MARKET BASKET ANALYSIS PROJECT

Cavallari Marco, Fiorio Federico

June 5, 2022

Abstract

"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."

Contents

1 Introduction

The purpose of this project is to implement a scalable solution for finding frequent itemsets (in our case pairs). In particular We implemented some of the algorithm seen in class, such as Apriori, PCY, Multi-Hash, Multi-Stage and SON. All of them were applied to a sub-portion (We will discuss about that) of the "Ukraine Conflict Twitter" dataset given by the professor. The analysis performed is also known as 'market-basket analysis' (or simply MBA).

What We did was preprocess the dataset and extract from it the baskets and the items that compose them. In this context the basket are tweets and the items are the hashtags inside them. Then starting from this set up, We started developing our solutions for the various algorithms cited before, starting from Apriori and arriving to SON. At the end, We generated in a very simple way the various association rules from our frequent pairs and calculated support, confidence, interest and lift for all of them.

2 Dataset analysis

Let's start saying that inside the file "ukraine-russian-crisis-twitter-dataset-1-2-m-rows.zip" there were many csv.gzip files. For sake of simplicity We took the first one: "0401_UkraineCombinedTweetsDeduped.csv.gzip". In Figure 1 are shown the first 2 rows (transposed) of the dataset. Many attributes were present (userid, username,etc..). Intuitively We are interested in one field of it: "hashtags". Theorically yes but practically no. The fact is that in some cases (like the very first row of our dataset) the filed "hashtags" is empty also if the filed "text" contain some hashtags inside it.

```
df[0:2].T
```

	0	1
userid	16882774	3205296069
username	Yaniela	gregffff
acctdesc	Animal lover, supports those who fight injusti...	NaN
location	Hawaii	NaN
following	1158	122
followers	392	881
totaltweets	88366	99853
usercreatedts	2008-10-21 07:34:04.000000	2015-04-25 11:24:34.000000
tweetid	1509681950042198030	1509681950151348229
tweetcreatedts	2022-04-01 00:00:00.000000	2022-04-01 00:00:00.000000
retweetcount	3412	100
text	⚡The Ukrainian Air Force would like to address...	Chernihiv oblast. Ukrainians welcome their lib...
hashtags	[]	[{'text': 'russianinvasion', 'indices': [77, 9...
language	en	en
coordinates	NaN	NaN
favorite_count	0	0
extractedts	2022-04-01 00:44:20.097867	2022-04-01 00:09:37.148770

Figure 1: Fist two rows (transposed) of the dataset used in this project.

What We did to resolve this was just not considering the attribute "hashtags" and directly extract the hashtags from the attribute "text". First we created the RDD rdd_text where each element was a text of a tweet. Then for all of them we applied a function "processHashtags" to extract its hashtags. We can see an example in Figure 2. The result was the RDD "hashtags_per_tweet", where each element was composed by a key (the tweetid) and a list of all the hashtags contained in it.

```
[24] def processHashtags(tweet):
       hashtags = []
       for i,word in enumerate(tweet):
         if word == '#':
           try:
             hashtags.append(re.sub(r'[^\w\s]','',tweet[i+1]).lower()) #remove punctuation with regexp and put them lowercase
           except:
             pass
       return hashtags

     hashtags_per_tweet = rdd_text.map(lambda x: (x[0], processHashtags(x[1])))
     hashtags_per_tweet.first()

     ('1509681950042198030', ['protectuasky', 'stoprussia', 'ukraineunderattack'])
```

Figure 2: Function processHashtags that takes as input a tweet and return the list of hashtags in it.

Finally, from this RDD We created "basket_file", where each element represented the list of hashtags inside one tweet. The Figure 3 shows the hashtags contained in the first 10 tweets. NB: in the project there is an intermediate passage from the "hashtags_per_tweet" to a Spark dataframe "new_df_spark", just to visualize better the situation.

```
basket_file.take(10)

[['protectuasky', 'stoprussia', 'ukraineunderattack'],
 ['russianinvasion',
  'standwithukraine',
  'ukraineunderattack',
  'ukrainewillwin',
  'putinisawarcriminal',
  'stopputin',
  'russianukrainianwar',
  'russiagohome',
  'россиясмотри',
  'нетвойне'],
 ['russianukrainianwar', 'china', 'taiwan'],
 ['anonymous', 'oprussia', 'ddosecrets'],
 ['nft', 'mint'],
 ['russia',
  'ukraine',
  'motivation',
  'netde',
  'edude',
  'delaware',
  'government',
  'usa'],
 ['ukraine', 'ukrainewar', 'russia', 'ukraineinvasion'],
 ['russian', 'moscow'],
 ['ukraine'],
 ['putin', 'medvedev', 'russia', 'ukraine']]
```

Figure 3: Fist ten baskets/tweets of the basket file. As expected, each basket is defined by a list of hashtags.

