Finding frequent topics on tweets through time

Data mining course final project

Ivan Martini, #207597
University of Trento, Italy
MSc in Computer Science [LM18], 2nd year
ivan.martini@studenti.unitn.com

ABSTRACT

This paper is the report for the work done on the final project of the course of Data Mining offered by the University of Trento, year 2020/2021.

The large use of smartphones and mobile devices increased a lot the capability of accessing the internet at any time. More or less 15 years ago, the web was already moving from the 1.0 to 2.0, having the final users able to produce content, but the birth of social network boosted this process in a consistent manner. In this scenario, huge quantities of data are produced, which could be (and actually are) analyzed in order to extract precious information. This paper will present a solution to analyze *tweets* (brief texts written on the personal homepage, on Twitter) in order to delineate topics that are recurrent and cyclical over time. In these pages, other than the formal definition of the task and a detailed explanation of both the implementation and the evaluation of the solution, there will be also a set of explanations on the choices that lead to the final result, in order to clarify as far as possible the motivation between each decision taken.

1 INTRODUCTION

During its 50-years long history, Internet underwent through a lot of changes and revolutions. Each one contributed to the face of the network as it is right now, deeply modifying the behaviours of the people which interfaces it. Since mid 2000s, users became also creator of contents, mostly but not only though the social networks. Facebook and Twitter are the big players in the field and, with GBs of data produced every second, they contributed to start the era of Big Data. They centered their business model on the users' data, but the data stream produced by their environments is so huge that is not feasible to be analysed in a traditional way. The effort spent in this direction led to the development of *data mining*, a branch which covers exactly the extraction of non-trivial information from huge quantity of data. Due to its many facets, data mining requires a solid background in many different field: computer science of course, but also statistics, economics, linguistics and studies on the human behaviour.

Through the *mining* of social networks' data, is possible to understand a lot about the users and their preferences. A classical example is the one of *trending topics*, which reflects directly the interest of a population. However, many challenges affect this field and makes it unique: the amount of data to work with, the quality of the data produced by users (which is usually lower than the one produced by machine) and last but not least the lack of a target. Indeed, if the data hold the answers, it is important to "*ask the right questions*", which is everything but trivial. This study finds its spot in this exact place: starting from a dataset of covid-related tweets, which ranges over a month in summer 2020 (from 24/7 to 30/8), the goal is to find the *most frequent topics though time*. After a brief explanation of the background required and the task given, there will be some sections dedicated to the explanation of the solution and to the tools used, concluding with a both qualitative and quantitative evaluation of the results.

2 RELATED WORK

Retrieving information out of raw and unstructured texts is an hot topic in this field, and different techniques can be used to achieve different results. To my knowledge, there is no given algorithm able to provide a temporal analysis over the frequency of words, which is the main reason behind the development of this solution. However there are two mining techinques that must be analysed in order to understand the final result: the *Frequent-Itemset* algorithm and the *TF-IDF weight function*. The former is actually used in the solution, while the latter is important as it inspired me during the development.

2.1 Frequent-Itemset Algorithm

The problem of finding *frequent itemset* is well known in data mining. Given a large set of items $I = \{i_1, \dots, i_n\}$, and a large set of baskets $B = \{B_1, \dots, B_m\}$, where $B_i \subset I$, the goal is to discover interesting relationship among each B_i holding with *some* certainty. An interesting relationship is defined as a set of items (i.e. itemset) which appear often together (i.e. frequent) and some *association rules*. An association rules is generally written as:

$$X, Y \subset I \quad , \quad X \Rightarrow Y$$

and it is read as "*the baskets containing X are likely to contain also Y*". The common solution takes two steps:

(1) Find frequent-itemset: To find frequent itemsets, it is important to identify the meaning of *frequent* and the meaning of *itemset*, even if they might seem trivial. The frequency is measured in term of **support**. The support of an item i is the fraction of the basket dataset B containing i. If the support required is 0.02 (i.e. 2%) and the dataset contains 1000 baskets, all the items in at least 20 baskets will be extracted. The itemset are instead just set of items, but there are two possible ways of using them:

- *Maximum itemset*: gives more pruning on the results, but the quality might be a little compromised. If the itemset $\{a, b, c\}$ is frequent, then also $\{a, b\}$, $\{b, c\}$ and $\{a, c\}$ will be frequent, but they are not considered
- *Closed itemset*: it produces more data, capturing more information, but might be harder to analyze (due to quantity). Again, if itemset $\{a, b, c\}$ is frequent, then also $\{a, b\}$, $\{b, c\}$ and $\{a, c\}$, but this time they are preserved, if the cardinality is different.

Different algorithm are used to extract the frequent itemsets. The first one is called **apriori**[6], which has a strong mathematic

background. Since an itemset is frequent iff all its subsets are frequent (*monotonicity*) and an itemset not being frequent means that all his supersets are not frequent (*anti-monotonicity*), it is possible to build the frequent itemsets in a *bottom up* fashion, identifying the frequent items (*singletons*) and building up pairs, then triples, and so on. Using these simple but effective rules many other algorithm have been implemented, including the one used in this implementation, the **FP-Growth** algorithm[7].

(2) Discover association rules: To discover association rules each frequent itemset is analyzed. The itemset $\{a, b, c, d\}$ could be represented by different rules: $\{a, b, c\} \Rightarrow \{d\}$, $\{a, b, d\} \Rightarrow \{c\}$, $\{a, c\} \Rightarrow \{b, d\}$ and so on so there is the need for measuring how *confident* a rule is. Indeed, the metrics to establish the goodness of a rule is called **confidence** and is measured as:

$$\text{conf}(X \Rightarrow Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X)}$$

which represents the portion of baskets containing X also containing Y. For the sake of the project however, just the section relative to finding the frequent itemsets is required, as no association rules are discoverd in the solution.

2.2 Term Freq. - Inverse Document Freq.

The *Term Frequency - Inverse Document Frequency* function, or more briefly **TF-IDF** function is a weight function commonly used in information retrieval to measure the importance of a word in a document, in a set of documents[9][8].

Given a set of documents $D = \{d_1, d_2, \ldots, d_k\}$ and a set of words $W = \{w_1, w_2, \ldots, w_n\}$ lets define as n_{wd} the occurencies of word w in document D and W_d as the set of words in D. The TF-IDF function is be defined as

$$TF(w, d) = \frac{n_{wd}}{\sum_{w \in W_d} n_{wd}}$$

$$IDF(w, D) = \log_{10} \frac{|D|}{|\{d | d \in D : w \in d\}|}$$

$$TF - IDF(w, d, D) = TF(w, d) * IDF(w, D)$$

Although being important, this function is not used in the solution proposed, but it has been fundamental for its development. While it is easy to understand that the TF function computes the fraction between the number of words w in the document and the number of word in the whole document, it could be harder to spot the importance of the IDF function. The inverse document frequency indeed rewards the words that are uncommon in the whole set of documents, at the point that, if each document contains the word w, its IDF is 0 (=logarithm of 1) as the term is obvious.

This idea, even if implemented in a different manner, is the core of the solution presented in this paper.

3 PROBLEM STATEMENT

When dealing with mining information out of text/documents many techniques can be applied, but none of them has any relationship with time. In the provided dataset, along with many data about the user, there is also the timestamp of each tweet, which allows to define a temporal ordering between a tweet and another. The task is *"to identify consistent topics in time, i.e., set of terms that become frequent together throughout time"*, which is somehow informal and requires a specification. First of all it is necessary to highlight two words, *topic* and *frequent through time*, which requires an additional description.

A **topic** is a set of words that appear in the same tweet together. While designing the solution, I also considered the possibility of analyzing cross-tweet topics (clustering for example by the topic and considering different tweets as a single set of words), but then I decided to proceed with the single tweet approach. In particular I decided to adopt the *Closed Itemset form* explained in Section 2 to identify the frequent topics, using a *low support* to capture more information. Since the lack of experience in data mining, I decided to use a policy that offer less pruning and more complete results, in order to have a wider range of opportunities to explore. The drawback is that this solution might produce confused result, harder to explain; this issue can however be avoided pruning the results afterwards.

