International Journal of Application or Innovation in Engineering & Management (IJAIEM)
Web Site: www.ijaiem.org Email: editor@ijaiem.org
Volume 3, Issue 8, August 2014 **ISSN 2319 - 4847**

A Survey of Association Rule Mining for Customer Relationship Management

Aji Pratap Singh Gautam[1], Dr. Natalia Valeriyavna Sharonova[2]

[1]Head, Department of Computer Application, T.E.R.I., P.G. College, Ghazipur, U.P
[2]Head, Department of Intelligence Computer Systems in National Technical University, Kharkov, Ukraine

Abstract

Data mining technique is most frequently used in Customer Relationship Management field. There are many data mining algorithm and methods can be used in solving CRM problems. Association Rule Mining technique is widely used in CRM application. Various algorithms has been proposed by various researches. This paper is an effort to analyze these algorithms in view of their implementation in Customer Relationship Management.

1. Introduction

Association rule is the method to analyze interesting relationship among items on a given set of data and identify patterns of customer behavior by different kinds of association tools. Since its introduce in 1991the task of association rule of mining has received a great deal of attention. A typical example of this would be Market Basket Analysis.

1.1 Association Rule and Frequent Itemsets: The market-basket analysis assumes we have some large number of itemses, e.g., "bread", "milk". Customers fill there market basket with some subset of the items, and we get to know that items people buy together, even if we don't know who they are. Marketers use this information to position items, and control the way a typical customer traverses the store. Association rules are statement of the form { X_1, X_2, ……X_n) → Y, meaning that if we find all of X_1, X_2, …….., X_n in the market basket, than we have a good chance of finding Y. The probability of finding Y for us to accept this rule is called the confidence of the rule. We normally would search only for rules that had confidence above a certain threshold. We may also ask that confidence be significantly higher than it would be if items were placed at random into basket. For example, we might find a rule like { milk, butter} → bread simply because a lot of people by bred. However, the beer/diapers asserts the rule { diapers } → beer holds with confidence significantly greater than the fraction of baskets that contain beer. Ideally, we would like to know that in an association rule of the presence of X_1,…….X_n, actually "causes" Y to be brought. However, "causality is an elusive concept, nevertheless, for market-basket data, the following test suggests what causality means. If we lower the price of diapers and raise the price of beer, we can lure diaper buyers, who are more likely to pick up beer while in the store, thus covering our losses on the diapers. That strategy works because " diapers causes beer." However, working it the other way round, running a sale on beer and raising the price of diapers, will not result in beer buyers buying diapers in any great numbers, and we lose money. In many situations, we only care about association rules or causalities involving sets of items that appear frequently in basket. For example, we can not run a good marketing strategy involving items that no one buys anyway. Thus, much data mining starts with the assumption that we only care about sets of items with high support; i.e., they appear together in many baskets, We then find associations rules or causalities only involving high support set of items i.e., { X_1,……X_n, Y} must appear in at least a certain percent of the baskets, called the support threshold.

1.2 Framework for Frequent Itemset Mining : We use the term itemset for " a set S that appears in at least fraction s of the baskets, "where s is some chosen constant, typically 0.01 or 1%.
We assume data is too large to fit in main memory. Either it is stored in a RDB, say as relation Basket(BID, item) or as flat file of the records of the form (BID, item1, item2,…..item). When evaluating the running time algorithms we:

- Count the number of passes through the data. Since the principal cost is often the best measure of running time of the algorithm.
- If a set of items S is frequent, than every subset of S is also frequent.

To find frequent itemsets, we can:
1. Proceed level wise, finding first the frequent items (sets of size 1), then the frequent pairs, the frequent triples, etc. In our discussion, we concentrate on finding frequent pairs because:
(a) Often, pairs are enough.
(b) In many data sets, the hardest part is finding the pairs; proceeding to higher level takes less time than finding frequent pairs. Level wise algorithms use one pass per level.

