

独立概率完全加权关联规则的并行挖掘算法

李成严¹, 李鑫宇¹, 张磊¹, 王广泽²

(1. 哈尔滨理工大学 计算机科学与技术学院, 哈尔滨 150080; 2. 哈尔滨理工大学 图书馆, 哈尔滨 150080)

摘要: 关联规则挖掘主要用于发现隐藏在数据中的知识。加权关联规则挖掘能更有效地挖掘出项目重要性不同的规则。针对人工赋权的方法存在一定的主观随意性, 没有充分利用数据本身特征且串行算法无法处理大数据集的问题。提出了独立概率完全加权关联规则的并行挖掘算法, 该算法以项在当前数据集中出现概率为依据进行完全加权模型构建, 以挖掘出更多用户所期待的关联规则。采用前缀划分、位图存储等技术分别解决加权频繁项集筛选、候选加权频繁项集生成所造成时间代价高的问题。引入分布式并行计算思想, 并在 Spark 框架下编程实现, 使算法可以在大数据环境下对加权关联规则进行高效挖掘。利用数值实例对该模型和算法进行了验证, 结果表明此算法可在保证算法时间效率优越的同时获得更多隐藏信息。

关键词: 关联规则挖掘; 完全加权; 独立概率; 并行计算

DOI: 10.15938/j.jhust.2023.06.013

中图分类号: TP399 **文献标志码:** A **文章编号:** 1007-2683(2023)06-0111-10

Parallel Independent Probability Fully Weighted Association Rule Mining Algorithm

LI Chengyan¹, LI Xinyu¹, ZHANG Lei¹, WANG Guangze²

(1. School of Computer Science and Technology, Harbin University of Science and Technology, Harbin 150080, China;
2. Harbin University of Science and Technology Library, Harbin University of Science and Technology, Harbin, 150080, China)

Abstract: Association rule mining is mainly used to discover knowledge hidden in the data. Weighted association rule mining can mine rules with different importance of the project more effectively. There is a certain subjective arbitrariness in the method of artificial weight assignment, existing the problem of not fully utilizing the characteristics of the data itself and the serial algorithm cannot deal with large data sets. A parallel mining algorithm for independent probability fully weighted association rules is proposed. The algorithm constructs a fully weighted model based on the probability of the item appearing in the current data set. In order to mine the association rules expected by more users. Technologies like prefix partition and bitmap storage are used to solve the problem of high time cost caused by weighted frequent item-sets filtering and candidate weighted frequent item-sets generation respectively. The idea of distributed parallel computing is introduced and implemented in Spark framework, which enables the algorithm to efficiently mine weighted association rules in big data environment. Numerical examples are used to verify the proposed model and algorithm. The results show that the proposed algorithm can obtain more effective hidden information and has higher time efficiency.

Keywords: association rule mining; fully weighted; independent probability; parallel computing

收稿日期: 2022-03-14

基金项目: 黑龙江省自然科学基金(LH2021F032).

作者简介: 李成严(1972—), 男, 教授, 硕士研究生导师;
张磊(1998—), 男, 硕士研究生.

通信作者: 李鑫宇(1998—), 男, 硕士研究生, E-mail: 602898111@qq.com.

0 引言

关联规则挖掘技术是数据挖掘研究中一个重要的方向^[1],主要用来发现隐藏在数据集中各项集之间的关联关系。该技术广泛的应用于教育行业、金融及零售业市场营销、互联网等领域。

目前在该领域已经有许多研究成果,较为经典的关联规则挖掘算法有 Apriori 算法^[2], FP-Growth 算法^[3]及 Eclat 算法^[4]。当经典的关联规则算法对大数据集进行挖掘时将会受到单机计算资源的限制,原因在于:传统的串行算法无法将大数据集一次性的放入到内存中,随着数据集规模的增加,挖掘频繁项集的效率会大大降低。针对此问题,有学者提出将经典关联规则算法移植到大数据分布式计算框架下运行,例如 MapReduce 框架。如一种按照等长切割数据集后在每一块数据集上使用 MapReduce 的 Apriori 算法或者 FP-Growth 算法的组合方法,提升了算法运行效率^[5]。一种基于快速且可扩展的频繁模式挖掘算法并在项集生成过程中使用位图结构存储项集,提升算法运算效率^[6]。以上算法虽然有效解决了大数据环境下的关联规则挖掘问题,但需要反复执行 Map 操作和 Reduce 操作且需要将中间计算结果保存到 HDFS(hadoop distribute file system)上,大幅度提升了时间代价。Spark 框架^[7]可以解决该问题,因为 Spark 框架是一个基于内存的并行计算框架,大幅降低集群的 I/O 负载。如一种基于增量并行的 Apriori 算法,通过 Spark 框架实现在进行数据集扩充时根据对之前的频繁项集进行更新,而不是重新计算整个数据集^[8]。如一种结合弹性分布式数据集的概念,通过利用数据集的垂直布局来减少候选项集的数量,在 Spark 框架下的迭代挖掘效率远高于 MapReduce 框架^[9]。

在以上提到的关联规则挖掘算法中,在确定项集支持度时只考虑项的出现频率,没有考虑项的重要性。如在真实事务数据库中项目间的重要程度如果具有一定的差异性,经典关联规则算法无论将支持度阈值设置的偏低或偏高都很难得到更有价值的关联规则或导致挖掘出的知识不可用。针对此问题,加权关联规则挖掘被广泛的探讨和研究,其思想是根据项目的实际意义由用户或领域专家为项目赋予合适的权重并参与到频繁项集的生成中。由于这些权重是独立于事务数据库之外的且在整个挖掘过程中是保持不变的,称为不完全的加权关联规则挖

掘算法。Cai 等^[10]提出了一种典型的加权关联规则挖掘算法,并在该算法中首次提出项的权重概念。Bay 等^[11]提出了一种基于加权项集树的加权频繁项集的挖掘算法,虽然提供了一种快速计算加权支持度的计算方法,但其所构建的树结构会浪费大量内存。

以上提到的不完全加权关联规则挖掘算法存在以下缺陷:项目的权值是固定不变且独立于事务集之外的,从而影响算法的稳定性和高效性。针对此缺陷,一种向量空间模型中完全加权关联规则的挖掘算法被提出,这也是完全加权的概念被首次提出,该算法在文本数据挖掘、信息检索领域中取得一定的研究的成果,但仅在特定领域内实现完全加权的权重赋予方式^[12]。如何实现一种依赖于事务集且能够适用于大多数实验数据集与真实数据集的加权方式,成为了加权关联规则挖掘领域内所重点研究的问题。

