CAREN - A Java based Apriori Implementation for Classification Purposes

Paulo J Azevedo

Technical Report - January 2003

Abstract

In this document a java based implementation of the well know Apriori algorithm is described. The association rule generator is constructed toward the generation of classifiers. A detailed description of the data structures to store the itemsets is given along with the most important steps of the algorithm. Benchmarking and discussion on the main features is also presented.

1 Introduction

The Apriori algorithm [Agrawal & Srikant94] is considered the classic approach for the derivation of association rules. Association rules are rules of the form

$$a_1 \ \& \ a_2 \ \& \ ... \ \& \ a_n \rightarrow c$$

Rules of this format describe association (or simply co-occurrence) between atomic elements present in data. These elements can be items bought in supermarket or genes present in a certain chromosome, or simply a pair of attribute/value items from a relational database table. Rules are made out of itemsets present in the dataset, i.e. combination of items. For instance, itemset $a_1a_2a_3...a_nc$ gave rise to the former rule. Quality of rules are measure through statistics metrics like support and confidence. Support describes incidence of the itemset in the dataset and confidence measure the predictability strength of the rule. Support is calculated by itemset counting among the transactions contained in the dataset. The main task of generating association rules is itemset counting. From the perspective of itemset counting, apriori is considered a bottom-up breath-first search algorithm.

Apriori counts itemsets by generating candidates and performing databases scans. For each group of itemsets (candidates) with cardinality k, a database scan is made. Itemsets who satisfy a minimal occurrence constraint are considered *frequent*. Candidates to frequent itemsets of size k are generated from frequent itemsets of cardinality k-1. The operation to generate candidates reduces to a joining between itemsets.

Apriori implements a downward closure property claiming that: "*an itemset can only be frequent if all its subsets are frequent*". Thus, only k-itemsets that have all its subsets of size k-1 frequent should be counted. In summary, for candidates generation one needs two procedures. First, joining frequent itemsets of size k-1 to generate candidates of size k, and secondly, check that all subsets of size k-1 belonging to candidate of size k are frequent itemsets.

The Caren system supports two version of the Apriori implementation. One is target for attribute/value format datasets and the other is for basket format datasets.

The first is oriented for ASCII files with several columns representing different attributes. The latter is traditionally oriented to represent supermarket shop baskets where each line is number with a transaction number and contains a set of products.

Datasets are always disk-resident. This means that a pass through the dataset is done using disk access. This option was taken bearing in mind the processing of large datasets. Thus, memory constraint are imposed by the size of the itemsets present and counted in the dataset, and not by the size of the dataset.