In the next section the techniques used to find frequent itemsets were performed only on a subset of "basket_file": the first 500 elements/tweet. This has been done to allow a fast computation (although the implementations of the various algorithms are scalable in a distributed enviroment).

3 Market Basket Analysis

In this section We will discuss more in detail about the various techniques used to retrieve frequent pairs.

3.1 Apriori Algorithm

The A-Priori Algorithm is designed to reduce the number of pairs that must be counted, at the expense of performing two passes over the basket file, rather than one pass[1]. In the first pass We calculate the absolute frequency of each item inside the basket file. This cost the first scan in the basket file. Then, between the fist and the second scan We examine the counts of the items to determine which of those are frequent as singletons (a frequent singleton is an item whose frequency inside the basket is above the fixed threshold). Now We start the second and last scan of Apriori. During the second pass, We calculate the frequency of all the pairs made of two frequent items for each basket of the basket file. Recall that a pair cannot be frequent unless both its members are frequent (by monotonicity). This allowed us to discard pairs that were surely not frequents without calculating their frequency inside the basket file.

Finally, at the end of the second pass, examine the structure of counts to determine which pairs are frequent.

In our implementation, We developed Apriori using the Map-Reduce paradigm. Let's see how:

1. First, We calculated the frequency of all the hashtags (singletons) in the basket file (1st pahse) performing a simple Map-Reduce job (row indicated with the comment "1st phase" in Figure 4),
2. then, between the the 1st and the 2nd phase we filtered out all the non frequent singletons,
3. finally in the 2nd phase We calculated all the possible pairs of frequent singletons, and after this We performed the 2nd and last scan on the basket file calculating the frequency of all the pairs made of frequent singletons in the basket file. At the end We filtered out all the pairs made of frequent singletons whose frequency in the basket file was below the threshold.

```
# define Apriori function to use in SON
from itertools import combinations
def Apriori(basket_file, threshold):
  singleton=basket_file.flatMap(list).map(lambda item: (item,1)).reduceByKey(lambda a,b: a+b) #1st phase
  freq_singleton=singleton.filter(lambda x: x[1]>=threshold)#between 1st and 2nd phase
  pairs=list(combinations(freq_singleton.map(lambda x: x[0]).toLocalIterator(),2)) #start of the 2nd phase
  flatted_couples = basket_file.map(lambda x: [(pair,1) for pair in pairs if set(pair).issubset(set(x))]).flatMap(lambda x: x).cache()
  reduced_elements = flatted_couples.reduceByKey(lambda a, b: a + b)
  freq_pairs = reduced_elements.filter(lambda x : x[1] >= threshold).cache()
  return freq_pairs
```

Figure 4: Apriori implementation.

NB: In the notebook, in the implementation of Apriori We preferred to use a "step-by-step" approach, so the instructions inside the function in Figure 4 are written in multiple cells (so not as a function) in order to give a detailed description of each line of code. The function in Figure 4 has been used when implementing the SON algorithm.

3.2 PCY algorithm

As second algorithm We decided to implement PCY. The A-Priori Algorithm is fine as long as the step with the greatest requirement for main memory – typically the counting of the candidate pairs C2 – has enough memory that it can be accomplished without thrashing (repeated moving of data between disk and main memory). Several algorithms have been proposed to cut down on the size of candidate set C2. Here, we consider the PCY Algorithm [2]. With this algorithm We exploit the fact that during the first scan there is a quite big part of main memory not used. What We can do is use an hash function and hash pair of objects inside buckets (associated with an integer). How many buckets will we have? all the buckets necessary to fill all the unused main memory. So in main memory We will have an Hash-map: a data structure which is a sort of array where each position is associated to one bucket of my hash function and the corresponding value is the number of pairs hashed to that position). During the 1st scan We will generate all the possible pairs of object of the basket We are processing. After this We pass to the hash function "h" the pair (i,j). h(i,j) will identify a position "k" in my array "A" and that cell will be incremented by 1. Note that "A[k]<threshold means that the frequency of the pair hashed to that cell is lower than the threshold. Note also that collisions are possible, so more than one pair could be hashed to one cell. how can we manage this situation? Rather intuitive:

1. if more pairs are hashed to k and (A[k]<s) then it means that the sum of their frequency is lower than the threshold so We can directly discard all of them
2. if more pairs are hashed to k and (A[k]≥s) then We cannot say anything. It could be possible that only a subset of the pairs hashed to k are frequent and the others not or it is possible that all of them are frequent.

This fact gives us an advantage on the second pass. We can define the set of candidate pairs C2 to be those pairs i, j such that:

1. i and j are frequent items.
2. (i, j) hashes to a frequent bucket (whose value≥threshold).

It is the second condition that distinguishes PCY from APriori: We have an additional criterion to filter out pairs. With this reasoning there is a problem: if We fill up all the available memory with this Hash-map We will not have space for storing my counters. The problem could be easily resolved "summarizing" my data structure in a bit-map.

In our implementation, We started defining an hashing function that took as input a pair and output its bucket. Then, in the same way we did in Apriori for the frequent singletons, We generated all the possible pairs between items (this time also not frequent) of each basket and stored the result in "pairs_first_pass" of Figure 5. After doing this, We applied a map job to create a tuple "(hashing(pair),1)" for each pair of each basket. Then We flatted the result and reduced it computing the frequency of each bucket. The result was stored in "hashtable_rdd" in Figure 5.

```
from sys import hash_info

HASH_TABLE_SIZE = 50000
hash_table = np.zeros(HASH_TABLE_SIZE, dtype=int)

def hashing(pair_to_hash):
  return int(abs(hash(frozenset(pair_to_hash))) % HASH_TABLE_SIZE)
```

```
hashtable_rdd = basket_file.map(lambda basket: [(hashing(pair), 1) for pair in list(combinations(basket, 2))]).flatMap(lambda x: x).cache()\
                           .reduceByKey(lambda a,b: a+b)
```

Figure 5: Fist pass of PCY algorithm. In hashtable_rdd it will be stored a list of key value pairs where the key is the bucket in which a pair has been hashed and the value is the number of pairs hashed in that bucket.

Subsequently We used a list hash_table (Figure 6) which correspond to the array "A" We discussed before and for the space problem discussed previously We passed from that structure to a bit-map (1 if hash_table[i]≥threshold, 0 otherwise)

```
hashtable_list = list(hashtable_rdd.map(lambda x: x).toLocalIterator())

for pair in hashtable_list:
  hash_table[pair[0]] = pair[1]

bitmap_freq = [hash_table[i]>=threshold for i in range(HASH_TABLE_SIZE)] #bitmap_freq[i]==False if hash_table[i]<threshold, True otherwise
```

Figure 6: Fist 5 element of hashtable_rdd. (3810,20) means that 20 pairs have been hashed to the bucket 3810.

In the 2nd step, We did the same thing We did with Apriori BUT this time, in order to be considered a "candidate frequent pairs" We also need to ensure that:

$$bitmap\_freq[hashing(x)] == True$$

In other words, the pair x must be hashed to a frequent bucket as We can see in Figure 7.

```
#candidate pairs = pairs of freq singleton and the pair is freq in the hash_table
candidate_pairs = pairs_rdd.filter(lambda x : bitmap_freq[hashing(x)] == True)
candidate_pairs_list = list(candidate_pairs.map(lambda x: x).toLocalIterator()) #pr
candidate_pairs_list[:10]
```

Figure 7: In order to be a candidate pair, it must be hashed to a frequent bucket.

3.3 Multistage and Multihash algorithms

Having already implemented the Apriori algorithm and PCY, It was quite straightforward the implementation of both multi-stage and multi-hash algorithms. The Multistage Algorithm improves PCY by using several successive hash tables to reduce further the number of candidate pairs. Now, a pair, in order to be consider a candidate pair must satisfy the following requisites:

- both the items that form the pair must be frequent singletons (We already gave the definition of frequent singletons in the previous sections),
- For all the hash tables considered, (i,j) must be hashed in a frequent bucket.

