

UNIVERSITA' DEGLI STUDI DI MILANO

DATA SCIENCE AND ECONOMICS (DSE)



Algorithms for massive data

Prof. Dario Malchiodi

Project 2

Stefano Zanlucchi
Matr.No. 22288A

Accademic Year 2022-2023

Contents

1	PROJECT n.2	1
1.1	Introduction and goals	1
1.2	Dataset description	1
1.3	Dataset organization	2
1.4	Pre-processing techniques	2
1.5	Algorithms	3
1.5.1	A-priori	4
1.5.2	Park, Chenk and Yu	5
1.6	Scalability of the solution	6
1.7	Experiments description	6
1.7.1	Step by step description	7
1.7.2	Experiments results	10
1.8	Comments and discussion on the experimental results	13
	Bibliography	15

List of Figures

1.1	Overview of columns of the file <code>full-data.csv</code>	2
1.2	Data preparation	3
1.3	A-priori: first and second pass	4
1.4	A-priori: patter for k-size frequent items search	5
1.5	A-priori: output of frequent singletons and pairs	7
1.6	A-priori: output of frequent singletons and pairs	7
1.7	A-priori: output of frequent triples count	8
1.8	PCY: Generate all the pairs and count the occurances	8
1.9	PCY: Count the pairs and add index	8
1.10	PCY: Hash pairs to bucket and keep frequent one	9
1.11	PCY: Summarize hash table as a bitmap a keep records related to frequent bucket with bit value = 1	9
1.12	PCY: Generation of pairs from frequent singletons	9
1.13	PCY: Reorganization of frequent pairs with link to frequent bucket (bit 1)	10
1.14	PCY: Final output of frequent pairs	10
1.15	A-priori: Lists in output of frequents pairs for comparison	11
1.16	PCY: explanation of average bucket count in the dataset	12
1.17	PCY: explanation of tuning of parameters in the dataset	12
1.18	Frequent pairs comparison sorted by occurrences	13

List of Tables

- 1.1 Results of A-priori, summarized for all the types of dataset considered . . . 11
- 1.2 Results of PCY and A-Priori algorithm on the modified sample dataset . . 13

1.1 Introduction and goals

The problem that I would like to take into account is the study of frequent itemsets in a dataset, which is common in marketing for example in online retailers as Amazon to recommend stuffs to customers, focusing on frequent products and itemsets.

The model to be used is the market basket analysis (abbreviation *MBA*) which is a many-to-many relationship between items and baskets. Each basket contains a set of the items and the number of items in a basket is usually smaller than the total amount of items in the data set.

The main purpose is to find the sets of items that appear as frequent and to do that we will focus on two main algorithm:

- A-priori
- Park Chen Yu (PCY)

These algorithms were explained during lesson of this course [2] and in the reference textbook [1].

1.2 Dataset description

In this analysis we use the *MeDAL* dataset [4], which is a large medical text used for the task of medical abbreviation disambiguation and Natural language techniques.

From this dataset we consider the file `full-data.csv` focusing just on the column `TEXT`, in which each row corresponds to abstract of an article. We consider each string contained in the `TEXT` columns as a basket and the words as items of the basket itself. In this way we are able to replicate the standard set-up to apply the market basket analysis.

To evaluate the correctness of the proposed approaches developed in the algorithm and to clarify how they work, instead of the full dataset I consider a smaller one presenting intermediate outputs of the steps of the algorithms. Later I call this simplified dataset as *sample dataset*.

1.3 Dataset organization

In order to perform this analysis we use following tools:

- Python and Pyspark module
- GoogleColab notebook

We start downloading of the dataset directly from kaggle link. Selecting only our relevant dataset `full-data.csv`, this file shows in fig.1.1:

- number of records or transaction: 14.393.620
- following 3 attributes: **TEXT** is the string of words of the abstract of the under analysis, **LOCATION** refers to the location index of the abbreviation, **LABEL** shows the words substituted at the given location
- the only relevant attribute for our project is the column **TEXT**

text string	location sequence	label sequence
alphanisabolol has a primary antipeptic action depending on dosage which is not caused by an alteration of the phvalue the proteolytic activity...	[56]	["substrate"]
a report is given on the recent discovery of outstanding immunological properties in ba ncyanoethyleneurea having a low molecular mass m...	[24, 49, 68, 113, 137, 172...	["carcinosarcoma", "recovery", "reference",...
the virostatic compound nddiethyloxotetradecylimidazolidinylethylpiperazinecarboxamidehydrochlo...	[55]	["substrate"]
rmi rmi and rmi are newly synthesized nrdibenzobfoxepinylnmethylpiperazinemaleates which show interesting...	[25, 82, 127, 182, 222]	["compounds", "compounds", "inhibitory", "lethal doses",...
a doubleblind study with intraindividual comparisons was carried out to investigate the effects of mg of ralphahydroxyisopropylalphatropanium...	[22, 26, 28, 77, 90, 144,...	["oxazepam", "placebo", "oral administration",...

