Data Mining project 2020-2021

Extraction of frequently correlated terms in time

Davide Zanella
University of Trento
2nd year CS, ID 211463
davide.zanella-1@studenti.unitn.it

1 INTRODUCTION & MOTIVATION

Nowadays, social networks have a key role in our society and daily routine. A lot of people spend hours every day scrolling posts, photos, and tweets on Facebook, Twitter, or Instagram. This leads to the fact that social networks are becoming the mirror of reality: simply looking at the trending topics or most shared posts we can understand what are the most popular news or scandals.

From a Data Mining point of view, this information can be really useful, in fact, collecting tweets for a sufficient period allows us to obtain a raw but valuable dataset. For example, from it we can extrapolate the trend topics throughout time, the trend topics of a really large period, like years or decade or the most used words by users, and the correlation between words and users. An interesting study from such a dataset could even be related to the lifetime of some topics, from their born to their end.

1.1 My problem interpretation

The purpose of this project is to find, starting from a dataset containing tweets, consistent topics in time. To better understand what the previous statement means, we should focus on the definition of the word topic. A topic is a set of terms, and terms, since we are referring to tweets, are any word present in a Twitter's post, so even hashtags or usernames. By consistent topics in time, it means topics that, when they become popular, they are frequently together. Let's analyze an example to better clarify the concept behind it. The term "earthquake" becomes popular rarely, but most of the time that it is popular, the word "emergency" becomes popular too. Therefore, we want to extrapolate some dependencies between popular terms such as "earthquake" → "emergency". Note that the dependency has a strict direction, so the contrary ("emergency" → "earthquake") is not always true, like in this case.

The results obtained from an algorithm applied over the Twitter dataset could lead to unexpected results, maybe we can discover an underline relationship between two or more normally unrelated words. Specifically, we can find out that the majority of the people on Twitter associate one word with another one, even if they normally are unrelated.

A key point of the problem is the temp dynamic. We should take into consideration that the date and time of the tweet rule the popularity calculation. In fact, we are interested in popularity that is relative to the time.

We can transpose this problem to another domain to view it from a different point. We can think of the tweet as a basket containing products of a supermarket. We collect all the receipts of a supermarket, and with the list of bought products and the date, we have a similar dataset to the original one. Without changing the main algorithm that will solve the problem of this project, we could obtain some meaningful results: the dependency between products in the period when they are mostly requested. For instance, we could discover that, in the Christmas period, when the purchase of Pandoro is frequent, even the Nutella is bought more frequently than in normal periods. This is an invented example, but some unexpected results could help supermarkets to increase some prices instead of others and increase their incomes.

© 2021 Association for Computing Machinery. Published in Proceedings of the ACM Conference, July 2017, ISBN 978-x-xxxx-xxxx-x/YY/MM on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

2 RELATED WORK

To solve the introduced problem, we have to rely mainly on the *Frequent Itemset* algorithms. While designing an algorithm for a Data Mining purpose, we have to respect some constraints regarding the memory used and the time of execution of our algorithm. Both of them are derived from the huge amount of data that we are working on, and moreover, the algorithm should work in a reasonable time, both if we are using a 1K rows dataset or a 1M rows one.

2.1 Frequent itemset algorithms

In this section we will focus on the main algorithms starting from a cumbersome version and ending with a more sophisticated one.

2.1.1 ***Naïve Algorithm.*** This is the simplest possible algorithm to solve this kind of problem. It starts reading every basket and it starts generating all the possible pairs of items contained inside the basket. We can memorize the count for each pair in a triangular matrix or using a dictionary structure. The best approach in terms of memory consumption depends on the number of possible pairs. If less than 1/3 of the possible pairs actually occur then the dictionary is a better solution, even if it requires more bytes to store every single key-value pair. Going from pairs to triples computation, the resources and time needed increase drastically, indeed this algorithm is not recommended most of the times.

2.1.2 ***A-Priori Algorithm.*** This algorithm tries to limit the memory required and it is based on the following idea: if an item i does not appear in s baskets, then all the sets (pairs, triples, etc.) including i will appear in less than s baskets. The flow of the A-Priori algorithm is composed of a number of passes equal to the number of maximum elements per set. 2-pass algorithm:

- **Pass 1**: it starts reading all the baskets and counting the occurrences of each item. At the end of this pass, only the items that appeared at least **s** times are kept;
- **Pass 2**: all the baskets are red again, but this time only the pairs of items where both of them where marked as frequent are counted (count >= **s**). As done in *Pass 1* we filter out the collected frequency numbers and keep only the ones with at least **s** appearances.

Going on with other steps we can increase the maximum number of final elements in the sets.