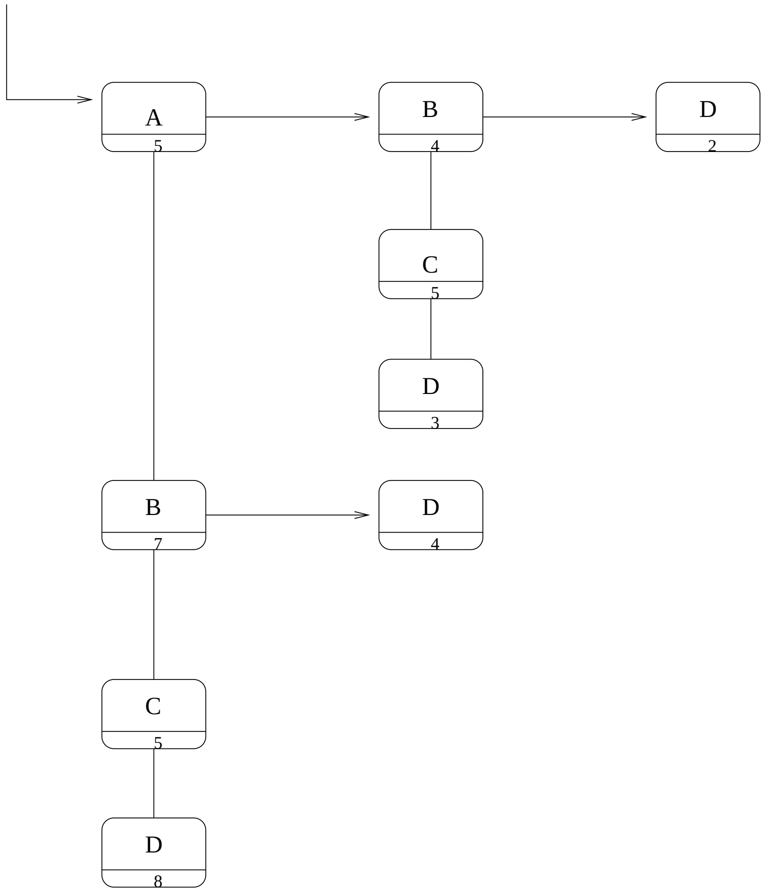

Figure 1: Trie representing itemsets $\{A, B, C, D, AB, AC, AD, BD, ABD\}$

2 Data Structures

Itemsets are stored in a *trie structure* (also know as prefix-tree) [Brin et al.97]. This data structure is organized as an directed acyclic graph where nodes represent items names. A sequence of nodes represents an itemset. Each node contains a counter representing the incidence of the itemset. In the case of the attribute/value version, items are represented by pairs of nodes. Each pair is composed of an attribute node and a value node.

Figure 1 pictorially represents itemsets $\{A, B, C, D, AB, AC, AD, BD, ABD\}$. The depth on the trie indicates the size of the itemset. That is, nodes of the depth 2

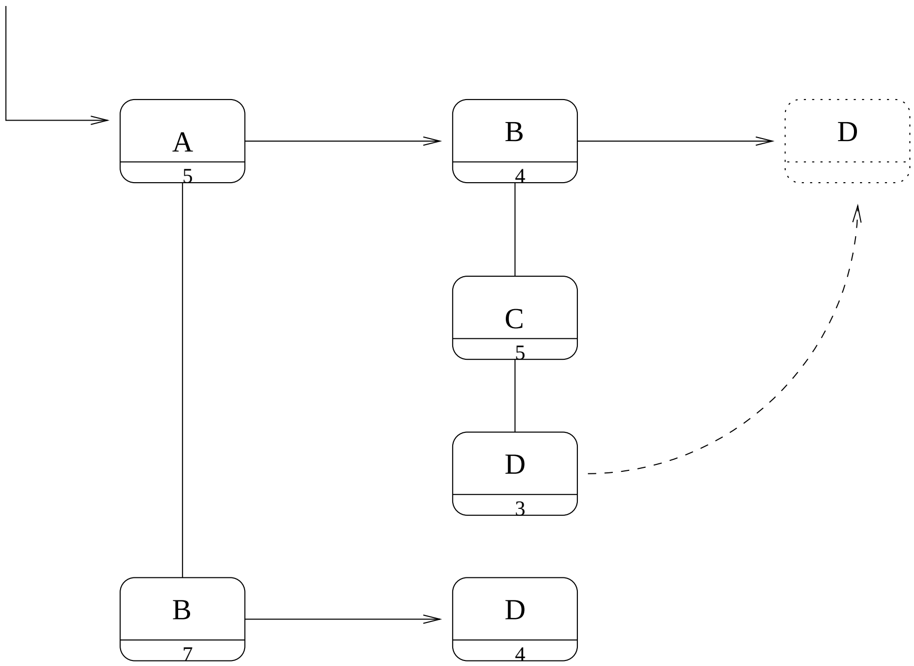

Figure 2: Joining AB with AD to obtain ABD

represent 2-itemsets. Thus, when counting 2-itemsets only nodes of depth 2 in the trie are visited. Notice that items are always lexicographically ordered along the same level (depth).