Frequent through time instead is a more vague concept, so it is crucial to specify how I decided to address this requirement. The frequency itself is expressed, as seen above, as the support (i.e. the fraction) of each element inside the dataset, which is a fine explanation but lacks the relationship with time. To be *frequent though time* a topic must be frequent over different periods, but not always frequent, as in that case it is *obvious* and could be spotted with a simple *frequent itemset algorithm*. The idea of rewarding topics that appear different times but not too often comes directly from the `TF-IDF` weight function described above, even if implemented in a different way (the data are not multiplied by the logarithm, but filtered by their count).

PROBLEM STATEMENT: Given a set $S = \{S_1, S_2, , \ldots, S_k\}$, containing k items $S_i = (W, d)$ where $W = \{w_1, w_2, \ldots, w_n\}$ are words, d is a date and a time period t, identify all the topics $T \subset W$ that are frequent with a certain support s in at least a time periods and at most b time periods, $a > 1, b < k$

Input: The set S, the time period t, the frequency boundaries a and b and the support s.

Output: The set $T^* = \{T_1, \ldots, T_n\}$

The idea of the time period, or time window might be a little confusing, so a few more words will be spent to clarify it. Each user spends it's time on the platform accordingly with it's daily routine, so the first step was discarding the hour of publication from the analysis, considering **the single day** as time unit. Even though a day seemed a good fit, this could not be enough. Indeed a simple analysis day by day on the dataset would produce the same identical result of a permutation on the days. This result could be perfectly fine, but it means that the temporal relationship is basically lost.

The *time period* covers this issue. It is possible indeed, to expect a topic to be frequent over a couple days together, so the frequency is measured day by day, over 2 days. In other words, a topic is considered frequent the day i, if it has enough support over the days i and i+1 combined. In this way the temporal relationship is preserved (day i is linked to i+1, which is linked to i+2, and so on..) and the support for the frequent-itemset can be raised a bit, pruning further the results. These parameters can be tweaked to perform differently (for example, working on data spread though years, an analysis day by day could be meaningless, but a week by week could work better).

4 SOLUTION

The pipeline of my solution is divided in two stages, which interact in order to produce the final output. There is also a third stage, which is parallel and it is used only for the baseline evaluation. The picture below explains graphically the logical structure.

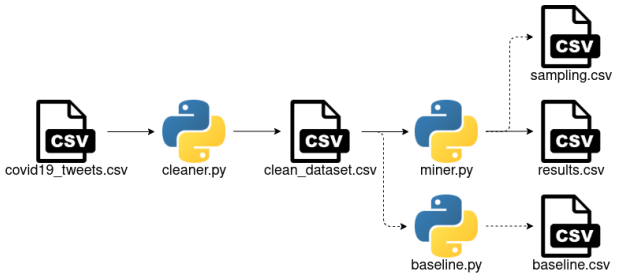

Figure 1: A graphical and intuitive representation of the pipeline which addresses the task

All the stages are independent and standalone. The reason behind this division is the logical consistency of the sections. In the first stage the database is cleaned and preprocessed in order to interact with the second stage. In the second stage the actual algorithm is applied to produce the expected results. At this point the algorithm for the baseline evaluation is executed too. The intermediate results are stored in another `CSV file`, which in a real system implies writing on disk and wasting "a lot" of time. However for the purpose of the project, this approach is more than suitable, as it allows a better understanding/explanation of the results obtained, simplifying also the task of reading the code. While the next two sections (5 and 6) are dedicated to the actual implementation of the solution, here an high level explanation is proposed instead.