2.1.3 ***PCY (Park-Chen-Yu) Algorithm.*** This algorithm is an improved version of the *A-Priori* one. It tries to equalize the unbalanced memory usage between the first and the second pass. To do so, it fills the free memory in the first pass with as many buckets as possible, and, while counting the occurrences of an item, it applies a hash function to each possible pair of items. The result of the hash function is used to identify a bucket, and the count inside it is incremented by 1. We can use the counts inside the buckets to determine which pairs to discard, indeed if a bucket has a frequency less than **s**, then all the pairs falling inside it will be not frequent. In the second pass of the algorithm, we can now just count the appearances of the pairs that felt inside the frequent buckets and where both the items were frequent, reducing the needed memory. To reduce the necessary space in memory we can use a bitmap that states for every bucket if it is above or below the **s** threshold.

There are two main improvements of this algorithm to reduce the number of false positives. The *Multistage* variants introduces an extra pass between the previous two that rehashes to a new bitmap only the pairs where both the items are frequent and the pair was hashed to a frequent bucket in pass 1. In the new pass 3, it counts only the pairs composed by frequent items and where the pair is hashed to a frequent bucket of both the bitmaps computed in pass 1 and 2. In the *Multihash* variants, in the first of the two main passes, instead of just one bitmap we introduce two or more of them, using different hash functions. Clearly, this technique is risky, since the number of buckets in each bitmap will be halved, and more pairs will be kept as frequent. The goodness of the hash functions is a key point.

2.1.4 ***Random Sampling.*** This technique relies on the concept of avoiding disk I/O by loading a random sample of the baskets into the main memory. Then an algorithm from the ones explained above can be used but without paying the Disk I/O operations. The big problem of this algorithm is that we are not using the entire dataset and so we could catch only the set frequent in the sample but not the ones frequent in the whole dataset.

2.1.5 ***SON Algorithm.*** In this algorithm, we divide the dataset into subsets of baskets and we compute the frequent itemsets in each one of the subsets using the preferred algorithm. This phase can be done in sequence by loading and computing for every subset, or in parallel by using several nodes. At the end of the first computation, only the itemsets that are frequent in at least one of the subset of basket are kept and with the most frequent of them are validated over the whole dataset. Here again, the main idea is that an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset.

2.2 The association rules

In order to decide if an itemset is frequent, we can just assure that the number of appearances is greater than a threshold **s**. As explained in the previous section, we are interested in finding some correlations between the items, like "earthquake" → "emergency". This is an association rule, and it means that if a basket contains the term "earthquake", then it likely to contain even the term "emergency". To discover these associations we could rely on the confidence metric and it is defined as below:

$$conf(I \rightarrow j) = \frac{\#appearances(I \cup j)}{\#appearances(I)}$$

The *Interest association rule*, which ensures that a high confidence value is not derived from a high frequency of the j item, is not suitable for our project, because, as the problem description states, the term j (e.g. "emergency") could be frequent even with other terms.

2.3 Clustering algorithms

To solve a problem found while developing the algorithm, I used a really simple but useful clustering technique. Clustering algorithms are used to make groups of items looking at their characteristics. With some of them, we must know from the beginning the number of clusters that we want (like in our case), while in others we should find a condition to decide when to stop merging or dividing groups.

3 PROBLEM STATEMENT

In this section, a formal definition of the problem is given. We start defining as **A** the algorithm that we want to obtain in order to solve the problem.

The algorithm takes the following inputs:

- A set **T** of pairs `<tweet, timestamp>`, where tweets are themselves set of words;
- A minimum confidence value $\mathbf{C} \in [0, 1]$;
- A days support value **D**;

As an output our algorithm **A** will return the following elements:

- A set **O** of pairs `<topic, score>`, where topics are set of terms, containing the most consistent topics in time;

Wrapping all together we obtain:

$$\mathbf{A} : T, C, D \rightarrow O$$

In the next sections the algorithm **A** will be described in details, and it will be shown how it changed from its initial version, to versions where it was enriched with features that allows it to perform in a more efficient way. All the versions of the algorithm **A** take the same exact inputs and results in the same output type. The main difference between them is the efficiency.

4 SOLUTION

To find the frequent itemsets, of course, we have to rely on one of the algorithms explained in the previous section. It's important to notice that in this project we are interested in frequent itemsets in subsets of the period that our dataset covers since we don't want frequent itemsets in the whole dataset. The problem now moves to decide how to split the dataset in order to exploit frequent itemsets. The more reasonable approach seems to be, in my opinion, to split the dataset by days and analyze the frequent itemsets of every day. Even if around the world there are different time-zones and people tweets constantly every possible hour of the day (in fact when one part of the world is sleeping another one is awake and tweeting), splitting by days seems a good trade-off between every other possible division, like clustering which doesn't make sense in this case. Splitting the dataset brings a non-obvious advantage in the computation: we can execute it in parallel and, if wanted, in a distributed fashion. As we will see later, this leads to a massive improvement.