Note that the algorithm consider an hash table per scan. So if for example We want to use 2 hash tables We would need to do 3 scans. The presented situation is well depicted in Figure 8 (figure taken by the book "Mining of Massive Datasets" by A. Rajaraman e J. Ullman).

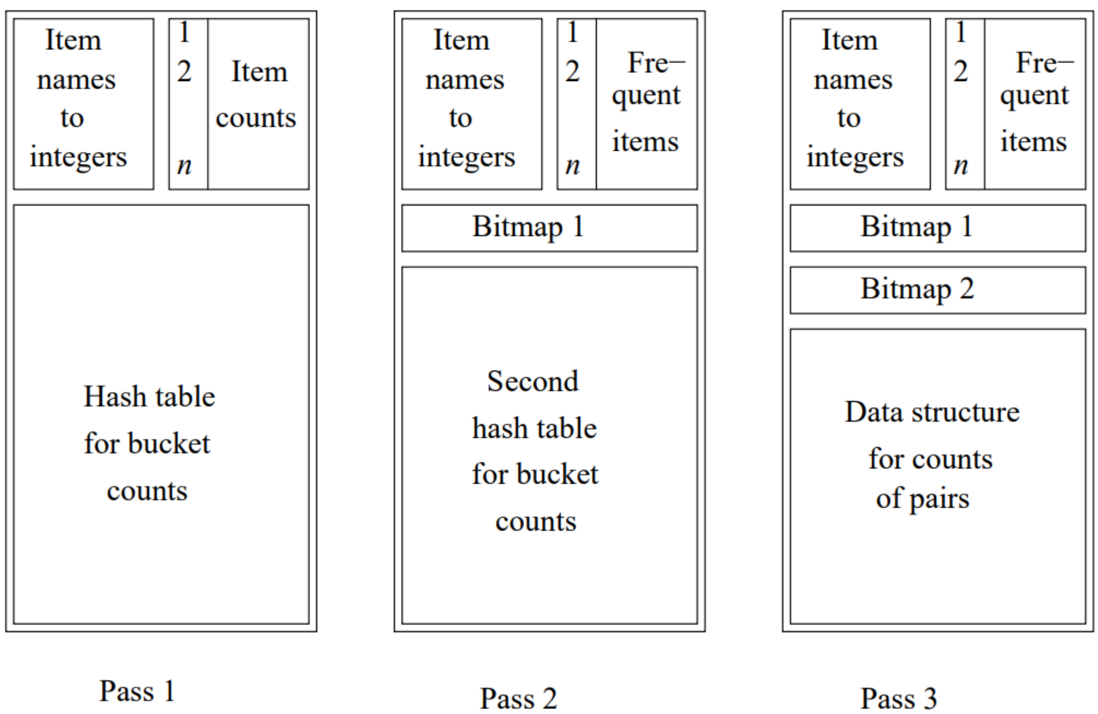

Figure 8: 3 pass of Multistage algorithm.

It is possible to have all the advantages of the Multistage Algorithm in a single pass. This variation of PCY is called the Multihash Algorithm. In this algorithm We consider multiple hash tables in a single scan. The disadvantage is that now the space that in Multistage was assigned to a single hash table now is shared between more of them. A consequence of this is that the number of buckets per hash table will be lower than before (the half in case of two hash map) and there will be more possibility of collision. An extreme of this is that all the buckets will be frequent buckets. **The implementation**, is very similar to the one of the PCY (obviously applying some changes). The first phase of Multistage was the same as PCY: We considered a single hash table and proceeded as in PCY. In the 2nd phase (Figure 9) We hashed the pairs of the basket file using the function "hashing_1". Then We considered the output (a set of pairs hashed in frequent bucket stored in "pairs_first_check_list" of Figure 9) as input for a second hashing function "hashing_2". At the third stage We were in a situation where We had the 2 bitmaps (obtained from the hash tables created previously). What has been done was filtering out from the pairs formed by frequent singletons all the pairs not hashed both in frequent bucket by the two hashing functions to obtain a set of candidate pairs. Finally We performed a last Map-Reduce job to calculate the frequency of all of them and discard all tho ones below the threshold.

```
#check against bitmap 1
pairs_first_check = pairs_rdd.filter(lambda x : bitmap_1[hashing_1(x)] == True) #pairs from freq singletons and resulted freq in bitmap_1
#hash them again to hashtable 2

pairs_first_check_list = list(pairs_first_check .map(lambda x: x).toLocalIterator())#creating all pairs

#2nd scan, usual count (counting candidates in the baskets)
hashtable_rdd_2 = basket_file.map(lambda x: [(hashing_2(pair),1) for pair in pairs_first_check_list if set(pair).issubset(set(x))])\
                          .flatMap(lambda x: x).cache().reduceByKey(lambda a,b: a+b)

hashtable_list_2 = list(hashtable_rdd_2.map(lambda x: x).toLocalIterator())

for pair in hashtable_list_2:
  hash_table_2[pair[0]] = pair[1]

len(hashtable_list_2) #should be less than in stage 1
```

```
75
```