4.1 Cleaner

The **cleaner module** starts from the original dataset and applies filters and corrections in order to rise the quality of the input data and, together, the quality of the results. The dataset used as reference can be retrieved at https://www.kaggle.com/gpreda/covid19-tweets and contains 178683 tweets from the 24th of July 2020 to the 30th of August 2020. The expected output is a three columns `CSV file` structured as follows:

- **id**: a simple counter, it is not required, but many implementation automatically produce it
- **tokens**: the set of relevant words (i.e. stop-words excluded) in a string, each one separated by spaces
- **date**: the date of the tweet, written as a string in the format `YYYY-MM-GG`.

Again, having the code divided as mentioned above, means that another dataset, structured in a completely different manner, can be integrated just by substituting the preprocessor with a brand new one. The only requirement is to produce the output following the schema presented above.

4.2 Miner

The **miner module** is the actual solution. Starting from the results obtained by the previous module, this module solves the task formalized in Section 3.

Algorithm 1 Miner module: frequent itemsets through time

```
1:  data ← LOAD_FROM_CSV(id, tokens, date)
    // load dataset
2:  start ← MIN(data.date), end ← MAX(data.date)
3:  minrange ← 7, maxrange ← 14
4:  delta ← 2
    // time period, 2 days
5:
6:  merged ← []
7:  while start < (end − delta) do
8:      current ← data.filter(data.date >= start)
9:      current ← current.filter(current.date <= start+delta)
        // current contains only the tweet published in the period
        from start to end
10:
11:     partial ← FREQ_ITEMSET(current, supp)
12:     merged ← merged ∪ partial
        // find the frequent topics for each day, over the delta days
        and merge the results
13:     start ← start + 1
14: end while
15: counted ← GROUP_BY(merged, "topics").COUNT()
16: final ← counted.filter(counted.count > minrange)
17: final ← final.filter(final.count < maxrange)
    // run an aggregation over the whole merged dataset and count
    the occurences. Remove the occurrences that appear too often
    and the ones which appear only few times
```

The pseudocode presented above is a simple description of the actual solution. The data from the preprocessed `CSV file` is loaded in memory and both the *start* and *end* date are extracted. Then the frequency boundaries are defined, where I decided to pick 7 and 14. This means that a topic must appear at least 8 times and at most 13 times to be consider as frequent through time.

Then the analysis is carried out, day by day, filtering the result (the frequency of day i is computed over day i and day i+1, according to delta, to preserve the temporal relationship) and running a plain frequent-itemsets algorithm, exactly as described in Section 2. At each step all the results are concatenated and stored and then they are filtered again to produce the dataset of itemset which are frequently recurring through time.

4.3 Baseline

A few words must also be spent for the third module, which is the **baseline module**. This module loads the same identical data preprocessed by the *cleaner module* and performs a vanilla frequent itemsets algorithm with an higher support, in order to avoid a huge quantity of results. In Section 7 the results of this algorithm will be confronted with the customs solution, in order to produce a qualitative analysis over the data taken into consideration.

5 IMPLEMENTATION

Each step of the pipeline is written in `Python` and it is composed by 3 files, each one representing a module descripted in Section 4. Along with some typical data-oriented libraries as `numpy`[4] and `pandas`[5], the solution requires the use of `pyspark`, which is the python implementation of the **Apache Spark** framework[1][2].

Apache spark is a framework for the distributed computation, maintained by the Apache Foundation. Originally it inherited a lot by the Map-Reduce model, solving many issues related to the iterated computation, as the original model stores every intermediate step on the disk. This approach clearly results in terrible performance of task which require some form of iteration (as the automated learning).

At each step the data is kept in a *pyspark dataframe*, and any operation over it is performed using pyspark operation, in order to let the framework to exploit the CPU as it fits better. Both the *miner* and the *baseline* are similar: after reading the parameters a local spark context is initialized. The data are loaded and splitted, then a **fpGrowth** model is used to produce the frequent itemsets.