2. Find all maximal frequent itemsets (i.e., sets S such that no proper superset of S is frequent) in one pass or a few passes.

2. The A-Priori Algorithm

L_1 = { Large itemsets);

for ($k = 2$; $L_{k-1} \neq \emptyset$; k++) **do begin**

C_k = apriori-gen(L_{k-1});// New candidates
forall transactions $t \in D$ **do begin**
C_i = subset(C_k, t);
forall candidates $c \in C_t$ **do**
c.count++
end
L_k = { $c \in C_k$| c.count $\geq$ minsup}
end
Answer = $\cup_k L_k$

Apriori Candidate Generation:

The apriori –gen function takes as argument L_{k-1}, the set of all large (k-1) itemsets. It returns a superset of the set of all large k – itemsets. The function works as follows. First in he join step we join L_{k-1} . First in join step we join L_{k-1} with L_{k-1} Inscrt C_k

select $p.item_1$, $p.item_3$,…..$p.item_{k-1}$, $q.item_{k-1}$
from L_{k-1} p , L_{k-1} q
where $p.item_1 = q.item_1$……, $p.item_{k-2} = q.item_{k-2}$,
where $p.item_1 = q.item_1$,……$p.item_{k-2} = q.item_{k-2}$,
$p.item_{k-1} < q.item_{k-1}$;
Next in the prune step we delete all itemsets $c \in Ck$ such some (k-1) subset c is not in Lk-1

forall itemsets $c \in Ck$ do

forall (k-1) subset s of c do

if ($s \notin L_{k-1}$) then
delete c from Ck

This algorithm precedes level wise.

1. Given support threshold s, in the first pass we find the items that appear in at least fraction s of the baskets. This set is called L_1, the frequent items. Presumably there is enough main memory to count occurrences of each item, since a typical store sells not more than 100,000 different items.
2. Pairs of items in L_1 become the candidate pairs C_2 for the second pass. We hope that size of C_2 is not so large that there is not room for an integer count per candidate pair. The pairs in C_2 whose count reaches s are the frequent pairs, L_2.
3. The candidate triples, C_3 are those sets { A, B, C} such that all of {A,B}, {A,C} and {B,C} are in L_2. On the third pass count the occurrences of triples in C3; those with a count of at least s are the frequent triples. L_3.
4. Proceed as far as you like. Li is the frequent set of i ; C_{i+1} is the set of sets of size i+1 such that each subset of size i is in Li.

Consider the following SQL on a Baskets(BID; item) relation with 10^8 tuples involving 10^7 baskets of 10 items each; assume 100,000 different items.

```
SELECT b1.item, b2.item, COUNT(*)
FROM Baskets b1, Baskets b2
WHERE b1.BID = b2.BID AND b1.item < b2.item
GROUP BY b1.item, b2.item
HAVING COUNT(*) >= s;
```

Note: s is the support threshold and the second term of the WHERE clause is to prevent pairs of items that are really one item, and to prevent pairs from appearing twice.

In the join Baskets ∞ Baskets, each basket contributes $\binom{10}{2}$ = 45 pairs, so the join has 4:5 x 10^8 tuples. A-priori "pushes the HAVING down the expression tree," causing us first to replace Baskets by the result of

```
SELECT *
FROM Baskets
GROUP by item
HAVING COUNT(*) >= s;
```

If s = 0:01, then at most 1000 items' groups can pass the HAVING condition. Reason: there are 10^8 item occurrences, and an item needs 0:01 ù $10^7 = 10^5$ of those to appear in 1% of the baskets. Although 99% of the items are thrown away by a-priori, we should not assume the resulting Baskets relation has only 10^6 tuples. In fact, all the tuples may be for the high-support items. However, in real situations, the shrinkage in Baskets is substantial, and the size of the join shrinks in proportion to the square of the shrinkage in Baskets.

Improvements to A-Priori: There are two ways, which can improve the performance of this algorithm.

