Algorithms in Data Mining

Distributed Association Rule Learning for Multi-Core Computing

---

Submitted by:

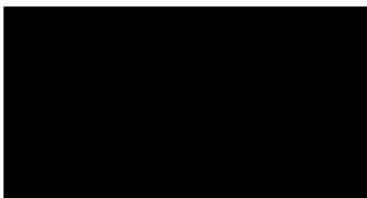

Table of Contents

1. Introduction 3

2. Project Specification 4

2.1. Project Description 4

2.2. Project Purpose 4

3. Association Rule Learning 5

3.1. Problem Definition 5

3.2. Algorithm Description 6

3.2.1. Apriori Algorithm 6

3.2.2. Related Works 8

3.2.3. Modified Algorithm 9

3.3. Analysis 11

3.4. Conclusions 13

3.5. Final Remarks 13

4. References 14

1. Introduction

Machine learning is a branch of artificial intelligence that deals with designing algorithms that allow computers to adapt the way they behave based on input given to them over time. Such machines can use data and try to deduce characteristics from it regarding the data set. The “rules” inferred from the data will often depend on heuristics of the algorithm, and different parameters to the algorithms may vary the results. The largest focus on research in this field is automatically recognizing patterns and making decisions based solely on the data. Hence, the learner algorithm must know how to generalize all of its’ given inputs in order to produce output useful for such purposes.

This field of research is vastly used in the field of data mining, in which algorithms are designed in order to discover patterns from large data sets. These patterns are then used for extracting of interesting unknown knowledge or for reaching useful conclusions.

Association rule learning is a well researched topic in data mining; it is a method for extracting useful relations between variables that appear numerously in a large data set. A great example for this would be finding correlations between purchased products by looking at transaction data stored by supermarkets. For example, such a rule could be deduced: {buns, lettuce, ketchup} → {hamburger}. This rule would imply that if a costumer were to buy buns, lettuce and ketchup he would be likely to buy hamburgers. It is simple to see that such rules can be very useful as the basis for making marketing decisions. The supermarket manager could decide to place the above items close by, or to make special promotional prices. There are many more interesting examples where association rule learning is used, but for the purpose of this project we will focus on this example.

2. Project Specification

2.1. Project Description

In this project we will look at the problem of efficiently learning association rules by looking at a well known algorithm (Apriori[1]) and modifying it. There has already been a large amount of research in the field of association rule learning, and many papers have been published on improvements to this well known algorithm.

We will first understand the classic Apriori algorithm and inspect its' faults. Upon understanding where Apriori falls short, we will survey relevant works on improving it. We shall then represent our new algorithm and analyze what it grants us over other algorithms, and what is the tradeoff.

The last stage will be reaching relevant conclusions on the usefulness the modification we made to Apriori.

2.2. Project Purpose

As mentioned in the introduction, association rule learning in our given scenario can be extremely useful as the basis for making marketing decisions. These decisions may very well be profitable in the sales sector.

The problem with many known algorithms is that they fall short in terms of running speed. If one were to use such algorithms to provide a service to numerous clients he could be limited by the running time of his chosen algorithm. We aim to provide an algorithm that is both accurate in the patterns that it finds, and also runs with low time complexity in real world situations. We intend for it to be able to be run on a single desktop computer, and not force the use of setting up a network of computers. This will mean a low setup and maintenance cost, and will grant us a more reliable solution. Such an algorithm could be very useful in the software-as-a-service industry.

A problem with many previous works on this topic is that they are rather theoretical, complex to implement, and have a large space and time overhead in real life situations. We attempt to provide a simpler and more reliable solution.

3. Association Rule Learning

3.1. Problem Definition

Our input is a data set of transaction data stored by supermarkets. The data set is potentially very large – it may contain hundreds of thousands of transactions stored over time. Our aim is to extract useful relations from this data. To be more precise, we would like to deduce rules of the sort {buns, lettuce, ketchup} → {hamburger}, which would imply that if a costumer were to buy the item set of products {buns, lettuce, ketchup} he would be likely to buy the item set {hamburger}.

We cannot expect our algorithm to return these rules with 100% certainty – this simply would not be the case in real life situations. Instead we will introduce some concepts for analyzing the usefulness and certainty of rules.

- *Support:* supp(X) of an item set X is defined as the proportion of the transactions that the item set X appears in out of all the transactions. The more support an item set has, the more we can feel secure about using it to deduce rules.
- *Confidence:* conf(X→Y) of a rules is defined as supp(X∪Y)/supp(X). It is the probably of finding Y in a transaction if X is in it. The higher it is, the more accurate the rule we found is.

