Induction of Association Rules: Apriori Implementation

Christian Borgelt and Rudolf Kruse

Department of Knowledge Processing and Language Engineering
School of Computer Science
Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, D-39106 Magdeburg, Germany

Abstract. We describe an implementation of the well-known *apriori algorithm* for the induction of association rules [Agrawal *et al.* (1993), Agrawal *et al.* (1996)] that is based on the concept of a *prefix tree*. While the idea to use this type of data structure is not new, there are several ways to organize the nodes of such a tree, to encode the items, and to organize the transactions, which may be used in order to minimize the time needed to find the frequent item sets as well as to reduce the amount of memory needed to store the counters. Consequently, our emphasis is less on concepts, but on implementation issues, which, however, can make a considerable difference in applications.

Keywords. association rule, apriori algorithm, prefix tree, item coding

1 Introduction

Association rule induction is a powerful method for so-called *market basket analysis*, which aims at finding regularities in the shopping behavior of customers of supermarkets, mail-order companies, on-line shops and the like. With the induction of association rules one tries to find sets of products that are frequently bought together, so that from the presence of certain products in a shopping cart one can infer (with a high probability) that certain other products are present. Such information, expressed in the form of association rules, can often be used to increase the number of items sold, for instance, by appropriately arranging the products in the shelves of a supermarket (they may, for example, be placed adjacent to each other in order to invite even more customers to buy them together).

The main problem of association rule induction is that there are so many possible rules. For the product range of a supermarket, for example, which may consist of several thousand different products, there are billions or even trillions of possible association rules. It is obvious that such a vast number of rules cannot be processed by inspecting each one in turn. Efficient algorithms are needed that restrict the search space and check only a subset of all rules, but, if possible, without missing important rules.

The importance of a rule is usually measured by two numbers: Its *support*, which is the percentage of transactions that the rule can be applied to (or, alternatively, the percentage of transactions, in which it is correct), and its *confidence*, which is the number of cases in which the rule is correct relative to the number of cases in which it is applicable (and thus is equivalent to an estimate of the conditional probability of the consequent of the rule given

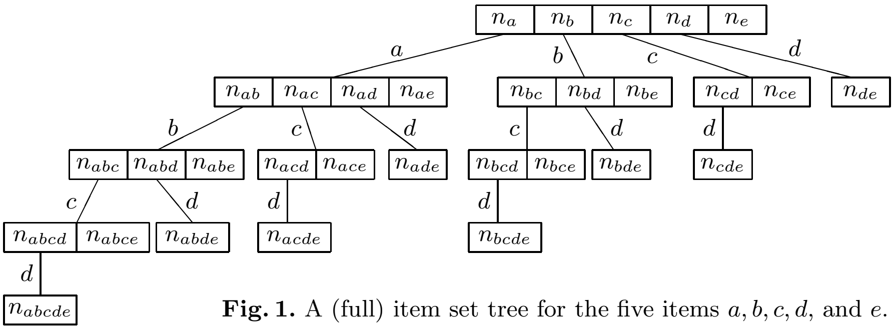

Fig. 1. A (full) item set tree for the five items a, b, c, d, and e.

its antecedent). To select interesting rules from the set of all possible rules, a *minimum support* and a *minimum confidence* are fixed.

Among the best known algorithms for association rule induction is the *apriori algorithm* [Agrawal *et al.* (1993), Agrawal *et al.* (1996)]. This algorithm works in two steps: In a first step the *frequent item sets* (often misleadingly called *large item sets*) are determined. These are sets of items that have at least the given minimum support (i.e., occur at least in a given percentage of all transactions). In the second step association rules are generated from the frequent item sets found in the first step. Usually the first step is the more important, because it accounts for the greater part of the processing time. In order to make it efficient, the apriori algorithm exploits the simple observation that no superset of an infrequent item set (i.e., an item set not having minimum support) can be frequent (can have enough support).

In this paper we describe an implementation of the apriori algorithm that has gained some popularity in the research community. This program was written by the first author of this paper and can be retrieved free of charge at `http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html`.