3 Generating Candidates

One major advantage of a trie-based representation arises from the fact that frequent itemsets and apriori candidates are materialized into the same nodes. Thus, there is no duplication of an itemset when it turns from candidate to frequent itemset. Candidates are "generated" by joining itemsets of the same depth. For instance to derive the candidate ABD we could join AB with AD. It is a straightforward process since it is a simple copying of nodes from the same level where we want to expand. The new copies are located in the sequel of the expanded node e.g. in figure 2 AB is the expended node and D is copied, obtaining ABD. Obviously it requires checking for subset occurrence to ensure that a generated candidate has all its subsets as frequent itemsets (this is referred as the *prune step* in [Agrawal & Srikant94]). Since k is the number of subsets of size (k-1) that a k-itemset contains, performing subset verification is reduced to subset finding in the try. That is, traversing the try looking for subsets belonging to the candidate. For instance, to verify that ABD is a valid candidate we count the number of subsets of size 2 present in the trie. It is a valid candidate if the number of subsets is 3, which actually is since AB, AD and BD occurs.

Trie structures are quite convenient for counting itemsets in a transaction (this operation is referred as *subset* in [Agrawal & Srikant94]). Checking which candidates a transaction contains is obtained by traversing the transaction guided by the trie. The trie is traversed according to the sequence of itemsets that exist in the transaction.

Whenever a node is visited, its counter is incremented. We assume that transactions are ordered according to the order of the trie i.e. lexicographical order. The depth explored in the trie coincides with the size of the candidates of be counted.

Contrasting with the original Apriori approach, CAREN does not generate and materializes subsets out of each transaction. Rather it inspects each transaction checking for candidates (driven by the Trie-structure) without any additional subset generation. Considerer the itemsets of figure 2 and a transaction with the items $ABCD$. We start from A on the trie and try to match the itemset beginning at node A in the transaction. Thus, node A is visited together with AB, (ABD is not visited since the size is bigger than 2), AC and AD, by this order. Then we follow to node B. Since candidates of size 2 are being counted, only the counters for itemsets AB, AC, AD and BD are incremented.

Some authors e.g. [Hipp et al.00], claim that a major drawback of trie-based representations is that itemsets that are not prefixes of a candidate in the trie impose an overhead when counting occurs. In fact this is a pertinent overhead and benchmarking had shown some performance degradation. We also observed that candidate counting is not particularly efficient in this approach (neither is in the original Apriori implementation). However, in the near future we will present a novel proposal for candidate counting with the aim of overcoming both overheads.
Filtering itemsets that are *large* [Agrawal & Srikant94] ($support > minsup$) and consequently are preserved in the trie is also very simple to implement. For iteration k, the nodes of depth k in the trie are visited and the support constraint is applied to each counter. CAREN also implements some database trimming by excluding counting transactions that cannot contain candidates e.g. testing transactions size against candidates size.

In terms of itemset counting and considering this itemset representation, the Apriori algorithm has a $\mathcal{O}(kn - \frac{1}{2}k^2 + \frac{1}{2}k)$ behavior where n is the size of the transaction and k the biggest itemset present in the dataset.

4 Discretization Processes

With the purpose of dealing with numeric attributes, CAREN has two discretization processes implemented. The first is a binary discretization method which mimics the discretization approach of the C4.5 [Quinland93]. Given a class attribute (default is the last described attribute of the dataset) and a target attribute, the process selects a pivot value among the range of values in the domain of the target attribute. The selection is performed following an entropy measure, guided by the class attribute. The pivot value defines two partitions in the attribute data. In the sequel, this attribute is interpreted as having only two artificial values: "$\leq pivot$" and "$> pivot$". The class and target attributes are loaded into an array to perform the entropy based counting and pivot selection. Thus, some memory restrictions exists for this discretization method.

The second method is the discretization method described in [Srikant & Agrawal96]. The basic idea is to derived intervals from the range of values present in the attribute to be discretized. The new derived intervals replace the old range of values in the attribute. The size and number of intervals obey to a principle of information loss (partial completeness measurements). The user supplies a parameter K (level of partial completeness) which represents the degree of freedom for the partition process. K is used in the configuration of intervals for each discretized attribute. The loss of information grows with the size of K. Partitioning is always equi-depth. That is, the discretization process always generates intervals of equal size.