In Algorithm 1 is reported in pseudo-code how the computation of the frequent itemsets per day is done. Starting from the subset of the tweets posted on the specific day, we start looping and every time we increment the length of the itemsets. So the first time we count only the frequency of the single words. The

Algorithm 1: Frequent itemsets computations per day

```
Result: Sets of the frequently correlated terms of the day
Data: tweets: Tweets of one day
Function GetFrequentItemsetsInDay(tweets, day):
    allItemsets = set()
    itemsets = set()
    itemsetsLength = 1
    do
        itemsets = getFrequentItemsets(tweets, itemsets,
          itemsetsLength)
        allItemsets.update(itemsets)   // update the total
          count with the count of the specific length
        tweets = tweetsThatContains(itemsets)      // we
          keep only the tweets that contains at least one
          frequent itemset
        itemsetsLength++
    until itemsets is empty
    return allItemsets
```

Algorithm 2: Frequent correlated itemsets computations, starting from the previous itemsets

```
Data: minConfidence: is the minimum amount of
        confidence we want
Function getFrequentItemsets(tweets, prevItemsets,
  itemsetsLength):
    allItemsets = set()
    allowedWords = wordsIn(prevItemsets)      // we use
      only the most frequent words
    foreach tweet in tweets do
        possibleWords = tweet.words ∩ allowed_words
        if possibleWords.length ≥ itemsetsLength then
            itemsetsCount =
              countNewItemsetsStartingFrom(possibleWords,
              prevItemsets) // count itemsets starting from
              the old itemsets and adding one frequent word
            allItemsets.update(itemsetsCount)
        end
    end
    minSupport = calcSupportThreshold(allItemsets)
    remove from allItemsets where
      support < minSupport or
      confidence < minConfidence
    return allItemsets
```

second time we use only the tweets where at least one frequent word is present, and we start counting tuples formed by frequent words. Going on, the logic is the same, we form and count only the triples that are composed of a frequent pair and a word that is inside a frequent pair. We stop exit the loop when no new frequent itemsets are created. Algorithm 2 shows how the itemsets of a specific length are calculated. In particular, to avoid useless computation, we check that inside the tweet under inspection, there are enough frequent words, otherwise, we know that the tweet cannot have itemsets of the desired length. Once finished the counting of every itemset, we have to cut-off the less frequent or less interesting of them. To do so, firstly we compute the support threshold value, which will later be explained how it works, and then we remove the topics that have a support or a confidence value less than the thresholds. The confidence value is the most valuable metrics in this project since it reveals the correlation between two sets of words in base of theirs support value.

4.1 Sorted dataset

While developing the algorithm, I noticed how much it was slow with a huge dataset, so I decided to focus on that and in particular to sort the entire dataset by the date fields. This is a heavy computation but you have to do it only one time. Once done, I implemented a binary search algorithm to speed up the loading of the tweets for a specific day. This leads to an improvement on the biggest dataset from more or less 35 minutes to 16 minutes.

4.2 Support threshold calculation

A critical part of the algorithm regards how to choose a support threshold that leaves out the non frequent itemset and includes the frequent ones. The following paragraphs will explain in an incremental way the metodologies used.

A fixed value. The first and easiest try that I made was to use a fixed value for the minimum support value. This reveals to be not a good choice since it suffers from generalization. We cannot decide a value that will work fine both for a dataset where the most frequent itemset appears millions of times and for another set where the most frequent appears only hundreds of times.

A fixed percentage value. The second trial focused on the idea of keeping only a percentage of the itemsets that are the most frequent, but once again with a high number of itemset we would keep too many of them, and, likely we are keeping even itemsets that are not enough frequent for our goal. Another idea related to this one was to keep only the itemsets where its support value is high relative to the range of possible supports, as described above:

$$support\_threshold = min\_support + (max\_support - min\_support) * percentage$$

With this simple formula, we would keep only the support values that are high enough, but the problem is related to situations where we have the maximum or minimum support value which is an outlier with respect to the other values. It could happen, in the covid19 dataset for example, that only one topic will be marked as frequent.

A Gaussian model. A good try was to model the distribution of the supports as a Gaussian and then, using the *percent point function* take the support value given a cumulative distribution probability. The idea seems to work, but problems arise when executing the code, in fact, the maximum support outlier that caused problems in the percentage support calculation, once again didn't help us. In particular, using a probability of 0.99999 over the ppf function returns a too low threshold support value which marked too many topics as frequent.