Figure 9: 2nd pass of Multistage algorithm. All the pairs hashed to a frequent bucket by bitmap_1 are hashed also according the function hashing_2

Multihash is very similar but this time we considered two hash functions with the half of the size with respect to the one of the Multistage algorithm (to simulate the split of the available main memory into 2) and We considered the two functions "in parallel".

3.4 SON algorithm

The last one implemented was the Savasere, Omiecinski, and Navathe algorithm. The idea is to divide the input file into chunks and consider all the baskets in a chunk as a sample. What is done is run Apriori (or any of its variations) to any chunk and merge the results. This will be the set of our candidate pairs. From this set then all the false positive are discarded (checking the frequency of all the pairs in the basket file). The interesting property of this algorithm is that it does nor have false negative either. **The implementation**, We started dividing the basket file in 2 chunks and the We applied Apriori (Figure 4) on them obtaining the frequent pairs of each chunk. Then We merged the result. the two nested for in Figure 10 are used in order to keep only one pair between for example "('protectusky', 'stoprussia')" and "('stoprussia','protectusky')" .

```
all_pairs = spark.sparkContext.parallelize([]) #empty RDD
scaled_threshold = 0.9 * (SIZE_CHUNK/num_baskets) * threshold

for chunk in chunks:
  rdd_chunk = spark.sparkContext.parallelize(chunk) #rdd_chun contain thecurrent chunk
  freq_pairs = Apriori(rdd_chunk, threshold=scaled_threshold)
  all_pairs = all_pairs.union(freq_pairs)

#all pairs at the end of the for will contain the results of Apriori run on the two chunks

all_pairs = all_pairs.map(lambda x : x[0]).distinct()  #elimination of duplicates (from (a,b),(a,b) to (a,b))
candidate_pairs_list = list(all_pairs.map(lambda x: x).toLocalIterator()) #convert to list

# It is possible that Apriori output from a chunk a frequent pair ("Hello","Computer) and Apriori output ("Hello","Computer) for another chunk.
#We need to delete also the duplicates of this from.

candidate_pairs = candidate_pairs_list

for i, pair_1 in enumerate(candidate_pairs_list):
  for j, pair_2 in enumerate(candidate_pairs_list):
    if (i!= j and frozenset(pair_1) == frozenset(pair_2)): # I got something like (a,b) (b,a) but I'm intrested only in one of them.
      candidate_pairs.remove(pair_2) #remove pair_2 from list
```

Figure 10: Fist part of the SON algorithm: merging (removing duplicates of the result of Apriori on each chunk.

Finally We checked our "candidate_pairs" against false positive. For each chunk We calculated the frequency of each candidate_pairs and We appended the result to "final_rdd". Then all the pairs in "final_rdd" whose frequency was under the original threshold were discarded. The results were the frequent pairs (Figure 11).

```
#CHECK AGAINST FP

final_rdd = spark.sparkContext.parallelize([]) #empty RDD

for chunk in chunks:
  rdd_chunk = spark.sparkContext.parallelize(chunk)
  flatted_couples_chunk = rdd_chunk.map(lambda x: [(pair,1) for pair in candidate_pairs if set(pair).\
                                                 issubset(set(x))]).flatMap(lambda x: x).cache()
  final_rdd = final_rdd.union(flatted_couples_chunk) #simulate shuffling, I need to do this because im using 1 machine and multiple chunks


reduced_elements = final_rdd.reduceByKey(lambda a, b: a + b)
freq_pairs = reduced_elements.filter(lambda x : x[1] >= threshold).cache()
result_SON = list(freq_pairs.map(lambda x: x).toLocalIterator())
freq_pairs.collect()
```

Figure 11: 2nd pass of SON algorithm.