Even though going over all possible options for rules and analyzing them will give us the best chances for finding all possible patterns, in practice we will not be able to go over them all. Also, many item sets are not worth considering simply because their support is too low – the item set does not appear often enough for any real conclusions. In the Apriori algorithm (that we shall soon introduce) these concepts will be used in order to find a smaller set of candidate item sets that need to be examined.

3.2. Algorithm Description
3.2.1. Apriori Algorithm

The Apriori[1] algorithm requires us to define a *minimum support* and *minimum confidence*. The algorithm generates association rules in two separate steps:

1. Find frequent item sets in the data set that satisfy minimum support.
2. Use these item sets to form rules. Only keep those with at least minimum confidence.

Looking at the first step – searching all possible item sets is difficult, since that would mean going over the power set of the items (which grows exponentially as the number of items grows). The Apriori algorithm manages efficient search by using the *downward-closure property* of support. This property guarantees that if an item set is frequent (i.e. has at least minimum support) then all of its' subsets are also frequent. This is straight forward. We conclude that if an item set is infrequent then all of its' super-sets must also be infrequent. Apriori uses a *bottom-up* approach in which we attempt to extend a frequent item set one item at a time in order to find larger frequent item sets. We use this property in order to see if the larger superset should be considered for discovery of rules, or if they it is irrelevant (does not meet minimum support demands).

We will now examine the pseudo code for each of the two stages of the algorithm.

Find frequent item sets in the data set that satisfy minimum support:
C_k represents candidate item sets of size k that the algorithm will consider at the k'th iteration
L_k represents frequent item sets of size k that the algorithm found at the k'th iteration from C_k

1. L_1 = {frequent items}
2. k = 1
3. while L_k is not an empty set
    a. C_{k+1} = candidates generated from L_k (see explanation below)
    b. for each transaction *t* in the data set
        i. increment the count of all candidates in C_{k+1} that are contained in *t*
        ii. L_{k+1} = candidates in C_{k+1} that satisfy minimum support
    c. k++
4. return $\cup_k L_k$

There are some parts of the frequent item set finding algorithm that need elaborating. Step 1 is simply done by going over all transactions, counting, and checking if minimum support is achieved. C_k is generated by using the *join* operation on C_{k-1} with itself, followed by a *pruning* step. Pruning is done by looking at all subsets of C_k and checking that all of them are frequent item sets found by the algorithm in the first k-1 steps. If one of them is not – the downward closure property implies that C_k should be pruned. L_k is calculated by going over all transactions, counting the appearances of each item set from C_k, and keeping only those for which the minimum support demand is met.

Form rules that satisfy minimum confidence:

1. for each frequent item set *l*
   a. generate all non-empty subsets of *l*
2. for each non-empty subset *s* of *l*
   a. if conf(*s* → *l**s*) = supp(*l*) / supp(s) satisfies minimum confidence
      i. form the rule *s* → *l**s*

We can see that once we found all frequent item sets, forming rules is simple. It is a much less intense operation.

We have now been introduced to the Apriori algorithm. We can see that it is rather simple and straight forward. The algorithm does attempt to lower running time by using the downward closure property for pruning irrelevant item sets and lowering the number or item sets that need to be checked. Alas, the running time is still very high. The biggest bottle neck for running time in the Apriori algorithm is the counting step. Going over all the transactions for every iteration is very exhaustive. This is the biggest pit fall of the algorithm.

3.2.2. Related Works

In the past two decades many attempts have been made to lower the Apriori running time. Work has been made in the following directions:

- Using sampling on the transaction data set
- Reducing the number of passes over the data set
- Parallelization of the computation (distributed computing)
- Partitioning the data set
- Reducing irrelevant transactions
- Adding extra constraints on the patterns

We will be focusing on two of the most researched directions – sampling and distributed computing.

Using sampling was first proposed in the mid 90's [2] [3]. It is a rather straight forward approach in which we use only a small part of the data set to find rules. It is not hard to see that sampling will reduce running time by a large factor (especially if the sample size fully fits into the main memory), but will come at the cost of the accuracy of results. Hannu Toivonen put forth research [3] on the size of the sample needed to reach a desired accuracy in results. The major problem is that the size of the sample is data dependant. This spawned research on progressive sampling algorithms [4] [5] [6] that change the sample size in each iteration according to progress made in previous iterations.