1. Cut down the size of the candidate sets Ci for i ≥ 2. This option is L_1, important, even for finding frequent pairs, since the number of candidates must be su3ciently small that a count for each can fit in main memory.
2. Merge the attempts to find L_2, L_3,…….. into one or two passes, rather than a pass per level.

2.1. Algorithm AprioriTid

The Apriori algorithm also uses the apriori-gen function to determine the candidate itemsets before the pass begins. The interesting feature of this algorithm is that the database D is not used to counting support after the first pass. Rather the set $\bar{C}_k$ is used for this purpose. Each member of set $\bar{C}_k$ is of the form < TID,{X_k} >, where each X_k is a potentially large k-itemset present in the transaction with identifier TID. For k=1, $\bar{C}_1$ corresponds to the database D, although conceptually each item I is replaces by the itemset{i}. For K> 1, $\bar{C}_k$ is generated by the algorithm (step 10). The member of $\bar{C}_k$ corresponding to transaction t is <t.TID, { c ∈ $\bar{C}_k$ | c contained in t} >. If the trisection does not contain any candidate k-itemsetm, then $\bar{C}_k$ will not have an entry for this transaction. Thus the number of entries in $\bar{C}_k$ may be smaller than the number of transaction in the database, especially for large value of k. In addition, for large value of k, each entry may be larger than the corresponding transaction because an entry in Ck includes all candidates in the transactions

Algorithm

```
L1 = {large l-itemsets };
C̄k = databsa D
     for ( k =2; Lk-1 ≠ ∅ ; k++) do begin
     Ck = ariori-gen(Lk-1); // New candidates

C̄k = ∅;
forall entries t ϵ C̄k-1 do begin
```

// determine candidate itemsets in Ck contained in the transaction with identifier t.TID

```
Ct = {c ϵ Ck | c-c[k]} ϵ t.set-of-itemsets ∧ (c-c[k-1]) ∈ t.set-of-itemsets};
forall candidates c ∈ Ct do
c.count++
if (Ct ≠ ∅ ) then C̄k +=<t,TID,Ct>;
end
Lk = { c ∈ Ck| c.count ≥ minsup}
End

Answer = ∪k Lk ;
```

It is not necessary to use the same algorithm in the all the passes over data. In the earlier passes, Apriori does better than AprioriTID. But AprioriTID beats Apriori in later passes because both algorithms are using same candidate generation procedure and therefore count the same itemsets. Based on these observation a hybrid algorithm has been proposed known as AprioriHybrid. That uses Apriori in the initial passes and switches to AprioriTID when it expects that the set $\bar{C}$k at the end of the pass will fit in the memory.

3. PCY Algorithm

Park, Chen, and Yu proposed using a hash table to determine on the first pass (while L_1 is being determined) that many pairs are not possibly frequent. Takes advantage of the fact that main memory is usually much bigger than the number of

items. During the two passes to find L_2, the main memory is laid out as in Fig. 1.

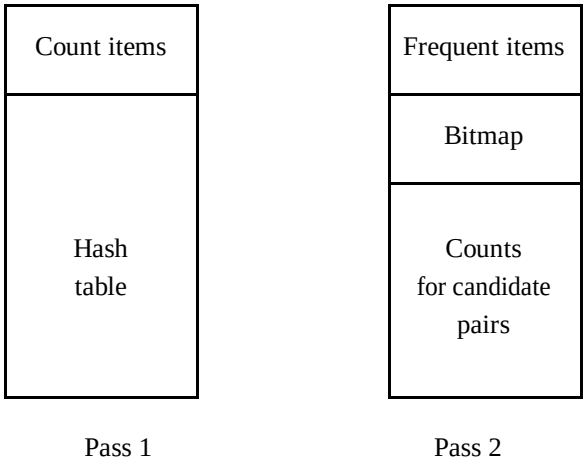

Figure 1: Two passes of the PCY algorithm

Assume that data is stored as a ɜat ɜle, with records consisting of a basket ID and a list of its items.

Pass 1:

(a) Count occurrence of all items.