本文的主要工作如下:①充分考虑项在事务集中的权重,提出了一种独立概率完全加权的加权方式,使加权结果可以适用于当前事务集以挖掘出更深层次的关联规则。②将前缀划分和位图存储等方法应用于本文算法中以提高算法的运行效率,并结合 Spark 框架,实现在大数据集下的快速并行挖掘。③将独立概率完全加权思想应用于 Spark 框架下的 Eclat 算法中,提出了一种并行的独立概率完全加权关联规则挖掘算法—W-SpEclat 算法,实现对数据集关联规则进行高效并可可靠的挖掘。④通过数值实验,将本算法与 MapReduce 框架下的加权关联规则挖掘算法进行对比,验证本文算法的运行高效性;与 Spark 框架下的非加权关联规则挖掘算法进行对比,验证本文算法可挖掘出更多的关联规则。

1 相关概念

1.1 加权关联规则

设 D 为事务数据库, $T = \{T_1, T_2, \dots, T_n\}$ 为 D 中的事务的总数, $I = \{i_1, i_2, \dots, i_n\}$ 为 D 中全部项集, 权重集 $W = \{w_1, w_2, \dots, w_n\}$ 为与 I 中项目对应的权重, 权重 w_j 可表示项目 i_j 的重要性。最小支持度设为 minsup 。

设 $X \subseteq I, Y \subseteq I$, 并且 $X \cap Y \neq \emptyset$ 。合集 X, Y 中的项目均参照项目合集中的整体序列排列。以 X 为例, $X(i)$ 表示 X 的第 i 个项目, X 的支持度设为 $\text{sup}(X)$ 。设合集 X 的权值为 $w(X)$, $w(X)$ 为集合 X 中排序在

第一位的项目的权值来表示。对于合集 X , $\text{sup}(X)$ = 包含特定集合 X 的事务在 D 中的总个数/ D 中的事务总数。加权支持度的定义为

$$\text{wsup}(X) = w(X) \text{sup}(X) \quad (1)$$

若合集 X 的加权支持度 $\text{wsup}(X)$ 大于指定的支持度阈值 minsup 时,即 $\text{wsup}(X) \geq \text{minsup}$,则可称 X 为加权频繁项集。当 X 含有 k 个项目时称为加权频繁 k -项集,其中蕴含的形如 $X \Rightarrow Y$ 称为加权关联规则。

1.2 SPARK 框架

Spark 是基于内存的大数据并行计算框架,在兼顾 MapReduce 优点的同时又弥补了它的不足,通过对两种框架的缓存、通信、调度机制和迭代操作等方面进行性能的综合分析,得出以下结论:基于内存计算的 Spark 要优于 MapReduce。Spark 还提供了丰富的内置操作,不仅有常规的 $\text{map}()$ 和 $\text{reduce}()$,其自身还带有很多算子,可以将一个弹性分布式数据集 RDD(resilient distributed datasets)通过特定操作,将原 RDD 转换为一个新的 RDD;相对于 Hadoop 的备份和回滚机制,Spark 中的某一个 RDD 转换失败或者出现内容丢失的情况,可通过前置 RDD 进行重建,保证算法良好的容错性^[13]。所以,Spark 更适合进行频繁项集挖掘这类需要进行迭代计算和交互式处理的任务,可以大幅度提高算法的性能。

1.3 ECLAT 算法

Eclat 算法的主要思想是:首先是扫描数据库,将数据格式从水平数型数据结构转化为垂直数型数据结构,即数据形式为 $\{\text{item}; \text{TIDSet}\}$ 的形式。然后通过项目所对应的事务集进行集合的交运算,从而得到项集和该项集的支持度。Eclat 算法在产生 $(k+1)$ -项集($k > 1$)时,利用到了先验的性质,提高了算法的效率,另外因为每个 k -项集的 TID 集包含了计算支持度的完整信息,所以该算法不需要扫描数据库来确定 $(k+1)$ -项集的支持度。但是,在特定情况下,TID 集的长度可能过长消耗大量的内存空间,在进行集合的交运算时需要消耗大量的运算时间,所以 Eclat 算法对于大型数据集数据的关联规则挖掘的挖掘效率不是很高。这也是目前学者针对 Eclat 算法效率改进的主要研究方向。

1.4 前缀划分策略

将频繁 k -项集提取前 $(k-1)$ 项作为前缀,并按照具有相同前缀原则对其进行划分。将得到的频繁 k -项集($\text{fre-}k$)进行划分,假设不同前缀个数为 s 个,

记为 $\text{fre-}k = \{\text{fre-}k_1, \text{fre-}k_2, \dots, \text{fre-}k_s\}$,在划分后的每个块中除前缀外计算频繁 2-项集,每个块中得到的频繁 2-项集分别为 $\text{fre-}k_1-2, \text{fre-}k_2-2, \dots, \text{fre-}k_s-2$,将得到的每个频繁 2-项集加上该块的前缀取并集就是所有的频繁 $(k+1)$ -项集。采用该策略的剪枝技术可以有效减少求交集运算的数据量,提高关联规则挖掘算法的运行效率^[14]。

2 W-SpEclat 算法

2.1 独立概率完全加权方法

加权关联规则挖掘方式的提出是对传统关联规则挖掘方式的改进,在加权关联规则的研究中,项目重要程度的具体表示—权值,一直是研究工作的重要话题,因为权值的赋予具有很大的主观随意性通常无统一标准,所以在算法中引入权值后往往会破坏算法的封闭性,从而造成频繁项集的子集不一定是频繁项集的后果,导致无法利用该性质进行候选项集的剪枝操作,最终降低算法的整体运行效率和有效规则的产出率。所以权值赋予方法是加权关联规则至关重要的一环,为了保证权值的可靠性应当遵循以下两点原则:避免人为主观的为项赋予权值;权值的设置不能独立于事务之外,且权值需体现出项在事务中的重要程度。称遵循以上原则的加权方式为完全加权方式。现有的一些研究,不能针对公共数据集及各行业的真实事务集进行权值统一化赋予,且改进后的算法虽然在运行时间效率上有了提升但是规则产出数量却有下降,在对有限数量的事务集进行挖掘时不能保证有效规则的产出。针对此问题,以项为单位进行独立概率计算的方法可以有效解决,对存在于事务中但是出现概率比较小、很有价值的关联规则进行有效挖掘且能够实现对于同一项目在不同的事务集中被赋予的权值也不同,可以更好的体现该项的重要程度,主要思想是:某一个项目 i_j 的权值 $w(i_j)$ 仅与该项目在事务集中出现的概率有关,在含有多个项目的 k -项集 I 中($k > 1$), k -项集 I 的权值为该项集中所有项目权值的均值,即 $w(I) = (w(i_1) + w(i_2) + \dots + w(i_k))/k$ 。采取设置加权支持度阈值 wsp 的方式对候选频繁项集进行筛选。以达到能够很好的挖掘出隐藏在数据库中的高价值规则的目的。称采取该方法进行关联规则挖掘的算法为独立概率完全加权关联规则算法,对该算法有如下定义。

