UNIVERSITÀ DEGLI STUDI DI MILANO

Data Science for Economics

Algorithms for Massive Data

Market Basket Analysis over LinkedIn Job Postings

Professor: Dario Malchiodi

Student: Guglielmo Berzano, id: 13532A

E-mail: guglielmo.berzano@studenti.unimi.it

April 2024

Abstract

This work aims to implement the A-Priori algorithm for a problem of Market Basket Analysis (MBA). The objective is pursued through the utilization PySpark, the Python version of the Apache Spark engine for dealing with BigData. After a brief description of the dataset and of the preprocessing methods, the report continues by explaining how the custom `apriori()` function works, illustrating and deepening all of its parameters. As a result, given the computational power at my disposal, the function was able to consistently find every frequent itemset until the quadruplets.[i]

[i] 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.

TABLE OF CONTENTS

1 Introduction

In this project I will create a Python function to deal with a Market Basket Analysis (MBA) problem of finding the frequent skills requested in job postings on LinkedIn. In particular, I will implement the **A-Priori** algorithm over the Kaggle dataset that can be found at this link[1].

In the following sections I will deeply explain the methodology and how I decided to tackle the problems I faced. The Python code is completely available on my GitHub profile.

1.1 Exploratory analysis and preprocessing

Among the many tables the aforementioned dataset is composed by, I will focus on one in particular: the *job_skills* table which has the following structure:

Table 1: Job_skills' table structure

Job_link	**Skills**
https://www.linkedin.com/jobs/[...]	Building Custodial Services, [...]
⋮	⋮
https://www.linkedin.com/jobs/[...]	Communication skills, Teamwork, [...]

where:

- **Job_link** is a column of strings relating to the LinkedIn job positions.
- **Skills** is a column of string relating to the skills required for each position.

First of all, I imported this table in the PySpark session and dropped the Job_link column due to a lack of meaningful information for our study-case. Furthermore, I dropped the Not-Available (NA) cells in Skills, resulting in $\mathbf{1,294,374}$ total rows.

[1] Licensed under the Open Data Commons (OCD) Attribution licence, click here to know more.

Among these jobs, the one with the highest number of skills required is a *clinical theater manager* with a whopping 463 total skills required. Still, average number of skills per-job is 20.789, the minimum is 1 and the total number of different skills is $2,770,596$. No further preprocessing is done.

2 A-Priori Algorithm implementation

In order to implement the A-Priori algorithm, I decided to create a function called `apriori` which takes the following inputs[2]:

- `df`: a PySpark DataFrame.
- `sampling`: a boolean variable indicating if the sampling step is desired or not. Notice that, for this application, the default PySpark `sample` function will be used. The default value is `True`.
- `fraction`: used only if `sampling == True`. This is a casual fraction of the dataset to keep. The default value is `0.01`.
- `seed`: used only if `sampling == True`. This is number used to do the sampling, useful for reproducibility. The default value is `1234`.
- `s_threshold`: the *support threshold* is needed to understand which itemset is frequent and which is not. Essentially, if the count of an itemset is larger than this value, it will be considered frequent. Usually this value is around 1% or 2% of the total `df` size – after sampling, if it was applied. The default value is `2`.
- `last_frequent`: last frequent itemset to check for. If it is equal to zero it will find frequent itemsets of every cardinality (singletons, pairs, triplets, ...) until there are no more. If it is different from 0, for example 1, it will only check for frequent singletons, if equals 2 for pairs and so on. The default value is `4`, thus checking up to quadruplets.

[2]Clearly I set default values for each entry except for the `df`. Still, each single input can be modified freely.

The main idea behind this function is the sampling, achieved by the implementation of the `sample()` function of Spark. In particular, I set `with_replacement = False` and the other parameters `fraction` and `seed` equal to the entries of their relative variable inside the `apriori` function. By taking just a small portion of the original dataset, every computation will be significantly sped up.