(b) For each bucket, consisting of items $\{i_1 \ldots . i_k\}$, hash all pairs to a bucket of the hash table, and incremented the count of bucket by 1.

(c) At the end of the pass, determine L_1, the items with counts at least s.

(d) Also at the end, determine those bucket with counts at least s.

- Key point: a pair (i, j) cannot be frequent unless it hashes to a frequent bucket, so pairs that hash to other buckets need not be candidates in. C_2 Replace the hash table by bit map, with one bit per bucket: 1 if the bucket was frequent , 0 if not.

Pass 2:

(a) Main memory holds a list of all the frequent items, i.e. L_1.

(b) Main memory also holds the bitmap summarizing the result of the hashing from pass 1.

- Key point: The bucket must use 16 or 32 bits for a count but these are compressed to 1 bit. Thus, even if the hash table occupied almost the entire main memory on pass 1, its bitmap occupies no more than 1/16 of main memory on pass 2.

(c) Finally, main memory also holds a table with all the candidate pairs and their counts. A pair (i, j) can be a candidate in C_2 only if all of the following is true:

i. i is in L_1.

ii. j is in L_1

iii. (i, j) hashes to a frequent bucket

It is the last condition that distinguishes PCY from straight apriori and reduces the requirements for memory in pass 2.

(d) During pass 2, we consider each basket, and each pairs of items, making the test outlined above. If pair meets all three conditions, add to its count in memory, or create an entry for it if one does not yet exist.

- When does PCY beat apriori? When there are too many pairs of items from L_1 to fit a table of candidate pairs and their counts in main memory, yet the number of frequent buckets in the PCY algorithm is suɜciently small that it reduces the size of C_2 below what can ɜt in memory (even with 1/16 of it given over to the bitmap).
- When will most of the buckets be infrequent in PCY? When there are a few frequent pairs, but most pairs are so infrequent that even when the counts of all the pairs that hash to a given bucket are added, they still are unlikely to sum to s or more.

3.1. The Iceberg" Extensions to PCY

1. Multiple hash tables: share memory between two or more hash tables on pass 1, as in Fig. 2. On pass 2, a bitmap is stored for each hash table; note that the space needed for all these bitmaps is exactly the same as what is needed for the one bitmap in PCY, since the total number of buckets represented is the same. In order to be a candidate in C_2, a pair

must:
(a) Consist of items from L_1 , and
(b) Hash to a frequent bucket in every hash table.

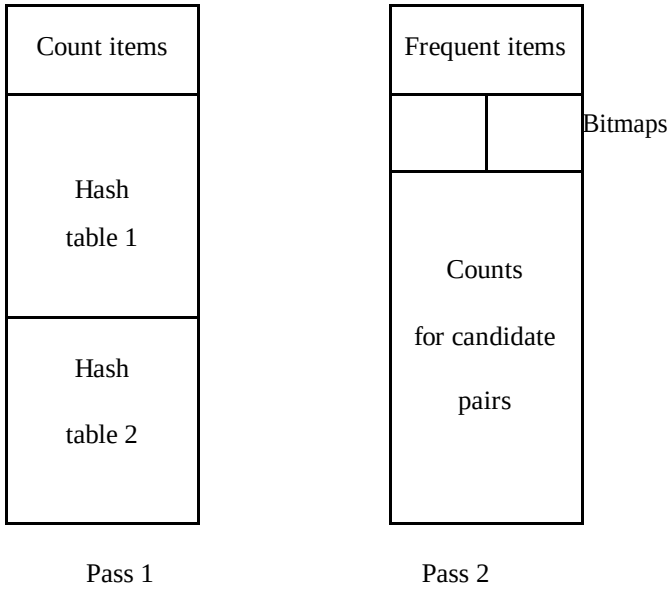

Figure 2: Multiple hash tables memory utilization

2. Iterated hash tables Multistage: Instead of checking candidates in pass 2, we run another hash table (different hash function)in pass 2, but we only hash those pairs that meet the test of PCY; i.e., they are both from L_1 and hashed to a frequent bucket on pass 1. On the third pass, we keep bitmaps from both hash tables, and treat a pair as a candidate in C_2 only if:

(a) Both items are in L_1.
(b) The pair hashed to a frequent bucket on pass 1.
(c) The pair also was hashed to a frequent bucket on pass 2.