定义 1(独立概率加权) 项目属性 i_j 的权是与项目特性有关的值,记做 $w(i_j)$ 。在基于概率分布

的加权关联规则挖掘算法中,项目属性 i_j 的权与项目属性 i_j 在数据集中出现的概率有关。在本文中被定义为数据事务集 D 中该项目属性出现概率的频数。即 $w(i_j) = P(i_j)$, 其中 $P(i_j) = |\{T: i_j \in T, T \in D\}| / |D|$, 是项目属性 i_j 在事务集 D 中出现的概率。

定义2(完全加权重值) 项目属性集 I 的权值是项目数据集 I 所有项目属性的权的均值记作 $w(I)$, 如公式(2)所示。其中 m 为项目属性集 I 中项目属性的个数。

$$w(I) = \frac{\sum_{i=1}^m w(i_j)}{m} \quad (2)$$

定义3(完全加权支持度) 项目属性集 I 的加权支持度记作 $wsp(I)$, 如公式(3)所示。其中 $s(I)$ 为项目属性集 I 的支持度。

$$wsp(I) = s(I)w(I) \quad (3)$$

2.2 W-SPECLAT 算法思想

针对 Apriori 算法的多次读取数据库从而对内存和 I/O 造成大量消耗等问题,采用 Eclat 算法进行独立概率完全加权关联规则挖掘。针对 Eclat 算法中需要反复、大量的进行集合交运算的问题,采用位图结构存储项集,并采取共享前缀划分策略对频繁项集进行划分,使算法效率进一步的提升。针对单机运算的局限性和基于 MapReduce 分布式框架运算的缺点,采取 Spark 分布式框架进行运算使算法更能适应于对大数据集的挖掘。本文提出了 W-SpEclat 算法—并行的独立概率完全加权关联规则挖掘算法,该算法在 Spark 框架下实现并行运行。W-SpEclat 算法共分成两个部分:分别是数据预处理、权值计算并过滤得到加权频繁 1-项集;计算加权频繁 k -项集。

在算法的第一阶段中,首先从分布式文件系统 HDFS 中将数据以 Spark RDD 的形式加载出来并保存。然后将水平数据库转化为本文所需要的垂直数据库,此时数据以 (item TidSet) 的垂直形式存储于 RDD 中,在将处理好的数据保存回 HDFS 中。通过 RDD 再次迭代垂直数据,使用 map() 将所有事务中包含的每一个项 (item) 及该事物所对应的事务编号 (Tid) 形成键值对,使用 reduce() 将每一个项及所对应的所有事务编号进行合并。对所有事务编号进行计数,即为该项在数据集中的出现频数 (TidSet-Count) 作为该项的权值。通过权值与该项的支持度进行乘积运算,将所得结果与给定的最小加权支持度阈值进行比较,如果小于则该项为非加权频繁项;

如果大于则该项为加权频繁项。把数据集中的非加权频繁项过滤掉,就得到了加权频繁 1-项集。最后,将加权频繁 1-项集的 TidSet 保存在位图中,如 TidSet 为 {1,3} 时,则将位图的第 1、3 位置上标记为 1,其余位为 0。

在算法的第二阶段中,首先要计算加权频繁 2-项集,在根据加权频繁 2-项集得出加权频繁 k -项集 ($k > 2$),在计算加权频繁 2-项集时,通过循环遍历所有加权频繁 1-项集,对所有可能的候选项集的两个项的 TidSet 进行位图交集操作,该交集的长度与该候选 2-项集的权值进行乘积运算,所得结果即为该 2-项集的加权支持度,与最小加权支持度阈值进行比较,大于或等于的即为加权频繁 2-项集。在对加权频繁 k -项集进行计算时,需要先使用前缀划分策略对加权频繁 2-项集进行前缀划分,得出共享前缀,各分布式计算节点在计算加权频繁 3-项集时,只需在共享前缀的基础上,重复计算除共享前缀外的加权频繁 2-项集,在与共享前缀进行拼接,生成候选 3-项集,筛选得到加权频繁 3-项集。频繁 2-项集的计算开销,远远小于计算频繁 3-项集,以此类推得到新的共享前缀,这将大大减少加权频繁 k -项集的计算规模,减少通信开销从而提高算法效率。W-SpEclat 算法流程图,如图 1 所示。

2.3 数据预处理

Eclat 算法虽然利用 ($k-1$)-项集作为生成 k -项集的前缀,减少了扫描数据库的时间消耗,提高了算法效率,但是在当数据量的规模较大或最小支持度阈值较小时,候选项集的数目会随之增大,生成 k -项集的前缀数量也会增多,当进行交集操作的同时,会给 CPU 带来巨大的压力。即使在分布式计算框架下,如果当前结点的计算能力不足以完成被分配的任务,则系统会自动将该任务分配到计算能力更强的结点上,此调度过程所需的代价是巨大的。所以在数据预处理阶段,先将数据库形式转换为垂直数据结构,以便于在对频繁 k -项集进行计算时,可以大幅度减少中间候选集的数量,避免频繁任务调度所造成大量计算资源的消耗,提高算法效率。水平数据库向垂直数据库的转换过程,如图 2 所示。

每一个事务 TID 都对应了一个项集 itemSet,表示为在该事务中分别出现的项。将每一个项拿出来并将该项出现的全部 TID 放入到 TidSet 中就得到了垂直数据库。由图 2 可见,对于项 D 分别出现在 TID 为 1,2,3,4 的事务中,所以项 D 所对应的 TidSet 为 {1,2,3,4}。

