Market-Basket Analysis using Apriori and PCY Algorithms

Jad YEHYA

Abstract

The dataset used in this paper is the yelp dataset downloaded in June 2023

1 Introduction

With data being everywhere and in enormous quantities, the need for tools to extract meaning from it has became more and more important. This is where data mining comes in. Data mining is the process of extracting useful information from large data sets (commonly called "Big-Data"). In this context, researchers have developped algorithms to help us achieve this. The objective of this paper is to talk about Market-Basket Analysis, finding frequent itemsets and association rules. We will see two of them. First we will see a basic implementation of the Apriori algorithm [1] which has some limitations. To overcome the limitations, the Park-Chen-Yu (PCY) algorithm [2] was introduced in 1995. We will then compare the two algorithms and see which one is better in terms of execution time and memory usage. Finally, we will see how we can parallelize the PCY algorithm using the MapReduce framework.

2 Preprocessing

To be able to process files as big as the yelp dataset that we have, we needed to find some special ways to process the data. This is when we know we entered "Big Data" world. We first read the file in chunks of size 100000. Then we tokenized every bucket. For this we used the library `nltk`. We also removed any stop words (such as "I", "and", "or" etc.). We did this using the MapReduce paradigm and multiprocesses (8 in our case) to speed up the work. We save this in a seperate file that will become our focus, since we don't any other columns in the original dataset. Now that we have clean and preprocessed data, we can start the actual work of the algorithms.

3 Apriori Algorithm

3.1 Finding Frequent Itemsets

The Apriori algorithm is a classic algorithm for frequent itemset mining, which is used to discover associations or patterns in a dataset. It operates on a transactional dataset, where each transaction consists of a set of items.

The main idea behind the Apriori algorithm is to generate candidate itemsets of increasing length (k) and prune them based on their support (frequency of occurrence) in the dataset. The algorithm iterates through multiple passes, progressively increasing the length of the itemsets until no more frequent itemsets can be generated. The algorithm is described in Algorithm 1. The implementation is straightforward, we first initialize the itemsets with the single items in the dataset. Then, we generate the candidate itemsets of length k, count the support (frequency of occurrence) of each candidate itemset, and prune them based on their support (this is done to reduce the computational time). These itemsets are discarded as they cannot be frequent. The remaining itemsets are added to the list of frequent itemsets. The algorithm terminates when no more frequent itemsets can be generated.

Algorithm 1 Apriori Algorithm

```
procedure APRIORI(transactions, min_support)
    itemsets ← [[item] for item in set(item for transaction in transactions for item in transaction)]
    frequent_itemsets ← []
    k ← 1
    while itemsets ≠ ∅ do
        candidates ← generate_candidate_itemsets(itemsets, k)
        itemsets ← []
        for candidate in candidates do
            is_frequent, count ← calculate_support(transactions, candidate, min_support)
            if is_frequent then
                itemsets.append(candidate)
                frequent_itemsets.append((candidate, count))
            end if
        end for
        k ← k + 1
    end while
    return frequent_itemsets
end procedure
```

3.2 Finding Association Rules

Association rules are patterns that describe relationships between items in a data set. It is commonly used in data mining and shopping cart analysis to find meaningful connections and correlations between different items. The strength of the association

rule is measured by two of her indicators: support and trust. This support measures the frequency or percentage of transactions in a dataset containing both X and Y. Confidence measures the conditional probability that Y exists in a transaction given that X exists.

We iterate over the $frequent\_itemsets$ then we generate the all possible subsets of the itemset. We will these the antecedents. Then for each subset generated, we calculte the consequente (right side of the association) by doing the set substraction between the itemset and the antecedent. We then calculate the support and confidence of the association rule. If the support and confidence are greater than the minimum support and minimum confidence respectively, we add the association rule to the list of association rules. The algorithm returns the list of tuples (association rule, support, confidence).

This algorithm is very basic and the most efficient. We will see in the next section the PCY algorithm which is more efficient than the Apriori algorithm.

4 PCY Algorithm

4.1 Finding Frequent Itemsets

The PCY algorithm is another algorithm for finding frequent itemsets. It uses a hash table to count the number of times a pair of items appear together. This hash table is called the *bitmap*. The bitmap is used to determine which pairs of items are potentially frequent. The PCY algorithm is faster than the Apriori algorithm because it does not generate candidates that are not potentially frequent. The PCY algorithm is also more memory efficient than the Apriori algorithm because it does not store the entire dataset in memory. It only stores the bitmap in memory. The PCY algorithm is also parallelizable. We will see how we can parallelize it using the MapReduce framework. There is 2 steps in the PCY : the first pass and the second pass.

4.1.1 First Pass

We iterate over every bucket and count the occurrences of every individual item. We then filter out (prune) the items that don't meet the minimum support threshold.

