Association Rule Mining

Yew-Kwong Woon
Nanyang Technological University, Singapore

Wee-Keong Ng
Nanyang Technological University, Singapore

Ee-Peng Lim
Nanyang Technological University, Singapore

INTRODUCTION

Association Rule Mining (ARM) is concerned with how items in a transactional database are grouped together. It is commonly known as market basket analysis, because it can be likened to the analysis of items that are frequently put together in a basket by shoppers in a market. From a statistical point of view, it is a semiautomatic technique to discover correlations among a set of variables.

ARM is widely used in myriad applications, including recommender systems (Lawrence, Almasi, Kotlyar, Viveros, & Duri, 2001), promotional bundling (Wang, Zhou, & Han, 2002), Customer Relationship Management (CRM) (Elliott, Scionti, & Page, 2003), and cross-selling (Brijs, Swinnen, Vanhoof, & Wets, 1999). In addition, its concepts have also been integrated into other mining tasks, such as Web usage mining (Woon, Ng, & Lim, 2002), clustering (Yiu & Mamoulis, 2003), outlier detection (Woon, Li, Ng, & Lu, 2003), and classification (Dong & Li, 1999), for improved efficiency and effectiveness.

CRM benefits greatly from ARM as it helps in the understanding of customer behavior (Elliott et al., 2003). Marketing managers can use association rules of products to develop joint marketing campaigns to acquire new customers. The application of ARM for the cross-selling of supermarket products has been successfully attempted in many cases (Brijs et al., 1999). In one particular study involving the personalization of supermarket product recommendations, ARM has been applied with much success (Lawrence et al., 2001). Together with customer segmentation, ARM helped to increase revenue by 1.8%.

In the biology domain, ARM is used to extract novel knowledge on protein-protein interactions (Oyama, Kitano, Satou, & Ito, 2002). It is also successfully applied in gene expression analysis to discover biologically relevant associations between different genes or between different environment conditions (Creighton & Hanash, 2003).

BACKGROUND

Recently, a new class of problems emerged to challenge ARM researchers: Incoming data is streaming in too fast and changing too rapidly in an unordered and unbounded manner. This new phenomenon is termed data stream (Babcock, Babu, Datar, Motwani, & Widom, 2002).

One major area where the data stream phenomenon is prevalent is the World Wide Web (Web). A good example is an online bookstore, where customers can purchase books from all over the world at any time. As a result, its transactional database grows at a fast rate and presents a scalability problem for ARM. Traditional ARM algorithms, such as Apriori, were not designed to handle large databases that change frequently (Agrawal & Srikant, 1994). Each time a new transaction arrives, Apriori needs to be restarted from scratch to perform ARM. Hence, it is clear that in order to conduct ARM on the latest state of the database in a timely manner, an incremental mechanism to take into consideration the latest transaction must be in place.

In fact, a host of incremental algorithms have already been introduced to mine association rules incrementally (Sarda & Srinivas, 1998). However, they are only incremental to a certain extent; the moment the universal itemset (the number of unique items in a database) (Woon, Ng, & Das, 2001) is changed, they have to be restarted from scratch. The universal itemset of any online store would certainly be changed frequently, because the store needs to introduce new products and retire old ones for competitiveness. Moreover, such incremental ARM algorithms are efficient only when the database has not changed much since the last mining.

The use of data structures in ARM, particularly the trie, is one viable way to address the data stream phenomenon. Data structures first appeared when programming became increasingly complex during the 1960s. In his classic book, *The Art of Computer Programming* Knuth (1968) reviewed and analyzed algorithms and data structures that are necessary for program efficiency.

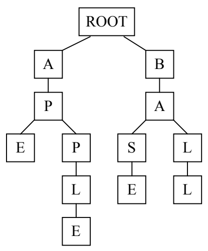
Since then, the traditional data structures have been extended, and new algorithms have been introduced for them. Though computing power has increased tremendously over the years, efficient algorithms with customized data structures are still necessary to obtain timely and accurate results. This fact is especially true for ARM, which is a computationally intensive process.

The trie is a multiway tree structure that allows fast searches over string data. In addition, as strings with common prefixes share the same nodes, storage space is better utilized. This makes the trie very useful for storing large dictionaries of English words. Figure 1 shows a trie storing four English words *(ape, apple, base,* and *ball).* Several novel trielike data structures have been introduced to improve the efficiency of ARM, and we discuss them in this section.

Amir, Feldman, & Kashi (1999) presented a new way of mining association rules by using a trie to preprocess the database. In this approach, all transactions are mapped onto a trie structure. This mapping involves the extraction of the powerset of the transaction items and the updating of the trie structure. Once built, there is no longer a need to scan the database to obtain support counts of itemsets, because the trie structure contains all their support counts. To find frequent itemsets, the structure is traversed by using depth-first search, and itemsets with support counts satisfying the minimum *support threshold* are added to the set of frequent itemsets.

Drawing upon that work, Yang, Johar, Grama, & Szpankowski (2000) introduced a binary *Patricia trie* to reduce the heavy memory requirements of the preprocessing trie. To support faster support queries, the authors added a set of horizontal pointers to index nodes. They also advocated the use of some form of primary threshold to further prune the structure. However, the compression achieved by the compact Patricia trie comes at a hefty price: It greatly complicates the horizontal pointer index, which is a severe overhead. In addition, after compression, it will be difficult for the Patricia trie to be updated whenever the database is altered.

Figure 1. An example of a trie for storing English words



The *Frequent Pattern-growth* (FP-growth) algorithm is a recent association rule mining algorithm that achieves impressive results (Han, Pei, Yin, & Mao, 2004). It uses a compact tree structure called a *Frequent Pattern-tree* (FP-tree) to store information about frequent 1-itemsets. This compact structure removes the need for multiple database scans and is constructed with only 2 scans. In the first database scan, frequent 1-itemsets are obtained and sorted in support descending order. In the second scan, items in the transactions are first sorted according to the order of the frequent 1-itemsets. These sorted items are used to construct the FP-tree. Figure 2 shows an FP-tree constructed from the database in Table 1.

FP-growth then proceeds to recursively mine FP-trees of decreasing size to generate frequent itemsets without candidate generation and database scans. It does so by examining all the *conditional pattern bases* of the FP-tree, which consists of the set of frequent itemsets occurring with the suffix pattern. Conditional FP-trees are constructed from these conditional pattern bases, and mining is carried out recursively with such trees to discover frequent itemsets of various sizes. However, because both the construction and the use of the FP-trees are complex, the performance of FP-growth is reduced to be on par with Apriori at support thresholds of 3% and above. It only achieves significant speed-ups at support thresholds of 1.5% and below. Moreover, it is only incremental to a certain extent, depending on the FP-tree *watermark* (validity support threshold). As new transactions arrive, the support counts of items increase, but their relative support frequency may decrease, too. Suppose, however, that the new transactions cause too many previously infrequent itemsets to become

Table 1. A sample transactional database

TID	Items
100	*AC*
200	*BC*
300	*ABC*
400	*ABCD*

Figure 2. An FP-tree constructed from the database in Table 1 at a support threshold of 50%