2 Item Set Trees

In order to find the frequent item sets, we have to count the transactions they are contained in. Our implementation is based on the idea to organize the counters for the item sets in a special kind of *prefix tree*, which not only allows us to store them efficiently (using little memory), but also supports processing the transactions as well as generating the rules. The structure of a (full) item set tree as our implementation uses it is shown in Figure 1. Each n_S denotes a counter for an item set S. The edge labels on a path from the root to some node specify the common part of the item sets for which there are counters in that node. Since the common part would be a prefix if we were dealing with sequences instead of sets, such a data structure is commonly called a *prefix tree*. That we are dealing with sets, not sequences, is the reason why this tree structure is unbalanced: n_{abc}, for instance, is the same as n_{bca} and therefore only one of these counters is needed.

During the first step of the apriori algorithm this tree is created level by level. That is, in a first traversal of the transactions the frequencies of the one element sets are determined (the root node is created), in a second traversal the two element sets are checked (the second tree level is created—the children of the root) and so on. Of course, in doing so, some branches of the tree can be pruned, because by exploiting the simple observation mentioned above we can find out whether a branch can contain frequent item sets or not.

3 Node Organization

There are different data structures that may be used for the nodes of this tree. In the first place, we may use simple vectors of integer numbers to represent the counters. The items (note that we only need one item to distinguish between the counters of a node, see above) are not explicitly stored in this case, but are implicit in the vector index. Alternatively, we may use vectors, each element of which consists of an item identifier (an integer number) and a counter, with the vector elements sorted by the item identifier.

The first structure has the advantage that we do not need any memory to store the item identifiers and that we can very quickly find the counter corresponding to a given item (we simply use the item identifier as an index), but it has the disadvantage that we may have to add "unnecessary" counters (i.e., counters for item sets, of which we know from the information gathered in previous steps that they must be infrequent), because the vector may not have "gaps". This problem can only partially be mitigated by enhancing the vector with an offset for the first element and a size, so that unnecessary counters at the margins of the vector can be discarded. The second structure has the advantage that we only represent the counters we actually need, but it has the disadvantage that we need extra memory to store the item identifiers and that we have to carry out a binary search in order to find the counter corresponding to a given item.

A third alternative would be to use a hash table per node. However, although this reduces the access time, it increases the amount of memory needed, because for optimal performance a hash table must not be too full. In addition, it does not allow us to exploit easily the order of the items (see below). Therefore we do not consider this alternative here.

Obviously, if we want to optimize speed, we should choose simple counter vectors, despite the gap problem. If we want to optimize memory usage, we can decide dynamically, which data structure is more efficient in terms of memory, accepting the costs for the binary search if necessary.

It should also be noted that we need a set of child pointers per node, at least for all levels above the currently added one (in order to save memory, we should not create child pointers before we are sure that we need them). For organizing these pointers we have basically the same options as for organizing the counters. However, if the counters have item identifiers attached, we have an additional possibility. We may draw on the organization of the counters, using the same order of the items and leaving child pointers nil if we do not need them. This can save memory, even though we may have unnecessary nil pointers, because we need not store item identifiers a second time.

4 Item Coding

It is clear that the way in which the items are coded (i.e., are assigned integer numbers as identifiers) can have a significant influence on the gap problem for counter vectors mentioned above. Depending on the coding we may need large vectors with a lot of gaps or we may need only short vectors with few gaps. A good heuristic approach to minimize the number and size of gaps seems to be the following: It is clear that frequent item sets contain items that are frequent individually. Therefore it is plausible that we have only few gaps if we sort the items w.r.t. their frequency, so that the individually frequent items receive similar identifiers (and, of course, infrequent items are discarded entirely). In this case it can be hoped that the offset/size representation of a vector can discard the greater part of the unnecessary counters, because these can be expected to cluster at the vector margins.

