Big Data Project Report
Professor Mauro Castelli

A-Priori Implementation in MapReduce

By:
Bernardo Galvão - m2015041
Dário dos Santos - m2015029
Diogo Reis - m2015395
José Machado - m2015361

Nova Information Management School
Spring Semester 2015 / 2016

The goal of this project is to implement the A-Priori counting algorithm to be able to efficiently compute Association Rules in a basket dataset. As one usually finds out when implementing anything in code, we faced new challenges that we would like to point out:

- How to efficiently generate candidate itemsets from the bottom-up, and reliably?
- How to compute the Association Rules with the implementation of Anti-Monotone property?

These will be addressed along the following pages. Below is a macro picture of our MapReduce design, composed by three Map-Reducers hereby named after the packages contained in our Java src.

1. baseCreator
2. frequentItemsetsFinder
3. ruleExtractor

Furthermore, there are two extra user parameters: booleans MONO and ANTI_MONO. When set to true, they make the program use our mappers and reducers that respect the properties they are named after - and thus the mappers and reducers of our final solution. They are built upon the mappers and reducers that do not follow the properties - which are accessed when the booleans are set to false - but provided the basis for the final mappers and reducers to be built. We left it in the code as we wanted to easily make performance comparisons if necessary or just out of curiosity.

---

baseCreator

We figured that the best way to carry information throughout the Map-Reducers was by the lines, as each line represents a basket that can be coded as an integer - a basket-key. Thus the baseCreator.map( ) collects as value the line number as it scans the document and outputs as key any item that was included in that line. The baseCreator.reduce( ) then proceeds to gather all of the basket-keys belonging to a single key (i.e. item) and only allows those items with a sufficient number of basket-keys (i.e. above minimum support) to be written in the context.

It is also at this stage that one gathers and stores information necessary for the iterative Map-Reducer that follows - and they are all related to our solution for building itemsets:

- TreeMap indexToSinglet: a sorted map with the singlets corresponding to an index key
- TreeMap singletToIndex: another sorted map with the index key corresponding to the singlets that are supported in the dataset. The reverse of the one made above for convenience.
- HashMap singletIndexToLines: for a key (index representing a supported singlet), it stores the sorted set of the basket-keys, containing therefore information about which baskets a singlet is included in.

frequentItemsetsFinder

In short, this Map-Reducer follows the exact same fashion of the baseCreator, with the difference being the input file being in the convenient format of <Single Item, Basket-Keys>. The following iterations will output <Itemset, Basket-Keys> and increase the size of the itemset until no itemset has a sufficient number of basket-keys - technically, until no line is written in the Context.

Here lies the core of our implementation of the A-Priori algorithm. It works under the requirement of a sorted set of items - which is why we keep the TreeMaps right at the beginning. Before describing the algorithm, let us define a few terms - that follow the naming choice in our code.

- Current-itemset: the key the mapper is reading.
- Current-lines: the basket-keys of the key the mapper is reading.
- Last-item: the last item of the current-itemset - not only due to appearing on the right-hand-side of the current-itemset, but also and strictly because it takes the last position of a sorting order.
- Next-to-last: the name is self-explanatory, but note that this item is not in the current-itemset.

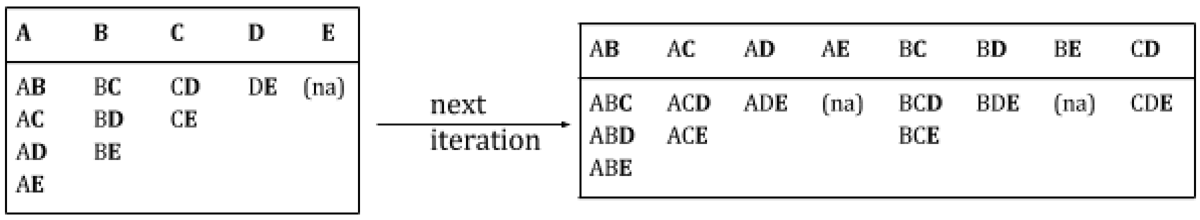
Let us consider that the Current-itemset is {A[1]}. The Last-item is A and the Next-to-last is B. Remind that we stored information about B in singletIndexToLines at the baseCreator stage. To construct the itemset of size 2 {AB} and filter it correctly, we map any basket-key that A and B have in common and then in the reducer we collect all of the basket-keys for the key {AB} - if this itemset has a sufficient number of basket-keys, then it is supported and written to the context.

The next Next-to-last is C - outputting the key {AC}. This itemset building process goes on until we reached the last supported single-item standing “after” the Last-item. So let us say that there are items A through E that are supported - refer to Fig.1 - and that we are at the first iteration of this Map-Reducer, meaning we are building pairs from single items[2]. All the pairs that we are building and considering are in the lower part of the table on the left. In the next iteration, not all pairs are supported and the candidate triples are built by the same mechanism.