4 Results

The previously analyzed algorithms were all "exact algorithms" so we expected that they all would gave converged to the same result. So it was. Since the result was the same for all of them We decided to pick one algorithm (PCY) and retrieve from the frequent pairs found the association rules. Given a frequent itemset I, We can generate all non-empty subsets of it and for every non-empty subset s of I, generate rule s$\rightarrow (I - s)$.

For example if We would have a frequent triple "ABC" all the possible association rules from this itmeset would have been:

- AB→C
- AC→B
- BC→A
- A→BC
- B→AC

- C→AB

In our case, since We just found frequent pairs "AB", the association rules generate were:

- A→B
- B→A

Where A and B rappresent an item . For each generated association rule We calculated:

- Support: fraction of transactions including both the component of a frequent pair,
- Confidence: fraction of transactions including A that also include B (given the associuation rule A→B). In other words, this correspond to the conditional probability P($B \mid A$),
- Interest: The difference between the confidence of a rule A→B and the support of B (intended as the fraction of transactions including the singleton B),
- Lift: the probability of all of the items in a rule occurring together (otherwise known as the support) divided by the product of the probabilities of the items on the left and right hand side occurring as if there was no association between them.[3]. In the case of the rule A→B the lift would be: S(A→B) / (S(A)*S(B))

At the end We stored the results in a pandas dataframe as we can see in Figure 12.

```
df
```

	Rule	Support	Confidence	Lift	Interest
0	protectuasky-->stoprussia	0.040	1.000	15.625	0.936
1	stoprussia-->protectuasky	0.040	0.625	15.625	0.585
2	russianinvasion-->standwithukraine	0.040	1.000	9.804	0.898
3	standwithukraine-->russianinvasion	0.040	0.392	9.804	0.352
4	russianinvasion-->ukrainewillwin	0.040	1.000	22.727	0.956
...	...	...	...	...	...
149	energy-->mining	0.020	1.000	50.000	0.980
150	mining-->coal	0.020	1.000	50.000	0.980
151	coal-->mining	0.020	1.000	50.000	0.980
152	ukrainerussiawar-->ukraine	0.036	0.545	1.274	0.117
153	ukraine-->ukrainerussiawar	0.036	0.084	1.274	0.018

154 rows × 5 columns

Figure 12: dataset containing the values of each rule for support, confidence, lift and interest.

5 Scalability and Performance

For sake of simplicity, in order to be able to perform multiple and fast experiments, We used a really small part of our dataset: just 500 baskets. After having developed the algorithms using this small portion of data, We tried to run them on bigger baskets file and with different threshold values.

In Figure 13 and Figure 14 are reported the results. We can clearly see that, as expected, the computational time decrease when the threshold value increase. This because, for example in Apriori, We filter out more singletons in between the two passes and as consequence there will be less pairs to check in the second pass. Figure 14 shows also the relation between the basket file size and the time. The time grows as the basket file to be analyzed grows. From these plots We can clearly see that PCY, Multisatge and Multihash algorithm are the ones which performs better. For example PCY,Multistage and Multihash take all less than 50 seconds to collect the frequent pairs on the entire basket file (254.626 baskets) with threshold=1.000.