4.1.2 Second Pass

Here I implented the second pass to search for pairs or triplets and not bigger itemsets. We create a hash table that will be used the occurences of the candidate itemsets. We then iterate over every bucket and generate candidate pairs and filter the candidate pairs by keeping the ones that have both element in the frequent itemset. The result becomes our candidate pairs. We then count pairs in the hash table. Same process for the triplets. We then iterate over all the candidates and filter out the ones that don't meet the minimum support threshold and the count in the hashtable. The result is our frequent itemsets.

Algorithm 2 PCY Algorithm

Require: *transactions*: List of transactions, *min_support*: Minimum support threshold, *hash_table_size*: Size of the hash table

Ensure: *frequent_itemsets*: List of frequent itemsets

single_item_counts $\leftarrow \{\}$
for *transaction* in *transactions* **do**
    for *item* in *transaction* **do**
        $single\_item\_counts[item] \leftarrow single\_item\_counts.get(item, 0) + 1$
    end for
end for
$frequent\_single\_items \leftarrow \{item$ for $item, count$ in $single\_item\_counts.items()$ if $count \geq min\_support\}$
candidate_counts $\leftarrow \{\}$
hash_table $\leftarrow$ np.zeros((*hash_table_size*,), dtype=int)
for *transaction* in *transactions* **do**
    items $\leftarrow$ set(*transaction*)
    pairs $\leftarrow$ combinations(*items*, 2)
    candidate_pairs $\leftarrow$ $[(item1, item2)$ for $item1, item2$ in *pairs* if $item1 \in frequent\_single\_items$ and $item2 \in frequent\_single\_items]$
    for *pair* in *candidate_pairs* **do**
        hash_value $\leftarrow$ hash(*pair*) mod *hash_table_size*
        $hash\_table[hash\_value] \leftarrow hash\_table[hash\_value] + 1$
        $candidate\_counts[pair] \leftarrow candidate\_counts.get(pair, 0) + 1$
    end for
    triplets $\leftarrow$ combinations(*items*, 3)
    candidate_triplets $\leftarrow [(item1, item2, item3)$ for $item1, item2, item3$ in *triplets* if $(item1, item2) \in candidate\_pairs$ and $(item1, item3) \in candidate\_pairs$ and $(item2, item3) \in candidate\_pairs]$
    for *triplet* in *candidate_triplets* **do**
        hash_value $\leftarrow$ hash(*triplet*) mod *hash_table_size*
        $hash\_table[hash\_value] \leftarrow hash\_table[hash\_value] + 1$
        $candidate\_counts[triplet] \leftarrow candidate\_counts.get(triplet, 0) + 1$
    end for
end for
$frequent\_itemsets \leftarrow [(itemset, count)$ for $itemset, count$ in $candidate\_counts.items()$ if $count \geq min\_support$ and $hash\_table[hash(itemset) \mod hash\_table\_size] \geq min\_support]$
    return *frequent_itemsets*

5 Implementation

5.1 Apriori Algorithm

We made a non optimized version of the apriori algorithm so that we can see the difference between this and the PCY algorithm. The implementation we have is the one described in the section 3.

The inference time is really long, as we can see on the figure 1. We didn't bother to go past 1000 buckets because it would have taken too much. The point is made.

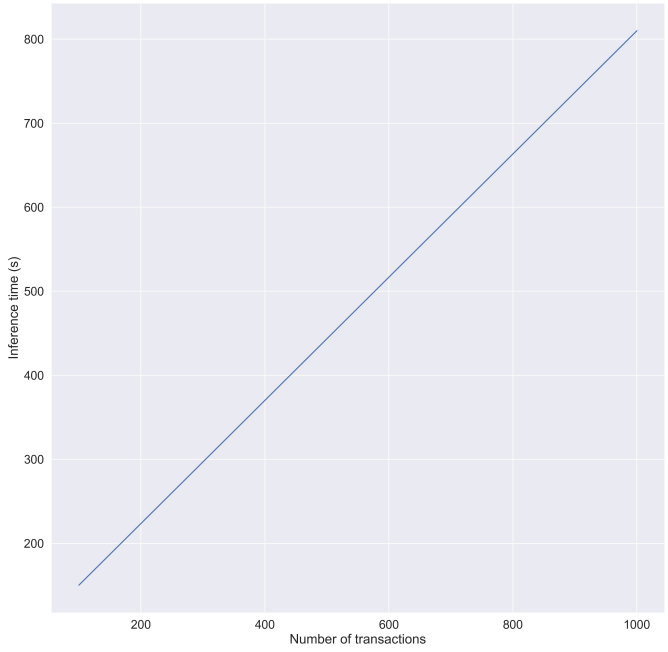

Fig. 1 Apriori inference time

5.2 PCY Algorithm

For the PCY algorithm, we made 2 implementations, first an implementation that is not parallelized and then a parallelized version using the MapReduce framework. The implementation we have is the one described in the section 4.