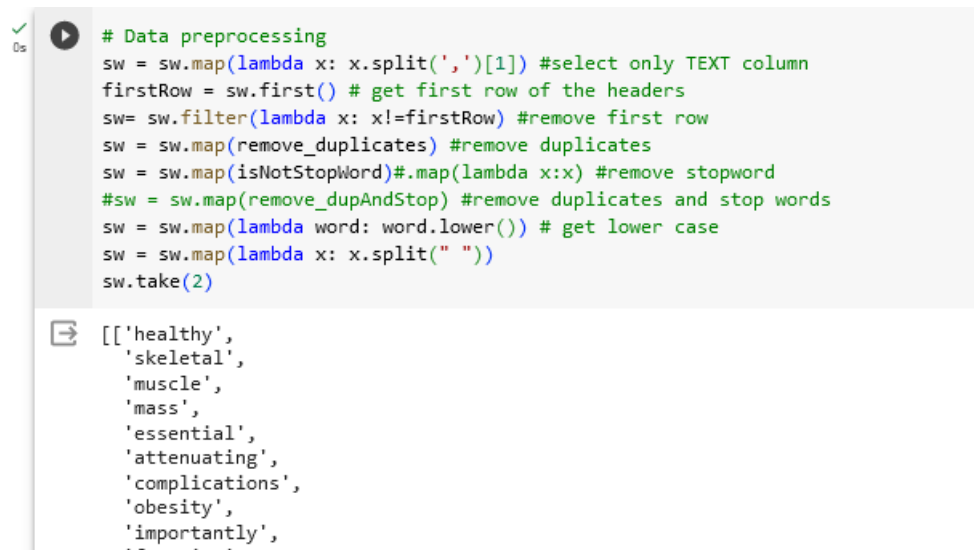
Figure 1.1: Overview of columns of the file `full-data.csv`

The *sample dataset* is taken instead from the `test.csv` dataset and there are two ways of use that we will clarify later, but basically they represent a smaller or very smaller part of the full dataset.

1.4 Pre-processing techniques

In order to analyze correctly the **TEXT** column we recommend that some important remarks are taken into account as shown also in line of code of fig.1.2:

1. according to market-basket model in principle items can appear only one in each basket. Since there could be the repetition of the same word in each string of text that is considered as basket, we need to get rid of these possible multiple copies of the items and we developed a short function called `remove_duplicates`
2. in strings of text there are included stop words, that is a set of words (as "a", "the", etc.) that do not bring much useful information. For this purpose there we created a short function called `IsNotStopWords` based on the natural language processing kit of `nltk` library
3. string of words could include items that are different just due to upper or lower case sensitivity, so capitalization should be ignored. In the dataset the column `TEXT` considers already normalized words, but abbreviation are distinguished with upper case digits
4. before feeding the dataset to the algorithms there is just a final re-organization of each basket as a list of words instead of keeping them as a string of words



```
# Data preprocessing
sw = sw.map(lambda x: x.split(',')[1]) #select only TEXT column
firstRow = sw.first() # get first row of the headers
sw= sw.filter(lambda x: x!=firstRow) #remove first row
sw = sw.map(remove_duplicates) #remove duplicates
sw = sw.map(isNotStopWord)#.map(lambda x:x) #remove stopword
#sw = sw.map(remove_dupAndStop) #remove duplicates and stop words
sw = sw.map(lambda word: word.lower()) # get lower case
sw = sw.map(lambda x: x.split(" "))
sw.take(2)
```

```
[[ 'healthy',
   'skeletal',
   'muscle',
   'mass',
   'essential',
   'attenuating',
   'complications',
   'obesity',
   'importantly',
   ...]]
```

Figure 1.2: Data preparation

1.5 Algorithms

Before digging into the algorithms explanations, we can start presenting some useful definitions related to this study:

- support: given a set of items called I , its support is the number of baskets for which I is a subset

Algorithms for massive data

- frequent itemsets: it is a set of items whose occurrence in the baskets is above a defined threshold called *support threshold*. There could be frequent singleton, doubleton, triples and so on

We can choose the support of I reasonably high as 1% or 2% of the total number of baskets and it is common to adjust this threshold in order to get a reasonable number of frequent itemsets in output for the analysis.

Another relevant aspect is the processing time:

- the workload to analyse each basket is proportional to the size of the file
- baskets are read sequentially, so each algorithm can be characterized by

$$\text{number-of-passes-through-baskets} * \text{file-size} \quad (1.1)$$

The amount of data are always given, so the algorithms proposed will focus only on the number of passes.

1.5.1 A-priori

A-priori is the first algorithm to implemented in this project and its main steps are shown in figure 1.3 and described here below:

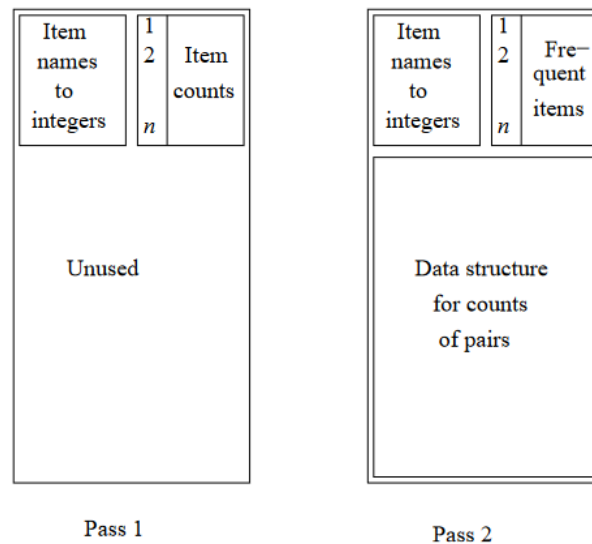


Figure 1.3: A-priori: first and second pass

1. 1st pass: count each item as we go ahead reading the baskets
 - after the first pass we want to know which are the frequent items (singletons) with respect to the support threshold that are used to fill in the frequent-items table

2. 2nd pass: follows the tasks

- in each basket we filter only items which are in the frequent-items table
- we generate all combination of frequent items in each basket
- we add 1 to each pair and then sum the pairs up to count the occurrences
- in the end we look at counter associated to each pairs, to get only frequent ones

A-priori algorithm relies on the monotonicity of the items: so if a set I is frequent, then also its subset are frequent. On the contrary if a set I is not frequent, then all the possible sets that contains I are infrequent.