The number of partitions (intervals) is given by the formula:

$$num\_int = \frac{2 \text{ x } n}{m \text{ x } (K-1)}$$

where n is the number of quantitative attributes, m is the minimal support value and K is the partial completeness value. After generating partitions, intervals can be merged according to a Maximal support constraint. Adjacent intervals are merged provided the resulting interval support does not overcome maximal support.

Discretization is always performed after 1-itemset counting and before pruning. The convenience of this situation comes from the fact that at this point attribute values are already available and easy to handle. After the discretization is performed the new derived values have to be counted. This happens for each discretized attribute in the itemsets trie. New attribute values derived by the discretization processes are represented as constraint e.g. $X \leq pivot$, $X \in [a, b]$. This requires reformulation of the counting process. Counting is implemented by checking the constraint against each of the original dataset values. Hence, discretized attributes are always slower to count since we pass from a simple equality checking to a constraint satisfaction checking.

5 Rules Generation

Rules are generated according to the standard algorithm [Agrawal & Srikant94]:

1) For each frequent itemset l generate all $l \subset a$.
2) For each a generate a rule

$$a \Rightarrow (l - a)$$

if $s(l)/s(a) \geq minconf$

where $minconf$ is the confidence filter.

At the moment, Caren only generates rules where $|l - a| = 1$ due to the purpose of this implementation (generate classification rules where the consequent is the class attribute). Moreover, rules where consequents have more than one item are quite difficult to interpret. However, we are planning to expand, in the future, the generation procedure to derive rules where $|l-a| > 1$. Rules are stored in a Hash table where the keys represent the head of the rule. The corresponding values are lists of pointer to arrays. Each array represents the set of items that composed a rule's body. The node (in the list) representative of the pointer also contains the support and confidence of the rule. The complete rules hash table can be saved via the normal Java Streams. It can also be reloaded by a classifier that can reorganize the rules according to some classification strategy (for instance to implement a decision list formed out of a set of classification rules). Figure 3 pictorially represents the hash table for a set of association rules. Notice that a keys corresponds to several rule bodies, where each body is a set of items stored in an array.

Caren implements rule filtering through the values of minimal confidence. However we plan to have in the future other strength metrics for rule selection. At the moment it is possible to perform rule filtering by measuring redundancy between rules. Instead of measure strength of rules we detect redundancy among rules with the same consequent and related antecedent. The used metric is *improvement* [Bayardo et al.00]. A rule gives rise to improvement if it increases the strength measure (in our case confidence) when compared to a simplification. Given a minimal value of improvement, Caren selects only specific rules that contribute to a significant predictive advantage (or are in fact relevant to the analysis).

Several different output formats are implemented in Caren. Rules can be accessed as a Java Stream file (that can be loaded by another Java application) or as a text file in several different formats. At the moment, Caren generates ASCII and CSV format for describing generated rules. A Prolog format is available, where rules are represented by Prolog clauses. PMML file format (a XML format to represent predicative models) is also available.