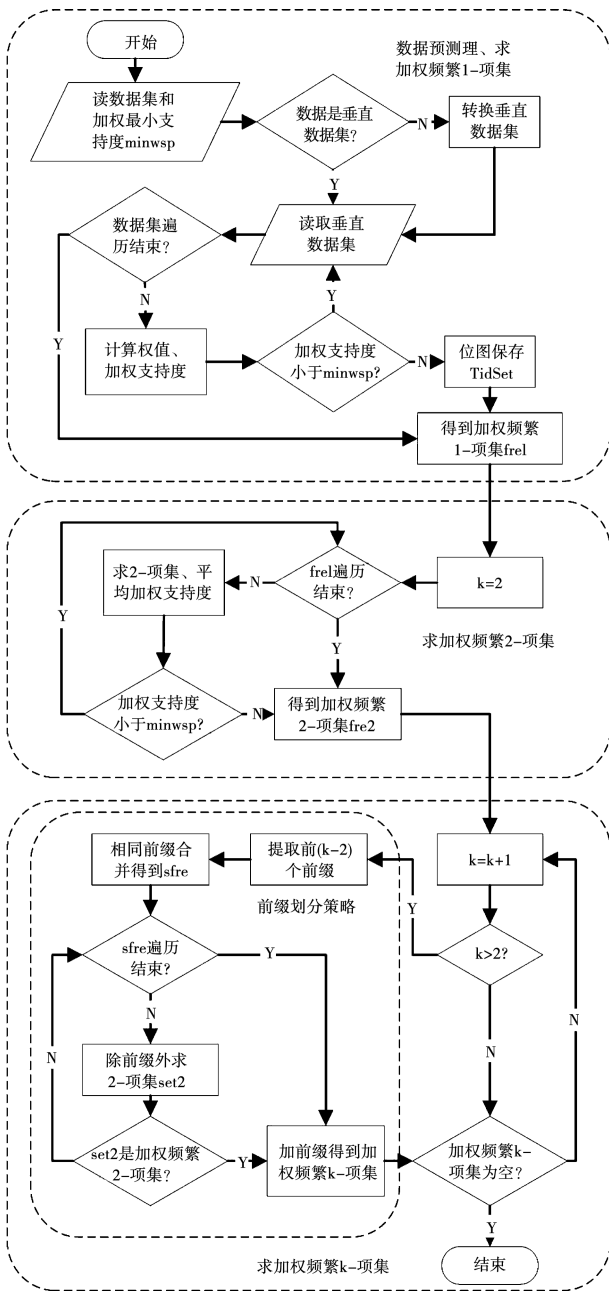


图1 W-SpEclat 算法流程图

Fig. 1 The flowchart of W-SpEclat algorithm

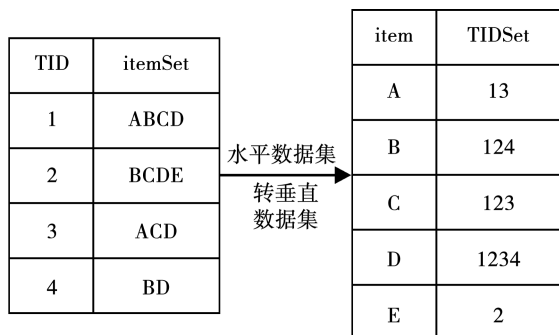


图2 水平形式数据库转化得到垂直形式数据库示意图

Fig. 2 Schematic diagram of horizontal database conversion to vertical database

2.4 权值计算及生成加权频繁 1-项集

为了提高算法的运行效率,在数据预处理阶段生成的垂直数据库已经是每一个项目 item 对应了事务编号集 TIDSet 的形式,在权值计算时,可以应用此结果,对每一个 TIDSet 进行计数后得到了该项所对应的事务数据库中的频数 $P(i_j)$,即 i_j 项的权值 $w(i_j)$,计算结果存放到哈希结构中。为降低由数据集庞大或者最小加权支持度过低所导致的调度代价,在权值计算完成后,直接对加权支持度小于最小加权支持度的数据进行过滤得到加权频繁 1-项集,从而减少整个算法计算过程中产生的中间候选集。图 3 给出了由垂直数据库得到项的权值并在最小加权支持度为 4 时,过滤加权频繁 1-项集的计算过程。

item	TIDSet		item	weight	wsp		item	TIDSet
A	13	计算概率 权值及 项集加权 支持度	A	2	4	过滤加权 频繁 1- 项集	A	13
B	124		B	3	9		B	124
C	123		C	3	9		C	123
D	1234		D	4	16		D	1234
E	2		E	1	1			

图3 权值计算及加权频繁 1-项集生成示例

Fig. 3 Example of Weight Calculation and Weighted Frequent 1-Itemset Generation

由图 3 可见,项 E 的 TIDSet 为 {2}, 只有一个元素即元素 E 只在数据库中的第 2 行出现一次,所以项 E 的权值为 1,根据上述 wsp 计算公式,可以得到 1-项集 {E} 的加权支持度为 1,由于小于设定的最小加权支持度阈值 4,所以将该项集从垂直数据库中删除,剩余的 A, B, C, D 满足要求,留在垂直数据库中,就得到了过滤后的加权频繁 1-项集,对于加权频繁 1-项集,采用位图的形式存放 TIDSet,由于位图是以位进行操作的,所以将位图中与 TIDSet 中的值对应的位置标号设为 1 即可。当生成候选加权 2-项集时,只要两个加权频繁 1-项集的对应 TIDSet 位图中对应位都为 1,则候选加权 2-项集的 TIDSet 位图对应位为 1,否则为 0,以加快候选项集的生成速度及支持度计算。

2.5 计算加权频繁 2-项集

对加权频繁 1-项集进行遍历,通过迭代器对象,获取事务项及该项所对应的 TIDSet,采用 Eclat 中对两个项集取交集的思想同时对两个加权频繁

1-项集组合生成候选 2-项集 itemSet 及对所对应的两个位图取交集得到该候选 2-项集的 TIDSet。两个事务项分别获取对应的权值,根据定义 2,计算出该候选 2-项集的权值。根据定义 3,得到该候选 2-项集的加权支持度并与最小加权支持度阈值进行比较,过滤得到加权频繁 2-项集。由于在权值计算中,已经将数据项作为 key,该项的权值作为 value 存放到哈希结构中,所以在计算候选 2-项集的平均权值时,只需要以数据项作为搜索条件,直接从哈希结构中读取相对应的权值。采取该种对候选 2-项集平均权值的动态计算方法,就可以避免预先计算出所有可能的候选 2-项集的平均权值所消耗的大量计算资源及存储资源。图 4 给出了用图 3 的加权频繁 1-项集来计算加权频繁 2-项集的过程。