- the **baseline module** runs a simple fpGrowth model (support = 0.006) on the whole dataset, without any regard on the dates. The results are stored in *data/output/baseline.csv*
- the **miner module** filters the data accordingly to the dates, as the frequency of a day is calculated over the day itself and the day after. Then, the fpGrowth model (support = 0.004) fits the data day by day, and the record are stored and merged inside an empty dataframe. When the computation over the dates has finished, the final merged results are aggregated and counted. Only the items which appear at least 8 times and at most 13 are considered valid. The support in this model is lower, as the results are further filtered in the last steps. This little margin potentially gives the model more power in order to produce a finer result (theoretically, further analysis are required). The results are stored in *data/output/results.csv*.

In the code there are also parts relative to the evaluation, storing intermediate results and performing other tasks. More on this topic in Section 7.

6 DATASET

The last section relative to the solution is the one about the **dataset**. Although looking as a simple job, cleaning and preparing the data for the analysis is actually one of the biggest task to address. Indeed if the data are *dirty*, than the result cannot be better than *dirty*.

The dataset used at each step is the one required by the project constraints, located at https://www.kaggle.com/gpreda/covid19-tweets. Checking the statistics online the dataset is said to keep the data about 178683 different tweets, but downloading the file in the `CSV` format resulted in more than 323158 entries. Indeed, just by reading some random rows in the file, it is possible to guess that something went wrong during the generation or the compression. Downloading the dataset different times did not solve the issues

The **cleaner module** uses the same libraries cited above and the *Apache Spark* framework, keeping the module efficient and consistent with the other stages of the pipeline. The only additional library is the **natural language tool-kit** (`nltk`)[3] in order to handle the sentences and remove the *stop-words* (i.e. the most common words in a language, which are used to articulate the sentences but doesn't really carry the meaning). As explained in Section 4, the cleaner and the actual solution are divided in order to ease the integration of different solutions or different dataset.

Each dataset will require an ad hoc cleaner, which should follow this flow:

(1) **Load the data**. In this case the `CSV file` is loaded into a spark dataframe. The spark native function `read` is able to directly handle `CSV files`, setting properly *headers* and terminators.
(2) **extract the data**. Only the data relative to the content of the tweet and the date should be preserved. All the null entries must be discarded. To produce finer results also the invalid date entries or null-text entries should be discarded too.
(3) **clean the data**. At this point the dataset has been filtered from the bad entries. However correct tweets must be checked again. This step removes from the text:
   - punctuation
   - stopwords
   - hashtag symbols (#)
   - tag symbols (@)
   - uppercase letters (which are just transformed to lowercase)
   - duplicates (in the same tweets)

   In this particular case, I also decided to transform any "*covid19*" occurence in "*covid*" giving more consistency to the term. Table 1 shows the result obtained at each step.
(4) **write the output**. The cleaned dataset must be stored as described in Section 4.1. The expected output is a `CSV file` containing 3 columns: *id, tokens, date*. The field **id** is expected to be an `integer`, the field **tokens** a string, containing all the tokens separated by space and the field **date** a string, in the format `YYYY-MM-GG`.

Cleaning step	Count
Total entries	323158
Non-null date and text	162475
Valid date and text	152379
In range date	152356

Table 1: This table tracks the count of entries kept at each step. In the end there are more than 150K valid tweets, which are however 30K less than the 178683 reported on the website.

Furthermore, there is a gap of three days (23 August - 25 August) which contains no tweets, in the clean dataset, and a just few, wrong, tweets on the original dataset. As expected those tweets have been pruned, resulting in an incomplete dataset. This issue could lead to a wrong analysis in a real study case, but it is quite fine in this academical setting.

7 EXPERIMENTAL EVALUATION

Evaluate a solution in this environment is clearly a hard task. The multi-field nature of data mining requires a certain mastery in different subject in order to produce an accurate description of the work. After the implementation of the solution, an analysis both qualitative and quantitative has been done, which is reported in this section.

7.1 Qualitative aspects

To perform the qualitative analysis two different path have been pursued, developing both a *baseline algorithm* and *verifying the*

heuristics behind the initial idea. As described above, the baseline solution is a plain frequent itemsets algorithm.

Baseline algorithm: the baseline algorithm produces 57 results, while the solution 50, so the quantity are pretty consistent.