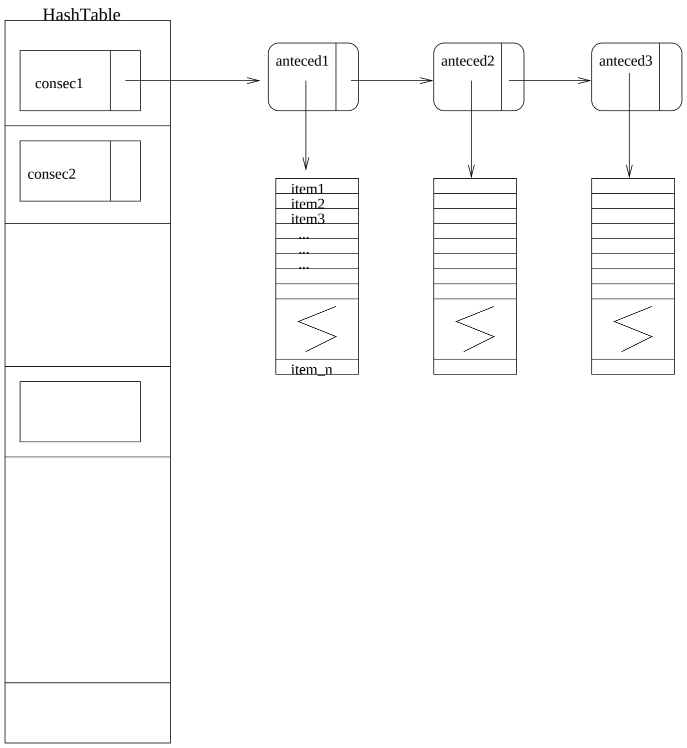

Figure 3: Hash Table for Association Rules

6 Benchmarking

Results presented here refer to values based on average time from batches of 10 runs. Datasets are stored in ASCII files and all scans are performed from disk. The hardware used is a PC INTEL P3 based clocked at 1.2 GHz under RedHat Linux. Java runtime

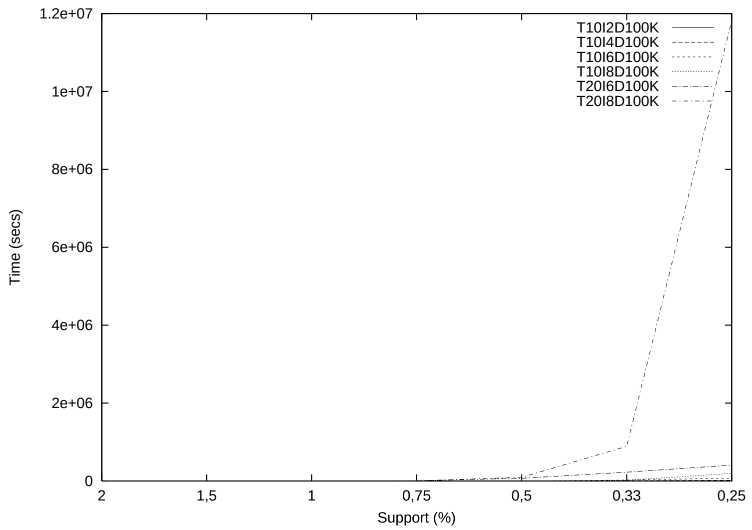

Figure 4: Number of rules generated

environment is JDK1.3.1_02. Time data is obtained by the Linux *time* command. For accuracy sake we use the *user time* produced by this command. This clears up the twisting caused by the unstable *system time*. Each run calculates all frequent itemsets for a specific *minsup* and derive all rules for a value of $minconf = 50\%$. It also includes writing all derived rules to a text format file. The synthetic datasets from [Agrawal & Srikant94] were used. Considering file T10I8D100K, T10 refers to datasets with an average transaction size 10 and (T20 means size 20). I8 means an average size of maximal potentially frequent itemsets of 8. All datasets have 100,000 transactions (D100K). Results show that this apriori implementation scales linearly with values of minimal support and frequent itemsets size. Notice how well computational time mimics the graph of number of derived rules (figure 4).

7 Conclusions

Caren was developed with the aim of producing a tool capable of generating classification rules. That is, rules that could be used with a classification purpose where a classifier makes use of these rules in the form of a decision list. Caren is a disk-based rule engine where the dataset is always consulted from the disk. One should bear in mind this fact and that Caren is Java-based implementation, when comparing performance with implementations like [Agrawal & Srikant94] or even [Hipp et al.00]. However, benchmarking shown that the algorithm scales along the number of derived rules (calculated frequent itemsets), which is the expected apriori behavior.

Future work will be carried on along the lines of performance improvement. As an example we plan to add new "tricks" for clever itemset counting (or not counting), novel itemsets representations and new mechanisms for candidate counting using improved transactions representation.