A clustering technique. The only way that I found to overcome the previous issues adapting to the supports distribution without using some fixed percentage or fixed values, was to rely on clustering. As previously said, clustering allows us to split the dataset into groups based on the characteristic of the data. In this particular case, I used the k-means clustering and specifically the 2-means one. The goal of this approach was to create two clusters:

one for the most frequent topics and the other one for the less frequent ones. In the initialization phase, I put the most frequent itemset in one cluster and all the others in another one. Then I start computing the centroids of the two clusters and decide which topics move to the other cluster based on their support value. In the end, so when no modifications are done, I obtain the two needed clusters. I had to introduce a fix to avoid obtaining a cluster of only one or two elements. If the cluster with the most frequent topics has less than three topics, I recompute the 2-means clustering excluding the already found frequent topics.

5 IMPLEMENTATION

I developed the entire project using python3 with the addition of some libraries to facilitate the development. The following is a list of the main python libraries used with their roles:

- **nltk**: it is a Natural Language processing library that, given a sentence, analyses every word of it and is capable, among other things, of understanding its role in the phrase;
- **csvsort**: this library is able to sort a CSV file splitting it in chunks of rows and process the chucks individually. This library is particularly suitable for large CSV datasets;
- **matplotlib**: this powerful library was used to create plots of statistics and metrics collected from the datasets;
- **multiprocessing**: this is a build-in python library which was used to parallelize the computation using several processes instead of just one.

6 DATASET

In this section, I will focus on which datasets I used, how I obtained and cleaned them, and why I made these choices.

In the description of this assignment, it was suggested to use the *covid19-tweets* which is a light dataset composed of 179 087 tweets with the *#covid19* hashtag recorded in the period between 24/07/2020 and 30/08/2020. This dataset doesn't satisfy me at all and, moreover, going further in the development of the algorithm, I wanted to test it over a bigger and more realistic dataset. In fact, from my point of view, a dataset where all its tweets for sure are related to the same topic could not lead to interesting results. We will see if it is true in the next section.

To obtain a different dataset I started from the Twitter stream files. Files where all the events (like new tweets, delete tweets) happening on Twitter's servers are logged. I took the stream files of June 2020 and from this big amount of data (more than 70 GB) I created a python script that filters all the events and keeps only the new tweets written in English. This leads to a raw dataset composed of 17 708 944 tweets.

At this point, I had two raw datasets containing a lot of tweets that need to be cleaned in order to be passed as input to the algorithm. I created a python script to meet this purpose, and it has the following objectives:

- *Remove the **urls***: It happens that in a tweet is inserted an http url, or, if the content of the tweet is too long, Twitter breaks it and adds a url where we can see the full tweet on their website. To remove all the urls, I used a simple regex that removes all the strings starting with "http".
- *Remove the **emojis***: Nowadays, emojis are present in most of the tweets since they can add more colors and let the tweet be more attractive. For our goal, emojis are useless and we want to remove them all. Even in this case I used a regex that removes all the text that is not printable in a console.
- *Remove the **punctuation***: With a simple regex I removed all the most used punctuation since they are useless from the objective point of view.
- *Extract the **usernames** and **hashtags***: Since the problem description states that we have to consider even usernames and hashtags, I decided to extract them as soon as possible from the tweets' text because, most of the times, they are not part of phrases, but just put at the end of the