We are often interested in small frequent itemsets up to the size 2 or 3, but A-priori algorithm can carry on the search for all the frequent sets of larger sizes that are present in the baskets. In this case the pattern to move from one size of the items to another bigger can be summarized through the figure 1.4.

In the project we will stop at size equal 3 for simplicity and based on the theory we know that the biggest requirement for the main memory is to calculate candidate pairs and find the frequent pairs.

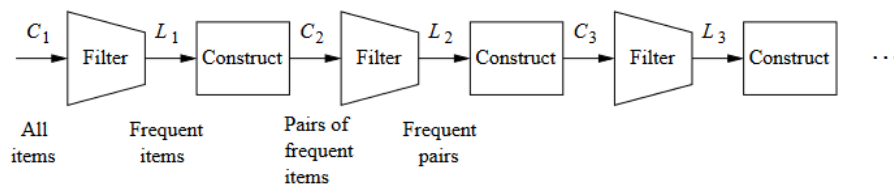


Figure 1.4: A-priori: pattern for k-size frequent items search

1.5.2 Park, Chenk and Yu

This algorithm starts with the awareness that during the first pass of A-priori the memory is mostly unused. The idea is to change in the following way:

1. 1st pass:

- we count the frequent items (singletons) as in the A-priori
- we generate all the pairs in a basket
- we hash each pair to a bucket and increase the count with regards to the hashed bucket
- sum all the number of pairs that are hashed onto a bucket
- define the set of frequent buckets above the threshold

2. between the 2 passes the hash table is summarized with a bitmap, an array that has the length of the hash table with value = 1 if the bucket is frequent and = 0 if is infrequent. So we complete the 2nd pass as A-priori, but using triples method, and in the result obtained are the frequent pairs that we are looking for

The main idea of this algorithm is that if a bucket results to be frequent, it means that the sum of counts of pairs that are hashed to the bucket is bigger than the threshold.

The benefit to use this PCY algorithm is that in the second pass there is a possible reduction in memory workload because most of the buckets are labelled as infrequent.

1.6 Scalability of the solution

The presented algorithms can work also when:

- the size of the file cannot fit the main memory
- the dataset is not stored on a single machine

In fact in case of large scale data we need to use tools as:

- Distributed File System (DFS): to store the file dividing it into chunks across several nodes or computers with redundancy
- MapReduce techniques: a style of computing to process large-scale of data

In the project we use Spark and its main pillar in Spark is the *Resilient Distributed Dataset (RDDs)* which is an immutable object that can be distributed across multiple nodes. It is called resilient because it allows to recover missing partitions due to failures, distributed because of the concept of multiples nodes linked into a cluster.

The algorithms make use of RDDs objects, on which two type of operations can be executed:

- transformation: are applied on RDDs to return other RDDs (e.g. map, filter, ...)
- actions: are used to compute a result in output (e.g. collect, reduce, take, ...)

These operation are widely used in the Python code and this allows us to scale up to large-scale data, in deed data is divided into smaller chunks, called partitions, and processed in parallel across multiple nodes in a cluster.

1.7 Experiments description

In this section I would like to present the results of the algorithms dividing the project into two parts. In the first part it is shown a step by step description of the operations of the algorithms performed on the a very smaller part of the *sample dataset*. So we can gain confidence on the intermediate passes to reach the implementation of a frequent itemsets search. In the latter instead the experiment is performed both on the *sample dataset* and the full dataset The scripts used for this project are made available into Google Colab Notebook [3].

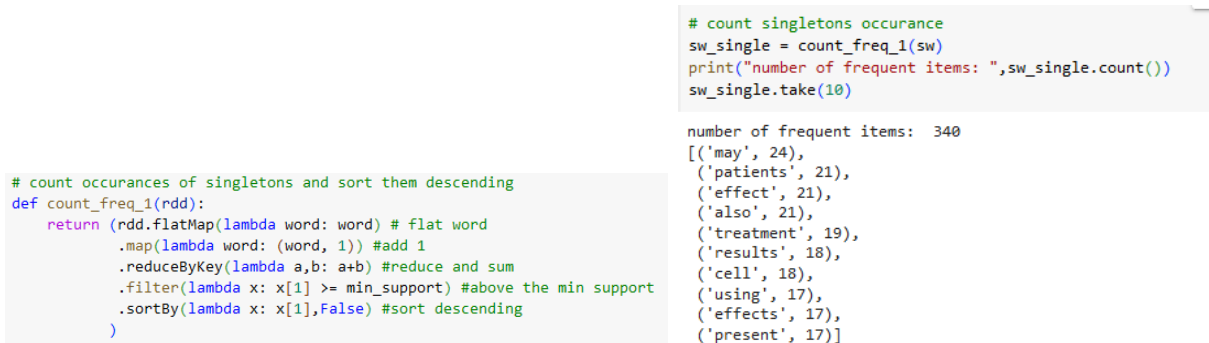
1.7.1 Step by step description

For simplicity the very small *sample dataset* used to tune the functions and the code in the project is considering only 87 baskets and a 5% of the number of the baskets as support threshold.

