Frequent Itemsets Extraction using PySpark

Adriano Meligrana

June 12, 2024

Abstract

This report describes a PySpark implementation of the APriori algorithm, useful to search for frequent itemsets occuring in a set of baskets. Its computational performance was empirically tested by applying it to the LinkedIn Jobs and Skills dataset. The results show that the proposed implementation is able to find all frequent itemsets in reasonable time on a commercial laptop, even without using any sampling technique to reduce the size of the dataset.

I declare that this material, which I now submit for assessment, is entirely my 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 work, and including any code produced using generative AI systems. I 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 or any other person for assessment on this or any other course of study.

1 Introduction

The search for frequent itemsets can be seen as the preliminary step for data mining applications such as market basket analysis. Through the extraction of frequent itemsets, namely, itemsets whose support is above a specific threshold, we already have enough information to compute the confidence and the interest of an association rule. Since the APriori algorithm is computationally expensive, having a worst case space and time complexity of $O(2^d)$ where d is the number of unique items in the dataset, it is a good candidate for a MapReduce implementation.

2 MapReduce Implementation of the Apriori Algorithm

The Apriori algorithm exploit a crucial property when searching frequent sets in a list of records: monotonicity. This property says that a set of items can't be frequent if any of its subsets is not frequent. Intuitevely, this means that searching with a bottom-up approach from smaller subsets to bigger ones should improve the performance of checking if a set is frequent because many can be removed because some of its subsets are not frequent and so it can't be frequent too. Indeed, the Apriori algorithm at each step k interlaces two passes:

- The first pass searches for the k-set frequent itemsets L_k;
- The second pass constructs the candidate set of itemsets C_{k+1} for the next search constructed from the combinations of items of $k+1$ elements in L_k.

In this way the number of combinations to check when k gets larger should decrease steeply and be much less than the $C(d, k)$ sets one would need to inspect if a brute-force approach was used instead.

For the algorithm developed using PySpark we needed to adapt the approach to make it more amenable to a MapReduce implementation:

- First, we compute L_k with a mapping function which, for each basket in the RDD (Resilient Distributed Datasets), computes key-value pairs having as key each combination of size k of items and a value of 1. This makes it possible to use a distributed reduce function which just adds the values for each key in order to compute the frequency of the k-subsets over the baskets;
- We then remove from each basket the items which could not be part of a frequent itemsets of size $k+1$ by employing the monotonicity property; to do so, we actually create a new basket which we substitute to the old one with all the unique elements which are part of at least a frequent itemset of size k. This is analogous to the creation of the candidate set C_{k+1}.

This is the main functioning of the algorithm developed. From the implementation point of view the *apriori* function developed has the following signature:

df	a PySpark DataFrame object
basket_col	the column of the dataframe containing baskets
support_threshold	the percentage value of the support threshold
max_size	the last set size of frequent itemsets to check
sample_fraction	sample fraction of baskets which will be used to find frequent itemsets
seed	seed used when sampling
cache	flag which decides if caching should be applied when running the algorithm
quiet	flag which decides if some information on the progression will be printed on screen.

which highlights some more interesting aspects of the implementation: a sample of the baskets could be used instead of the all dataset, which can potentially make it possible to run it on much more massive dataset, because running the algorithm on a sample requires less memory, in proportion to

the size of the sample. In the implementation, a SRS schema is used. If the algorithm is run on a cluster, a stratified sampling could be preferable, because there is no need to collect the entire dataset in memory on a single node before extracting it, but a fraction of the sample is extracted on each partition, helping to reduce the memory for this initialization phase. Moreover, caching in RAM of the main data structures used throughout the algorithm can be used to reduce the running time at the cost of a greater memory usage.

3 Application to the LinkedIn Jobs and Skills Dataset

To test the APriori implementation previously described we will use the LinkedIn Jobs and Skills Dataset available on Kaggle. In particular we are interested in the *job_skills.csv* table which contains some string records containing the skills required by a number of job posts on LinkedIn.

To be able to run the APriori algorithm we first needed to do some preprocessing: we dropped the tiny fraction of empty records, we uniformized some skills substrings having the same semantic meaning such as "decisionmaking" and "decision making", and then we splitted the string in a list containing each of the skills in a posts. The PySpark dataframe containing these lists of items is then passed to the *apriori* function described previously.