Distributed computing is a fast growing branch of research in computer science with very practical uses. Most association rule learning algorithms that use distributed computing at the first stage of the algorithm do so by dividing the transaction data set between multiple nodes. The way these nodes interact with each other varies. One possibility is for each node to calculate its' own frequent subsets and communicate with all other nodes [7]. Another option is to distribute the counting and for all nodes to transmit their local counts to all other nodes so that global counts could be calculated by all nodes [8]. A variation of this is to transmit the local counts to one central node, which calculates the global counts and transmits the frequent item sets back to all the nodes [7].

In our modified algorithm we will use some of the techniques described above with our own tweaks that better suit our purposes.

3.2.3. Modified Algorithm

In this chapter we will first have an overview of the concepts used in our algorithm, and then look at its' pseudo code. Let's notice that our modification is intended for tweaking the first part of the Apriori algorithm. The second part does not demand many computing resources and will be left unchanged.

Our algorithm uses the distributed approach – we would like to have a number of threads working together to help us with CPU intense operations. We would like our algorithm to be able to run on one multi-core computer. Instead of using the approach that most distributed algorithms focus on – splitting *all our data set* between nodes – we would like to only use a ***sample*** of the data set that fits into our main memory, and split that for all threads to work on. Using our main memory for all CPU's is called the ***shared-memory*** approach. It has many advantages we will discuss later.

We want each core to be responsible for a part of the data set, and so each core will have a thread running on it. A major point that we would like to improve on with relation to other distributed algorithms is the ***communications*** between the different threads. Primitive algorithms use a protocol that sends messages from each thread to all others. More effective algorithms send all messages to one thread, which sends replies back to them. We would like to use a similar concept, only simpler – the ***master-slaves approach***. We will have one *master-thread* responsible for finding the association rules, but we allow it to use all other *slave-threads* when needed. The master thread will run a variant of Apriori on it. When it needs to count item sets in transactions it will ask all other threads to calculate their own local counts, and relay them back (the master itself will also count part of the data set). The master thread will add these together to calculate global counts. It will now be able to continue with the first stage of the Apriori algorithm, until it will needs the slave threads again for its' next iteration. The second part of the algorithm will be run by the master alone.

For that sake of formality and clarity, we will now see the pseudo code for the master:

1. Calculate the percentage of transactions data that can fit into the main memory and choose a sample of the data that is 1-ϕ of this size
2. Send each slave what part of the transaction data set it is responsible of (by dividing the sample of the data set equally between all threads, including itself)
3. Ask slaves to count their local frequencies for single items and transmit back
4. The master counts local frequencies for single items in its' own partition of the data set
5. Sum up global frequencies for single items – L_1 = {frequent items}
6. k = 1
7. While L_k is not an empty set
    a. C_{k+1} = candidates generated from L_k. Store in common memory that all threads can access.
    b. Ask slaves to count their local frequencies for item sets in C_{k+1} and transmit back
    c. The master counts local frequencies for item sets in C_{k+1} in its' own partition of the data set
    d. Sum up global frequencies for candidates – L_{k+1} = candidates in C_{k+1}, whose global counts satisfy minimum support
    e. k++
8. return $\cup_k L_k$

Steps 1 and 2 can even be ran in O(1) – it depends on the method used for choosing a sample (we will not discuss this here). The second part of the algorithm is run only by the master, and is unchanged from the original Apriori.

The algorithm itself is simple to understand and to implement. In the next chapter we will analyze its' usefulness and analyze what we sacrificed for efficiency.

3.3. Analysis

We will first independently analyze sampling, distribution, using one multi-core computer, memory sharing, and thread communications. We will then look at how these parts come together. The analysis will come in the form of discussion, and not by using rigorous mathematical proofs.

Using sampling when searching for association rules has been vastly researched and many papers have been published on the topic. It is simple to see that sampling will reduce running time by a large factor. It will give us at least a linear improvement, and if we consider that finding frequent item sets (the counting stage) may take up to exponential time in relation to the data set size – the improvement is potentially very big. Since we are only looking at a part of the data base, sampling will come at the cost of accuracy. Hannu Toivonen showed that relatively small samples help us reach a desired accuracy in results [3] [4]. He showed both theoretically and empirically that the sampling method is very precise, with a low chance of failure. This is very attractive for us, and the great improvement in running time is worth a negligibly small chance of failure.

