Implementation and evaluation of market basket analysis techniques

Giacomo Turati

June 15, 2021

Course:	Algorithms for Massive Datasets
Professor:	Dario Malchiodi
Course of Study:	Master Degree in Computer Science
Institute:	Università degli Studi di Milano

Disclaimer: *I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.*

Abstract

This report is aimed to analyze the implementation from scratch of four algorithms that solve the problem of finding frequent itemsets occuring in a sequence of transactions (collections of items, called *baskets* in literature(1)) according to a *support threshold*, that is the minimum number of baskets in which the itemset must occur to be declared frequent. Algorithms' performances are compared in terms of execution time, scalability over a growing number of transactions and memory requirements. The implementation was entirely written in Python in the form of a Jupyter notebook and results were validated against the *Apriori algorithm* implementation found in the *apyori*, library released under the MIT License.

1 Dataset

The chosen dataset is called *"Old newspapers"* and is a collection of more than 6 GB of old newspaper's articles written in various languages, made available on *Kaggle*. In the context of market-basket analysis, articles are baskets and words that they contain are items, so that the goal of the analysis is to find sets of words occuring frequently together in articles.

1.1 Preprocessing

Although data could be used as they are in principle, some basic preprocessing is required to have meaningful results. The following list resumes the main preprocessing aspects that were taken into account:

a. Only articles written in the same language, or in languages which have many terms in common, should be considered during the analysis, as long as if two languages are completely disjoint, words occuring in one language won't occur in the other.

b. Numbers and all symbols but ASCII alphabet's character should be removed because they're likely to be very rare and dispread, thus irrelevant.

c. Since articles written in natural language are supposed to have a correct syntax and a sensible semantics, words (items) haven't equal probability to be part of a document (basket); in particular, stopwords like articles, prepositions etc... happen to be naturally frequent in well-structured sentences although they don't point out any specific semantics. This fact may result in the need of setting a very high support threshold to effectively rely on the assumption that frequent itemsets made by more than three elements are so unlikely, that they can be ignored in complexity evaluations.

For all these reasons, functions that build a dataset with the previously described characteristics were coded and the resulting basket file is a simple plain text file in which every line is a basket and every line is a sequence of words separated by commas. The *nltk* Python library was used to get dictionaries of stopwords available in most languages.
At last, another important aspect that should be considered when treating this kind of dataset, is the number of items per basket. Indeed, in the reference literature(1) it is clearly stated that the number of pass over the basket file, which is read from disk, is the bottleneck of the analysis and so the complexity of proposed algorithms is measured in pass over the entire dataset. Although it is reasonable in theory, if there are many items per basket the computational time required to generate all combinations of k elements in baskets in a combinatoric fashion, could kill the I/O operations cost and so test txt datasets can be created with a limited number of items per basket for the sake of the analysis.

2 Algorithms implementation

Four algorithms that solve the frequent itemsets problem were implemented:

a. A-priori algorithm: exploits the monotonicity property to solve the problem without keeping in memory all possible itemsets counters.

b. *PCY* algorithm: A-priori variant whose goal is to further reduce the number of candidates couples to be considered during the second basket file pass, which are likely to dominate in complexity over the count of itemsets of different size.

c. SON Algorithm: exploits the ability of counting independently frequent itemsets in chunks of the basket file by adapting the support threshold. It is naturally designed for a distributed computation in a map-reduce framework.

d. Toivonen algorithm: tries to solve the problem with only one pass over the entire basket file plus a run of market-basket analysis over a sample of the basket file.

In the following subsections we're gonna look at the relevant implementation's details.

2.1 Apriori algorithm

As previously said, this algorithm rely on the monocity property of an itemset's support: if we call

$$s(I) = \#\text{ of baskets in which I occurs}$$

the *support* of the itemset I and s_t the support threshold, then it's easy to see that

$$\forall J | J \neq \emptyset \wedge J \subset I$$

$$s(I) <= s(J) \implies \text{if } s(I) >= s_t \implies s(J) >= s_t$$

So the algorithm iterates over the dimension k of the itemsets till no more frequent itemsets are found, alternating two phases:

- *Build candidates set*: read baskets from disk and check for every possible k-itemsets extractable from a basket, provided that the empty set is always frequent, if all of its *immediate subsets* ($k-1$-itemsets) are frequent itemsets. By induction of the monotonicity property, this implies that all of its proper subsets are frequent too and so it's worth counting the itemset's occurence in baskets.

- *Filter candidates set*: check for every candidates set I if $s(I) > s_t$.