Acknowledgments: Thanks to Alipio Jorge for all the suggestions. This work is supported by the POSI/2001/CLASS Project sponsored by Fundação Ciência e Tecnologia.

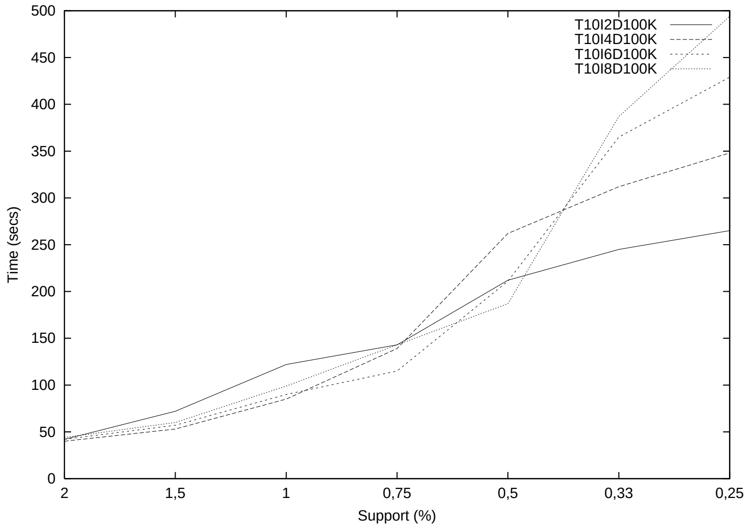

Figure 5: Performance over T10 datasets

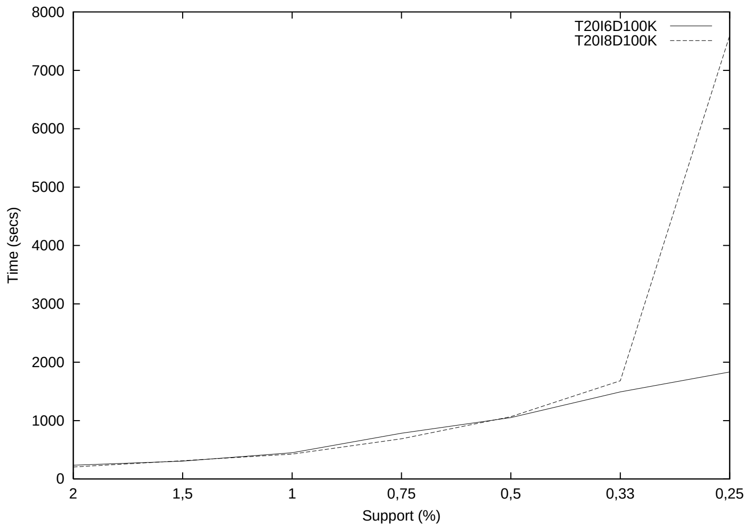

Figure 6: Performance over T20 datasets

References

[Agrawal & Srikant94] Agrawal R., Srikant R.
Fast Algorithms for Mining Association Rules
in Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile, Sept. 1994

[Bayardo et al.00] Bayardo R., Agrawal R., Gunopulos D.
Constraint-Based Rule Mining in Large, Dense Databases
in Journal of Data Mining and Knowledge Discovery, Vol 4, Number 2/3, pag 217-240, July 2000.

[Brin et al.97] Brin S., R. Motwani J. Ullman and S. Tsur
Dynamic Itemset Counting and Implication Rules for Market Basket Data
in Proceedings of ACM SIGMOD International Conference in Management of Data, 1997

[Hipp et al.00] Hipp J., Guntzer U., Nakhaeizadeh G.
Algorithms for Association Rule Mining - A General Survey and Comparison
in SIGKDD Explorations June 2000, Vol 2, Issue 1, 2000.

[Quinland93] Quinland J. R.,
C4.5: Programs for Machine Learning
Morgan Kaufman 1993.

[Srikant & Agrawal96] Srikant R., Agrawal R.
Mining Quantitative Association Rules in Large Relational Tables
in Proceedings of the SIGMOD International Conference in Management of Data, 1996