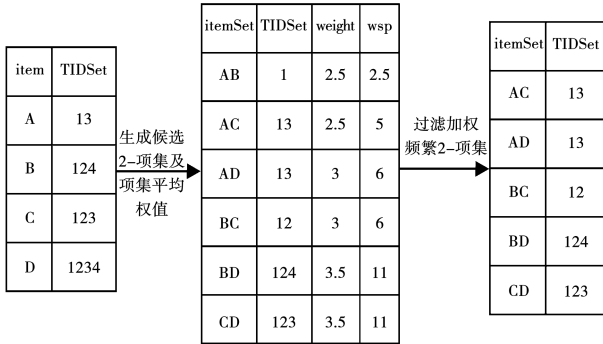


图 4 加权频繁 2-项集生成示例

Fig. 4 Weighted Frequent 2-Itemset Generation Example

由图 4 可见,通过对图 3 中的加权频繁 1-项集得到新的候选 2-项集: $\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}$ 。由于 $\{A, B\}$ 的加权平均支持度为 2, 小于规定的加权最小支持度阈值 4, 所以将该候选 2-项集过滤掉后, 获得加权频繁 2-项集。

2.6 计算加权频繁 $k(k > 2)$ -项集

在进行加权频繁 $k(k > 2)$ -项集的计算前, 需先对加权频繁 $(k-1)$ -项集进行前缀划分: 首先将加权频繁项的前第 $(k-1)$ 个项提取出来, 作为该加权频繁项集的前缀项集, 将提取之后剩余的第 k 项与当前加权频繁项集的 TIDSet 组合生成新的哈希表结构, 在 Spark 框架中, 被拆分的前缀项集和剩余的哈希表结构通过 `map()` 被存放在一个 RDD 中, 将所有加权频繁项完成拆分之后, 执行 `reduceByKey()` 算子对有相同前缀项集的 RDD 进行合并成一个 RDD, 该 RDD 包含两部分, 第 1 部分为前缀项集作为 key 值, 第 2 部分为拥有该前缀项集的哈希表结构所构成的大哈希表结构。

通过前缀划分我们在对计算加权频繁 $k(k > 2)$ -项集计算时, 只需对加权频繁 $(k-1)$ -项集进行前缀划分, 然后对每个加权频繁项集剩余的第 k 项进行加权频繁 2-项集的计算, 再与对应的前缀项集进行拼接即得到了加权频繁 k -项集。由于对 2-项集的挖掘要远远小于对 $k(k > 2)$ -项集挖掘所需消耗的计算资源, 所以前缀划分策略可以大幅度提高算法的执行效率。图 5 为图 4 中得到的加权频繁 2-项集使用前缀划分策略进行划分的过程示例。

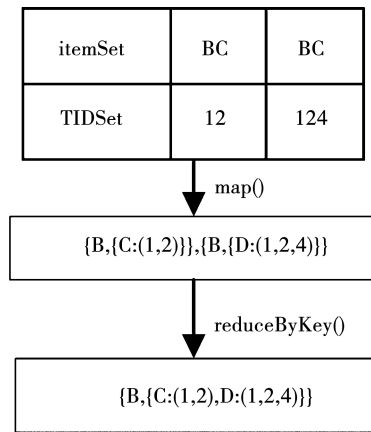


图 5 前缀划分策略示例

Fig. 5 Example of Prefix Partitioning Policy

图 5 是对图 4 中以 B 为前缀的加权频繁 2-项集 $\{B, C\}: (1, 2), \{B, D\}: (1, 2, 4)$ 进行前缀划分的过程。首先通过 Spark 中的 `map()` 转换算子进行前缀提取得到 $\{B, \{C: (1, 2)\}\}, \{B, \{D: (1, 2, 4)\}\}$, 然后通过 `reduceByKey()` 行动算子进行合并得到最后的前缀划分结果 $\{B, \{C: (1, 2), D: (1, 2, 4)\}\}$ 。

根据对加权频繁 2-项集的计算, 在得到一个新的前缀项集的同时, 会生成一个拥有共同前缀且规模为 1 的候选项集, 对此项集重复进行上述加权频繁 2-项集的操作, 在对候选加权频繁 3-项集进行过滤时, 需要将新计算的候选 2-项集与其前缀项集进行拼接, 组成一个候选加权频繁 3-项集, 计算该项集的平均权值得到加权支持度, 与最小加权支持度阈值进比较后进行过滤, 筛选出符合规定的项集, 生成加权频繁 3-项集。对图 5 前缀划分后除去前缀之外的部分 $\{C: (1, 2), D: (1, 2, 4)\}$ 进行加权频繁 2-项集的计算, 得到 $\{C, D\}: (1, 2)$ 。由于该 2-项集的加权支持度为 7, 大于最小加权支持度的阈值, 所以只需将前缀 B 加入到 $\{C, D\}$ 中, 就得到了加权频繁 3-项集 $\{B, C, D\}: (1, 2)$ 。

以此类推,当对频繁 k -项集的计算时以自底向上的形式并行的进行迭代,用加权频繁 k -项集生成加权频繁 $(k+1)$ -项集。在每一次进行前缀划分前,进行一次判断,如果当前项集已经是空集则终止对加权频繁 $(k+1)$ -项集的迭代,反之则继续进行迭代计算。即通过 Spark 框架,将项集分配给不同的计算结点,进行前缀划分计算,重复进行计算加权频繁 2-项集等步骤,将前缀项集与加权频繁 2-项集进行拼接,生成加权频繁 k -项集。此过程中,前缀项集中项的个数为 $(k-2)$ 个,与新生成的加权频繁 2-项集拼接后就得到了规模为 k 的加权频繁项集。