Although it's possible to write a generic code for finding frequent itemsets for every given k, the best solution in terms of efficiency that was found during the implementation's test is to write ad-hoc code for counting candidates singletons and couples and a general iteration for any other k-itemset. This was done because there's nothing to check while counting candidates singletons, as long as the empty set is always frequent, and filtering out frequent singletons from a basket before generating couples it's enough to have the monocity property checked for all of them, avoiding further checks while scanning the basket file.
Another little efficiency improvement implemented is to keep already discovered frequent itemsets separated for k, to allow a slightly faster check of the monotonicity property on smaller dictionaries useful for high values of k when there are many frequent itemsets.
The following list briefly resumes the main functions that implement the A-priori computation to guide the reader while looking at the code:

- `first_pass(basket_file)`:

  scan the basket file and build candidates singletons set by keeping a counter for every item in a dictionary.

- `second_pass(basket_file,freq_it_sets)`:

  build candidates couples set by removing every unfrequent item from each basket before generating couples. Keep the couples' counters in a dictionary.

- `get_ck(basket_file,freq_it_sets,k)`:

  build candidates k-itemsets set for every $k > 2$. Scan the basket file, filter unfrequent singletons from baskets, generate all possible k-itemsets and check if all their immediate subsets are frequent.

2.2 PCY

One of the main parameters to take into account in the market-basket analysis is the size of the candidates set to filter. PCY tries to reduce the number of candidates couples during the second pass exploiting the memory space left unused during the first pass. While counting candidates singletons, all possible couples are hashed to a table whose buckets are associated to counters and every time a couple is hashed to a specific bucket, increment by one its counter. Then the hash table is compressed to a bitmap in which every bit b is set if its counter $c(b) >= s_t$, indicating a frequent bucket. During the second pass a couple's counter is kept in memory if and only if the couple is formed by all frequent items and hashes to a frequent bucket.
The implementation was straightforwardly built upon the A-priori structure previously defined with some variations:

- `pcy_first_pass(basket_file,bm_size):`

  initialize a table of size bm_size with all zeroes and scan the basket file building candidates singletons set. Every time a basket is read, generate all possible couples, hash them to the table and increment the bucket's counter.

- `pcy_second_pass(basket_file,bm):`

  scan the basket file, remove unfrequent items from each basket, generate couples and hash them to the bitmap bm; if $bm[hash(couple)]$ is set, keep the counter else skip the couple.

Between the first and second pass the bitmap is built by checking which buckets of the table returned after the first pass exceed the support threshold. Depending on the parameter bm_size, there will be more or less collisions while hashing, resulting in a lower or bigger reduction of the number of candidates couples.

2.3 SON

As previously introduced, this algorithm is designed for a distributed computation over a map-reduce framework, so the implementation rely on the python API *pyspark* of the analytics engine *Apache Spark*. Explaining in details how this framework manages the computation is beyond the scope of this report, it's sufficient to say that algorithms must be thought as a sequnce of map steps, which take an *RDD* (i.e. a distributed collection) and compute the same function for every item in it, and reduce steps, which take the collection and reduce it to a single item. Input data are typically divided in chunks and computations are performed independently on the single chunks.
The SON algorithm divides the discovery of frequent itemsets into four main tasks:

- First map: perform a market-analysis, no matter how, independently on chunks whose size is a fraction p of the entire basket file, with an adapted threshold ps_t. This task was implemented by the function's call

  ```
  basket_file_rdd.mapPartitions(lambda chunk: apriori(chunk,int((1/num_p)*s))
  ```

  which computes the Apriori algorithm on every partition with a threshold rescaled by $1/num\_p$, where num_p is the number of partitions. It returns a list of couple (key,value) where key is an itemset in the form of a tuple and the value is the occurency counter of the itemset in the chunk.
  To complete the map task, we just call

  ```
  freq_in_chunks.map(lambda fis: (fis[0],1))
  ```

  to output a list of couples (itemset_frequent_in_chunk,1).

- First reduce: take in input the previously defined list and discard any duplicate. This task is implemented by calling

  ```
  first_map.reduceByKey(lambda el1,el2: 1)
  ```

  so that (key,value) couples with the same key are assigned to the same reducer, which outputs only one couple (key,1) for all of them.
  Now we have the list of all the itemsets found to be frequent in at least one chunk, which are the only one that will be considered next.

- Second map: scan independently every partition of the basket file and compute the occurence of any candidate itemset (i.e. itemset frequent in at least one chunk). Implemented by

  ```
  basket_file_rdd.mapPartitions(lambda chunk:count_occurence(chunk,candidates))
  ```

  returns a list of couples (itemset_freq_in_chunk,occurency_counter).

- Second reduce: sum all counters associated to the same itemset and filter out itemsets that has a support lower than the threshold. The following code implements this task, just for simplicity, into two API calls

  ```
  second_map.reduceByKey(lambda el1,el2: el1+el2).filter(lambda el: el[1]>=s)
  ```

  using the same concept of reduction by key explained before.