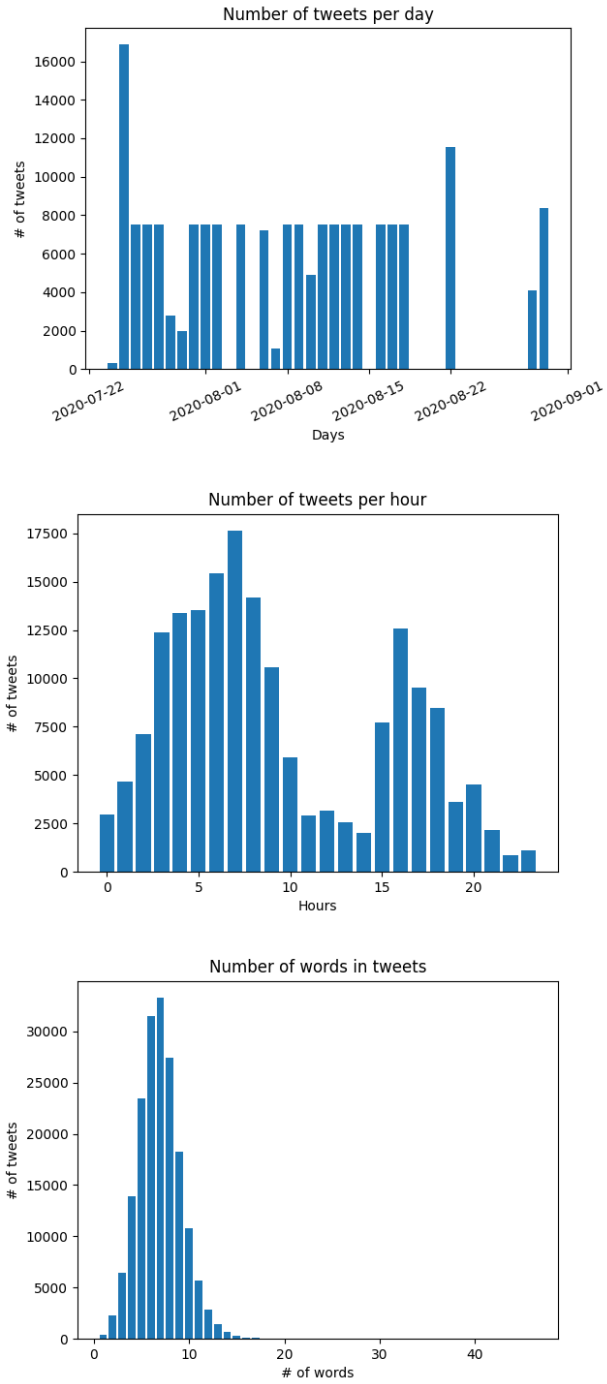

Figure 1: Plots of the distributions of the covid19 dataset.

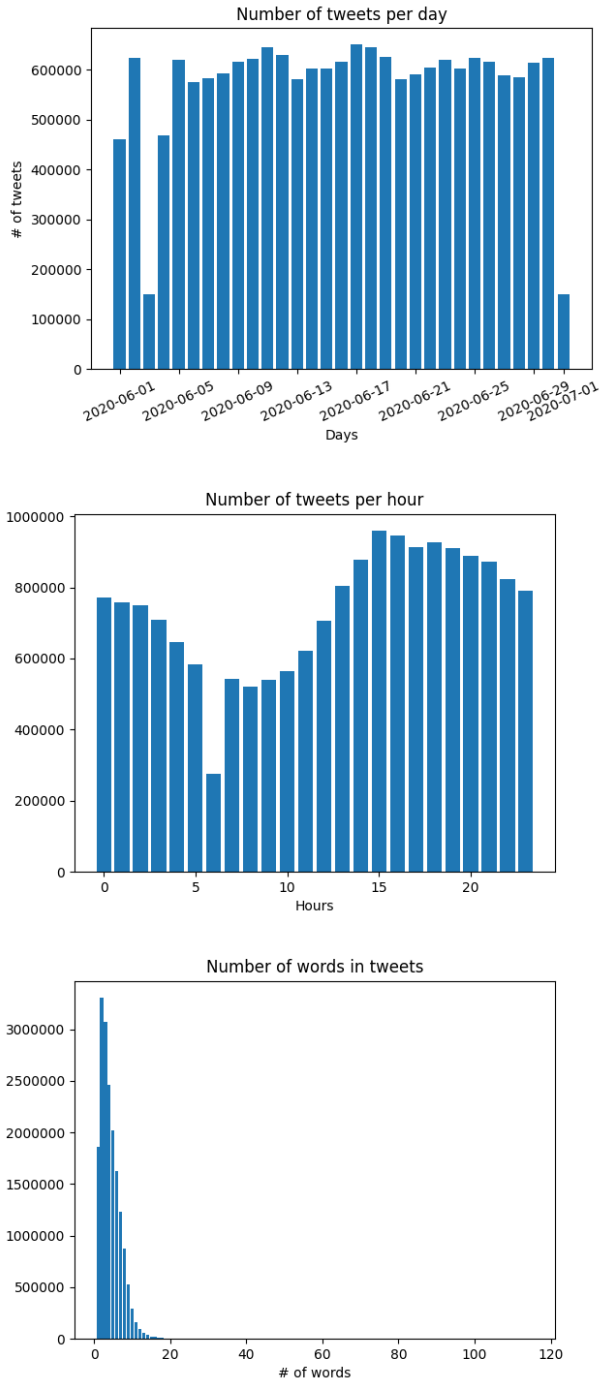

Figure 2: Plots of the distributions of the dataset of tweets collected from the Twitter stream of June 2020.

tweets. This decision helped to avoid mistakes in the nouns extraction as explained in the next bullet point.

- *Extract all the **nouns** from the tweets*: We are not interested in part of the phrases such as articles or verbs, but just in nouns and pronounces. If, for instance, we kept even the articles, the output of our algorithm would be very likely only sets of them, since they are the most frequent words used while speaking or writing.

To finalize the dataset cleaning process I decided to sort all the tweets in each dataset by the date and time of publication. The reason has been given in the previous section. The covid19 dataset, which is smaller than the second one, can be sorted easily with python. The problem arises with the dataset that I created from the Twitter stream because it was too big to be sorted entirely in RAM. To overcome this issue I used a python library called *csvsort*, which sorts csv files loading only some little chunks of data at the time.