2.7 算法实现

数据预处理阶段,需要对初始数据集的存储形式从水平型数据库,转换为垂直型数据库。垂直型数据库的形式,便于进行生成 TIDSet 及对项的权值计算。该过程需要对数据库进行一次完整的遍历,对每一个数据项进行判断,对应行号在相同项的 TIDSet 位图的对应位置 1 或是以当前项为 *key* 值,创建新的 TIDSet 位图结构。数据预处理的结果以哈希结构的形式,以项作为 *key* 值,以该项对应的所有行号所组成 TIDSet 位图作为 *value* 值存放在内存中,以便于项的权值计算及加权频繁 1-项集的生成。数据预处理的伪代码如算法 1 所示:

算法 1 数据预处理算法

输入 水平形式数据集路径 *path*

输出 垂直形式数据集 *dataprocess*

```

1  javaRDD←sc. textFile( path)
2  dataprocess←new HashMap
3  for row ∈ javaRDD do
4    count←1 //设置行号
5    set1←row. split(“”)
6    for i ∈ set. length do
7      if set1[ i] ∈ dataprocess. key then //item
        已存在
8        dataprocess. put ( set1 [ i] , set1 [ i]
        . values + count)
9      else //item 不存在
10       dataprocess. put( set1 [ i] , count)
11       count + +
12     for i ∈ dataprocess do
13       s←s + i. key + i. value + ”\n”
14     returndataprocess

```

对数据预处理后存放在内存中的哈希结构,生成一个新的 RDD 算子,通过迭代器遍历每一个项及

其 TIDSet 并对 TIDSet 内的元素进行计数运算。以项作为 *key*,计数结果作为 *value* 将计算结果储存在哈希结构中。权值计算、生成加权频繁 1-项集的伪代码如算法 2 所示:

算法 2 权值、加权频繁 1-项集计算

输入 垂直形式数据集 *dataprocess*、最小加权支持度 *wsp* 最小加权支持度 *wsp*

输出 满足最小加权支持度的频繁 1-项集

w_fre1

```

1  javaRDD←sc. parallelize( dataprocess)
2  w_map←new HashMap
3  str←“”
4  w_key←map. keyset()
5  w_vlaue←map. values()
6  while w_key, w_vlaue do
7    str←w_vlaue. split(“”)
8    w_map. put ( w_key, str. length ) //项的
    权值
9  for i ∈ dataprocess do
10   weight←w_map. get( s + i. key)
11   s←s + i. key + i. value + ”\n”
12   w_fre1←new HashMap()
13   for i in map. size() do
14     if s[ 1: ]. size * weight > = wsp do //加权
        频繁 1-项集
15       w_fre1. add( s[ 0] , BitSet( s[ 1: ]. size)
        //转位图形式
16   return w_fre1

```

在加权频繁 2-项集的计算中,通过项值在哈希结构中获取当前项的权值,计算该候选 2-项集的平均权值,从而计算出加权支出度。将过滤后的加权频繁 2-项集存放在内存中,方便与后续加权频繁 k -项集的运算。计算加权频繁 2-项集的伪代码如算法 3 所示。

算法 3 计算加权频繁 2-项集算法

输入 加权频繁 1-项集 *w_fre1*、最小支持度 *wsp*

输出 满足最小加权支持度的频繁 2-项集

w_fre2

```

1  w_fre2←new HashMap()
2  while i ∈ w_fre1 do
3    bitset2 ← i. values()
4    while j ∈ w_fre1 and i < j
5      bitset3←j. values()
6      bitset4←bitset2&&bitset3

```

```

7  Weight ← w_map.get(i.key()) + w_map.get(j.key())/2
8  if bitset4.size() ≥ wsp do //加权频繁 2-项集
9      w_fre2.put(w_fre1.get(i) + “,” + w_fre1.get(j), bs4)
10 return w_fre2

```

进行加权频繁 k -项集运算前,先对加权频繁 $(k-1)$ -项集进行前缀划分,然后迭代进行规模为 2 的加权频繁项集挖掘。重复多次的加权频繁 2-项集的挖掘所消耗的计算资源远小于加权频繁 k -项集的计算资源。计算频繁 k -项集的伪代码如算法 4 所示:

算法 4 计算加权频繁 k -项集 ($k > 2$) 算法

```

输入  频繁  $(k-1)$ -项集  $w\_frek-1$ 、最小支持度  $wsp$ 
输出  满足最小支持度的频繁  $k$ -项集  $w\_frek$ 
1  map = new HashMap()
2  while w_fre1-1.key hasNext
3      pre = w_fre1-1.key[0:k-2] //提取 k-2 个前缀
4      suf = new HashMap()
5      suf.put(w_fre1-1.key-pre, w_fre1-1.get(w_fre1-1.key)) //除前缀项外都作为后缀
6      map.put(pre, suf)
7  javaRDD = sc.parallelizePairs((List) map)
8  map1 = new HashMap()
9  mappreadd ← javaRDD.reduceByKey(i.suf + j.suf) //前缀相同时后缀合并
10  fre2 ← getfre_2(mappreadd.value, key, wsp)
11  w_fre2 = mappreadd.key + w_fre2
12  return w_fre2

```

2.8 算法时间复杂度分析

对一些值进行假设,以数学公式的形式表示算法的时间复杂度。设数据集中事务总数为 T ,分布式框架包含计算节点数为 N ,频繁 1-项集的总数为 t ,每条事务的平均长度为 g 。对于本文算法,数据预处理中遍历每一行水平数据,得到每个项的 TID,时间复杂度为 $O(t^2)$ 。生成加权频繁 2-项集的过程主要由双层 while 循环组成,其时间复杂度为 $O(t^2)$ 。生成加权频繁 k -项集的过程主要由对加权频繁 $(k-1)$ -项集的迭代与规模为 2 的加权频繁项集计算构成,其时间复杂度为 $O(t^3)$ 。在哈希结构中查询权值的时间为 $O(1)$,分布式框架映射器所用

时间为 $O(\frac{T}{N}g)$ 。所以本文算法的时间复杂度为 $O(t^3 + \frac{T}{N}g)$ 。而 Apriori 算法的时间复杂度主要取决于计算频繁 1-项集及频繁 k -项集的筛选和剪枝,所以 Apriori 算法时间复杂度为 $O(TN + t^4)$ 。本文算法的时间复杂度优于 Apriori 算法。

3 数值实验与结果分析

3.1 实验环境

使用两台服务器作为集群的主从节点,配置为第六代 Intel 处理器,128 GB 内存,3 TB 硬盘。集群的配置环境参数如下:Ubuntu 16.04;Hadoop version 2.5.1;Spark version 2.1.0,其中 Hadoop 集群的堆大小设置为 25 G,Spark 的 executor 内存设置为 25 G。开发语言为 Java 语言,JDK 的版本为 jdk1.7.0,使用 IntelliJ IDEA 进行代码开发、编译和运行。

