，在这种情况下我们需要进行重新采样，小样本导致的多次重复采样数据所花费的代价可能会很大。