Figure 3 suggests the use of memory. This scheme could be extended to more passes, but there is a limit, because eventually the memory becomes full of bitmaps, and we can't count any candidates.

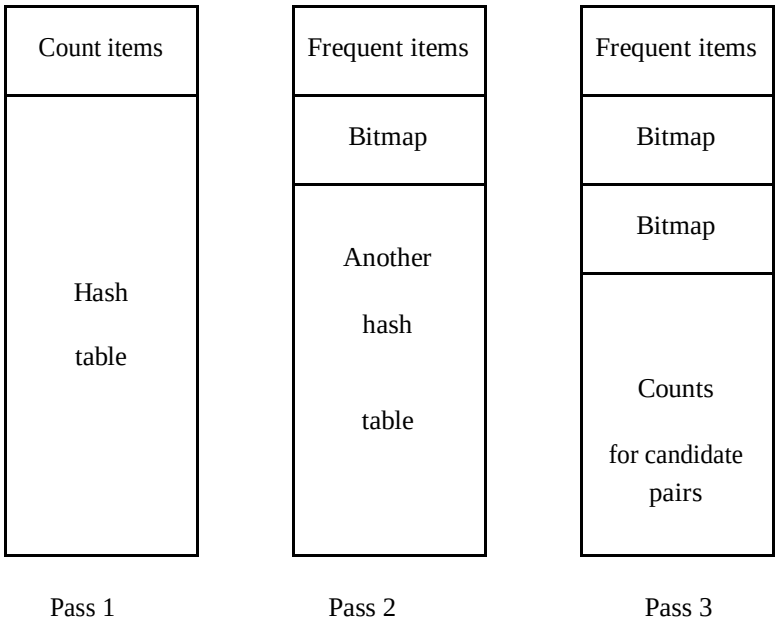

Figure 3: Multistage hash tables memory utilization

- When does multiple hash tables help? When most buckets on the first pass of PCY have counts way below the threshold s. Then, we can double the counts in buckets and still have most buckets below threshold.
- When does multistage help? When the number of frequent buckets on the first pass is high (e.g., 50%), but not all

buckets. Then, a second hashing with some of the pairs ignored may reduce the number of frequent buckets signi3cantly.

All Frequent Itemsets in Two Passes:
The methods above are best when you only want frequent pairs, a common case. If we want all maximal frequent itemsets, including large sets, too many passes may be needed. There are several approaches to getting all frequent itemsets in two passes or less. They each rely on randomness of data in some way.

1. Simple approach: Taka a main-memory-sized sample of the data. Run a levelwise algorithm in main memory (so you don't have to pay for disk I/O), and hope that the sample will give you the truly frequent sets.

- Note that you must scale the threshold s back; e.g., if your sample is 1% of the data, use s=100 as your support threshold.
- You can make a complete pass through the data to verify that the frequent itemsets of the sample are truly frequent, but you will miss a set that is frequent in the whole data but not in the sample.
- To minimize false negatives, you can lower the threshold a bit in the sample, thus finding more candidates for the full pass through the data. Risk: you will have too many candidates to 3t in main memory.

4. SON95 (Savasere, Omiecinski, and Navathe from 1995 VLDB; referenced by ToivonenÑ. Read subsets of the data into main memory, and apply the “simple approach" to discover candidate sets. Every basket is part of one such main-memory subset. On the second pass, a set is a candidate if it was identi3ed as a candidate in any one or more of the subsets.

- Key point: A set cannot be frequent in the entire data unless it is frequent in at least one subset.