3.2 数据集

为了验证 W-SpEclat 算法的正确性,在 T40I10D100K 数据集上与 MapReduce 下采取同一独立概率完全加权方式的 Eclat 算法进行结果对比,T40I10D100K 数据集是被广泛应用关联规则挖掘的数据集和比较算法的性能;在某大学计算机科学与技术学院 2013 – 2016 级全体学生成绩数据(StudentSC)上与无加权 SparkEclat 算法的处理结果进行对比。数据集的参数如表 1 所示。

表 1 数据集的参数

Tab. 1 Parameters of dataset

数据集	T40I10D100K	StudentSC
事务平均长度	39.6	6.3
项个数	942	244
事务总数	10 000	883
大小	0.16 GB	15 188 KB

3.3 算法挖掘时间效率比较

将本文的 W-SpEclat 的算法与 MapReduce 框架下的独立概率完全加权 Eclat 算法进行算法效率对比,使用 T40I10D100K 公共数据集。为了保证实验结果的正确性,MapReduce 集群环境与 Spark 集群环境均使用 3.1 节中的实验环境,设计 3 组不同加权支持度阈值进行实验,并在每次算法输出完全一致的情况下对 5 次算法运行时间取平均数作为最终实验结果。比较结果如图 6 所示。

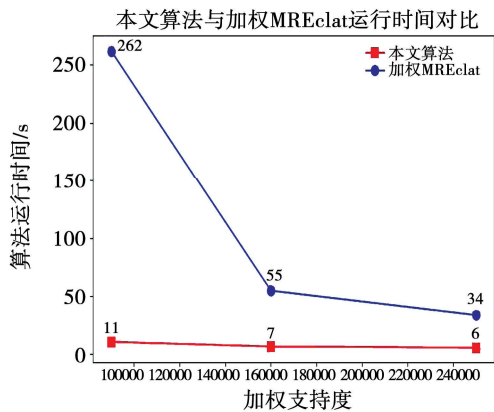


图6 不同框架下效率对比

Fig. 6 Efficiency comparison on different frameworks

由图6可见,在同一数据集、同一加权支持度阈值下,本文的 W-SpEclat 算法的运行时间要远小于 MapReduce 下的加权 Eclat 算法。当加权支持度阈值为1 000 000的时,本文算法优于对比算法约26倍,随加权支持度阈值的降低,加权频繁项集的数目和规模也随之增多,对比算法执行所消耗的时间近似于直线式增长,而本文的 W-SpEclat 算法的运行时间增长较为缓慢,基本保持水平。当加权支持度阈值为360 000时,本文算法优于对比算法约184倍,发生这种情况的原因是:本文算法采用的是 Spark 框架,将运算中间结果存入内存中,减少大量 I/O 通信的时间;使用位图结构加快候选加权频繁项集及加权支持度的计算速度;通过前缀划分的方式使规模为 k 的加权频繁项集运算降低为规模为2的加权频繁项集的运算,从而提升了运算效率,减少了算法运行时间。由此可知本文算法性能的提升会随数据集规模的增大而体现的愈发明显。

3.4 算法挖掘性能比较及实例分析

将本文的 W-SpEclat 算法与不加权的 SparkEclat 算法所生成的规则数量进行对比,使用 StudentSC 真实数据集以保证算法所挖掘出规则的实用性。为保证实验结果的正确性,在支持度阈值一致的条件进行了5组实验进行结果对比。比较结果如图7所示。

由图7可见,在同一数据集与支持度阈值的情况下,本文的 W-SpEclat 算法生成的规则数要远大于 Spark 下的非加权 Eclat 算法。当支持度为90时,本文算法通过加权挖掘过程,生成的规则数比 SpEclat 算法多84条,这种优势会随着支持度阈值的减小而逐渐变大。当支持度为30时,W-SpEclat 算法可以挖掘出2 409条有效规则相比 SpEclat 算法多挖掘出2 111条规则,因为对相同数量的1-项集进

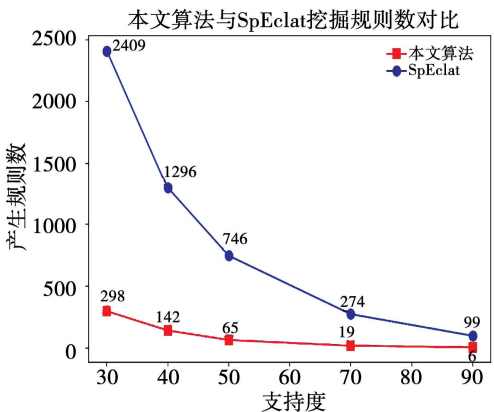


图7 加权前后的规则数对比

Fig. 7 Comparison of the number of rules before and after weighting

行关联规则挖掘时,W-SpEclat 算法可以获取更多的信息,这些信息往往是隐藏在项集之间且常规关联规则挖掘算法无法挖掘出来的规则。两种算法在真实数据集 StudentSC 中挖掘出的候选项集 HI 、加权频繁项集 FI 及关联规则数量比较如表2所示,其中的项分别代表不同的科目: $\{D1\}$ 电子技术(模拟)、 $\{N1\}$ 概率论与数理统计、 $\{F1\}$ 复变函数、 $\{F2\}$ 积分变换、 $\{G1\}$ 高等数学(一)、 $\{G2\}$ 高等数学(二)、 $\{X1\}$ 线性代数。

表2 两种算法在 StudentSC 中挖掘的频繁2-项集实例
Tab. 2 Examples of 2-itemsets mined by two algorithms in StudentSC