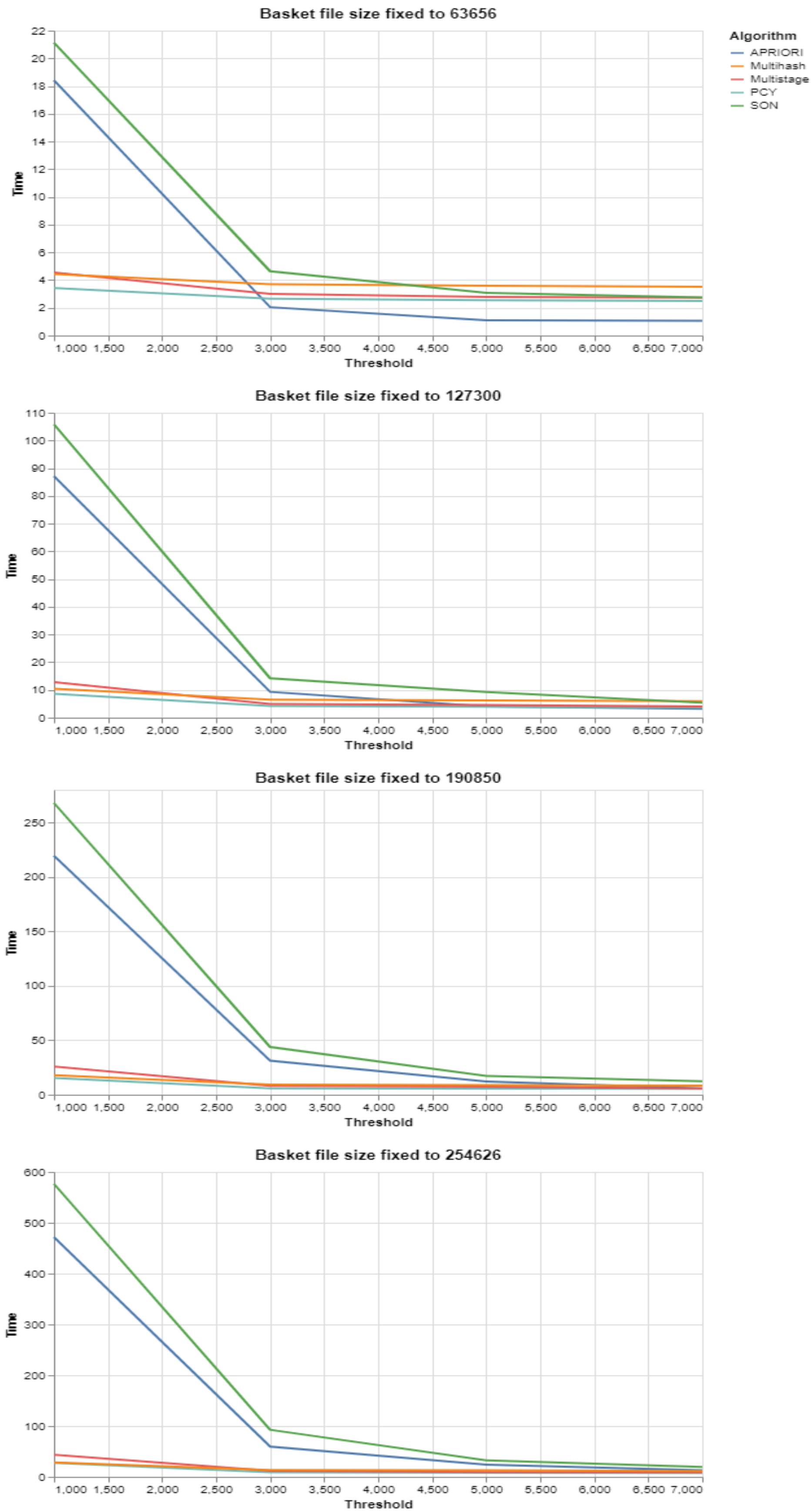

Figure 13: Relation between the threshold value and the time (given a fixed size for the basket file.