Baseline Results

#	items	freq
0	[cases, covid]	6380
1	[new, covid]	5312
2	[coronavirus, covid]	4226
...	...	...
18	[vaccine, covid]	1393
...	...	...
21	[reported, cases]	1366
...	...	...
30	[realdonaldtrump, covid]	1131

Table 2: Some results extracted from the baseline.

Solution Results

items	count
[last, deaths, cases]	12
[spread, help, covid]	10
...	...
[government, covid]	10
...	...
[wear, mask]	9
[spike, deaths]	8

Table 3: Some results extracted from the solution.

The results proposed in Tables 2 and 3 have been extracted to perform an empirical analysis on the result. The first thing that could be noticed is that the word "*covid*" appears in the 80% of the result produced by the baseline (46/57) but only in the 60% of the ones produced by the solution (30/50). Since they are all covid related tweets, this results had to be expected, but however the solution is able to discard the most obvious terms, which are expected to appear every day, therefore being costant and not recurrently frequent.

The proposed solution it is also able to produce different topics that are lost in the baseline algorithm. Different tweets reminded the use of the mask (`[wear, mask]`) or linked the contagions/death to mass events (`[spike, death]`), but they were simply not enough to appear in the most frequent itemset. By pure intuition, the results are able to space more and model also a wider segment of population, due to the pruning of obvious results. Even if all *covid* related, the results produced by the solution presented in this paper are more wide and able to give more information about the thought of the users.

Validation of the heuristics: the second aspect that I wanted to consider was the validation of the idea presented in this paper. The initial guess was that, using an analysis day by day (providing a temporal linkage between the days) and keeping only the results inside a certain range (8-13) of occurencies, those should have been the results frequently recurrent through time. To evaluate this guess, I decided to store the intermediate results of each day for 2 topics produced by the baseline and 3 topics produced by the solution.

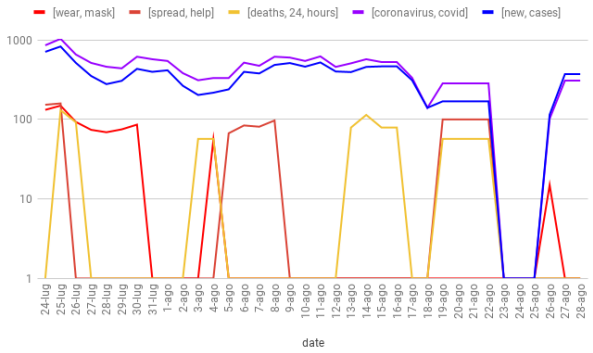

Figure 2: The graph representing the frequency of each topic evaluated. The warm colors are the one produced by the custom solution, while the cold ones are the baseline results.

In Figure 2 these results are plotted. The first thing to notice is the lack of data in the days between 23rd and the 25th of August, which were already pointed out in Section 6. The topic chosen were:

- in warm colors `[wear, mask]`, `[spread, help]` and `[deaths, 24, hours]`, produced by the solution
- in cold colors `[coronavirus, covid]` and `[new, cases]`, produced by the baseline

The scale is logarithmic, in order to compare the magnitude of the elements, but it is clear that the cold lines, which are pruned by the solution are actually always frequent, and don't really have any form or *recurrence through time*. The warm colors instead produce some spikes (4 the yellow line, 3 the red/brown ones), which is exactly the result wanted. This part of the analysis validates the initial idea and concludes the qualitative part of the evaluation.

7.2 Quantitative analysis

The last part of the evaluation of the algorithm is the quantitative analysis. Starting from the precedent analysis, the first observation to point out is the difference from the *baseline* and the *solution* in terms of time. From table 4 it is clear that the baseline is much faster, but since they perform a different task, and produce different results, this confront in not really meaningful.

	Baseline	**Solution**
Average run (3 runs)	11.819s	78.141s

Table 4: Average execution time of the two solution

The next step has been a more consistent evaluation on the performance, testing the *scalability* of the whole environment. Clearly the use of *Apache Spark* already solved the issue at the start, as the framework handles most of the complications due to distributed computing. However, a structured analysis in this sense has been carried out.