A possible improvement of this preprocessing step would be considering instead the edit distance between pairs of skills so that to be able to aggregate the ones with small distances. This is not trivial because there are more than 2 millions of unique skills in the dataset.

On a AMD Ryzen 5 5600H with 6 CPUs and 16 gb of RAM the search for frequent itemsets on the all dataset completed in nearly 90 seconds, finding, at a threshold of 1%, frequent itemsets up to order 6. The singletons with a support higher than the threshoild were 186; this plot represents the 10 singletons with the highest support:

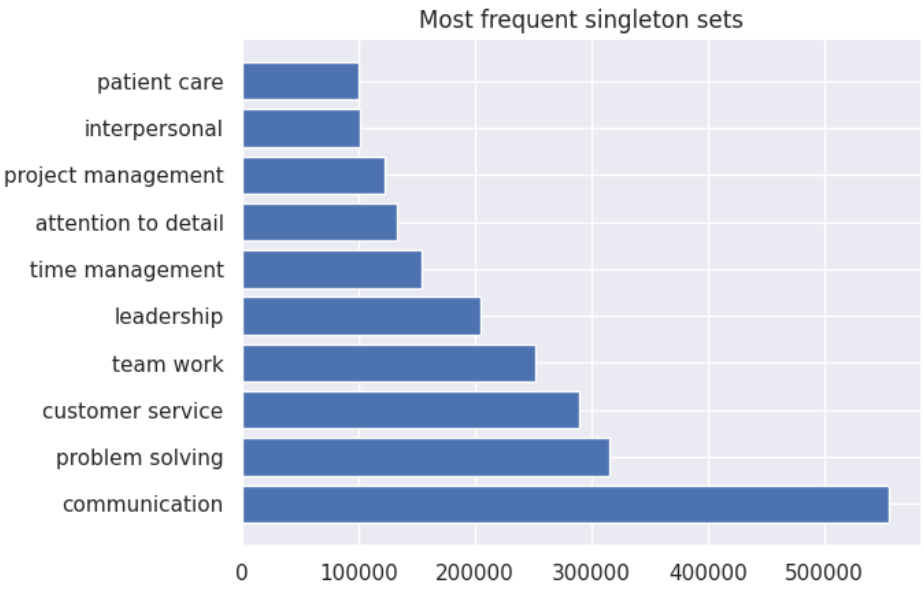

Figure 1: Enter Caption

Considering that the dataset contains nearly 1.3 million posts, these skills seem highly requested by the job market. The most frequent subsets on higher sizes were mostly combinations of the singletons already highlighted in the graph. There was just a subset of 6 elements above the 1% threshold, which was composed by "communication", "time management", "customer service", "team work", "problem solving" and "attention to detail".

It was also tried to verify that sampling was useful in reducing the running time of the algorithm. Since the time required dropped to nearly 10 seconds for a sample of 1% of the dataset, this means that sampling helps in reducing it. Though, it can be noted that the running time didn't drop to roughly 1/100 of the running time on the all dataset, but just to 1/10. This is because collecting the initial sample required most of the time of the entire process.

4 Conclusions

In this report, we successfully implemented a distributed version of the APriori algorithm which was able to return all frequent itemsets with a support threshold of 1% in less than 2 minutes on the all example dataset on a modest commercial laptop at our disposal. While untested, it should be possible to use the proposed algorithm on a cluster of nodes. We end this report with some of the limitations which would actually need to be lifted if one wants to really improve the space and time required by the algorithm: first, the implementation still uses a lot of user-defined function (UDF) which hinder the overall time performance of the algorithm because they need to be executed as interpreted Python code which is known to be slow compared to other languages. Secondly, while the items are transformed in integers by the algorithms, they are not as small as they could be because they are actually Python objects which require more than 20 bytes each. This means that the overall memory performance could be probably improved by almost an order of magnitude by avoiding Python data structures as much as possible. Unfortunately, it is not an easy task because dictionary and sets, which are heavily used in the implementation, are not available in easy-to-use accelerator such as NumPy or Numba. A possible viable option would be to use instead Cython because it allows to mix functions which are compiled to some fast C routines with other functions which instead are executed by the Python interpreter.

References

[JLU14] Anand Rajaraman Jure Leskovec and Jeffrey David Ullman. *Mining of Massive Datasets.* 2014.