A-priori

With the support of some developed function for RDDs that implements A-priori assumption, we can find at the frequent singletons as shown in fig.1.5.

In the same way we can go ahead finding frequent pairs as obtained in in fig.1.6. We use



```
# count occurrences of singletons and sort them descending
def count_freq_1(rdd):
    return (rdd.flatMap(lambda word: word) # flat word
            .map(lambda word: (word, 1)) # add 1
            .reduceByKey(lambda a,b: a+b) # reduce and sum
            .filter(lambda x: x[1] >= min_support) #above the min support
            .sortBy(lambda x: x[1],False) #sort descending
    )

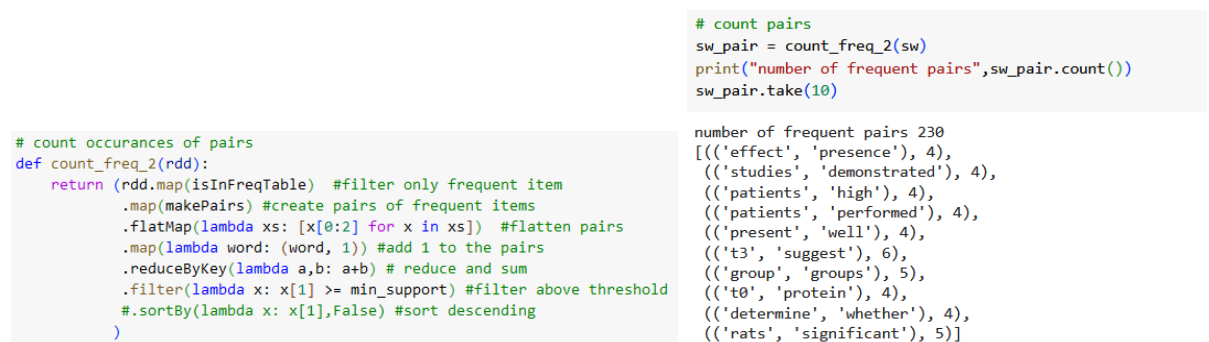
# count singletons occurrence
sw_single = count_freq_1(sw)
print("number of frequent items: ",sw_single.count())
sw_single.take(10)
```

```
number of frequent items: 340
[('may', 24),
 ('patients', 21),
 ('effect', 21),
 ('also', 21),
 ('treatment', 19),
 ('results', 18),
 ('cell', 18),
 ('using', 17),
 ('effects', 17),
 ('present', 17)]
```

Figure 1.5: A-priori: output of frequent singletons and pairs

two additional function: `isInFreqTable` to check if the item in the basket belongs to the frequent singletons and then after this filter we can create pairs of frequent items using the function `makePairs` that implements a double loop.

Since we are also interested in finding frequent itemsets up to size 3, it means that we



```
# count occurrences of pairs
def count_freq_2(rdd):
    return (rdd.map(isInFreqTable) #filter only frequent item
            .map(makePairs) #create pairs of frequent items
            .flatMap(lambda xs: [x[0:2] for x in xs]) #flatten pairs
            .map(lambda word: (word, 1)) #add 1 to the pairs
            .reduceByKey(lambda a,b: a+b) # reduce and sum
            .filter(lambda x: x[1] >= min_support) #filter above threshold
            .sortBy(lambda x: x[1],False) #sort descending
    )

# count pairs
sw_pair = count_freq_2(sw)
print("number of frequent pairs",sw_pair.count())
sw_pair.take(10)
```

```
number of frequent pairs 230
[('effect', 'presence'), 4),
 (('studies', 'demonstrated'), 4),
 (('patients', 'high'), 4),
 (('patients', 'performed'), 4),
 (('present', 'well'), 4),
 (('t3', 'suggest'), 6),
 (('group', 'groups'), 5),
 (('t0', 'protein'), 4),
 (('determine', 'whether'), 4),
 (('rats', 'significant'), 5)]
```

Figure 1.6: A-priori: output of frequent singletons and pairs

want to find also frequent triples. This result is shown in fig.1.7 and also for this pass we use two additional functions: `isInFreqTable` to determine if item of the baskets belong to frequent pairs and `makeTriples` to create combination of 3 items of these frequent items.

Algorithms for massive data

```
# function to count occurrences of triples
def count_freq_3(rdd):
    return (rdd.map(isInPairTable) #filter only frequent pairs
            .map(makeTriple) #makeComb
            .flatMap(lambda xs: [x[0:3] for x in xs]) #flatten triples
            .map(lambda word: (word, 1)) #add 1 to triples
            .reduceByKey(lambda a,b: a+b) #reduce and sum
            .filter(lambda x: x[1] >= min_support) # above support threshold
            )

# count triples
sw_triples = count_freq_3(sw)
print("number of frequent triples",sw_triples.count())
sw_triples.take(10)

number of frequent triples 4
[[('determine', 'whether', 'could'), 4),
 (('effects', 'also', 'results'), 4),
 (('effects', 'mgkg', 'significant'), 4),
 (('also', 'cell', 'may'), 4)]
```

Figure 1.7: A-priori: output of frequent triples count

Park Chen Yu

In this section I would like to present the intermediate outputs that can be obtained implementing the PCY algorithm.