Extending this scheme, we may also consider to code the items w.r.t. the number of frequent pairs (or even triples etc.) they are part of, thus using additional information from the second (or third etc.) level to improve the coding. This extension, however, is not yet incorporated in the publicly available version of our program (version 4.0 at the time of this writing).

5 Recursive Counting

The item set tree is not only an efficient way to store the counters, it also makes processing the transactions very simple, especially if we sort the items in a transaction ascendingly w.r.t. their identifiers. Then processing a transaction is a simple doubly recursive procedure: To process a transaction for a node of the tree, (1) go to the child corresponding to the first item in the transaction and process the remainder of the transaction recursively for that child and (2) discard the first item of the transaction and process it recursively for the node itself (of course, the second recursion is more easily implemented as a simple loop through the transaction). In a node on the currently added level, however, we increment a counter instead of proceeding to a child node. In this way on the current level all counters for item sets that are part of a transaction are properly incremented.

By sorting the items in a transaction, we can also apply the following optimizations (this is a bit more difficult—or needs additional memory—if hash tables are used to organize the counters and thus explains why we are not considering hash tables): (1) We can directly skip all items before the first item for which there is a counter in the node, and (2) we can abort the recursion if the first item of (the remainder of) a transaction is beyond the last one represented in the node. Since we grow the tree level by level, we can even go a step further: We can terminate the recursion once (the remainder of) a transaction is too short to reach the level currently added to the tree.

6 Transaction Representation

The most simple way of processing the transactions is to handle them individually and to apply to each of them the recursive counting procedure described in the previous section. However, the recursion is a very expensive procedure and therefore it worth considering how it can be improved. One approach is based on the fact that often there are several similar transactions, which lead to a similar program flow when they are processed. By organizing the transactions into a prefix tree (an idea that has also been used by Han *et al.* (2000) in a different approach) transactions with the same prefix can be processed together. In this way the procedure for the prefix is carried out only once and thus considerable performance gains can result. Of course, the gains have to outweigh the additional cost of constructing such a transaction tree. This extension, however, is also not yet incorporated in the in the publicly available version of our program (version 4.0 at the time of this writing).

7 Experimental Results

We conducted several experiments in order to study the effects of the different optimizations we discussed above. The results we present here were selected,

item sorting	node structure	transaction representation	transaction prep. (s)	subset check (s)	memory used (Mb)
none	vector	individually	—	30.5	152.1
indiv., descending	vector	individually	—	36.1	171.2
indiv., ascending	vector	individually	—	26.2	103.2
pairs, descending	vector	individually	—	23.9	83.5
pairs, ascending	vector	individually	—	18.9	63.3
none	dynamic	individually	—	37.6	51.2
indiv., descending	dynamic	individually	—	44.3	50.8
indiv., ascending	dynamic	individually	—	33.8	52.6
none	vector	prefix tree	0.1	24.1	152.1
indiv., descending	vector	prefix tree	0.1	31.5	171.2
indiv., ascending	vector	prefix tree	0.1	16.2	103.2
none	dynamic	prefix tree	0.1	28.6	51.2
indiv., descending	dynamic	prefix tree	0.1	38.0	50.8
indiv., ascending	dynamic	prefix tree	0.1	20.4	52.6

Table 1. Results on the webview dataset (max. 8 items, min. support 0.056%).

item sorting	node structure	transaction representation	transaction prep. (s)	subset check (s)	memory used (Mb)
none	vector	individually	—	187.9	167.4
indiv., descending	vector	individually	—	173.4	66.3
indiv., ascending	vector	individually	—	80.3	67.5
pairs, descending	vector	individually	—	176.4	73.5
pairs, ascending	vector	individually	—	89.0	69.6
none	dynamic	individually	—	219.1	59.8
indiv., descending	dynamic	individually	—	205.2	52.6
indiv., ascending	dynamic	individually	—	80.8	67.4
none	vector	prefix tree	0.2	30.3	167.4
indiv., descending	vector	prefix tree	0.2	22.9	66.3
indiv., ascending	vector	prefix tree	0.2	27.6	67.5
none	dynamic	prefix tree	0.2	42.7	59.8
indiv., descending	dynamic	prefix tree	0.2	32.6	52.6
indiv., ascending	dynamic	prefix tree	0.2	28.2	67.4

Table 2. Results on the census dataset (max. 8 items, min. support 0.05%).

because they demonstrate these effects most clearly.