算法	项集数	项集实例
SpEclat	HI 21	$\{D1, N1\}, \{D1, F1\}, \{D1, F2\}, \{D1, G1\}, \{D1, G2\}, \{D1, X1\}, \{N1, F1\}, \{N1, F2\}, \{N1, G1\}, \{N1, G2\}, \{N1, X1\}, \{F1, F2\}, \{F1, G1\}, \{F1, G2\}, \{F1, X1\}, \{F2, G1\}, \{F2, G2\}, \{F2, X1\}, \{G1, G2\}, \{G1, X1\}, \{G2, X1\}$
	FI 2	$\{G1, X1\}, \{G2, X1\}$
W-SpEclat	HI 21	$\{D1, N1\}, \{D1, F1\}, \{D1, F2\}, \{D1, G1\}, \{D1, G2\}, \{D1, X1\}, \{N1, F1\}, \{N1, F2\}, \{N1, G1\}, \{N1, G2\}, \{N1, X1\}, \{F1, F2\}, \{F1, G1\}, \{F1, G2\}, \{F1, X1\}, \{F2, G1\}, \{F2, G2\}, \{F2, X1\}, \{G1, G2\}, \{G1, X1\}, \{G2, X1\}$
	FI 6	$\{G1, X1\}, \{G2, X1\}, \{D1, X1\}, \{N1, X1\}, \{F1, X1\}, \{F2, X1\}$

由表2可见,在本文算法 W-SpEclat 和对比算法 SpEclat 的候选 2-项集 HI 相同的前提下,分析两个算法对相同候选 2-项集进行频繁 2-项集挖掘的结果可以得出:本文算法在挖掘出对比算法所挖掘的关联规则基础上可以对隐藏信息进行进一步的挖掘从而得到更多的关联规则。如频繁项集 $\{N1, X1\}$ 表示学生在概率论与数理统计与线性代数这两门课挂科之间体现了关联关系,其指导老师可以根据此规则对 $N1$ 成绩不合格的同学进行成绩预警,促进学生加权对 $X1$ 的学习、降低 $X1$ 的挂科率。发生以上情况的原因是:本文算法采用的是基于独立概率完全加权的挖掘方式,通过取项集平均权值的方式在判断候选 k -项集是否是频繁项集时降低了大概率事件的加权支持度,提高了小概率事件的加权支持度,使得这部分小概率事件可以参与到加权频繁 k -项集的生成过程中,而不是随着项集规模 k 的增大而被大量过滤掉,最后以关联规则的方式体现到运算结果中,所以本文算法可以发掘出更多的关联规则,更加全面的、真实的体现该数据集中隐藏的信息,最大程度的发挥关联规则在实际应用领域的指导意义。

4 结 论

针对 Eclat 挖掘效率低及不能在有限的数据集中挖掘出更有价值的关联规则的问题,给出了并行的独立概率完全加权关联规则算法,该算法在 Eclat 算法只需要遍历一次数据库的基础上通过采取位图存储频繁项集及通过前缀划分与设置支持度阈值共同剪枝的方法同时应用 Spark 分布式计算框架使算法的运行效率得到大幅度的提升,并且本文算法通过独立概率完全加权,提升一部分项集的加权支持度使得最终挖掘出的有效关联规则数大幅增加,得到原本隐藏在数据集中,但是未能被挖掘的有效信息。同时本文算法可以在低支持度或高支持度的情况下,保证算法的运行高效性及挖掘的有效性,可广泛应用于各行业的事务数据库的关联规则挖掘中。

参 考 文 献:

- [1] 李成严, 辛雪, 赵帅, 等. Sp-IEclat: 一种大数据并行关联规则挖掘算法[J]. 哈尔滨理工大学学报, 2021, 26(4):109.
LI Chengyan, XIN Xue, ZHAO Shuai, et al. Sp-IEclat: A Big Data Parallel Association Rule Mining Algorithm [J]. Journal of Harbin University of Science and Technology, 2021, 26(4):109.
- [2] GUO H, LIU H, CHEN J Y, et al. Data Mining and Risk Prediction Based on Apriori Improved Algorithm for Lung Cancer[J]. Journal of Signal Processing Systems, 2021:1.
- [3] 马月坤, 刘鹏飞, 张振友, 等. 改进的 FP-Growth 算法及其分布式并行实现[J]. 哈尔滨理工大学学报, 2016, 21(2):20.
MA Yuekun, LIU Pengfei, ZHANG Zhenyou, et al. Improved FP-Growth Algorithm and Its Distributed Parallel Implementation [J]. Journal of Harbin University of Science and Technology, 2016, 21(2):20.
- [4] VIGER P F, LIN C W, VO B, et al. A Survey of Itemset Mining[J]. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2017, 7(4):e1207.
- [5] WANG L. An Efficient Algorithm of Frequent Itemsets Mining Based on MapReduce[J]. Journal of Information & Computational Science, 2014, 11(8):2809.
- [6] CHON K W, KIM M S. BIGMiner: A Fast and Scalable Distributed Frequent Pattern Miner for Big Data [J]. Cluster Computing, 2018, 21:1507.
- [7] Shashi Raj, Dharavath Ramesh, Krishan Kumar Sethi. A Spark-based Apriori Algorithm with Reduced Shuffle Overhead[J]. The Journal of Supercomputing, 2020, 77(1):1.
- [8] WEN H, KOU M, HE H, et al. A Spark-based Incremental Algorithm for Frequent Itemset Mining[C]// Proceedings of the 2018 2nd International Conference on Big Data and Internet of Things October 2018:53.
- [9] Krishan Kumar Sethi, Dharavath Ramesh. HFIM: A Spark-based Hybrid Frequent Itemset Mining Algorithm for Big Data Processing[J]. The Journal of Supercomputing, 2017, 73(8):3652.
- [10] CAI C H, FU P, WAI K W, et al. Mining Association Rules with Weighted Items[C]// Database Engineering and Applications Symposium, 1998. Proceedings. IDEAS'98. International. IEEE, 1998:68.
- [11] Bay Vo, Frans Coenen, Bac Le. A New Method for Mining Frequent Weighted Itemsets Based on WIT-trees[J]. Expert Systems With Applications, 2013, 40(4):1256.
- [12] 黄名选, 蒋曹清. 基于完全加权正负关联模式挖掘的越-英跨语言查询译后扩展[J]. 电子学报, 2018, 46(12):3029.
HUANG Mingxuan, JIANG Caoqing. Vietnamese-English Cross Language Query Post-Translation Expansion Based on All-Weighted [J]. Acta Electronica Sinica, 2018, 46(12):3029.
- [13] LEE H, KANG M, YOUN S, et al. An Experimental Comparison of Iterative MapReduce Frameworks [C]// ACM. ACM, 2016.
- [14] WANG L. An Efficient Algorithm of Frequent Itemsets Mining Based on MapReduce [J]. Journal of Information & Computational Science, 2014, 11(8):2809.

(编辑:温泽宇)