First we should start as with A-priori finding the frequent singletons and with the unused memory also generate all the combinations of pairs in each baskets as shown in fig.1.8.

Since we have all the pairs generated, we count the pairs and prepare them before the

```
# 1.0 count items and find frequent one as A-priori
freq1 = count_freq_1(sw)
freq1_out = freq1.map(lambda x: x[0])
print('number of frequent items',freq1.count())
freq1.take(10)

number of frequent items 340
[('healthy', 5),
 ('sc', 5),
 ('chain', 4),
 ('significantly', 13),
 ('result', 4),
 ('used', 11),
 ('presence', 14),

# 1.a generate all pairs while examining each basket
nn_comb = 2
pairs = sw.map(lambda x: list(combinations(x,nn_comb))).
print("frequent pairs",pairs.count())
pairs.take(5)

frequent pairs 313016
[('healthy', 'skeletal'),
 ('healthy', 'muscle'),
 ('healthy', 'mass'),
 ('healthy', 'essential'),
 ('healthy', 'attenuating')]
```

Figure 1.8: PCY: Generate all the pairs and count the occurrences

hashing, indeed we need some sort of index to hash the pairs into a bucket, this is why we are using the transformation `.zipWithUniqueId()` and this step is in fig.1.9.

Now we are ready to hash pairs to the buckets with a very simple hash function as

```
# 1.b: count each pair in the baskets
count_pairs = pairs.map(lambda word: (word,1)) # add 1 to each pair
count_pairs = count_pairs.reduceByKey(lambda a,b:a+b).sortBy(lambda
count_pairs.take(10)

[ (('also', 'results'), 8),
  (('effect', 'results'), 7),
  (('also', 'suggest'), 7),
  (('t3', 'suggest'), 6),
  (('evaluated', 'results'), 6),
  (('presence', 'low'), 6),

# 1.c: before hashing pairs to buckets, attached an index
comb_index=count_pairs.zipWithUniqueId().sortBy(lambda x: x[1],True)
#comb_index=count_pairs.zipWithIndex() #in alternative
comb_index.take(20)

[ (('also', 'results'), 8), 0),
  (((('healthy', 'skeletal'), 1), 3),
  (((('effect', 'results'), 7), 4),
  (((('healthy', 'investigate'), 1), 7),
  (((('also', 'suggest'), 7), 8),
  (((('healthy', 'population'), 1), 11),
  (((('t3', 'suggest'), 6), 12),
  (((('healthy', 'fed'), 1), 15),
  (((('evaluated', 'results'), 6), 16),
```

Figure 1.9: PCY: Count the pairs and add index

presented in fig.1.10. In this figure it is clarified also the next step: we can rearrange the RDD in such a way that we can sum the counts of all the pairs that hash to a specific bucket. It is clear that all the buckets are frequent buckets, so all the pairs that hash to the buckets are considered frequent pairs based on the data that we have.

Few last steps are needed in the first pass: reorganize the RDD in such a way that the bucket number is summarized with a bit instead of an integer (1 if the bucket is frequent,

Algorithms for massive data

```
# 1.c: before hashing pairs to buckets, attached an index
comb_index=count_pairs.zipWithUniqueId().sortBy(lambda x: x[1],True)
#comb_index=count_pairs.zipWithIndex() #in alternative
comb_index.take(20)

[(((('also', 'results'), 8), 0),
  (((('healthy', 'skeletal'), 1), 3),
  (((('effect', 'results'), 7), 4),
  (((('healthy', 'investigate'), 1), 7),
  (((('also', 'suggest'), 7), 8),
  (((('healthy', 'population'), 1), 11),
  (((('t3', 'suggest'), 6), 12),
  (((('healthy', 'fed'), 1), 15),
  (((('evaluated', 'results'), 6), 16),
  ...))

# 1.d keep only bucket_id, value and reduce
buck_freq=pairs_buck.map(lambda t:(t[0],t[2])).reduceByKey(lambda a,b:a+b)
# 1.e filter frequent buckets
buckFreq = buck_freq.filter(lambda x:x[1]>=min_support).sortByKey()
buckFreq.take(20)

[(0, 14910),
 (1, 14904),
 (2, 14905),
 (3, 14906),
 (4, 14909),
 (5, 14904),
 (6, 14905),
 (7, 14906),
 (8, 14909),
 (9, 14903),
 (10, 14905),
```

Figure 1.10: PCY: Hash pairs to bucket and keep frequent one

0 otherwise) and then filter the RDD with keeping only records with bit = 1. This is what fig.1.11 shows.