After these operations, I obtained two CSV datasets composed of two columns: the date of publication and the interesting words composing the tweet. All of them are sorted by date and time.

Now, we can compare the two datasets examining Fig. 1 and Fig. 2. The same statistics were collected from the two datasets, in particular, analyzing the number of tweets per day plots, we can discover that the covid19 dataset is less uniform and complete since a lot of days has neither a tweet. The number of tweets per hour plots show how people tend to tweet mainly in specific hours of the day for the covid19 dataset, while for the Twitter stream one, users pretty constantly tweet at every hour. Comparing the plots of the number of words in tweets, we can see how in the covid19 dataset the distribution has a Gaussian shape, while in the Twitter stream dataset the peak appears more or less at the same x value but with a really higher maximum number of words.

7 EXPERIMENTAL EVALUATION

To collect metrics about my computer while running the algorithm, I modified a script found on GitHub[1]. This script is now capable of collecting metrics about CPU and memory usage of an executed command, and all its sub-processes, every 5 seconds and save these values into a CSV file. All the computations were performed on a laptop with the following environment:

- **CPU**: `4 × Intel® Core™ i7-6500U CPU @ 2.50GHz`;
- **Memory**: `15.5 GiB of RAM`;
- **Operating System**: `Manjaro Linux 64-bit`.

I started executing my algorithm over the covid19 dataset using different minimum interest parameters (0.7, 0.8, and 0.9). The days support parameter was always set to 1 to obtain more values as result. The results are reported in Table 1. As can be seen from the table, there are some correlations of terms that are redundant, in the sense that summing up the first four rows, the set composed by {alert, news, pandemic} is strongly related, and when one of them is popular the others are popular for sure. This set is created by the daily tweets informing about the new coronavirus cases.

The correlation `hours` → `cases` is well explained in Figure 3, wherein the top plot we can understand the number of appearances of the two terms, while in the bottom one we can see the confidence value between the two terms changing throughout days. The confidence value is always high, but it exceeds the 0.8 value only in one day, and this explains why the number of days of popularity drops increasing the minimum confidence value.

Another interesting correlation is between the word Putin and vaccine. In Figure 4 is reported the correlation plots. Only in one or two days there is a peak of tweets containing these terms, this is explained by the fact that in those days Putin announced that Russia is testing the first vaccine for the coronavirus.

In Table 2 are reported the results of the algorithm executed over the Twitter stream dataset. Since this dataset is not strongly

[1] https://github.com/davidezanella/process-metrics-collector

Terms	Min confidence: **0.7**		Min confidence: **0.8**		Min confidence: **0.9**	
	# days	**Mean confidence**	**# days**	**Mean confidence**	**# days**	**Mean confidence**
alert → news	1	1.0	1	1.0	1	1.0
alert → pandemic	1	1.0	1	1.0	1	1.0
(alert, news) → pandemic	1	1.0	1	1.0	1	1.0
(alert, pandemic) → news	1	1.0	1	1.0	1	1.0
#EarlyVoting → person	1	0.9495	1	0.9495	1	0.9495
(hours, #COVID19) → cases	2	0.8533	-	-	-	-
(putin) → vaccine	1	0.8074	1	0.8074	-	-
(russia) → vaccine	1	0.8006	1	0.8006	-	-
(staff) → #COVID19	1	0.7986	-	-	-	-
(hours) → cases	3	0.7866	1	0.8378	-	-
(person) → #EarlyVoting	1	0.7635	-	-	-	-
(hours, cases) → #COVID19	3	0.7616	-	-	-	-
(#coronavirus) → #COVID19	2	0.7511	1	0.8	-	-
(impact) → #COVID19	1	0.7446	-	-	-	-
(deaths, #COVID19) → cases	1	0.7443	-	-	-	-
(spread) → #COVID19	1	0.7338	-	-	-	-
(cases) → #COVID19	4	0.7336	-	-	-	-
(deaths) → #COVID19	3	0.7157	-	-	-	-
(hours) → #COVID19	2	0.71	-	-	-	-
Execution time (sec)	16.8526		11.9384		12.0848	

Table 1: Results of the algorithm over the covid19 tweets dataset. The column "# days" means the number of days the set of terms was popular, while the "Mean confidence" was calculated averaging their confidence values in the popular days.