At the beginning, the function takes the `df` and samples it according to the parameters passed, if any. Then, it transforms the result into a PySpark Resilient Distributed Dataset (RDD), finds the number of partitions and applies the following operations. Each basket, i.e. row, in the Skills column of Table 1 will be split according to the separator `(", ")` and transformed into lowercase singletons. To be more clear, we are going from a structure in which each row was stored as a string: "Communication skills, Teamwork, [...]" to a structure in which each row is a list containing singular lowercase skills ["communication skills", "teamwork", "[...]"]. After this, the words "skill" and "skills" are removed to avoid considering "communication" and "communication skills" as two different skills. Lastly, I changed "problemsolving" to "problem solving" to avoid the same problem depicted earlier. Then, the function proceeds by starting a `while` loop until certain conditions are not met.

2.1 Results

First of all the function will look for frequent singletons, just by counting how many times each skill appears in the baskets and checking if that frequency is higher than `s_threshold` which, in our case, means that a skill must appear at least 260 times in order to be considered frequent. The job was executed in around 30 seconds and a total of **77 frequent singletons** were found. The three most frequent are:

1. **Communication**, observed 5,618 times.
2. **Problem solving**, observed 3,167 times.
3. **Customer service**, observed 2,965 times.

The next step is to find the frequent pairs. In order to compute them, it is crucial to use the `combinations()` function of the `itertools` library in order to check, in this case, which pairs of frequent singletons are, themselves, frequent. Indeed the **monotonicity** property,

which states "if a set I of items is frequent, then so is every subset of I", see [1], must apply. To do so, we iterate over each ordered couple[3] formed by the skills in each basket, check if both of the elements are frequent singletons and eliminate those pairs which are below the *support threshold*. The job was executed in around 30 seconds and a total of **91 frequent pairs** were found with three most frequent being:

1. **Communication, Problem solving**, observed 2,597 times.
2. **Communication, Customer service**, observed 1,996 times.
3. **Communication, Teamwork**, observed 1,970 times.

Similarly to how we found frequent pairs, we can now find frequent triplets. In this case, the **monotonicity** property imposes us to check, for each triplet, if all the pairs generated by the elements composing the triplet are frequent and then check if the triplet itself is frequent, i.a. if its count exceeds the support threshold. Thus, a triplet must be composed only by frequent pairs which, in turn, must each be composed by frequent singletons. Again, to minimize the number of triplets to check, we will focus on those obtained after having used `sorted()`.

The function found **51 frequent triplets** in about 100 seconds and the top three most frequent are:

1. **Communication, Problem solving, Teamwork**, observed 1,271 times.
2. **Communication, Customer service, Problem solving**, observed 1,076 times.
3. **Communication, Leadership, Problem solving**, observed 935 times.

After triplets, we will continue and compute quadruplets using the same exact methodology. The function found **16 frequent quadruplets** in about 22 minutes and the top three most frequent are:

[3]We want to analyse ordered pairs because otherwise we may had the tuples `("communication", "teamwork")` and `("teamwork", "communication")` being considered as two distinct elements which should not be the case. To solve this we simply order the tuples.

1. **Communication, Customer service, Problem solving, Teamwork**, observed 1,271 times.

2. **Communication, Problem solving, Teamwork, Time management**, observed 1,076 times.

3. **Communication, Leadership, Problem Solving, Teamwork**, observed 935 times.

Since the default value for `last_frequent` was 4, the function stops here even though there may exist frequent pentaplets and maybe also hexaplets. Still, the computational power at my disposal did not allow me to proceed any further even after the drastic sampling described earlier.

3 Conclusions

Even by using a library like PySpark, the amount of time needed to perform every map-reduce operation was massive. This forced me to implement a sampling technique in order to speed up computations, just for the sake of this experiment. The whole `apriori()` function I created runs in about 26 minutes finding every frequent itemset up to quadruplets without being able to go further due to the missing computational power. Still, the function is perfectly able to analyse larger datasets and what I have shown in this project is just a demonstration of what it is possible to obtain.

References

[1] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets.* Cambridge University Press, 2nd edition, 2014.