We used two datasets: The BMS-webview-1 dataset, which was also used as a benchmark by Zheng *et al.* (2001) and which is derived from the KDD-Cup 2001 data [Kohavi *et al.* (2000)]. In contains click-stream data from Gazelle.com, a small leg-care company, which no longer exists. Each transaction describes a web session consisting of all the product detail pages viewed. The second dataset is the well-known census dataset (also known as adult dataset) from the UCI machine learning repository [Blake and Merz (1998)], which contains data from the US Census Bureau. Each record of this dataset describes one tax payer. We preprocessed this dataset by discretizing numeric attributes and replacing the values of a record by expressions of the form "*attribute*=*value*". These two dataset exhibit very different characteristics and are thus very well suited to show the differences between the optimizations studied above.

The results of our experiments are shown in Table 1 for the webview dataset and in Table 2 for the census dataset. The fixed parameters are a maximum number of eight items per frequent item set for both datasets and a minimal support value of 0.056% for the webview dataset and a minimal support value of 0.05% for the census dataset. The other parameters used are indicated in the first columns of the tables. Item sorting refers to whether items were sorted based on individual frequencies or on number of frequent pairs they are part of. The node structure is either a pure counter vector or a dynamic, memory minimizing choice between the options discussed above. The third column states whether transactions were processed individually or organized into a prefix tree. All times were measured on an AMD Athlon Thunderbird 1.33GHz system running S.u.S.E. Linux 7.3. The apriori program was compiled with gcc version 2.95.3. The memory used for the item set tree was computed using an estimate for the overhead of the memory management system. It is consistent with output of the `top` system utility.

From these tables it can be seen that sorting the items can considerably reduce memory usage as well as processing time, with ascending sorting faring best (conjectured reason: it leads to more nodes having shorter vectors thus mitigating the gap problem). Considering frequent pairs is not necessarily better than sorting w.r.t. individual frequencies. The dynamic node structure leads to minimal memory usage, but often at a considerable loss in performance. The gains from organizing the transactions in a prefix tree clearly outweigh the construction costs. In addition, with prefix trees, the performance loss of the memory minimization is less severe.

8 Conclusions

In this paper we discussed several ways to optimize the performance of the classic apriori algorithm for inducing association rules. As our experimental results confirm, considerable performance improvements can be achieved with relatively simple modifications. However, which optimizations are best for a given problem can depend heavily on the dataset used.

References

Agrawal, R., Imielienski, T., and Swami, A. (1993). Mining Association Rules between Sets of Items in Large Databases. In: *Proc. Conf. on Management of Data*, 207–216. New York: ACM Press

Agrawal, A., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A. (1996). Fast Discovery of Association Rules. In: Fayyad *et al.* (1996), 307–328

Blake, C.L. and Merz, C.J. (1998). *UCI Repository of Machine Learning Databases*. Dept. of Information and Computer Science, University of California at Irvine
`http://www.ics.uci.edu/ mlearn/MLRepository.html`

Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R., eds. (1996). *Advances in Knowledge Discovery and Data Mining.* Cambridge: AAAI Press / MIT Press

Han, J., Pei, J., and Yin, Y. (2000). Mining Frequent Patterns without Candidate Generation. In: *Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX).* New York: ACM Press

Kohavi, R., Bradley, C.E., Frasca, B., Mason, L., and Zheng, Z. (2000). KDD-Cup 2000 Organizers' Report: Peeling the Onion. SIGKDD Exploration 2(2):86–93.

Zheng, Z., Kohavi, R., and Mason, L. (2001). Real World Performance of Association Rule Algorithms. In: *Proc. 7th Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD'01)*. New York: ACM Press