Terms	Min confidence: **0.7**		Min confidence: **0.8**		Min confidence: **0.9**	
	# days	**Mean confidence**	**# days**	**Mean confidence**	**# days**	**Mean confidence**
(happy, lauren) → birthday	1	0.9884	1	0.9884	-	-
lauren → birthday	1	0.9851	1	0.9851	1	0.9851
(happy, birthday) → lauren	1	0.9778	1	0.9778	-	-
happy → birthday	1	0.8920	1	0.8920	-	-
happy → lauren	1	0.8824	1	0.8824	-	-
petition → sign	1	0.8418	1	0.8657	-	-
floyd → george	1	0.8418	1	0.8418	-	-
fathers → day	1	0.7596	-	-	-	-
george → floyd	1	0.7256	-	-	-	-
Execution time (sec)	997.49		908.47		878.81	

Table 2: Results of the algorithm over the Twitter stream dataset. The column "# days" means the number of days the set of terms was popular, while the "Mean confidence" was calculated averaging their confidence values in the popular days.

focused on a topic like the covid19 one, the number of results is low. The first five rows represent a set of strongly correlated terms regarding the birthday of Lauren Jauregui, a singer famous in the U.S.A.. Her birthday is on the 27 of June and as we can see in Figure 5, the peak of the confidence value of these terms was exactly on that day, while the word "happy" was pretty frequent even on other days, but without the other terms.

The set composed by "Floyd" and "George" has a strong confidence value but the flow of its popularity is going down, as reported in Figure 6. In Figure 7 there are two tweets extracted from the dataset regarding this set of terms where the common words are depicted with some colors.

Figure 8 and Figure 9 are the results of plotting the collected metrics while executing the algorithm. The CPU percentage maximum value is 400% since my laptop has four cores. Using the covid19 dataset, the quantity of RAM used was really low, the CPU usage was neither at maximum and the execution lasted more or less only 16 seconds. On the other hand, with the Twitter stream dataset, the CPU usage reaches almost the maximum value and the quantity of RAM used was about ten times the covid19 dataset memory used. The maximum execution lasted 997 seconds, so about 16 minutes.

7.1 The test dataset

In order to be sure that my algorithm works in the proper way, I created a simple test dataset that contains hand-written terms for two days. Recalling the example of the first chapter, I tried to achieve as the result the correlation set composed by `earthquake` → `emergency`. I was able to achieve my goal using several words but assuring that both the terms were popular in both the days

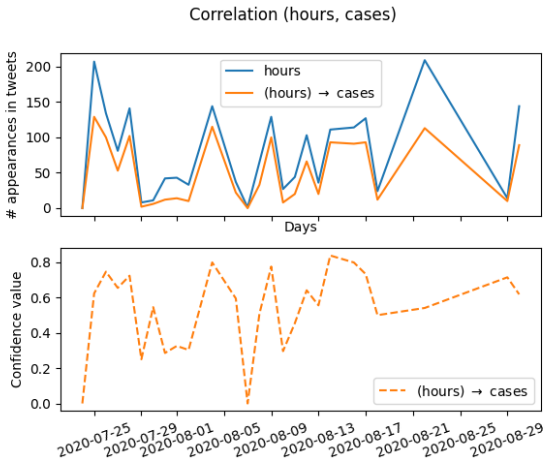

Figure 3: Plots showing the correlation *hours* → *cases* in the covid19 dataset, through the confidence value and the occurrences count.

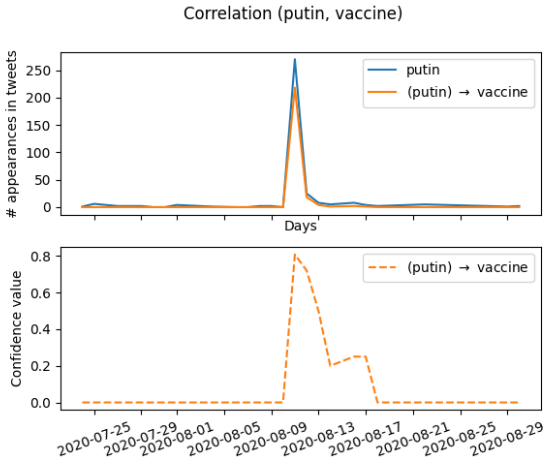

Figure 4: Plots showing the correlation *putin* → *vaccine* in the covid19 dataset, through the confidence value and the occurrences count.

and that the word earthquake appears rarely without the emergency one.

8 CONCLUSION

To develop the project I used the concepts of the A-Priory and SON algorithms adapted to fit the requirements. The confidence value reveals to be the main discriminator metric to decide if an itemset is interesting or not. The results came out from real datasets reveal to be related to real-word facts (like the Russian vaccine or the death of George Floyd) proving the goodness of the algorithm. In order to extract more interesting correlations, this algorithm should be executed over a bigger dataset, like one that covers one year of tweets. In this way could be found frequent itemsets that are frequent, for instance, every day of a month or two times a year.