{AE} and {BE} have no candidate triple generated from them because there is no supported item-letter coming after the Last-item letter. At first thought, one may think “What about building triples from {AE} with each of the other letters B, C and D?”. The answer is: that was already handled by the previous columns. The simplest of examples is found in the first table for letter E. Note how this allows for not considering candidate itemsets that were previously built by previous current-itemsets (columns).

[1] The sorting order of an item follows the alphabet.

[2] These single items are given by the first file generated by the baseCreator.



A	B	C	D	E
AB	BC	CD	DE	(na)
AC	BD	CE		
AD	BE			
AE				

next
iteration

AB	AC	AD	AE	BC	BD	BE	CD
ABC	ACD	ADE	(na)	BCD	BDE	(na)	CDE
ABD	ACE			BCE			
ABE							

Fig. 1 - Iterative process of building candidate itemsets. On the first line, the last-items (i.e. the last letters) are highlighted in bold. On the lines below, only the letters coming next to the last-items are highlighted. The first line of a table is the current-itemset the mapper is reading. On the lower section of each table, what we are doing is appending a frequent single item, which sorting order is after that of the last-item, to the current-itemset, resulting in a candidate itemset.

It does however have a defect. In the example above, {CE} did not make the cut. This algorithm generates a candidate itemset {CDE}, where {CE} is not a frequent subset. The Map-Reducer we designed will count to check if an impossible itemset makes the cut.

Order of the items does not matter in the definition of a candidate itemset - but when it comes to analysing our implementation, it does. For example, let us consider that {ABE} turned out to be a frequent itemset and that {ABC} did not turn out to be a frequent itemset. If we want to build a quadruplet from {ABE}, according to our mechanism we cannot add C to it, because:

- It would be the job of {ABC} to create the quadruplet by appending E, following the next-to-last rule - or simply put, following alphabetical order - resulting in {ABCE}.
- But if {ABC} was not a frequent itemset then {ABCE} could never be either - no itemsets would be built from {ABC} as a result, since this itemset would not even be present at the next iteration. This rule thus obeys "a little" to the monotone property (on the basis of sorting order only): it never considers such non-frequent candidate itemsets because it would forbid building {ABEC} (due to non-alphabetical order), even before counting the basket-keys such an itemset appears in.

The last bullet point describes a situation that compensates for the previously mentioned defect. Moreover, it adds to the efficiency, however is yet an "incomplete" monotone property. To effectively implement this property, a global blacklist - declared in the AssociationRules class - stores all itemsets that were found not to be frequent in the MONO_ItemsetsReducer - where the count is evaluated against the specified minimum support - and checks in the MONO_ItemsetsMapper if there is any subset of a candidate set that is blacklisted - if so, the mapper does not allow the candidate set to be sent to the reducer. The referred mapper and reducer are activated when MONO = true. This addresses the first challenge we pointed out.

One note that ought to be made is that the basket keys - integer numbers - corresponding to an itemset are sorted - for instance, check one of the "itemsetSize" files. We planned since the beginning to take advantage of sorting order. The first attempt to do this was by applying binary search[3], which was not efficient as there are lots of operations that have to be performed. It was even worse than our first basic mapper. So we took advantage of the sorted lines in a different way: as soon as it found an intersecting line, the loop would break and start from the last intersecting line - as opposed to starting the loop all over again.

[3] As proof that we implemented it, we included the DISCARDED_ItemsetsMapperBinarySearch.java in the frequentItemsetsFinder package. This mapper is not our final choice by any means.

ruleExtractor

The mapper collects the counts. The reducer collects the itemsets (as keys) and respective counts to a HashMap - itemsetsAndCounts - that will be used in the cleanup, which in its turn, computes the association rules.

The baseline strategy (i.e. when ANTI_MONO = false) is to iterate once through frequent itemsets, build the powerset (if applicable) and, iterate the subsets of the itemset, lookup the subset in itemsetsAndCounts and compute the confidence. The problem is with the last loop: it iterates through the subsets in a way that puts as many items on the right-hand side (RHS) of a rule first, hence going completely against the anti-monotone property.

To implement a smarter reducer-cleanup - accessed when ANTI_MONO = true - we adapted our baseline strategy to iterate through the subsets in a way that puts as few items on the RHS first. To this end, we collect the set of “RHS itemsets“ into a SortedMap, where the key is the size of the RHS-itemsets. The next natural step is to iterate through the keySet( ) of this map and iterate through the RHS-itemsets that have size = key.

Similarly to the process of finding frequent itemsets, we keep a Set blacklistedRHS, comprised of RHS-itemsets that do not satisfy minimum confidence. When iterating through RHS-itemsets and finding one that produces a low confidence rule (below minimum confidence) this RHS-itemset will be stored in the in blacklistedRHS. At the next size of RHS-itemsets, it will check if each new RHS-itemset has a subset present in this blacklist. This way, we granted respect to the anti-monotone property. This addresses our approach to the second challenge we pointed out.