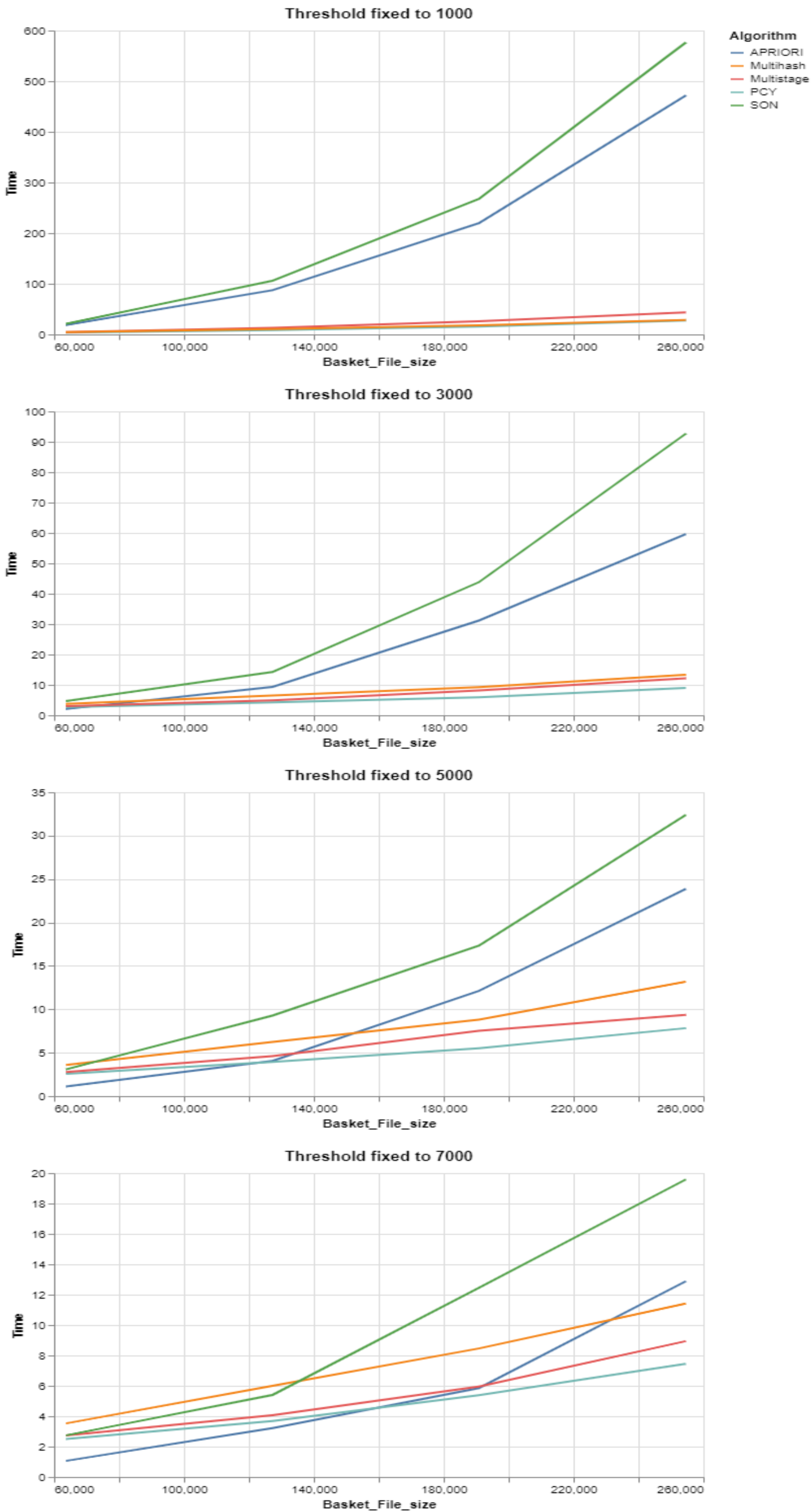

Figure 14: Relation between the basket file size and the time (given a fixed value for the threshold.

6 Conclusion

The goal of this project was to find frequent sets inside our basket file. We decided to concentrate on pairs but the algorithm can be extended in order to find frequent itemsets of higher cardinality. Anyway, We accomplished what we wanted: finding freq pairs.

We started extracting the needed information from the given dataset: the baskets (the tweets) and their items (the hashtags). Then We implemented the algorithm discussed is section 3 and all of them converged to the same results. In section 5 We also showed how the implementation behave on a larger dataset with a different thresholds.

At the end We applied some metrics on the association rules generated by the frequent pairs founded with PCY. For sake of simplicity We analyzed the association rules on the "toy dataset" of 500 baskets analyzed in the first part of our experiments.

References

[1] Mining of Massive Datasets, written by A. Rajaraman e J. Ullman (chapter 6.2.5 "The A-Priori Algorithm")

[2] Mining of Massive Datasets, written by A. Rajaraman e J. Ullman (chapter 6.3 "Handling Larger Datasets in Main Memory")

[3] Market Basket Analysis: Understanding Customer Behaviour written by Lynsey McColl (https://select-statistics.co.uk/blog/market-basket-analysis-understanding-customer-behaviour/)