The second pass start considering candidate pairs those pairs that have: both items

```
# 1.f: get list of freq pairs
freq_buck = buckFreq.map(lambda x: x[0]).collect()
# 1.g: prepare as bitmap: 1 if in freq bucket table 0 otherwise
bitmap = pairs_buck.map(lambda a:((a[1],a[2]),1 if a[0] in freq_buck else 0))
bitmap.take(20)

[(((('also', 'results'), 8), 1),
  (((('healthy', 'skeletal'), 1), 1),
  (((('effect', 'results'), 7), 1),
  (((('healthy', 'investigate'), 1), 1),
  (((('also', 'suggest'), 7), 1),
  (((('healthy', 'population'), 1), 1),
  (((('t3', 'suggest'), 6), 1),
  (((('healthy', 'fed'), 1), 1),
  (((('evaluated', 'results'), 6), 1),
  ...))

# 1.h: reorganize bitmap with bucket_id, pair, bit value
bitmap = bitmap.map(lambda a:(a[0][0],a[0][1],a[1]))
# 1.i: keep pairs and bitmap value = 1
bit1= bitmap.map(lambda x:(x[0],x[1],x[2]))
bits_1 = bit1.filter(lambda x: (x[1][1] ==1)) # filter bit=1
bits_1.take(10)

[(((('also', 'results'), (8, 1)),
  (((('healthy', 'skeletal'), (1, 1))),
  (((('effect', 'results'), (7, 1))),
  (((('healthy', 'investigate'), (1, 1))),
  (((('also', 'suggest'), (7, 1))),
  (((('healthy', 'population'), (1, 1))),
  (((('t3', 'suggest'), (6, 1))),
  (((('healthy', 'fed'), (1, 1))),
  ...))
```

Figure 1.11: PCY: Summarize hash table as a bitmap a keep records related to frequent bucket with bit value = 1

belonging to the frequent singletons and the pairs hashes to a frequent bucket. So we start generating the pairs of frequent singletons as shown in fig.1.12. In this case we have just modified the function to create pairs with one trick to have both possible orders of pairs from the combination that we can generate, this is way the generated pairs are double with respect to the binomial coefficient $\binom{N}{2}$, where N is the number of frequent singletons.

We can join these new pairs with the already available RDD of 1st pass which kept the

```
def makePairs2(x):
    # function to create pairs of frequent singletons
    # using nested loops with all possible order
    res = []
    n = len(x)
    for i in range(n):
        for j in range(i+1, n):
            res.append((x[i], x[j]))
            res.append((x[j], x[i]))
    return res

# 2.0: take frequent items and make pairs
tt = makePairs2(freq1_out.collect())
# it is a list that need to be transformed into rdd
rdd1 = sc.parallelize(tt)
rdd1 = rdd1.map(lambda x: ((x[0],x[1]),1))
print("Number of pairs generated with freq.items:", rdd1.count())
rdd1.take(2)

Number of pairs generated with freq.items: 115260
[(((('healthy', 'sc'), 1), ((('sc', 'healthy'), 1))
```

Figure 1.12: PCY: Generation of pairs from frequent singletons

rows with bit = 1 of frequent bucket. If we do some rearrangement and in the end we filter the value of pairs linking to the minimum support then we get the frequent pairs (see fig.1.13).

Usually we would expect to take advantage from the use of frequent buckets coming from

```
# 2.a: link the possible frequent pairs to bitmap with 1 in bit
freq2_bitmap = rdd1.join(bits_1) #join pairs (a,b) with their bucket_id
freq2_bitmap.take(10)

[ (('healthy', 'presence'), (1, (1, 1))),
  (('healthy', 'risk'), (1, (1, 1))),
  (('healthy', 'years'), (1, (1, 1))),
  (('healthy', 'identify'), (1, (1, 1))),
  (('healthy', 'mg'), (1, (1, 1))),
  (('assess', 'healthy'), (1, (1, 1))),
  (('assess', 'risk'), (1, (1, 1))),
  (('assess', 'years'), (1, (1, 1))),
  (('assess', 'identify'), (1, (1, 1))),
  (('assess', 'mg'), (1, (1, 1))) ]

# 2.b: reorganize freq2 rdd keeping pairs and count
freq2_bitmap = freq2_bitmap.map(lambda x: (x[0], x[1][1][0]))
freq2_bitmap = freq2_bitmap.filter(lambda x: (x[1] >= min_support))
freq2_bitmap.take(10)

[ (('rats', 'significantly'), 4),
  (('significantly', 'compared'), 4),
  (('c2', 'significantly'), 4),
  (('significantly', 'increased'), 4),
  (('level', 'significantly'), 5),
  (('presence', 'showed'), 4),
  (('presence', 'low'), 6),
  (('ine', 'raised'), 4) ]
```

Figure 1.13: PCY: Reorganization of frequent pairs with link to frequent bucket (bit 1)

the first pass, but this is not the case, because all buckets are frequent. We are exactly in the worst case in which the PCY Algorithm counts exactly the same number of pairs that A-priori does as well on the second pass, the figure 1.14 shows it.

```
# 2.c: get frequent itemsets
freq_2 = freq2_bitmap.map(lambda x : x[0])
print("number of frequent pairs", freq_2.count())
freq_2.take(5)

number of frequent pairs 230
[ ('rats', 'significantly'),
  ('significantly', 'compared'),
  ('c2', 'significantly'),
  ('significantly', 'increased'),
  ('level', 'significantly') ]
```

Figure 1.14: PCY: Final output of frequent pairs

1.7.2 Experiments results

In this part we present the results of the implemented algorithms: first of all looking at the A-priori algorithm and then at PCY.

We used Google Colab Compute Engine backend and on the dataset used except of the one use in the step by step description, the support threshold used is the 2% of the number of the basket, it means 287872.

A-priori

We start looking at the table 1.1, that presents the overview of the algorithm performances and results obtained with the A-priori algorithm.