The inference time is way better as we can see in the figure 2. Naturally, the parallelized version is faster because it uses multiple threads to do the work. The non parallelized version is faster than the apriori algorithm because it doesn't generate candidates that are not potentially frequent.

Table 1 Frequent itemsets

Itemset
going, order
need, clean, like, even, work
really, well, maybe, one, good
experience, order, usually, good, food

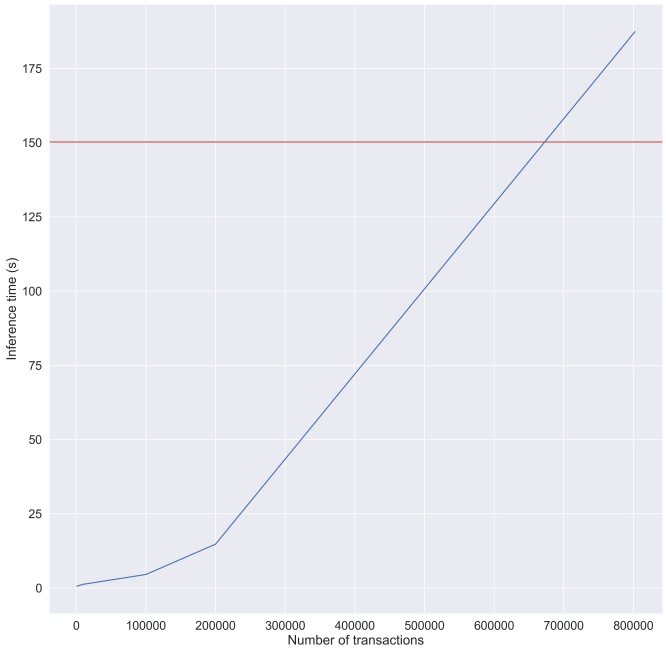

Fig. 2 PCY inference time

The red line we can see is the time taken by the apriori algorithm for a bucket of size 100. In the same time, the PCY algorithm can process buckets of size almost 7×10^5.

6 Results

We have ran the algorithms on the dataset and this is the main frequent itemsets and association rules that we found:

Now that we have the association rules and their confidence, we draw a graph showing the association rules and their confidence. The graph is shown in Figure 3.

Table 2 Association rules

Antecedent	Consequent	Support	Confidence
really, fun, well, maybe	one	0.000199	0.704846
menu	good	0.0461062	0.470149
experience order usually good	food	0.000428	0.666019

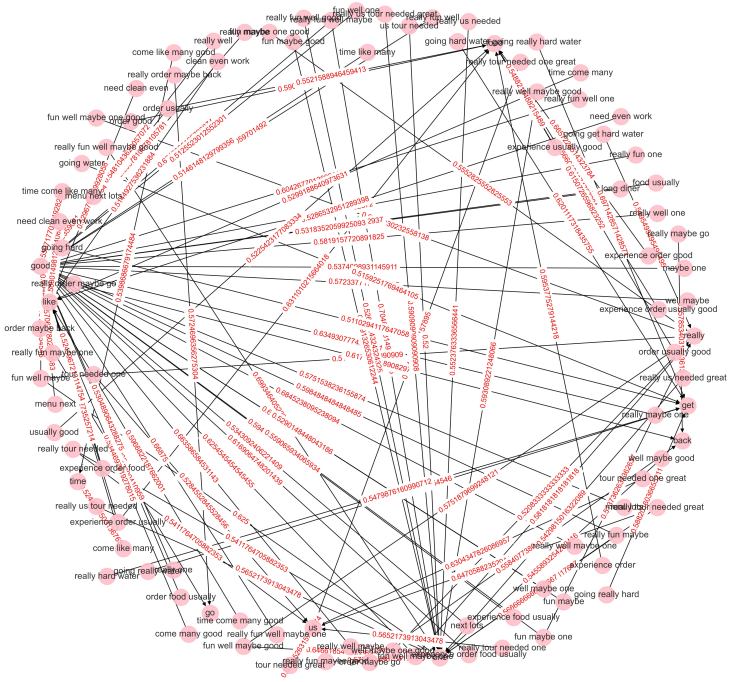

Fig. 3 Association rules

7 Conclusion

In conclusion, we saw that the PCY is faster than the apriori algorithm, but also that the PCY using MapReduce and multi processes is even faster. This is no news and was completely expected. Further optimizations of the PCY have been made to make inference even faster and to work with bigger datasets.

8 Declaration

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.

References

[1] Agarwal, R., Srikant, R., *et al.*: Fast algorithms for mining association rules. In: Proc. of the 20th VLDB Conference, vol. 487, p. 499 (1994)

[2] Park, J.S., Chen, M.-S., Yu, P.S.: An effective hash-based algorithm for mining association rules. ACM SIGMOD Record **24**(2), 175–186 (1995) https://doi.org/10.1145/568271.223813 . Accessed 2023-06-12