During the initialization of the *local* spark context it is possible to select the number of workers to simulate. By default the number is equivalent to the number of physical cores available on the machine (8 in this case, as I run all the tests on an Intel 7th Gen I7-7700K) but it can be changed as needed. Initial test with 1 or 2 workers produced the same results as the 8 workers tests. This is due to the parallelization environment of Spark, which recognizes that some cores are available and exploits them to perform better. To overcome this issue, I created some limited virtual machines to run the tests. The next tables show the results obtained.

2 cores virtualized results

input file size	number of tweets	computation time (sec)
~10MB	~150000	314.749s
~20MB	~300000	621.012s
~50MB	~750000	/

4 cores virtualized results

input file size	number of tweets	computation time (sec)
~10MB	~150000	147.882s
~20MB	~300000	280.507s
~50MB	~750000	667.110s

8 cores non-virtualized results

input file size	number of tweets	computation time (sec)
~10MB	~150000	78.141s
~20MB	~300000	136.762s
~50MB	~750000	325.066s

The initial idea was to perform these computation on 1, 4 and 8 cores, but even the 10MB dataset was very slow on the 1 core virtualized machine. This lead to the decision of running the benchmarks over 2, 4 and 8 cores. The results on the 2 core virtualized machine relative to the 50MB dataset was tainted due to thermal throttling, so I decided to discard them. The last thing to consider is that, the virtualized tests have some slight overhead due to the virtualization itself, so the actual results could be expected to be a little lower.

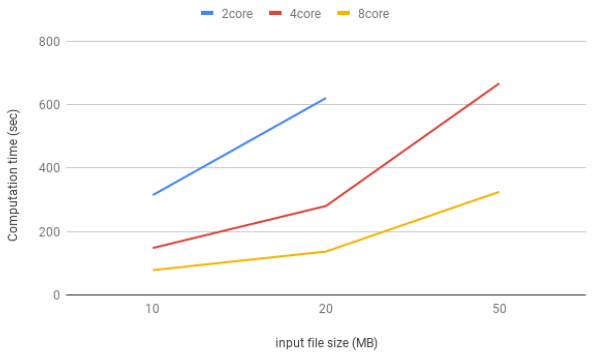

Table 8: Graphical representation of the quantitative analysis

8 CONCLUSIONS

In this paper has been proposed a solution for finding topics frequent though time, in a dataset of tweets. Other than the theoretical formulation of the task and the algorithm, an implementation has been developed and tested, in order to support the initial heuristics which contributed to the final solution. Through the confrontation with a standard frequent itemsets algorithm, it has been demonstrated that the proposed solution allows to model a wider segment of population, capturing frequently recurrent topics and pruning obvious results, which are frequent without any recurrence through time. The initial concept has been validated, with both a qualitative and quantitative analysis, opening the chances for a set of future works. Two examples are the implementation of a weight function to order the result produced and the search for a different frequent itemsets algorithm, which could improve the efficiency.

REFERENCES

[1] [n.d.]. *Apache Foundation.* https://www.apache.org/
[2] [n.d.]. *Apache Spark.* https://spark.apache.org/
[3] [n.d.]. *Natural language toolkit library.* https://www.nltk.org/
[4] [n.d.]. *Numpy library.* https://numpy.org/
[5] [n.d.]. *Pandas library.* https://pandas.pydata.org/
[6] Rakesh Agrawal and Ramakrishnan Srikant. 2000. Fast Algorithms for Mining Association Rules. *Proc. 20th Int. Conf. Very Large Data Bases VLDB* 1215 (08 2000).
[7] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. 2004. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery* 8, 1 (2004), 53–87.
[8] Karen Spärck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation* 28 (1972), 11–21.
[9] H. P. Luhn. 1957. A Statistical Approach to Mechanized Encoding and Searching of Literary Information. *IBM J. Res. Dev.* 1 (1957), 309–317.