Sincerely, I was not expecting so high amount of time needed to perform such computations, but since we are dealing with more that 14 million of transactions or baskets, then it should reasonable to have such a memory requirement. So we arm ourselves with patience and we arrive at the completion of the analysis after a lot of hours. The longer amount of time is needed to perform actions, indeed during A-priori steps we need to collect the frequent items and store them in a list. As we were expecting also most time demanding step is needed to find frequent pairs, which is what also theory says.

So it seems reasonable to look at the lists obtained in outputs from sample dataset of 10000 baskets and the one calculated on the full dataset (see fig.1.15). They should be quite similar detecting similar amount of frequent pairs.

Algorithms for massive data

	A-priori step	A-priori 10000	A-priori full
% of min. support	5%	2%	2%
T calc. freq. singletons [s]	n.a.	3.3	3003.8
T calc. freq. pairs [s]	n.a.	36.8	25835.3
T calc. freq. triples [s]	n.a.	40.3	13281.6
N. of freq. singletons	340	729	649
N. of freq. pairs	230	443	180
N. of freq. triples	4	0	0

Table 1.1: Results of A-priori, summarized for all the types of dataset considered

algo_Apriori2_output.txt	algo_Apriori-10000_2_output.txt
1 ('study', 'observed')	1 ('group', 'p')
2 ('human', 'results')	2 ('patients', 't3')
3 ('human', 'may')	3 ('different', 'also')
4 ('patients', 'two')	4 ('also', 'data')
5 ('significant', 'results')	5 ('t3', 'increased')
6 ('significantly', 'may')	6 ('analysis', 'results')
7 ('t0', 'results')	7 ('different', 'may')
8 ('study', 'analysis')	8 ('three', 'results')
9 ('found', 'results')	9 ('p', 'results')
10 ('results', 'also')	10 ('results', 'suggest')
11 ('effects', 'effect')	11 ('observed', 'may')
12 ('using', 'significant')	12 ('may', 'also')
13 ('patients', 'significantly')	13 ('two', 'one')
14 ('patients', 'years')	14 ('significantly', 'p')
15 ('study', 'patients')	15 ('one', 'two')
16 ('using', 'showed')	16 ('significant', 'observed')
17 ('cells', 'found')	17 ('present', 'using')
18 ('control', 'results')	18 ('cells', 'results')
19 ('patients', 'one')	19 ('cells', 'expression')
20 ('treatment', 'results')	20 ('rats', 'results')
21 ('patients', 'study')	21 ('using', 'results')
22 ('cell', 'cells')	22 ('significant', 'also')
23 ('play', 'role')	23 ('also', 'observed')
24 ('protein', 'also')	24 ('also', 'however')
25 ('found', 'may')	25 ('activity', 'results')
26 ('cell', 'may')	26 ('used', 'results')
27 ('using', 'compared')	27 ('used', 'significantly')
28 ('patients', 'results')	28 ('t0', 'higher')
29 ('using', 'found')	29 ('t0', 'observed')
30 ('using', 'used')	30 ('present', 'cells')
31 ('patients', 'however')	31 ('present', 'suggest')
32 ('study', 'effect')	32 ('cells', 'significantly')
33 ('clinical', 'patients')	33 ('showed', 'results')

Figure 1.15: A-priori: Lists in output of frequents pairs for comparison

PCY

Before digging into the experiments of PCY Algorithm we need to make an important observation about the fact that such a dataset creates a lot of difficulties in computations because in the first pass we would create the pairs from words in each basket, this would create a very big hash table:

- $\frac{n_{words}(n_{words}-1)}{2}$, where n_{words} is the number of words in each string of text, so in every basket
- $n_{baskets} \frac{n_{words}(n_{words}-1)}{2}$, If we multiply it by the number of baskets $n_{baskets}$, we can estimate the total number of possible pairs
- $\frac{n_{baskets} n_{words}^2}{2n_{buckets}}$, where we approximate $n_{words}(n_{words}-1) \approx n_{words}^2$ and we divide by the number of buckets $n_{buckets}$, so we get the average count that each bucket would get

Algorithms for massive data

This value is much higher than the support threshold so this is way we are not filtering out any buckets in the first pass.

Therefore we need to make an assumption forcing each string of text to have at most 5 words. Also lowering the support threshold helps to go to the right direction as shown in fig.1.16 and 1.17. In deed we have an average count of bucket that is higher than the threshold.

When we run the PCY algorithm on such a modified dataset we get at last some

<pre>n_words = 85 #n.of words in each string of text n_buck = math.floor(n_baskets/4) #number of buckets perc_min_supp = 0.02 min_support = round(perc_min_supp*n_baskets) #number of possible pairs poss_pairs = n_baskets*(n_words*(n_words-1)/2) #avg count of each bucket avg_buck_count= (n_baskets*n_words**2)/(2*n_buck) print('avg count of each bucket',avg_buck_count) s_exp = (n_baskets*n_words**2)/(2*min_support) print('Number of buckets expected:',s_exp)</pre> <p>avg count of each bucket 14450.0 Number of buckets expected: 180625.0</p>	<pre>n_words = 5 #n.of words in each string of text n_buck = math.floor(n_baskets/4) #number of buckets perc_min_supp = 0.0025 min_support = round(perc_min_supp*n_baskets) #number of possible pairs poss_pairs = n_baskets*(n_words*(n_words-1)/2) #avg count of each bucket avg_buck_count= (n_baskets*n_words**2)/(2*n_buck) print('avg count of each bucket',avg_buck_count) s_exp = (n_baskets*n_words**2)/(2*min_support) print('Number of buckets expected:',s_exp)</pre> <p>avg count of each bucket 50.0 Number of buckets expected: 5000.0</p>
--	--