5. Toivonen's Algorithm

(a) Take a sample that fits in main memory. Run the simple approach on this data, but a threshold lowered so that we are unlikely to miss any truly frequent item sets (e.g. if simple is 1% use s/125 as the support threshold).
(b) Add to the candidate of the sample the negative border, those sets of items S such that, S is not identified as frequent in the sample, but every immediate subset of S is. For example, if ABCD is not frequent in the sample, but all of ABC, ABD, ACD, and BCD are frequent in the simple, than ABCD is in the negative border.
(c) Make a pass over the data, counting all the candidate itemsets and the negative border. If no member of the negative border is frequent in full data, then the frequent itemsets are exactly those candidates that are above threshold.
(d) Unfortunately, if there is a member of the negative border that turns out to be frequent, then we don’t know whether some of its supersets are also frequent, so the whole process needs to be repeated(or we accept what we have and don’t worry about a few false negatives)

6. Conclusion

we presented here the various Association Rule Mining algorithms and compared their performance. The performance of these algorithms vary with the problem size. The best feature of two algorithm Apriori and AprioriTid is combined into a hybrid algorithm called AprioriHybrid . This new hybrid can be used in large database because Scale-up experiments showed that AprioriHybrid scales linearly with the number of transactions. In addition, the executions time decreases a little as the number of item increases in database. As the average transaction size increases, the execution time increases only gradually. Therefore AprioriHybrid is optimal solution for large database. Since we are dealing with Customer Relation Management and it consist of very large database usually, the use of this algorithm for Association Rule Mining will improve the efficiency of CRM software.

References

[1.] Alex Berson and Stephen J. Smith,” Data Warehousing, Data Mining & OLAP”, Tata McGraw-Hill Education 2004.
[2.] J.T. Tou and R.C. Gonzalez, Pattern Recognition Principals. London: Addison-Wesley,1974
[3.] James A. Freeman and David M. Skapura, “Neural Network: Algorithms, Applications, and Programming Techniques”, Pearson Education, 2001
[4.] Margret H.Dunham,” Data Mining: Introductory and Advanced Topics”, pp 89-138 Pearson Education, 2005.
[5.] R. Agrawal, T. Imienlniski, and A.Swami, “ Database mining: A performance perspective,” IEEE Trans. on Knowledge and Data Engineering, Vol 5, pp 914-925, 1993.
[6.] R. Agrawal , C Falotsos, and A. Swami. Efficient similarity search in sequence database. In Proc. of the Forth International Conferences on Foundation of Data Organization and Algorithm, Chicago, October 1993.

[7.] R.Agrawal, T. Imienlniski, and A.Swami, " Mining Association Rules between set of items in Large Database". Proc. 1993 ACM-SIGMOD Int. Conf. of Management of Data, pp 207-216, May 1993.

[8.] R. Agrawal and R. Srikant, Fast algorithm for mining association rule in large database. Research Report RJ 9839, IBM Almaden Research Center, San Jose, California, June 1994.

[9.] R. Agrawal, S. Ghosh, T Imielinski, B. Iyer and A. Swami. An interval classifier for database mining application. In Proc. of the VLDB Conference, Vancouver, British Colombia, Canada, 1992.

[10.] S. Muggleton and C. Feng. Efficient induction of logic programs In S. Muggleton, editor, Inductive Logic Programming. Academic Press, 1992.

[11.] Susmita Mitra and Tinku Acharya, " Data Mining: Multimedia, Soft Computing, and Bioinformatics, John Wiley & Sons, Inc., Hoboken, New Jersey.

Author:

Ajit Pratap Singh Gautam B.S. and M.S. degree from National Technical University, Kharkov, Ukraine in year 2000. He is presently working as head, Department of Computer Application, T.E.R.I., P.G. College, Ghazipur, U.P.

Dr. Natalia Valeriyvna Sharnova, working as head, Department of Intelligence Computer Systems in National Technical University, Kharkov, Ukraine. Her research area is Automated Natural Language Processing, Machine Translation, Documented Information Systems, Mathematical Fundamentals of Linguistics.