Our algorithm runs on a shared memory architecture, which has many desirable properties. Each processor has direct and equal access to all the system's memory, which means that parallel programs are easy to implement on such a system. Since each processor is responsible only for a part of the data set, we will expect each core (which has its' own cache) to have good data locality. This means a better running time. The big downside of using such architecture is that we are limited by the amount of memory we have available. This takes us back to sampling – working with a smaller data set data set that fits fully in the main memory means no page faults. This gives us another major improvement.

In our algorithm we used the distributed approach. In most computers sold today, a quad core processor is standard. By having threads running on all cores to help with counting (the most CPU intense part of the algorithm) we can give a boost that is linear in the number of cores. The small overhead of sending messages between the threads will be looked at soon, but it is negligible compared to the gain in run time that a multi-core environment gives us.

As mentioned earlier – by using a multi-core environment in which we have many threads communicating with each other, we get unwanted overhead. In previous works, this has often been overlooked; we often see algorithms in which all nodes send messages to all other nodes. Such methods are both difficult to implement, and have a large overhead. We used a much simpler approach where a master thread asks all other threads to give local counts for candidate sets stored in common memory, and they reply back. This is both simple to implement and has a low overhead compared with competing algorithms. Not having to send candidate sets to all slave threads is another improvement – after all, they simple need to access this memory for reading.

Overall, we built an algorithm with components that all work well on their own, but also fit together very well. We decided to parallelize heavy operations on a multi-core computer and use the advantages of shared memory. Being limited by main memory size led us to use sampling, which greatly decreased running time both because it made us work with a smaller data set, and because it reduced page faults and allowed the algorithm to access only the RAM. We managed to simplify the communication protocol between our threads to the bare minimum and hence our algorithm is both simple to implement and has a relatively low overhead.

Our analysis has led us to conclude that our modified algorithm for finding association rules is excellent for the purpose for which it was intended.

3.4. Conclusions

Let's recall our demands – we wanted an algorithm that is fast and simple to implement, which means that it is reliable. We wanted it to be able to run on an inexpensive and simple infrastructure - one multi-core computer. Lastly, we wanted our algorithm to work well in a real life environment, and not only for theoretical purposes. We will now review how we stood up to these criteria.

We first consider the implementation of this algorithm. The original unmodified Apriori algorithm is very simple to implement. Adding a sampling phase at the start does not add any complications. Making an algorithm run in a distributed environment usually adds complexity – but in our case this is rather minimal. Implementing a master thread that stores candidate item sets in a common memory area, sends messages to all other threads and receives their replies is rather simple to implement – this is all thanks to the simple communication protocol. A simple algorithm means fewer chances for bugs in the code – which means a reliable algorithm.

This algorithm was designed from the start to run a multi-core computer with shared memory and it meets these demands. This means that it can be run without great setup costs and without the need to deal with maintaining a network of computers just for this purpose. Using sampling and taking advantage of all the cores of the computer gave us the fast algorithm we wanted.

What we have in hand is an algorithm that is practical in the real world, and useful for providing services to sales departments for many different market sectors. Because of its' low time complexity, it can be used to serve numerous clients. This algorithm can be of great value in the software-as-a-service industry.

3.5. Final Remarks

As an end note – on paper, this algorithm does seem to meet all the criteria. We discussed how this algorithm would run in theory, but there is no substitute to doing real experiments and comparing empirical data. For future work, this algorithm should be run on real data sets and its' results compared to other distributed algorithms that can be made to run in a multi-core environment.

4. References

[1] Rakesh Agrawal and Ramakrishnan Srikant
Fast algorithms for mining association rules in large databases

[2] Heikki Mannila, Hannu Toivonen, A. Inkeri Verkamo
Efficient Algorithms Association for Discovering Rules

[3] Hannu Toivonen
Sampling Large Databases for Association Rules

[4] F. Provost, D. Jensen, and T. Oates
Efficient progressive sampling

[5] S. Parthasarathy
Efficient progressive sampling for association rules

[6] Kun-Ta Chuang, Ming-Syan Chen, and Wen-Chieh Yang
Progressive Sampling for Association Rules Based on Sampling Error Estimation

[7] Cheung, DWL; Han, J; Ng, VT; Fu, AW; Fu, Y
A fast distributed algorithm for mining association rules

[8] M. A. Mottalib, Kazi Shamsul Arefin, Mohammad Majharul Islam, Md. Arif Rahman, and Sabbeer Ahmed Abeer
Performance Analysis of Distributed Association Rule Mining with Apriori Algorithm