2.4 Toivonen

Like the SON algorithm, the goal of the Toivonen algorithm is to solve the frequent itemsets problem limiting the number of pass over the entire basket file. In particular, we can outline the structure of this algorithm into three main steps:

- Take a sample of the basket file which contains approximately a fraction p of the total number of baskets, save it in memory and perform market-basket analysis with an adapted threshold αps_t, where *alpha* is a scaling factor typically slightly lower than one (0.8-0.9). In the project code the sample is collected by scanning the entire dataset and selecting each basket with a probability p, thus generating numbers from a uniform distribution and checking if they're less than p.

- Build the *negative border* defined as the set of itemsets unfrequent in the sample but whose immediate subsets are frequent in it. Two main approaches were considered to compute this set: the first takes the set of all items, generates all the combinations of k items and checks their immediate subsets. The second inserts all non-frequent singletons in the negative border, then performs iteratively a self join of the set of $k-1$-itemsets frequent in the sample to build the set of k-itemsets that could

be in the negative border; then checks whether they're frequent in the sample or not. The latter was chosen under the assumption that the frequent itemsets found in the sample should be limited, if the threshold is well chosen, while the set of all items occuring in the basket file could be very big.

- Scan the entire dataset, count only itemsets in the candidates set, defined as the union of the set of itemsets frequent in the sample and the negative border, and filter out those whose support is below support threshold s_t.

The peculiarity of this algorithm is that it can't solve the uncertainty if some frequent itemsets found at the end are in the negative border and it has to rerun with a bigger sample or lower scaling factor α to grow the number of itemsets frequent in the sample.

3 Testing and experimental results

The whole code was tested with the *apyori* library as benchmark. Results were validated by checking that any algorithm implemented agreed with the library results in terms of frequent itemsets found and support computed as # of baskets in which itemset occurs divided by total number of baskets.
The threshold considered during the experiment was the 1% of basket file size, since it is a typical value proposed in literature (1).
Algorithms were compared in terms of time required to complete the whole task and size of the candidates set before any filtering operation.
The SON algorithm deserve a particular note: since the test was done in a local environment, it can't benefit of the parallelism given by the cluster computing system for which it's designed. Indeed, since the analysis was performed on a dual core machine, the only reasonable choice (that is the dafault choice) was to split the dataset in two and to use only two *workers* (i.e. parallel process) to compute frequent itemsets in chunks. This is the maximum degree of true parallelism that we can exploit in such an environment.

3.1 The experiment

The experiment consists in choosing all the english newspapers, which is likely to be the largest collection of documents written in the same language that we have within the whole set, building three txt datasets of different order of magnitude (10^3,10^6,10^6) and run the algorithms implemented togheter with *apyori*'s implementation of the Apriori algorithm, asking them to find all frequent itemsets up to couples, as suggested in literature(1). The biggest order of magnitude considered is 10^6 because we have approximately one milion english newspapers available and so we can't test a higher order of magnitude.
At last, we remove stopwords and limit to fifteen the maximum number of items per basket (which was seen to be reasonably near to the mean number) to speed up the analysis without loosing important information.

3.2 Results

The fastest algorithm in any case is the Apriori, which performs the whole computation over 10^6 baskets in approximately 20 seconds, against the 70 of PCY, the 60 of SON and the 80 of Toivonen, while the *apyori* implementation (benchmark) ends in 15 seconds, which is not very far from 20, especially considering that the MIT library require the basket file to be in memory. It's important to note that this timing results shouldn't be taken in absolute sense, because if we impose a limit on the number of items per basket, some items will be randomly discarded from baskets. Although this is true, it's sufficient to run the experiment many times to observe that these are reasonable approximation values and the order (by execution time) of the algorithms doesn't change.
Although maybe this test doesn't outline how algorithms would perform on a dataset of really extreme dimensions, most of the theoretic results that we expected are confirmed: first, given a sufficiently big number of buckets for PCY bucket table (in the experiment 40000 bit - 5KB), the size of the candidates couples set can be reduced over the 99%. Second, although the Toivonen algorithm is not the fastest, we can see that is approximately 115 times slower than Apriori on 10^3 baskets, but if we grow the size of basket file to 10^6 becomes only 4 times slower than it, letting us suppose that will get faster than Apriori if we take a basket file drastically bigger. Furthermore, should be noted that running Toivonen with a sample which is 0.4 of the entire dataset and a scaling factor of 0.8 it's enough to have the answer in most cases.
Concerning *association rules* we can only observe that "would" $\implies$ "said" with a confidence of 0.2, but this is due to the fact that only one frequent couple has been found. Unfortunately, "would" and "said" are very common terms in english language and so this doesn't reveal anything about the nature of the data collected. So another analysis with the Apriori algorithm is performed with the complete collection of english newspapers (without any limit on the number of items per basket), with a slightly lower threshold to see if we can have further insights on the dataset. We have now an interesting result: rules like "st" $\implies$ "louis", "new" $\implies$ "jersey" and "new" $\implies$ "york" let us suppose that we're dealing with american newspapers rather than british.

References

[1] Leskocev, Rajaraman, and Ullman (2010). *Mining of Massive Datasets.* Cambridge University Press.