The computation time seems to be reasonable but, maybe, could be even improved by finding a way to reduce the baskets' sizes, like splitting the days in more baskets. For sure, a c++

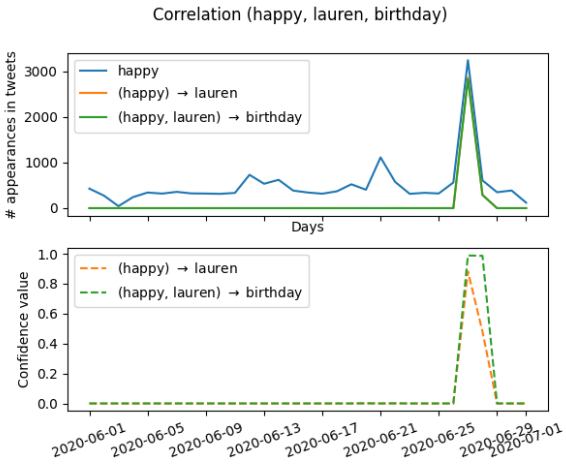

Figure 5: Plots showing the correlation (*happy*, *lauren*) → *birthday* in the Twitter stream dataset, through the confidence value and the occurrences count.

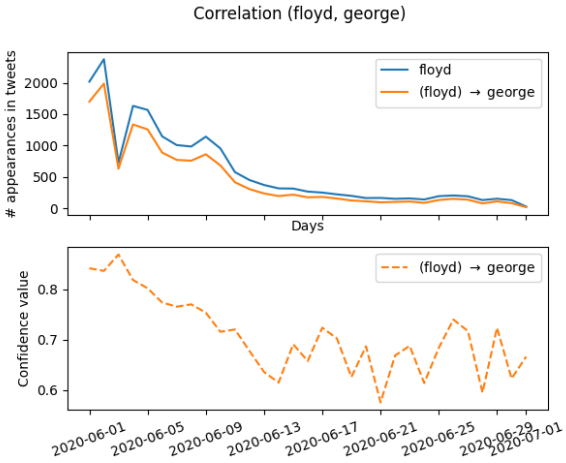

Figure 6: Plots showing the correlation *floyd* → *george* in the Twitter stream dataset, through the confidence value and the occurrences count.

implementation would achieve better performance since python is a slow programming language.

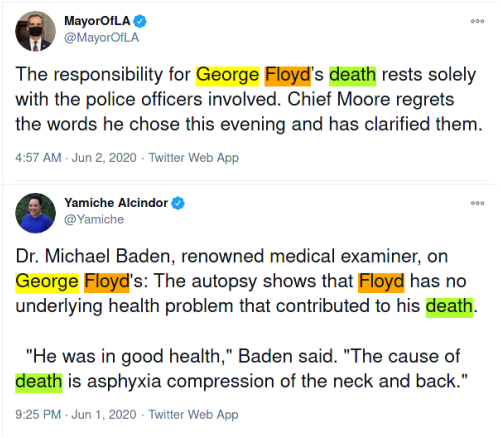

Figure 7: Two tweets talking about George Floyd. The common interesting terms are highlighted with different colors.

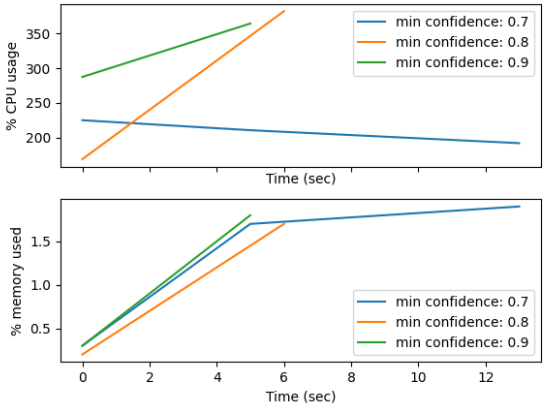

Figure 8: Metrics collected while executing the algorithm over the covid19 dataset.

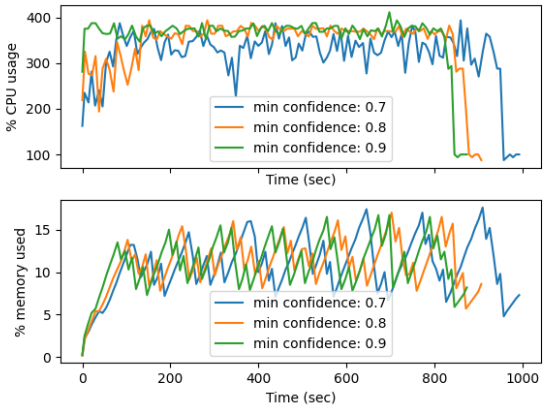

Figure 9: Metrics collected while executing the algorithm over the Twitter stream dataset.