Figure 1.16: PCY: explanation of average bucket count in the dataset

```
# important adjustment to limit word in strings of text
sw = sw.map(lambda x: x[0:5])

#define min support and number of buckets
n_baskets = math.floor(n_baskets/4)
#perc_min_supp = 0.02
perc_min_supp = 0.0025
min_support = round(perc_min_supp*n_baskets) #filter the value of t
#min_support = 5
print("Number of baskets considered in this project",n_baskets)
print("n_baskets for hashing:", n_baskets)
print("Min support considered is",f"{perc_min_supp:.0%}", "of the t
print("Total value of the min support",min_support)
```

Number of baskets considered in this project 10000
n_baskets for hashing: 2500
Min support considered is 0% of the total number of baskets
Total value of the min support 25

Figure 1.17: PCY: explanation of tuning of parameters in the dataset

reasonable results considering also the computation time and we can see that during 1st pass some buckets are infrequent. In addition we can think to use A-priori on the modified sample dataset, and in the table 1.2 there is this overview that show the comparison.

In conclusion we can look for example at the results obtained with the 2 algorithms and as we can see from fig.1.18. The pairs are word linked to the medical world and the purpose of studies done in this area.

	PCY 10000-mod	A-Priori 10000-mod
N. of baskets	10000	10000
N. of buckets	2500	n.a
Supp. threshold	25	25
T. 1st pass [s]	7.88	2.18
T. 2nd pass [s]	0.72	2.22
N. of freq. singletons	275	275
N. of freq. pairs	17	17

Table 1.2: Results of PCY and A-Priori algorithm on the modified sample dataset

Line	PCY Output	A-Priori Output
1	((('aim', 't0'), 58)	((('aim', 't0'), 58)
2	((('present', 'study'), 54)	((('present', 'study'), 54)
3	((('purpose', 'study'), 50)	((('purpose', 'study'), 50)
4	((('aim', 'study'), 47)	((('aim', 'study'), 47)
5	((('report', 'case'), 45)	((('report', 'case'), 45)
6	((('present', 't0'), 44)	((('present', 't0'), 44)
7	((('yearold', 'woman'), 40)	((('yearold', 'woman'), 40)
8	((('purpose', 't0'), 38)	((('purpose', 't0'), 38)
9	((('previous', 'studies'), 35)	((('previous', 'studies'), 35)
10	((('yearold', 'man'), 34)	((('yearold', 'man'), 34)
11	((('determine', 'whether'), 32)	((('t0', 'investigate'), 32)
12	((('t0', 'investigate'), 32)	((('study', 'determine'), 32)
13	((('study', 'determine'), 32)	((('determine', 'whether'), 32)
14	((('study', 'evaluate'), 30)	((('study', 'evaluate'), 30)
15	((('growth', 'factor'), 29)	((('growth', 'factor'), 29)
16	((('study', 'investigate'), 27)	((('study', 'investigate'), 27)
17	((('study', 'effect'), 25)	((('study', 'effect'), 25)
18		

Figure 1.18: Frequent pairs comparison sorted by occurrences

1.8 Comments and discussion on the experimental results

In this last paragraph I would like to make some comments on the result that we found, that I summarize in the following list:

1. the step by step analysis of the algorithms help us to understand the main steps of the presented algorithms
2. we use one pass for each size of itemsets (only in A-priori I tried also to find the frequent triples)
3. instead of looking at all the frequent itemsets we could also think about to find most of them with some approximated algorithm as the simple randomized algorithm. This is somehow what we did taking into consideration a sample dataset instead of the full one, believing that the baskets appear in random order in the file. In this case we should remember to adjust the threshold to $0.9ps$ to reduce the occurrence of false negative
4. as we were expecting the larger amount of time is used to calculate frequent pairs

Algorithms for massive data

5. we faced a relevant obstacle in using the PCY Algorithm and we found a work around to see that it works
6. the way how the algorithms are implemented in Spark, it allows us to perform analysis also on large-scale dataset as also shown on the analysis on the full dataset with A-priori

Next steps to further analysis could be:

1. implement a multistage or the multihash algorithms since we have already a good basis to further modify PCY algorithm into the other ones
2. It could be reasonable to develop if possible a recurrent algorithm that allows to calculates all the possible candidates C_k of k size

"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. 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."

Bibliography

- [1] A. Rajaraman J.Leskovec and J. Ullman. *Mining of Massive Datasets*. 2020.
- [2] Prof. Dario Malchiodi. Material from lessons of algorithm for massive data, 2023.
- [3] S.Zanlucchi. Project n.2. https://colab.research.google.com/drive/1fE3yHTfr4vVuUHtHAZI0zT8r_xq3tfS8?usp=drive_link, 2023.
- [4] Zhi Wen, Xing Han Lu, and Siva Reddy. MeDAL: Medical abbreviation disambiguation dataset for natural language understanding pretraining. In *Proceedings of the 3rd Clinical Natural Language Processing Workshop*, pages 130–135, Online, November 2020. Association for Computational Linguistics.